Turbo C®

User's Guide

Version 2.0

Copyright© 1988
All rights reserved

All Borland products are trademarks or registered trademarks of
Borland International, Inc. Other brand and product names are trademarks or registered
trademarks of their respective holders.
Copyright© 1988 Borland International.

This manual was produced with
Sprint® The Professional Word Processor

Table of Contents

Introduction 1

The Turbo C Package 1

What's New in Turbo C 2.0 2

Requirements 2

The Turbo C Implementation 3

Volume I: The *Turbo C User's Guide* 3

Volume II: The *Turbo C Reference Guide* 4

Recommended Reading 5

Typographic Conventions 6

Borland's No-Nonsense License Statement 6

How to Contact Borland 7

Chapter 1 Before You Begin 9

In This Chapter 10

The README File 10

Installing Turbo C on Your System 10

  If You Are Installing Turbo C on a Floppy-Disk System 10

  Running INSTALL 11

  Setting Up Turbo C on a Laptop System 11

  MicroCalc 12

Where to Now? 12

  Programmers Learning C 12

  Experienced C Programmers 12

  Turbo Pascal Programmers 13

  Turbo Prolog Programmers 13

Chapter 2 Getting Started 15

In This Chapter... 15

HELLO.C: Building and Running a Single-File Program 16

    Step 1: Load TC 16

    Step 2: Choose the Working Directory (Optional) 16

    Step 3: Set Up Your Working Environment 17

    Step 4: Load the Source File into the Editor 18

    Step 5: Build the Executable File 19

    Step 6: Run the Program 20

What Have You Accomplished? 20

Editing Your Program 21

If You Did Something Wrong 22
Sending Your Output to a Printer 22
Writing Your Second Turbo C Program 23
Writing to Disk 24
Running SUM.C 24

Chapter 3 Putting It All Together—Compiling and Running Your Program 25
In This Chapter... 25
Building Files in TC, Revisited 26
Debugging Your Program 26
Catching Syntax Errors: The Error-Tracking Feature 27
The Message Window 27
Correcting a Syntax Error 28
Catching Run-Time Errors: The Integrated Debugger 29
Projects: Using Multiple Source Programs 29
Building a Multi-Source File Program 30
Error Tracking Revisited 31
Stopping a Make 31
Syntax Errors in Multiple Source Files 32
Keeping and Getting Rid of Messages 33
The Power of Project-Make 34
Explicit Dependencies 34
Autodependency Checking 35
What? More Make Features? 36
External Object and Library Files 36
Overriding the Standard Files 37
Compiling and Linking from a Command Line 37
The TCC Command Line 38
Options on the Command Line 38
File Names on the Command Line 38
The Executable File 39
Some Sample Command Lines 39
The TURBOC.CFG File 40
The TCCONFIG.EXE Conversion Utility for Configuration Files 41
The MAKE Utility 42
BUILTINS.MAK 42
Running Turbo C Programs from the DOS Command Line 42
Moving Ahead with Turbo C 43

Chapter 4 Debugging Your Program 45
In This Chapter... 45
How the Integrated Debugger Works 46
Example 1: Debugging a Simple Program 48

Setting and Using a Breakpoint 51
Using *Ctrl-Break* 52
Stepping Over Function Calls 52
Evaluating an Expression 53
The nextword and wordlen Functions 54
Stop and Think 54
What You've Accomplished 55
The Default Expression in the Evaluate Window 56
Changing the Value of an Evaluated Expression 56
Qualifying Variable Names 57
Format Specifiers 57
Exercise 2: Finding the Bug in wordlen 62
Fixing the Bug 64
What You've Accomplished 64
More About Breakpoints 64
Exercise 3: Back to the Program 66
Editing and Deleting Watch Expressions 67
Zooming and Switching Windows 68
Scrolling Watch Expressions 68
Exercise 4: Debugging the Print Loop 68
Exercise 5: Working with Large Programs 70
Finding the Definition of a Function 70
The Call Stack 70
Returning to the Execution Position 71
About Multiple Source Files 71
Survey of Debugger Commands and Hot Keys 72
Guidelines for Effective Software Testing 74
Develop a Standard Approach 75
Test Modifications Thoroughly 76
Design Defensively 76
Debug from the Bottom Up 77
Look for Classes of Bugs 77
Debugging Inline Assembly Code 77

Chapter 5 The Turbo C Integrated Development Environment 79
In This Chapter 79
What You Should Read 80
How to Get Help 80
Part I: Using TC 82
TC Command-Line Switches 82
Finding Your Way around TC 84
The TC Hot Keys 85
Menu Structure 87

Menu-Naming Conventions 89
The Main Menu 90
The Quick-Ref Lines 91
The Edit Window 91
Quick Guide to Editing Commands 93
How to Work with Source Files in the Edit Window 94
Creating a New Source File 94
Loading an Existing Source File 95
Saving a Source File 95
Writing an Output File 95
The Message Window 96
The Watch Window 96
The Integrated Debugger 98
Controlling the Debugger 98
The Debugger Screen Display 99
Debugging Menu Commands and Hot Keys 99
Part II: The Menu Commands 102
The File Menu 102
Load 103
Pick 103
New 103
Save 103
Write To 104
Directory 104
Change Dir 104
OS Shell 104
Quit 104
The Edit Command 104
The Run Menu 105
Run 105
Program Reset 106
Go to Cursor 106
Trace Into 107
Step Over 107
The Compile Menu 108
Compile to OBJ 109
Make EXE File 109
Link EXE File 110
Build All 110
Primary C File 110
Get Info 111
The Project Menu 111
Project Name 112

Break Make On 112
Auto Dependencies 113
Clear Project 113
Remove Messages 114
The Options Menu 114
Compiler 115
The Model menu 115
Defines 116
The Code Generation Menu 116
The Optimization Menu 120
The Source Menu 122
The Errors Menu 123
The Names Menu 124
Linker 125
The Map File Menu 126
Initialize Segments 126
Default Libraries 126
Graphics Libraries 127
Warn Duplicate Symbols 127
Stack Warning 127
Case-sensitive Link 127
Environment 127
Message Tracking 128
Keep Messages 128
Config Auto Save 129
Edit Auto Save 129
Backup Files 129
Tab Size 129
Zoomed Windows 130
The Screen Size Menu 130
Directories 130
Include Directories 131
Library Directories 131
Output Directory 132
Turbo C Directory 132
Pick File Name 132
Current Pick File 132
Arguments 133
Save Options 133
Retrieve Options 133
The Debug Menu 133
Evaluate 134
Find Function 137

Call Stack 137
Source Debugging 138
Display Swapping 138
Refresh Display 139
The Break/Watch Menu 139
Add Watch 140
Delete Watch 141
Edit Watch 141
Remove All Watches 141
Toggle Breakpoint 141
Clear All Breakpoints 142
View Next Breakpoint 142
Part III: More about Configuration and Pick Files 143
What Is a Configuration File? 143
The TC Configuration Files 143
What Is Stored in TC Configuration Files? 144
Creating a TC Configuration File 144
Changing Configuration Files Midstream 145
Where Does TC.EXE Look for TCCONFIG.TC? 145
TCINST vs. the Configuration File: Who's the Boss? 145
What Does Options/Environment/Config Auto Save Do? 146
What Are Pick Lists and Pick Files? 146
The Pick List 146
The Pick File 147
When and How Do You Get a Pick File? 147
When Does Turbo C Save Pick Files? 148
Part IV: Additional Features and Editing Commands 149
More on Tabs 149
Autoindent, Unindent, and Optimal Fill 149
Examples 150
Pair Matching 151
A Few Details about Pair Matching 152
Directional and Nondirectional Matching 152
Nestable Delimiters 153
The Search for Comment Delimiters 154
Editing Key Assignment 155
Chapter 6 Programming in Turbo C 157
In This Chapter 157
The Seven Basic Elements of Programming 158
Output 159
The printf Function 159
The Format String 159
Other Output Functions: puts and putchar 161

Data Types 161
Float Type 162
The Three ints 163
Unsigned 163
Defining a String 163
Using a Character Array 163
Using a Character Pointer 164
Identifiers 165
Operations 166
The Assignment Operator 166
Unary and Binary Operators 166
Increment (++) and Decrement (--) Operators 166
Bitwise Operators 167
Combined Operators 168
Address Operators 169
Input 170
The scanf Function 170
Whitespace 170
Passing an Address to scanf 170
Using gets and getch for Input 171
Conditional Statements 172
Relational Operators 172
Logical Operators 173
More About Expressions 174
Assignment Statements 174
The Comma Operator 174
The if...else Statement 175
Loops 176
The while Loop 176
The for Loop 178
The do...while Loop 179
Functions 180
Breaking Down the Program 181
The get_parms Function 182
The get_ratio Function 182
The put_ratio Function 182
Global Declarations 183
Function Declarations 183
Function Definitions 184
Comments 185
Summary 185

Chapter 7 More Programming in Turbo C 187
In This Chapter... 187
A Survey of Data Structures 188
Pointers 188
Dynamic Allocation 190
Pointers and Functions 191
Pointer Arithmetic 192
Arrays 194
Arrays and Pointers 195
Arrays and Strings 195
Multi-Dimensional Arrays 196
Arrays and Functions 197
Structures 199
Structures and Pointers 200
The switch Statement 200
Control Flow Commands 203
The return Statement 204
The break Statement 204
The continue Statement 205
The goto Statement 206
The Conditional Operator (?:) 206
Streams and Stream I/O 207
What Are Streams? 207
Text vs. Binary Streams 208
Buffering Streams 208
Predefined Streams 209
Style in C Programming: Modern vs. Classic 210
Using Function Prototypes and Full Function Definitions 210
Using enum Definitions 211
Using typedef 212
Declaring void functions 212
Make Use of Extensions 213
String Literals 213
Hexadecimal Character Constants 213
signed Types 214
Pitfalls in C Programming 214
Path Names with C Strings 214
Using and Misusing Pointers 215
Using an Uninitialized Pointer 215
Strings 215
Confusing Assignment (=) with Equality (==) 217
Forgetting the break in switch Statements 218
Array Indexing 218

Failure to Pass-by-Address 219
Sailing Away 220

Chapter 8 Turbo C's Video Functions 221
In This Chapter... 221
Some Words about Video Modes 221
Some Words about Windows and Viewports 222
What Is a Window? 222
What Is a Viewport? 223
Coordinates 223
Programming in Text Modes 223
The Console I/O Functions 223
Text Output and Manipulation 224
Window and Mode Control 225
Attribute Control 226
State Query 227
Text Windows 227
The *text_modes* Type 229
Text Colors 229
High-Performance Output: the *directvideo* Variable 230
Programming in Graphics Mode 231
The Graphics Library Functions 232
Graphics System Control 232
A More Detailed Discussion 234
Drawing and Filling 236
Manipulating the Screen and Viewport 238
Text Output in Graphics Mode 239
Color Control 241
Pixels and Palettes 241
Background and Drawing Color 242
Color Control on a CGA 242
Color Control on the EGA and VGA 244
Error Handling in Graphics Mode 245
State Query 246

Chapter 9 Notes for Turbo Pascal Programmers 249
In This Chapter... 249
Program Structure 250
An Example 251
A Comparison of the Elements of
Programming 253
Output 253
Data Types 254
Operations 255

Input 257
Block Statement 258
Conditional Execution 258
Iteration 262
The while Loop 262
The do..while Loop 262
The for Loop 263
Subroutines 264
Function Prototypes 266
A Major Example 268
A Survey of Data Structures 271
Pointers 271
Arrays 272
Strings 273
Structures 277
Unions 279
Programming Issues 280
Case Sensitivity 280
Type-Casting 281
Constants, Variable Storage, Initialization 281
Constant Types 282
Variable Initialization 282
Variable Storage 283
Dynamic Memory Allocation 283
Command-Line Arguments 285
File I/O 286
Common Pitfalls for Pascal Programmers
Using C 288
PITFALL #1: Assignment vs. Comparison 288
PITFALL #2: Forgetting to Pass Addresses (Especially When Using
scanf) 288
PITFALL #3: Omitting Parentheses on Function Calls 289
PITFALL #4: Warning Messages 289
PITFALL #5: Indexing Multi-dimensional Arrays 290
PITFALL #6: Forgetting the Difference Between Character Arrays and
Character Pointers 290
PITFALL #7: Forgetting that C is Case Sensitive 291
PITFALL #8: Leaving Semicolons Off the Last
Statement in a Block 291

Chapter 10 Interfacing Turbo C with Turbo Prolog 293
In This Chapter... 294
Linking Turbo C and Turbo Prolog: An Overview 294

Example 1: Adding Two Integers 296
Turbo C Source File: CSUM.C 296
Compiling CSUM.C to CSUM.OBJ 296
Turbo Prolog Source File: PROSUM.PRO 297
Compiling PROSUM.PRO to PROSUM.OBJ 297
Linking CSUM.OBJ and PROSUM.OBJ 297
Example 2: Using the Math Library 299
Turbo C Source File: CSUM1.C 299
Turbo C Source File: FACTRL.C 299
Compiling CSUM1.C and FACTRL.C to .OBJ 300
Turbo Prolog Source File: FACTSUM.PRO 300
Compiling FACTSUM.PRO to FACTSUM.OBJ 301
Linking CSUM1.OBJ, FACTRL.OBJ, and FACTSUM.OBJ 302
Example 3: Flow Patterns and Memory Allocation 302
Turbo C Source File: DUBLIST.C 303
Calling Turbo Prolog from Turbo C 304
Lists and Functors 306
Compiling DUBLIST.C 308
Example 4: Drawing a 3-D Bar Chart 308
Turbo C Source File: CBAR.C 309
Compiling CBAR.C 309
Turbo Prolog Program: PBAR.PRO 309
Compiling PBAR.PRO to PBAR.OBJ 309
Linking PBAR.OBJ with the Module CBAR.OBJ 310
That's All There Is to It 310

Chapter 11 Turbo C Language Reference 311
In This Chapter... 311
Comments (K&R 2.1) 312
Identifier (K&R 2.2) 312
Keywords (K&R 2.3) 313
Constants (K&R 2.4) 313
Integer Constants (K&R 2.4.1) 313
Character Constants (K&R 2.4.3) 314
Floating Constants (K&R 2.4.4) 315
Strings (K&R 2.5) 316
Hardware Specifics (K&R 2.6) 316
Conversions (K&R 6) 317
char, int, and enum (K&R 6.1) 317
Pointers (K&R 6.4) 318
Arithmetic Conversions (K&R 6.6) 318
Operators (K&R Section 7.2) 319
Type Specifiers and Modifiers (K&R 8.2) 319

The enum Type 320
The void Type 320
The signed Modifier 321
The const Modifier 321
The volatile Modifier 322
The cdecl and pascal Modifiers 322
pascal 323
cdecl 323
The near, far, and huge modifiers 323
Structures and Unions (K&R Section 8.5) 324
Word Alignment 325
Bitfields 325
Statements (K&R 9) 326
External Function Definitions (K&R 10.1) 326
Function Type Modifiers (K&R 10.1.1) 326
The pascal Function Modifier 326
The cdecl Function Modifier 327
The interrupt Function Modifier 328
The near, far, and huge Function Modifiers 328
Function Prototypes (K&R 10.1.2) 329
Scope Rules (K&R 11) 333
Compiler Control Lines (K&R 12) 333
Token Replacement (K&R 12.1) 334
File Inclusion (K&R 12.2) 335
Conditional Compilation (K&R 12.3) 335
Line Control (K&R 12.4) 336
Error Directive (ANSI C 3.8.5) 336
Pragma Directive (ANSI C 3.8.6) 337
#pragma inline 337
#pragma warn 337
#pragma saveregs 338
Null Directive (ANSI C 3.7) 338
Predefined Macro Names (ANSI C 3.8.8) 338
Turbo C Predefined Macros 339
Anachronisms (K&R 17) 340

Chapter 12 Advanced Programming in Turbo C 341
Memory Models 341
The 8086 Registers 342
General-Purpose Registers 343
Segment Registers 343
Special Purpose Registers 343
Memory Segmentation 344

Address Calculation 344
Near, Far, and Huge Pointers 345
Near Pointers 346
Far Pointers 346
Huge Pointers 347
Turbo C's Six Memory Models 348
Mixed-Model Programming: Addressing Modifiers 353
Declaring Functions to Be Near or Far 354
Declaring Pointers to Be Near, Far, or Huge 355
Pointing to a Given Segment:Offset Address 357
Building Proper Declarators 357
Using Library Files 359
Linking Mixed Modules 360
Mixed-Language Programming 362
Parameter-Passing Sequences: C and Pascal 362
C Parameter-Passing Sequence 362
Pascal Parameter-Passing Sequence 364
Assembly Code Interface 366
Setting Up to Call .ASM from Turbo C 366
Defining Data Constants and Variables 367
Defining Global and External Identifiers 368
Setting Up to Call Turbo C from .ASM 369
Referencing Functions 369
Referencing Data 369
Defining Assembly Language Routines 370
Passing Parameters 371
Handling Return Values 371
Register Conventions 374
Calling C Functions from .ASM Routines 375
Low-Level Programming: Pseudo-Variables, Inline Assembly, and Interrupt Functions 377
Pseudo-Variables 377
Using Inline Assembly Language 379
Opcodes 382
String Instructions 383
Repeat Prefixes 384
Jump Instructions 384
Assembly Directives 384
Inline Assembly References to Data and Functions 384
Inline Assembly and Register Variables 385
Inline Assembly, Offsets, and Size Overrides 385
Using C Structure Members 385
Using Jump Instructions and Labels 387

Interrupt Functions 387
Using Low-Level Practices 388
Using Floating-Point Libraries 390
Emulating the 8087/80287 Chip 391
Using the 8087/80287 Math Coprocessor Chip 392
If You Don't Use Floating Point... 393
The 87 Environment Variable 394
Registers and the 8087/80287 395
Using matherr with Floating Point 396
Caveats and Tips 396
Turbo C's Use of RAM 396
Should You Use Pascal Conventions? 396
Summary 397

Bibliography 399

Index 401

List of Figures

Figure 4.1: Typical Steps in the Debugging Process 47
Figure 5.1: The Main TC Screen 82
Figure 5.2: The TC Menu Structure 88
Figure 5.3: The TC Main Menu Bar 90
Figure 5.4: The File Menu 102
Figure 5.5: The Run Menu 105
Figure 5.6: The Compile Menu 109
Figure 5.7: The Compile/Get Info screen 111
Figure 5.8: The Project Menu 112
Figure 5.9: The Project/Break Make On Menu 113
Figure 5.10: The Options Menu 114
Figure 5.11: The Options/Compiler Menu 115
Figure 5.12: The O/C/Model Menu 116
Figure 5.13: The O/C/Code Generation Menu 117
Figure 5.14: The O/C/Optimization Option 120
Figure 5.15: The O/C/Source Menu 122
Figure 5.16: The O/C/Errors Menu 123
Figure 5.17: Displaying the Common Errors 124
Figure 5.18: The O/C/Names Option 125
Figure 5.19: The Options/Linker Menu 126
Figure 5.20: The Options/Environment Menu 128
Figure 5.21: The Options/Directories Menu 131
Figure 5.22: The Debug Menu 134
Figure 5.23: The Break/Watch Menu 140
Figure 5.24: How Unindent Works 150
Figure 5.25: Search for Match to Square Bracket or Parenthesis 153
Figure 5.26: Nested Comments Toggled On—Forward Search with ^Q ^[....................154
Figure 5.27: Nested Comments Toggled Off—Forward Search with ^Q ^[....................155
Figure 5.28: Nested Comments Toggled Off—Backward Search with ^Q ^] 155
Figure 8.1: A Window in 80x25 Text Mode 228
Figure 12.1: 8086 Registers 342
Figure 12.2: Tiny Model Memory Segmentation 349
Figure 12.3: Small Model Memory Segmentation 350
Figure 12.4: Medium Model Memory Segmentation 350
Figure 12.5: Compact Model Memory Segmentation 351

Figure 12.6: Large Model Memory Segmentation 351
Figure 12.7: Huge Model Memory Segmentation 352

List of Tables

Table 4.1: Debugging Format Specifiers 58
Table 4.2: Priority and Defaults in Format Specifier Classes 61
Table 4.3: Debugger Commands and Hot Keys 72
Table 4.4: Menu Commands and Hot Keys Used with the Debugger 74
Table 5.1: Turbo C Hot Keys 86
Table 5.2: Watch Window Editing Commands 97
Table 5.3: Debugger Commands and Hot Keys 99
Table 5.4: Menu Commands and Hot Keys Used with the Debugger 101
Table 5.5: Format Specifiers Recognized in Debugger Expressions 135
Table 9.1: Pascal and C Operators 256
Table 9.2: File I/O Similarities 286
Table 11.1: Keywords Reserved by Turbo C 313
Table 11.2: Turbo C Integer Constants Without L or U 314
Table 11.3: Turbo C Escape Sequences 315
Table 11.4: Turbo C Data Types, Sizes, and Ranges 317
Table 11.5: Methods Used in Usual Arithmetic Conversions 319
Table 12.1: Memory Models 353
Table 12.2: Pointer Results 354
Table 12.3: Declarators without Typedefs 358
Table 12.4: Declarators with Typedefs 359
Table 12.5: Identifier Replacements and Memory Models 367
Table 12.6: Turbo C Pseudo-Variables 378
Table 12.7: Opcode Mnemonics 383
Table 12.8: String Instructions 384
Table 12.9: Jump Instructions 384

INTRODUCTION

Turbo C is for C programmers who want a fast, efficient compiler; for Turbo Pascal programmers who want to learn C with all the "Turbo" advantages; and for anyone just learning C who wants to start with a fast, easy-to-use implementation.

The C language is a structured, modular, compiled, general-purpose language traditionally used for systems programming. It is portable, so you can easily transfer application programs written in C from one system to another. You can use C for almost any programming task, anywhere. But while traditional C compilers plod along, Turbo C flies through compilation, and gives you more time to test and perfect your programs.

The Turbo C Package

Your Turbo C package consists of a set of distribution disks and the two-volume manual—the *Turbo C User's Guide* (this book) and the *Turbo C Reference Guide*. The distribution disks contain all the programs, files, and libraries you need to create, compile, link, and run your Turbo C programs; they also contain sample programs, several stand-alone utilities, a context-sensitive help file, an integrated debugger, and additional C documentation not covered in these guides.

The *User's Guide* is designed as a handbook and guide for the beginner and a useful refresher course for the experienced C user. The *Reference Guide* is first and foremost a detailed list and explanation of Turbo C's extensive library functions. It also contains information on the Turbo C editor, error messages, utilities (CPP, MAKE, TLINK, TLIB, GREP, BGIOBJ, and OBJXREF), command-line options, Turbo C syntax, and customization. Unless you are already a C programmer, you will probably want to begin with the *User's Guide* before wading into the deeper waters of the *Reference Guide*.

What's New in Turbo C 2.0

Turbo C 2.0 includes many new and improved features:

- Integrated debugging: Step and trace through code, set breakpoints, watch and evaluate expressions
- A faster compiler (20 to 30%) and linker
- EMS storage for the edit buffer: Gives you up to 64K more memory for compiling and running
- Faster memory allocation and string functions
- Faster floating-point emulation
- New **signal** and **raise** functions
- An **__emit__** feature that lets you insert machine code into your program at compile time.
- An enhanced BGI graphics library, with many new functions, including installable drivers and fonts
- Support for command-line wildcard expansion
- Linker can create .COM files for tiny model programs
- Support for Borland's new standalone debugger
- Autodependency checking for the MAKE utility
- Support for **long double** constants and variables
- New editor features, including block indent/unindent and optimal fill

Requirements

Turbo C runs on the IBM PC family of computers, including the XT, AT, and PS/2, along with all true IBM compatibles. Turbo C requires DOS 2.0 or higher and at least 448K of RAM; it will run on any 80-column monitor. One floppy disk drive is all that's required, although we recommend two floppy drives or a hard disk with one floppy drive.

Turbo C includes floating-point routines that let your programs make use of an 80×87 math coprocessor chip. It will emulate the chip if it is not available. The 80×87 chip can significantly enhance performance of your programs, but it is not required.

The Turbo C Implementation

Turbo C supports the Draft-Proposed American National Standards Institute (ANSI) C standard, fully supports the Kernighan and Ritchie definition, and includes certain optional extensions for mixed-language and mixed-model programming that allow you to exploit your PC's capabilities.

Volume I: The *Turbo C User's Guide*

The *Turbo C User's Guide* (this volume) introduces you to Turbo C, shows you how to create and run programs, and includes background information on topics such as compiling, linking, debugging, and project making. Here is a breakdown of the chapters in the *User's Guide*:

Chapter 1: Before You Begin tells you how to install Turbo C on your system. It also suggests how you should go about using the rest of the *User's Guide*.

Chapter 2: Getting Started teaches you basics about using the Turbo C integrated development environment (TC) to load, compile, run, edit and save a simple Turbo C program.

Chapter 3: Putting It All Together—Compiling and Running Your Program shows how to use the Turbo C **R**un command, and explains how to "make" (rebuild) a program's constituent files.

Chapter 4: Debugging Your Program introduces you to the Turbo C integrated debugger and walks you through a sample program with built-in bugs to demonstrate various features of the debugger.

Chapter 5: The Turbo C Integrated Development Environment explains Turbo C's text editor, integrated debugger, and menu system, and discusses pick files and configuration files.

Chapter 6: Programming in Turbo C introduces you to some of the basic steps involved in creating and running Turbo C programs and takes you through a set of short, progressive sample programs.

Chapter 7: More Programming in Turbo C provides summary explanations of additional C programming elements including arrays, pointers, structures, and statements.

Chapter 8: Turbo C's Video Functions first briefly discusses video modes and windows, then describes programming in text mode versus programming in graphics mode.

Chapter 9: Notes for Turbo Pascal Programmers uses program examples to compare Turbo Pascal to Turbo C, describes and summarizes the significant differences between the two languages, and gives some tips on avoiding programming pitfalls.

Chapter 10: Interfacing Turbo C with Turbo Prolog shows how to interface modules written in Turbo C with Turbo Prolog programs and provides several examples that demonstrate the process.

Chapter 11: Turbo C Language Reference lists all aspects and features of this implementation that differ from Kernighan and Ritchie's definition of the language, and details the Turbo C extensions not given in the current draft of the ANSI C standard.

Chapter 12: Advanced Programming in Turbo C provides details about the start-up code, memory organization in the different memory models, pointer arithmetic, assembly-language interface, and the use of floating-point.

Volume II: The *Turbo C Reference Guide*

The *Turbo C Reference Guide* is written for experienced C programmers; it provides implementation-specific details about the language and the run-time environment. In addition, it describes each of the Turbo C functions, listed in alphabetical order. These are the chapters and appendixes in the programmer's *Reference Guide*:

Chapter 1: Using Turbo C Library Routines lists Turbo C's #include (*.H) files and each of its library routines by category, discusses function **main** and its arguments, and concludes with a description of each of the Turbo C global variables.

Chapter 2: The Turbo C Library is an alphabetical reference of all Turbo C library functions. Each entry gives syntax, include files, an operative description, return values, and portability information for the function, together with a reference list of related functions and examples of how the functions are used.

Appendix A: The Turbo C Interactive Editor gives a more thorough explanation of the editor commands for those who need more information than is given in Chapter 5 of the *User's Guide*.

Appendix B: Compiler Error Messages lists and explains each of the error messages and summarizes the possible or probable causes of the problem that generated the message.

Appendix C: Command-Line Options lists the command-line entry for each of the user-selectable TCC (command-line compiler) options.

Appendix D: Turbo C Utilities discusses the utilities included in the Turbo C package; CPP, the preprocessor; MAKE, the program builder; TLINK, the Turbo Link utility; TLIB, the Turbo librarian; GREP, the file search utility; BGIOBJ, a conversion utility for graphics drivers and fonts; and the object module cross-referencer OBJXREF.

Appendix E: Language Syntax Summary uses modified Backus-Naur Forms to define the syntax of all Turbo C constructs.

Appendix F: TCINST: Customizing Turbo C takes you on a walk through the customization program (TCINST), which lets you customize your keyboard, modify default values, change your screen colors, and so on.

Appendix G: MicroCalc explains how to compile, run, and use MicroCalc, the sample spreadsheet program included on the Turbo C distribution disks.

Recommended Reading

You will find these documents useful supplements to your Turbo C manuals:

- The most widely known description of C is found in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie (New Jersey: Prentice-Hall, 1978).
- The ANSI Subcommittee X3J11 on Standardization of C is presently creating a formal standard for the language, and Turbo C supports this upcoming ANSI C standard.

If you are learning C for the first time, we recommend that you use Turbo C to work through the exercises in Kernighan and Ritchie. If you are experienced with C, you should have little difficulty using Turbo C.

Refer to the bibliography in the back of this manual for other books on C and Turbo C.

Typographic Conventions

All typefaces used in this manual were produced by Borland's Sprint: The Professional Word Processor, on a PostScript laser printer. Their uses are as follows:

`Monospace type`	This typeface represents text as it appears onscreen or in a program, or anything you must type (such as command-line options or switches).
[ ]	Square brackets in text or DOS command lines enclose optional input or data that depends on your system. *Text of this sort should not be typed verbatim.*
< >	Angle brackets in text or on DOS command lines enclose optional input or data that depends on your system. *Text of this sort should not be typed verbatim.* Angle brackets in the function reference section enclose the names of include files.
Boldface	Turbo C function names (such as **printf**) and structure names are shown in boldface when they appear in text (but not in program examples). This typeface is also used, in text but not in program examples, for Turbo C keywords such as **char**, **switch**, **near**, and **cdecl**.
Italics	Italics indicate variable names (identifiers) that appear in text. They are also used to emphasize certain words (especially new terms).
Keycaps	This special typeface indicates a key on your keyboard. It is often used to describe a particular key you should press; for example, "Press *Esc* to exit a menu."

Borland's No-Nonsense License Statement

This software is protected by both United States copyright law and international treaty provisions. Therefore, you must treat this software *just like a book* with the following single exception: Borland International

authorizes you to make archival copies of Turbo C for the sole purpose of backing up your software and protecting your investment from loss.

By saying, "just like a book," Borland means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another so long as there is **no possibility** of its being used at one location while it's being used at another. Just like a book that can't be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time. (Unless, of course, Borland's copyright has been violated.)

How to Contact Borland

The best way to contact Borland is to log on to Borland's Forum on CompuServe: Type `GO BOR` from the main CompuServe menu and choose "Borland Programming Forum B (Turbo Prolog & Turbo C)" from the Borland main menu. Leave your questions or comments there for the support staff to process.

Technical Support Department

INTERNATIONAL HEADQUARTERS :
BORLAND INTERNATIONAL
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95066-0001
U.S.A.
Tél. : (1) (408) 438-8400
Telex : 172373
Fax : (1) (408) 438-8696

EUROPEAN HEADQUARTERS :
BORLAND INTERNATIONAL FRANCE
43, Avenue de l'Europe
B.P. 6.
78141 VELIZY CEDEX
FRANCE
Tél. : (33) (1) 39-46-96-69
Telex : 698793
Fax : (33) (1) 39-46-81-60

U.K. OFFICES :
BORLAND INTERNATIONAL (U.K.) LTD
8 Pavilions
Ruscombe Business Park
TWYFORD, BERKSHIRE RG10 9NN
UNITED KINGDOM
Tel. : 0734-320022
Telex : 846616
Fax : 0734-320017

Please have the following information handy before you call:

- product name and version number
- computer make and model number
- operating system and version number

C H A P T E R

1

Before You Begin

Your Turbo C package actually includes two different versions of the C compiler: the integrated development environment version and a separate, stand-alone, command-line version. When you install Turbo C on your system, you copy files from the distribution disks to your working floppies or to your hard disk. There is no copy protection, and an installation program is included to make it simple to install Turbo C. The distribution disks are formatted for double-sided, double-density disk drives and can be read by IBM PCs and close compatibles. For reference, we include a list of the distribution files in the README file on the Installation Disk.

We assume you are already familiar with DOS commands. For example, you will need the DISKCOPY command to make backup copies of your distribution disks. **If you do not already know how to use DOS commands,** refer to your DOS reference manual before starting to set up Turbo C on your system.

You should make a complete working copy of the distribution disks when you receive them, then store the original disks away in a safe place. Do *not* run Turbo C from the distribution disks; they are your original (and only) backups in case anything happens to your working files.

If you are not familiar with Borland's No-Nonsense License Statement, now is the time to read the agreement in the Introduction (it's also at the front of this book) and mail us your filled-in product registration card.

In This Chapter ...

We begin this chapter with instructions for accessing the README file and installing Turbo C on your system. The rest of the chapter is devoted to some recommendations on which chapters you should read next, based on your programming language experience.

The README File

It is very important that you take the time to look at the README file on the Installation Disk before you do anything else with Turbo C. This file contains last-minute information that may not be in the manual. It also lists every file on the distribution disks, with a brief description of what each one contains.

To access the README file, insert the Installation Disk in Drive A, switch to Drive A by typing `A:` and pressing *Enter*, then type `README` and press *Enter* again. Once you are in README, use the *Up* and *Down arrow* keys to scroll through the file. Press *Esc* to exit.

Installing Turbo C on Your System

Your Turbo C package includes all the files and programs necessary to run both the integrated-environment and command-line versions of the compiler, along with start-up code and library support for six memory models and 8087/80287 coprocessor emulation. If you are installing Turbo C for the first time, or installing the upgrade from the previous version (1.5), the INSTALL program makes it easy.

If You Are Installing Turbo C on a Floppy-Disk System

If your system has one or two floppy disk drives but no hard drive, you must have a set of three *formatted, empty* disks ready *before* you run INSTALL.

Each time you run INSTALL, it will let you install Turbo C with one memory model. If you want to install more than one memory model, you

must have additional sets of disks, one for each memory model you want to install.

Running INSTALL

The Turbo C installation program INSTALL is designed to walk you through the installation process. All you have to do is follow the instructions that appear onscreen at each step. *Please read them carefully.*

To run INSTALL:

1. Insert the distribution disk labeled Installation Disk in Drive A.
2. Type `A:` and press *Enter.*
3. Type `INSTALL` and press *Enter.*

From this point on, just follow the instructions that INSTALL displays onscreen.

As soon as INSTALL is finished running, you are ready to start using Turbo C.

Note: After you have tried out the Turbo C integrated development environment, you may want to permanently customize some of the options. We give you a program called TCINST that will make this easy to do. See Appendix F in the *Turbo C Reference Guide* for instructions.

Setting Up Turbo C on a Laptop System

If you have a laptop computer (one with an LCD or plasma display), in addition to carrying out the procedures given in the previous sections you should set your screen parameters before using Turbo C. The Turbo C Integrated Development Environment version (TC.EXE) works best if you enter `MODE BW80` at the DOS command line before running Turbo C.

Alternatively, you can install TC for a black-and-white screen with the Turbo C customization program, TCINST. Refer to Appendix F in the *Turbo C Reference Guide.* With this customization program, you should choose "Black and White" from the Screen Modes menu.

MicroCalc

We have included the source code for a spreadsheet program called MicroCalc. Before you try to compile it, read Appendix G in the *Turbo C Reference Guide.*

Where to Now?

Now that you have finished installing Turbo C, you are ready to start digging into this guide and using Turbo C. But since this user's guide is written for four different types of users, certain chapters are written with your particular Turbo C programming needs in mind. Take a few moments to read the following, then take off and fly with Turbo C speed!

Programmers Learning C

If you are just now learning the C language, you will want to start with Chapters 2 and 3, which introduce you to the Turbo C integrated development environment and shows you how to load, compile, link, and run a simple Turbo C program, as well as how to edit and save your own creations. Chapter 4 will introduce you to the Turbo C integrated debugger. Next, read Chapters 6 and 7. These are written in tutorial fashion and take you through the process of creating and compiling C programs. If you are not sure how to use the integrated development environment, you will need to read Chapter 5. Chapter 8 will introduce you to Turbo C's exciting graphics features.

Experienced C Programmers

If you are an experienced C programmer, you should have little difficulty porting your programs to the Turbo C implementation. You will want to read Chapter 11, "Turbo C Language Reference," however, for a summary of how Turbo C compares to Kernighan and Ritchie and to the draft ANSI C standard. When you are ready to port or create C programs with Turbo C, you will need to read Chapter 3, "Putting It All Together—Compiling, Debugging, and Running," Chapter 4 about how to use the Turbo C integrated debugger, and Chapter 12, "Advanced Programming in Turbo C." If you are interested in exploring what you can do with Turbo C graphics, read Chapter 8.

Turbo Pascal Programmers

Chapter 9, "Notes for Turbo Pascal Programmers," is written specifically for you; in it, we provide some examples that compare Turbo Pascal programs with equivalent Turbo C programs, and we elaborate on some of the significant differences between the two languages.

If you have programmed with Turbo Pascal, you are familiar with the seven basic elements of programming. To get up to speed with Turbo C, you will want to read Chapters 5, 6, and 7. (If you have used another menu-driven Borland product, such as SideKick or Turbo Basic, you will only need to skim Chapter 5.) You should also look at Chapter 3, on compiling and running your Turbo C programs, and Chapter 4, on the Turbo C integrated debugger.

Turbo Prolog Programmers

If you have used Turbo Prolog and would like to know how to interface your modules with Turbo C, you should read Chapter 10.

C H A P T E R

2

Getting Started

Now you have Turbo C installed on your system, and you are ready to start programming. But first you have to find out a few basics, like how to run Turbo C, how to use a text editor to create and modify your programs, and how to compile and run them.

You can use any ASCII text editor to create your programs, and then compile and run them from the DOS command line using the command line compiler (the TCC version of Turbo C). However, you will probably find it easier, at least at first, to work in the Turbo C integrated development environment (the TC version of Turbo C), which provides you with an editor, a menu system of Turbo C commands, an integrated debugger, and a built-in Project-Make facility that lets you compile and run your programs from within the TC environment.

Note: We will explain as we go along how to use the TC menus to perform the exercises in this chapter. If you want a comprehensive explanation of the whole TC menu system, refer to Chapter 5 in this manual.

In This Chapter...

We start out by teaching you a few basic skills that you will need to use Turbo C: loading the Turbo C integrated development environment (TC), loading a program into Turbo C, and building and running the program.

Next, we show you how to modify your program, using the TC Editor.

Finally, we show you how to create an all-new Turbo C program and save it to its own file before you build and run it.

HELLO.C: Building and Running a Single-File Program

Let's start out easy. Before you plunge in and start writing programs of your own in Turbo C, let's practice using the integrated development environment (TC) version of Turbo C with a program that already exists.

In the directory where you installed your example programs, there is a file called HELLO.C that contains the source code for a very simple program. Working with it will demonstrate for you the six steps to building and running a single-file Turbo C program.

Step 1: Load TC

If you installed Turbo C with the INSTALL program, TC should already be in your main Turbo C directory. Just go into that directory and load TC by typing `TC` on the DOS command line and pressing *Enter*.

Note: If you want to do your programming in a separate working directory instead of in the directory that contains TC, you have to tell DOS where to find the TC program:

- Specify the directory with TC in it by using the DOS PATH command. (See PATH in your DOS manual; be careful that you do not destroy existing PATHs when you enter the TC path.).
- In 3.x versions of DOS, you can type the path name to TC directly on the command line, for example, `\TURBOC\TC`.

Step 2: Choose the Working Directory (Optional)

If your current directory is the one that contains HELLO.C, you can skip this step.

Choose the drive and directory that contain HELLO.C, the source file you want to load. Do this by going to the pull-down **F**ile menu (press *F10*, then *F*, or just press *Alt-F*). Select **C**hange Dir (use the arrow keys to position the highlight bar and press *Enter*, or just type *C*). When the New Directory

prompt box appears, type in the name of the directory that contains HELLO.C and press *Enter*. This directory becomes the current directory.

Note: When the New Directory prompt box comes up, it lists the current directory. That means you can use the File/Change Dir option to check what directory you are in; simply choose it, so that the New Directory prompt box appears, then press *Esc* to get back to the menus without changing the current directory.

Step 3: Set Up Your Working Environment

If you used the INSTALL program to install Turbo C on your system, the working environment has been set up for you already. You may want to read this section anyway, to verify that your environment is set up correctly.

To set up and save your working environment, press *F10*, then *O* (or press *Alt-O*) to invoke the **O**ptions menu from the main menu bar. Then select **D**irectories to bring up the **D**irectories menu. You will need two of the items on this menu: **I**nclude Directories and **L**ibrary Directories.

Choose **I**nclude Directories, then type the name of the drive and directories that contain the Turbo C standard include files (.h files), separating the directory names with semicolons. The include directories will usually be C:\TURBOC\INCLUDE and C:\TURBOC\INCLUDE\SYS; you would type

```
C:\TURBOC\INCLUDE;C:\TURBOC\INCLUDE\SYS
```

and press *Enter*.

Now choose **L**ibrary Directories, and type in the name of the drive and directory that contains the Turbo C library and startup files. (This will usually be C:\TURBOC\LIB.) Other directory names may be entered; separate them with semicolons.

Note: At this point, if you wish, you can set the output directory (where your compiled program will be stored) with the **O**ptions/**D**irectories/**O**utput Directory command. If you do choose an output directory, all compiler and linker output will be written to that directory instead of to the directory you are currently in. In our present example case, it is not necessary to set an output directory.

For most simple cases, this is all the setup necessary for building C programs.

You can save the settings for your working environment in a configuration file that is loaded automatically when you start TC. Press *Esc* to get back to the **O**ptions menu, then choose **S**ave Options to write the current options to a configuration file on disk. The default file, TCCONFIG.TC, will be written to the current directory.

Note: When it starts up, TC looks in the current directory for a file called TCCONFIG.TC, which it loads if it is present. If you wish, you can give the configuration file another name by typing in the new name and pressing *Enter*. If you do this, however, you will need to load this configuration file *explicitly* the next time you enter TC, either by typing its name on the TC command line with the `/c` switch (see Chapter 5 on TC command-line switches), or by using the **O**ptions/**R**etrieve Options command.

Note: When you are working on a particular program, it is useful to have a default configuration file in the same directory as the program and to start TC from that directory. However, if the configuration file is not found in the current directory, TC will also look in the Turbo C directory. This means that you can keep one general purpose configuration file in the Turbo C directory and others in your various source file directories that use different settings.

Step 4: Load the Source File into the Editor

Now load HELLO.C. Choose the **L**oad command from the **F**ile menu, or press *F3*, the file load hot key. A prompt box will appear containing the wildcard notation `*.C`. Type in `HELLO` (you do not need to include the .C extension), and press *Enter*.

Note: If you aren't sure about the name of the file you want to load, or you want to see a listing of all the source files in the current directory, just press *Enter* instead of typing in the file name. TC will display a menu of all the available .C source files in the directory. To choose a file, use the arrow keys to move the highlight bar to the name of the file you want, then press *Enter*.

The HELLO.C file is now displayed in the TC Editor window. It should look like this:

```
/*   HELLO.C -- Hello, world */

#include <stdio.h>

main()
{
   printf("Hello, world\n");
}
```

Note: It is possible to load TC, the source file, and the configuration file from the command line, which eliminates having to bother with Steps 2, 3, and 4. The integrated development environment accepts two command-line arguments that accomplish this: the file name of a source code file to be loaded into the editor, and a /c switch immediately followed by the name of the configuration file you want to load with the source file. These two arguments can be in any order. Thus,

```
tc hello /cmyconfig
```

will place HELLO.C in the editor and load the configuration file MYCONFIG.TC. (Note that there is no space between the /c switch and the file name, and that the default extension .C is assumed for the edit file and the default extension .TC is assumed for the configuration file.)

Step 5: Build the Executable File

When you *build* a program, you first compile the source code to create an *object file* (a machine code file with an .OBJ extension). Then you send the object file to the *linker* to be converted to an *executable file* with a .EXE extension. The linker copies into your object file certain necessary subroutines from the standard run-time library files. (You told Turbo C where to find these library files, remember, back when you set up your working environment.)

In this simple case of a single-file program, you can build and run the program without creating a *project file* (more on project files in Chapter 3).

Though there are other approaches, the easiest way to build your program is to press *F10*, then *C* to bring up the Compile menu (or press *Alt-C*), then choose Make EXE File (or press *F9*, the make .EXE file hotkey). Observe that the Compile menu tells you the name of the object (.OBJ) file that will be compiled and the .EXE file that will be built.

The Compiling window will appear on the screen, and Turbo C should successfully compile and link your program. If all went well, the Compiling window will give you a flashing `Press any key` message.

Note: If there is anything wrong with your program, you will see error messages or warnings in the Message window at the bottom of the screen. If this happens, make sure that your program is typed in exactly as it is above, then compile it again.

Step 6: Run the Program

At this point, you should have an executable program.

Now, to run your program, choose **Run** from the **Run** menu, or press *Ctrl-F9*, the run program hot key.

What happened? You saw the screen flash, and then you were back in the main TC screen. To see the output from the program, select **Run/User** Screen, or press *Alt-F5*. This brings up the User screen, which is where your screen output went.

The User screen should contain the message,

```
Hello, world
```

After you have examined your program output, press any key to return to the TC screen.

What Have You Accomplished?

Now get out of Turbo C (choose the **Quit** command from the **File** menu or press *Alt-X*).

Let's look at what you've created.

At the DOS prompt, type `dir hello.*` and press *Enter*. You'll get a list of files that looks something like this:

```
HELLO    C       104   5-11-88   2:57p
HELLO    OBJ     458   5-11-88   3:01p
HELLO    EXE    8884   5-11-88   3:01p
```

The first file, HELLO.C, is the source for your program. It contains the text (the *source code*) of your program. You can display it on the screen; just enter (at the DOS prompt) the command `type hello.c`. As you can see, HELLO.C isn't very big—only 104 bytes.

The second file, HELLO.OBJ, is your *object file*. It contains the binary machine instructions (the *object code*) produced by Turbo C compiler. If you use the DOS TYPE command to display this file on the screen, you'll get mostly gibberish.

The last file, HELLO.EXE, is the actual *executable file* produced by the Turbo Linker. It contains not only the code in HELLO.OBJ, but also all the necessary support routines (such as **printf**) that the linker copied in from

the library file. To run any executable file, you just type its name at the DOS prompt, without the .EXE extension.

To run HELLO.EXE, type `hello` at the DOS prompt, and press *Enter*. The message `Hello, world` will appear on the screen, and the DOS prompt will come back again.

Editing Your Program

Tradition has it that your first C program should always be the *Hello, world* program found in the classic work, *The C Programming Language* by Kernighan and Ritchie. This is the little HELLO.C program you have just finished building and running.

Are you feeling brave? Now that you are somewhat familiar with the Turbo C integrated development environment, let's try doing some programming of your own. We'll start by making some modifications to the HELLO.C program. To do this, you must learn to use the TC Editor.

If you're not already there, get back into Turbo C by typing `tc hello` at the DOS prompt. You'll find yourself back in TC, with your program already loaded in.

Now let's modify your program so that you can interact with it a little.

Notice the flashing cursor in the upper left corner of the screen. You can move this cursor around the Edit window with the arrow keys. To enter code, just move the cursor to the right spot and type in the code. You can delete a line of code by typing `Ctrl-Y`, and insert a line by typing `Ctrl-N`. Make sure you are in Insert mode (the word `Insert` should appear in the status line at the top of the Edit window; if it doesn't, press *Ins* to toggle it on). (For complete information on how to use the TC Editor, see Chapter 5 in this manual and Appendix A in the *Turbo C Reference Guide.*)

Go ahead and edit your program so it looks like this:

```
#include <stdio.h>

main()
{
   char   name[150];

   printf("What's your name?\n");
   scanf("%s",name);
   printf("Hello, %s\n",name);
}
```

You've added three lines to HELLO.C. The first line (char name[150];) declares a variable named *name*, which can hold a string of up to 150 characters (letter, digit, punctuation, etc.). (Position 150 is reserved for a special character that we'll tell you about later.) The second line you added calls the function **printf** to write out the message What's your name?. The third new line calls the function **scanf** to read a name into the variable *name*.

Next, press *Ctrl-F9* to run your program. Notice that Turbo C is smart enough to know that you have modified your source code, so it recompiles the program before running it.

This time when your program runs two things happen: The User screen appears, with the message What's your name? and the cursor sits waiting on the next line. Type in your name and press *Enter*. Press *Alt-F5*. The User screen now says Hello, <your_name>. Note that it only read the first name you typed in; you'll learn why in Chapter 6. For now, press any key to return to the TC screen.

If You Did Something Wrong

As you write programs, you will make errors or receive warnings. An *error* is a mistake in your program that prevents Turbo C from compiling it to make object code. A *warning* is just that: a message that points out a possible problem. Errors and warnings appear in the Message window at the bottom of the TC screen. There are many different errors and warnings; they are covered in more detail in Appendix B of the *Turbo C Reference Guide*.

Sending Your Output to a Printer

Are you wondering how to send your HELLO.EXE program output to a printer instead of to the screen? We'll show you how here, although we won't go into the details of how this works just yet; you have plenty to learn for now, and we want to save some of the fun for later.

Load HELLO.C into the Editor, and modify it to look like this:

```
#include <stdio.h>

main()
{
   fprintf(stdprn,"Hello, World\n");
}
```

Make sure your printer is ready, then compile and build your program just as you did before, by pressing *Ctrl-F9*. Your printer should print out the message `Hello, world.`

Note that this time we've used the **fprintf** function instead of **printf**. As you gain more expertise with Turbo C and venture into the *Turbo C Reference Guide*, you'll learn more about these elements we've added.

Writing Your Second Turbo C Program

Now let's modify your HELLO.C program some more, and store it in a new file. You should still be in the Editor, but if you aren't (there is no flashing cursor), either press *Alt-E* for the quick shortcut, or press *F10* to activate the menu system, and *E* to select the Editor. Now change your program so that it looks like this:

```
#include <stdio.h>

main()
{
   int  a,b,sum;

   printf("Enter two numbers:  ");
   scanf("%d %d",&a,&b);
   sum = a + b;
   printf("The sum is %d \n",sum);
}
```

You have made five changes to the original program. You have

- replaced the line defining *name* with one defining other variables (*a*, *b*, and *sum*, all integers)
- changed the message in the **printf** statement
- changed the format string and variable list in the **scanf** statement
- added the assignment statement `sum = a + b;`
- changed the format string and argument list in the final **printf** statement

Don't let the percent signs (%), ampersands (&), and backslashes (\) confuse you; we'll explain what they mean in Chapter 6.

Writing to Disk

Now, do *not* press the *F2* function key. If you do, this program will be saved as HELLO.C (you are going to save it under a different name).

Instead, press *Alt-F* to get to the **F**ile menu. Press *W* to select the **W**rite To command. Turbo C will ask you to type in the new name for this program; type `sum.c` and press *Enter*. Your second program has now been saved on disk as SUM.C.

Running SUM.C

Press *Ctrl-F9*. Turbo C will compile your program. If there are any errors, go back into the editor and be sure that what you've typed in matches exactly what is given in the example.

Once there are no errors, Turbo C will link in the appropriate library routines and then run your program. The User screen will appear, with this message:

```
Enter two numbers:
```

Your program is waiting for you to enter two integer values, separated by blanks and/or tabs and/or carriage returns. Be sure to press *Enter* after typing the second value. Your program now prints the sum of those two values on the User screen; select **R**un/**U**ser screen (or press *Alt-F5*) to see the result. Press any key to return to Turbo C.

Congratulations! You've now written two completely different Turbo C programs using several of the basic elements of programming. Are you wondering what those elements are? You can find out by reading Chapter 6, before going on to Chapter 7.

C H A P T E R

3

Putting It All Together—Compiling and Running Your Program

Now that you have had some experience using Turbo C, let's move on to some more complicated issues—more advanced features of the Turbo C integrated development environment, and command-line Turbo C.

Turbo C provides a flexible environment for C program development; it comes with default option settings to get you started, but you can easily change these defaults to best meet your programming needs. Turbo C also provides various support tools to perform the routine chores associated with program development, such as error tracking and file-system management.

If you are not familiar with Borland's easy-to-use integrated development environment (TC), you should look over Chapter 5 before compiling and running your programs through TC's menu system. It is a logical and easy system to learn, and it won't take long for you to feel comfortable using it.

In This Chapter...

Because you can compile and run your Turbo C programs either from the integrated development environment or from a standard DOS command line, we discuss both processes in this chapter. However, because the integrated development environment is a complete package, powerful and easy to use, we think you will want to know about it first.

We begin this chapter with a brief review of how you compile and link Turbo C source files through the integrated development environment to produce executable programs. This is followed by a brief discussion of TC Turbo C's debugging features.

We then demonstrate how to run your programs from the integrated development environment; we also introduce TC's built-in Project facility, Project-Make, and demonstrate how to use it.

After showing you how to run programs within the integrated development environment, we explain how to use the command line for compiling, linking, making (rebuilding), and running your Turbo C programs. In addition to the integrated development environment version of Turbo C, your package includes a stand-alone compiler (TCC), linker, and MAKE utility. Specific details on these stand-alone programs are given in Appendixes C and D of the *Turbo C Reference Guide.*

Building Files in TC, Revisited

Building a new program in the Turbo C integrated development environment (TC) usually entails going through the following steps.

1. Set directory options so the compiler and linker know where to find and store things.
2. Load the program you want to build into the TC Editor. (**Note:** If the program consists of more than one module, you need to create a *project file* that lists the names of your modules.)
3. Build the executable program file.

The exact procedure in these general steps differs depending on whether you're working with one file or several files as your source.

Debugging Your Program

Finding and fixing errors in your programs is always one of the more frustrating aspects of programming. The Turbo C integrated development environment (TC) makes your job a lot easier by providing debugging features to help you out on both the compile-time and the run-time levels.

Catching Syntax Errors: The Error-Tracking Feature

One of the best reasons to use TC is that it lets you fix syntax (compile-time) errors and evaluate any warnings the compiler gives you. TC collects compiler and linker messages in a buffer and then displays them in the Message window. This lets you look at all the messages at once while you still have direct access to your source code.

To try this out, add some syntax errors to the HELLO.C program. Remove the # from the include statement on the first line. Next take out the trailing quotation mark in the **printf** string on the fifth line. The now-buggy file should look like this:

```
include <stdio.h>

main ()
{
   printf("Hello world\n);
}
```

Now compile the file again by pressing *Alt-F9* (the compile to .OBJ hot key). The Compiling window will tell you how many errors and warnings you have introduced (there should be two errors and no warnings).

The Message Window

When you see the message `Press any key` in the Compiling window, press the spacebar. The Message window will become active, and a highlight bar will be placed on the first error or warning. Since the first error occurred in the file that is currently in the editor, you will also see a highlighted line in the Edit window. This marks the place in your source code where the compiler generated the error or warning.

At this point you can use the cursor keys to move the Message window's highlight bar up and down to view other messages. Notice how the highlight bar in the Edit window tracks where the compiler thinks each error occurred in your source. When you place the highlight bar on the "compiling" message, the editor shows you your last position in that file.

If the text in the Message window is too long to see, you can use the *Left* and *Right arrow* keys to scroll the message horizontally. To view more messages at once, you can zoom the Message window by pressing *F5*. When the Message window is zoomed, you cannot see the Edit window, so no tracking occurs. For now, leave the windows in split-screen mode.

Correcting a Syntax Error

To correct an error, place the Message window highlight bar on the first error message and then press *Enter*. Your cursor shifts to the Edit window and is placed at the spot that generated the error message. Notice that the status line of the editor shows the message you chose (this is useful when you work in zoomed mode). You can now correct the error that generated the message. (You'll have to put the # you took out earlier back in the first line.)

Since there is more than one error message, there are two ways to proceed to fix the next error.

The first method is to return to the Message window by pressing *F6* and choosing the next message you want to fix, as previously described.

However, you do not need to return to the Message window to get to the next error. Instead, you can simply press *Alt-F8* and the editor will place the cursor at the location of the error listed next in the message window. You can also move backward to the previous error by pressing *Alt-F7*.

There are certain advantages to both these methods, and usually circumstances dictate which method is preferable. Sometimes one silly mistake in the source can confuse the compiler, producing many messages. In this case, choosing and fixing the first message makes the next few error messages meaningless. When this happens, it is more convenient to use method one—to return to the Message window after fixing the first error, scroll down to the next meaningful message, then choose it. In other cases, however, you may wish to check each message in sequence; pressing *Alt-F8* is more effective in such situations.

Remember that *Alt-F7* and *Alt-F8* are hot keys; that is, they work from anywhere within TC. Thus if you are in the Message window and you press *Alt-F8*, you don't get the message that is currently highlighted but the one after it. (If you want to choose the current message, press *Enter*.) If there are no further compiler messages, *Alt-F8* has no effect.

Note: You cannot choose linker messages this way, and they will not track in your source.

In the course of fixing syntax errors, it is often necessary to add and delete text. The editor keeps track of this: When you proceed to the next error, it correctly positions the cursor on the error. You don't need to remember line numbers or keep track of added or deleted lines of text.

Catching Run-Time Errors: The Integrated Debugger

Once you have fixed all the syntax errors, your program will compile perfectly well. But it still may not run the way it should, because it may contain *logic* (run-time) errors. The error-tracking feature is no help in finding these.

To catch run-time errors, TC features an *integrated debugger*. You can run your program through the debugger, stop it at any point, check the value of variables, and even change values to test how your program will react. For a tutorial on how to use the TC integrated debugger, read Chapter 4 in this manual.

Projects: Using Multiple Source Programs

One of the great things about TC is its ability to handle separate compilation of multiple source files. And TC's Project-Make facility makes it even more effective.

In the examples in Chapter 2, you were working with only one source file, so you could just use the Compile/Make EXE File command to make an executable file. When you build a program from more than one C source file, however, you have to tell TC exactly which files are involved. That means you have to create a *project file.*

Creating a project file is as simple as listing the names of your C source files. Even though, as you will see, you can list a lot of different files in your project file, let's keep it simple for now with a two-file program.

One basic case is to have a main program file and a support file that contains functions or data referenced from the main file. For example, the main file called MYMAIN.C might look like this:

```
#include <stdio.h>

main (int argc, char *argv[])
{
   char *s;

   if (argc > 1)
      s = argv[1];
   else
      s = "the universe";

   printf("%s %s.\n",GetString(),s);

}
```

And the support file called MYFUNCS.C might look like this:

```
char ss [] = "The restaurant at the end of";

char *GetString(void)
{
   return ss;
}
```

Go ahead and create MYMAIN.C and MYFUNCS.C.

These two files now give you something to work with to build a project file. The project file will simply contain two lines naming the files to be compiled and linked. Create a new file, and type in the two file names, like this:

```
mymain
myfuncs
```

You don't have to type in the .C extensions. TC assumes any file without an extension is a .C file (though you may add the .C if you want to). The order of the files is not important either, except that it determines the order in which files are compiled. The following project file would have the same end result as the previous:

```
myfuncs
mymain
```

Now save your file as MYPROG.PRJ (select **Write to** from the **File** menu). That's all.

Notice that the name of the project file (MYPROG.PRJ) is not the same as the name of the main file (MYMAIN.C). The two names *could* have been the same (but not the extensions), but they do not have to be. The important thing to remember is that the name of your executable file (and any map file produced by the linker) will be based on the project file's name. In this case the executable file will be MYPROG.EXE (and possibly a map file called MYPROG.MAP).

Also note that you can specify complete path names for any of the files listed in the project file. In this way, you can build a program without having all the source files in the same directory.

Building a Multi-Source File Program

Now that you have a project file, all you need to do is tell TC what project you want to make. This is done by entering the name of the project file on

the project menu. Press *Alt-P* to get to the **Project** menu and choose **Project name**. You can explicitly type in the name of your project file or you can use wildcards to find it in a list of file names in a specified directory. (But remember, if you haven't saved the file, it won't be on disk.) Once your project name is entered, you can simply press *F9* (Make) to make the executable file. To run this program, press *Ctrl-F9* (**Run/Run**).

Note that running a program includes doing a make. This means that pressing *Ctrl-F9* can initiate a compile and link cycle if the files in the project need to be recompiled. This means you could have omitted the explicit make (*F9*). Select **Run/User screen** (or press *Alt-F5*) to see your output. Press any key to return to the TC Editor.

Error Tracking Revisited

In the example of a single-file program, you saw that syntax errors that generate compiler warning and error messages can be viewed and chosen from the Message window. Likewise, the Message window handles errors from multiple-file compilations (or makes).

To see this, introduce some syntax errors into the two files, MYMAIN.C and MYFUNCS.C. From MYMAIN.C, remove the first angle bracket in the first line and remove the *c* in **char** from the fifth line. These changes should generate three errors and three warnings in MYMAIN.

Now load MYFUNCS.C and remove the first *r* from `return` in the fifth line. This change will produce two errors and one warning.

Editing these files makes them out-of-date with respect to their object files, so doing a make will recompile them. Since you want to see the effect of tracking in multiple files, you need to modify the criterion that Project-Make uses to decide when to stop. This is done by setting a special toggle in the **Project** menu.

Stopping a Make

There are several reasons why the make cycle stops in TC. Obviously, Project-Make stops once an executable file has been produced. But Project-Make will also stop to report some type of error.

For example, Project-Make will always stop if it can't find one of the source files (or one of the dependency files—to be discussed later) listed in the project file. You can also force Project-Make to stop by pressing *Ctrl-Break*.

A make can also stop when the compiler generates messages. You can choose the type of message you want it to stop on by setting the **Project/Break Make On** menu toggle. The **Break Make On** menu defaults to **Break Make On...Errors**—which is normally the setting you'll want to use.. However, you can have a make stop after compiling a file with warnings, with errors, or with fatal errors, or have it stop before it tries to link.

The usefulness of each of these modes is really determined by the way you like to fix errors and warnings. If you like to fix errors and warnings as soon as you see them, you should set **Break Make On** to **Warnings** or maybe to **Errors**. If you prefer to get an entire list of errors in all the source files before fixing them up, you should set the toggle to **Fatal Errors** or to **Link**.

Syntax Errors in Multiple Source Files

To demonstrate errors in multiple files, set **Project/Break Make On** to **Fatal Errors**. To do this, press *Alt-P* to get to the **Project** menu, and choose **Break Make On**. Now choose **Fatal Errors** from the **Project/Break Make On** menu.

At this point, you should have introduced syntax errors into MYMAIN.C and MYFUNCS.C. Press *F9* (Make) to "make the project." The Compiling window will show the files being compiled and the number of errors and warnings in each file and the total for the make. When the `Press any key` message flashes, press the spacebar.

Your cursor should now be positioned on the first error or warning in the Message window. And if the file that message refers to is in the editor, there will be a highlight bar in the Edit window showing you where the compiler detected a problem. Again, you can scroll up and down in the Message window to view the different messages. Note that there is a "Compiling" message for each source file that was compiled. These messages are not errors or warnings but serve as "file boundaries," separating the various messages generated by each file.

When you scroll down past a file boundary, the Edit window may or may not track in the next file, depending on the setting of the Message Tracking toggle in the **Options/Environment** menu. The default value is to track only in the current file.

Thus, moving to a message that refers to a file other than the one in the editor causes the Edit window's highlight bar to turn off. If you choose one of these messages (that is, press *Enter* on it), TC will load the file it references into the Editor and place you in the Editor with the cursor on the

error. If you then return to the Message window by pressing *F6*, tracking will resume in that file.

But by setting the Message Tracking toggle to All Files, you can track messages across file boundaries. This means that, when you scroll through the Message window, TC will automatically load the appropriate file into the editor so you can see where each message refers. Try it.

You can also turn tracking off completely, by setting the Message Tracking toggle to Off. In this case, you simply choose the message you wish to work on and then press *Enter*. The file the message refers to will then be loaded into the editor with the cursor placed on the error.

Note that *Alt-F7* and *Alt-F8* (Previous error and Next error) are not affected by the setting of the Message Tracking toggle. These hot keys will always find the next or previous error and will load the file if necessary.

Keeping and Getting Rid of Messages

Normally, whenever you start to make a project, the Message window is cleared out to make room for new messages. Sometimes, however, it is desirable to keep messages around between makes.

Consider the following example: You might have a project that has many source files and you have Break Make On set to stop on Errors. In this case, you may get several warning messages in several files, but then one file contains an error so that the make stops. You fix that error and want to find out if the compiler will accept the fix. But if you just do a make or compile again, you will lose your earlier warning messages, which you may yet want to look at. How can you avoid this? All you have to do is to turn on the Keep Messages toggle in the Options/Environment menu.

When the Keep Messages toggle is set to On, messages are not cleared out when you start up a make. The only messages removed are the ones that result from the files you *re*compile. Thus, the old messages for a given file are replaced with any new messages that the compiler may generate.

If at some point you are done with the messages, you can get rid of them by choosing Project/Remove Messages. This zaps all the current messages. Setting Keep Messages to Off and running another make will also get rid of any old messages.

It's a good idea to get into the habit of clearing the messages when you change projects. To facilitate this, there is a shortcut in the Project menu, called Clear Project, that clears both the project name and the current

messages. After choosing Project/Clear Project, you can define a new project or compile and run single-file programs by simply loading them into the editor or defining the primary .C file name with the Primary C File command.

The Power of Project-Make

In the last description of making a project, you dealt with the most basic situation: just a list of C source file names. Project-Make provides a lot of power to go beyond this simple situation. To see this you need to understand how a make works.

A make works by comparing the date of the source file with the date of the object file generated by the compiler. This comparison of creation dates defines several *implicit dependencies* in a simple project list.

Given the earlier example using MYPROG.PRJ, you have the following dependencies:

MYMAIN.OBJ	is dependent on MYMAIN.C
MYFUNCS.OBJ	is dependent on MYFUNCS.C
MYPROG.EXE	is dependent on MYMAIN.OBJ, MYFUNCS.OBJ, and MYPROG.PRJ

This means the object file MYMAIN.OBJ is out-of-date if MYMAIN.C is newer than MYMAIN.OBJ; thus MYMAIN.C will be recompiled. Notice the executable file is always dependent on all object files in the project and on the project file itself. This means that, if any of the objects or the project file MYPROG.PRJ itself has a newer date than MYPROG.EXE, the make will relink MYPROG.EXE. These implicit dependencies arise from the simple list of file names of the C files in your project.

Explicit Dependencies

However, bigger projects require a more sophisticated make facility that allows you to specify explicit dependencies. This is useful when a particular C source file depends on other files. It is common for a C source to include several header files (.H files) that define the interface to external routines. If the interface to those routines changes, you would like the file that uses those routines to be recompiled. This is done with explicit dependencies.

For example, say you have a main program file, MYMAIN.C, that includes a header file MYFUNCS.H. A make will recompile MYMAIN.C and MYFUNCS.C if MYFUNCS.H changes—if you specify the following dependencies in your project file:

```
MYMAIN.C (MYFUNCS.H)
MYFUNCS  (MYFUNCS.H)
```

Notice that this project file makes the MYFUNCS.C file dependent on the MYFUNCS.H file. This is a good consistency check for your files. So now you have the same implicit dependencies as well as some explicit dependencies, like so:

MYMAIN.OBJ	is dependent on MYMAIN.C and MYFUNCS.H
MYFUNCS.OBJ	is dependent on MYFUNCS.C and MYFUNCS.H
MYPROG.EXE	is dependent on MYMAIN.OBJ, MYFUNCS.OBJ, and MYPROG.PRJ

Any C file listed in a project file can have as many explicit dependencies as it needs. Simply place the files you want the C source to be dependent on in parentheses, separated by blanks, commas, or semicolons.

For example if you want MYMAIN.C to be dependent on MYFUNCS.H, YOURS.H, and OTHER.H, you would type

```
MYMAIN.C (MYFUNCS.H, YOURS.H, OTHER.H)
```

Note: When autodependency checking is on, any included file will be checked, but explicit dependencies will not.

Autodependency Checking

Project-Make has the capacity for automatically checking dependencies between source files in the project list and their corresponding object files. Project-Make opens the .OBJ file and looks for information about files included in the source code. This information is always placed in the .OBJ file by both TC and TCC when the source module is compiled. Then every file that was used to build the .OBJ file is checked for time and date against the time/date information in the .OBJ file. The .C source file is recompiled if the dates are different.

Note: In order for this feature to work, you must toggle On the **Project/Auto Dependencies** switch in the integrated environment.

That is all there is to dependencies. This method gives you the power of more traditional makes without all the hassle of a complicated make syntax.

What? More Make Features?

There are two other features that add to the power of the make:

- you can specify external object and library files to be linked into your project
- you can override the standard startup files and libraries

External Object and Library Files

From time to time, you might want to use some routines that came from another source, such as assembly language or another compiler. Or maybe you have some library files that perform special functions not provided in the standard libraries. In these cases, you can include the name of the object or library files in your project with an explicit extension, like this (note that, when listing files, the order is not important):

```
MYMAIN  (MYFUNCS.H)
MYFUNCS (MYFUNCS.H)
SPECIAL.OBJ
OTHER.LIB
```

When Project-Make sees a file with an explicit .OBJ extension, it simply includes that file in the list of files to be linked together. It does not try to compile it or find its source, or read in autodependency information. Similarly, a name in your project file with a .LIB extension gets put into the list of libraries the linker searches when trying to resolve external references. Again, it does not try to compile or build the library.

Note that files of these types cannot have explicit dependency lists (they will be ignored). However, you can include these names in your C source dependency list like any other file you want your source to depend on.

For example,

```
MYMAIN  (MYFUNCS.H, SPECIAL.OBJ)
MYFUNCS (MYFUNCS.H, OTHER.LIB)
SPECIAL.OBJ
OTHER.LIB
```

What this means is that, if for some reason these .OBJ or .LIB files become updated, the C source will be recompiled.

Overriding the Standard Files

In some cases, it is necessary to override the standard startup files or libraries. This is usually recommended only for experienced users, and is not a common practice for beginners. But if you ever feel the need, here's how to do it.

To override the startup file, you must place a file called C0*x*.OBJ as the *first* name in your project file—where *x* stands for any DOS name (for example, C0MINE.OBJ). What is critical is that the name start with C0, that it is the first file in your project, and that it have an explicit .OBJ extension.

To override the standard library, all you need to do is place a special library name anywhere in the list of names in your project file. The name of the library must start with a C, followed by a letter rperesenting the model (such as `S` for the small model); the remaining characters, up to six, may be anything you want for a file name. You must use an explicit .LIB extension (for example, CSMYFILE.LIB or CSNEW.LIB).

When the standard library is overridden, make does not try to link in the math libraries as based on the Floating Point toggle setting in the **O/C/** Code Generation menu. If you wish to have these libraries linked in when you override the standard library, you must explicitly include them in your project file.

Compiling and Linking from a Command Line

In addition to using the integrated development environment, you can run your Turbo C programs with the old-fashioned type of command-line interface. While the integrated development environment mode is best for developing and running your programs, you may sometimes prefer to use the command line; in some advanced programs, the command-line

interface may be the only way to do something intricate. For example, if your Turbo C programs include inline assembly code, you will need to use the command-line version of Turbo C (TCC) rather than TC, the integrated development environment version.

TCC compiles C source files and links them together into an executable file. It works similarly to the UNIX CC command. TCC will also invoke TASM to assemble .ASM source files. Note that to *compile only* you have to use the `-c` option at the command line.

The TCC Command Line

To invoke Turbo C from the command line, enter `TCC` at the DOS prompt and follow it with a set of command-line arguments. Command-line arguments include compiler and linker options and file names. The generic command-line format is

```
tcc [option option option ...] filename filename ...
```

Options on the Command Line

Each command-line option is preceded by a hyphen (-), and separated from the `tcc` command, other options, and following file names by at least one space. You can explicitly turn a command-line option *off* by following the option with a dash. (For example, `-K-` explicitly turns the **unsigned chars** option *off*.) Turbo C's command-line options are described in Appendix C of the *Turbo C Reference Guide*.

File Names on the Command Line

After the list of options, type file names on the command line. The compiler compiles files according to the following set of rules:

filename	compile *filename.c*
filename.c	compile *filename.c*
filename.xyz	compile *filename.xyz*
filename.obj	include as object at link time
filename.lib	include as library at link time
filename.asm	invoke TASM to assemble to .OBJ

The compiler will then invoke the linker and supply the linker with the names of the appropriate C startup file and standard C libraries.

The Executable File

Normally, the compiler derives the name of the executable file from the first source or object file name supplied on the command line. The executable program is given that first file name with the .EXE extension.

If you want to specify a different name for the executable file, use the `-e` option. After the `tcc` command and before any file names, enter `-e` *immediately followed* by the name you want to give the executable file (no whitespace between the *e* and the file name).

Some Sample Command Lines

The following example illustrates proper syntax for invoking Turbo C from the DOS command line:

```
tcc -IB:\include -LB:\lib -etest start.c body.obj end
```

For this example command line, the command `tcc` invokes Turbo C at the DOS prompt. Turbo C then interprets the command-line options as meaning

- the include directory is B:\INCLUDE (`-IB:\include`)
- the libraries are in the B:\LIB directory (`-LB:\lib`)
- the executable result should be placed in a file called TEST.EXE (`-etest`)

Turbo C interprets the listed files to mean that this program consists of

- a source file called START.C to be compiled
- an object file called BODY.OBJ to be included at link time
- another source file called END.C to be compiled

Here is another example of a Turbo C compile-time command line:

```
tcc -IB:\include -LB:\lib2 -mm -C -K s1 s2.c z.asm mylib.lib
```

This compile-time command line directs Turbo C to

- look for the include files in the B:\INCLUDE directory (`-IB:\include`)
- look for the libraries in the B:\LIB2 directory (`-LB:\lib2`)
- use the Medium memory model (`-mm`)
- allow nested comments (`-C`)
- make **chars** unsigned (`-K`)

Turbo C interprets the list of file names to mean

- the source files called S1.C and S2.C are to be compiled
- the file Z.ASM is to be assembled (using TASM)
- the executable file will be named S1.EXE
- the library file MYLIB.LIB is to be linked in at link time

The TURBOC.CFG File

You can set up a list of options in a configuration file called TURBOC.CFG, which can be used in addition to options entered on the command line. This configuration file contains options as they would be entered on the command line.

If you've listed your commonly used options in TURBOC.CFG, you won't need to enter them on the command line when you use TCC.EXE. If you don't want to use certain options that are listed in TURBOC.CFG, you can override them with switches on the command line.

You create the TURBOC.CFG file using any standard ASCII editor or word processor (such as the Turbo Editor in the integrated development environment version). You can list options (separated by spaces) on the same line or list them on separate lines. Then, when you compile your program from the command line, Turbo C uses the options supplied in TURBOC.CFG, in addition to the ones given on the command line.

When you run TCC, it looks for TURBOC.CFG in the current directory. If it doesn't find it there *and* if you're running DOS 3.x, it then looks in the start directory (where TCC.EXE resides). Note that TURBOC.CFG is not the same as TCCONFIG.TC, which is the default integrated development environment version of a configuration file.

Options given on the command line override the same options specified in TURBOC.CFG. This ability to override configuration file options with command-line options is an important one. If, for example, your configuration file contains several options, including the `-a` option (which you want to turn *off*), you can still use the configuration file but override the `-a` option by listing `-a-` in the command line.

How are command-line options and TURBOC.CFG options combined and overridden? There are two kinds of TURBOC.CFG options:

- the `-I` and `-L` options
- all the other options in the file

Under any circumstances, command-line options are evaluated from left to right, and the following rules apply:

- For any option that is *not* an `-I` or `-L` option, a duplication on the right overrides the same option on the left. (Thus an *off* switch on the right cancels an *on* switch to the left.)
- The `-I` and `-L` options on the left, however, take precedence over those on the right.

When the options from the configuration file are combined with the command-line options, the `-I` and `-L` options from TURBOC.CFG are appended to the right of the command-line options, and the remaining TURBOC.CFG options are inserted on the left of the command line's list of options, immediately after the `tcc` command.

Thus, because of the way the command line and TURBOC.CFG are combined, the TURBOC.CFG `-I` and `-L` options are on the extreme right, so the include and library directories specified in the command line are the first ones that Turbo C searches for the include and library files. This gives the `-I` and `-L` directories on the command line priority over those in the configuration file. All other options from the TURBOC.CFG file are inserted to the left of the command-line options, which again, correctly, gives the command-line options priority over them.

The TCCONFIG.EXE Conversion Utility for Configuration Files

The integrated environment and command-line compiler have a number of common options, listed in Table C.1 of Appendix C in the *Turbo C Reference Guide*, "TCC Command-Line Options." TCCONFIG.EXE takes a configuration file created by one environment and converts it for use by the other.

The conversion command is

```
TCCONFIG SourceFile [DestinationFile]
```

TCCONFIG automatically determines the direction of the conversion: It examines the source file to see whether it is an integrated environment (TC) configuration file or a command-line compiler (TCC) configuration file.

The destination file name is optional. If you don't specify a file name, TCCONFIG uses the default name TCCONFIG.TC or TURBOC.CFG, depending on the conversion direction. You can give any file name; the command-line compiler, however, only looks for a file named TURBOC.CFG when it runs. It won't run on any other name.

When it creates the TCCONFIG.TC file, TCCONFIG uses default values for any items not specified by the command-line compiler configuration file (TURBOC.CFG). Going in the other direction, it includes in TURBOC.CFG only the options in TCCONFIG.TC that differ from the default values.

TCCONFIG returns you to the DOS prompt when the conversion is done.

The MAKE Utility

Turbo C's stand-alone MAKE utility, a more powerful version of Project-Make, permits you to describe source and object file dependencies. It is based on the UNIX MAKE utility. The MAKE utility evaluates those dependencies to ensure that the files are correctly compiled and linked.

What is the advantage to using a MAKE utility? As with Project-Make, you do not have to keep track of which program components have changed since you last compiled them. Stand-alone MAKE is more powerful than Project-Make, however, because it is a general-purpose program builder. Before linking your complex program's object files, MAKE recompiles any files that need to be updated. Then it simply incorporates the newly compiled files with those that did not need to be recompiled and creates a new, executable program file.

Appendix D in the *Turbo C Reference Guide* contains a detailed explanation of the stand-alone MAKE utility.

BUILTINS.MAK

BUILTINS.MAK is an optional file in which you can store MAKE macros and rules that you use again and again, so you don't have to keep typing them into your makefiles. See Appendix D of the *Turbo C Reference Guide* for information on how to write makefiles and set up a BUILTINS.MAK file.

Running Turbo C Programs from the DOS Command Line

To run executable Turbo C programs from the DOS command line, simply type the executable file name at the DOS prompt. It is not necessary to include the .EXE extension. For example, to execute the program TEST.EXE,

you would just type `test` at the DOS prompt and press *Enter*. The TEST program would then run (execute).

Moving Ahead with Turbo C

Now that you have seen how to compile, link, run, and make your Turbo C programs, both with the integrated development environment and through standard command lines, you are ready to put Turbo C through its paces. As you expand your knowledge about the language and about this implementation, you will want to refer to the second volume of this handbook, the *Turbo C Reference Guide*, for information about the run-time environment, the library files, and Turbo C's implementation of the C language, as well as a number of useful standalone utilities included with Turbo C to make your job easier. Check out Chapter 4 for a guided tour of Turbo C's integrated debugger, Chapters 6 and 7 for an overview of the C programming language, Chapter 8 for an introduction to the Turbo C graphics functions, and Chapter 12 on advanced programming techniques.

If you know Turbo Pascal or Turbo Prolog, you will also want to read Chapters 9 and 10, respectively, for some tips on how to use either of these languages with this fast, powerful C programming package.

C H A P T E R

4

Debugging Your Program

When you first run a program after correcting compile and link errors, it's unlikely to work correctly. A new program almost always contains numerous bugs—errors in design and coding—that you must identify and fix. Finding and fixing a program's bugs is called *debugging.*

It's hard to find bugs merely by watching the afflicted program's behavior, so most programmers use a debugger to help them locate the bugs in their programs. A debugger is a piece of software that lets you take control of your program as it runs. You can stop your program's execution at any point, run it one statement at a time, and inspect the data it is processing.

In This Chapter...

Turbo C's integrated environment contains a debugger, called the integrated debugger. In this chapter, we explain how to use the Turbo C integrated debugger.

The chapter begins with a series of examples that demonstrate how to use the integrated debugger. In the first example, we show you how to use the debugger's simplest features to identify an "easy" bug. The following examples illustrate more advanced features of the debugger.

There follows a survey of the debugger menu commands and related menu commands, with the corresponding hot keys or hot key combinations, and a description of what each command does.

Finally, we present some guidelines that will make debugging easier for you. Many of the guidelines concern how to design and write a program, as well as how to debug one. You can apply most of these ideas to any computer language, not just to Turbo C.

How the Integrated Debugger Works

Turbo C's integrated debugger is a source-level debugger. This means that you control the debugger with the same "language" you use to write your programs. For example, you might display the value of an element in an array by telling the debugger to display the value of an expression like

```
rptr->image[nptr+0x80]
```

You debug a program simply by running it with the **Run/Run** menu option (hot key: *Ctrl-F9*). The debugger takes over if the program was compiled with the Source Debugging toggle set to On. (To set this toggle, choose Debug/Source Debugging.)

Before you run the program, you can set breakpoints on one or more lines in the source file. When the running program encounters a breakpoint, it halts just before executing the first statement on the line bearing the breakpoint and returns control of the debugger to you.

While the program is halted, you can study and manipulate it in many ways. For example, you can

- display the value of a variable or expression
- set up a list of expressions in a special window and observe how their values change
- change the value of a variable
- clear existing breakpoints or set new ones
- run the program one line at a time
- edit files, recompile and relink the program, or use any other feature of Turbo C's menu system
- make the program continue running until it encounters another breakpoint

Figure 4.1 illustrates the typical course of a debugging session. (Note that it doesn't show everything you can do with the integrated debugger at each step.)

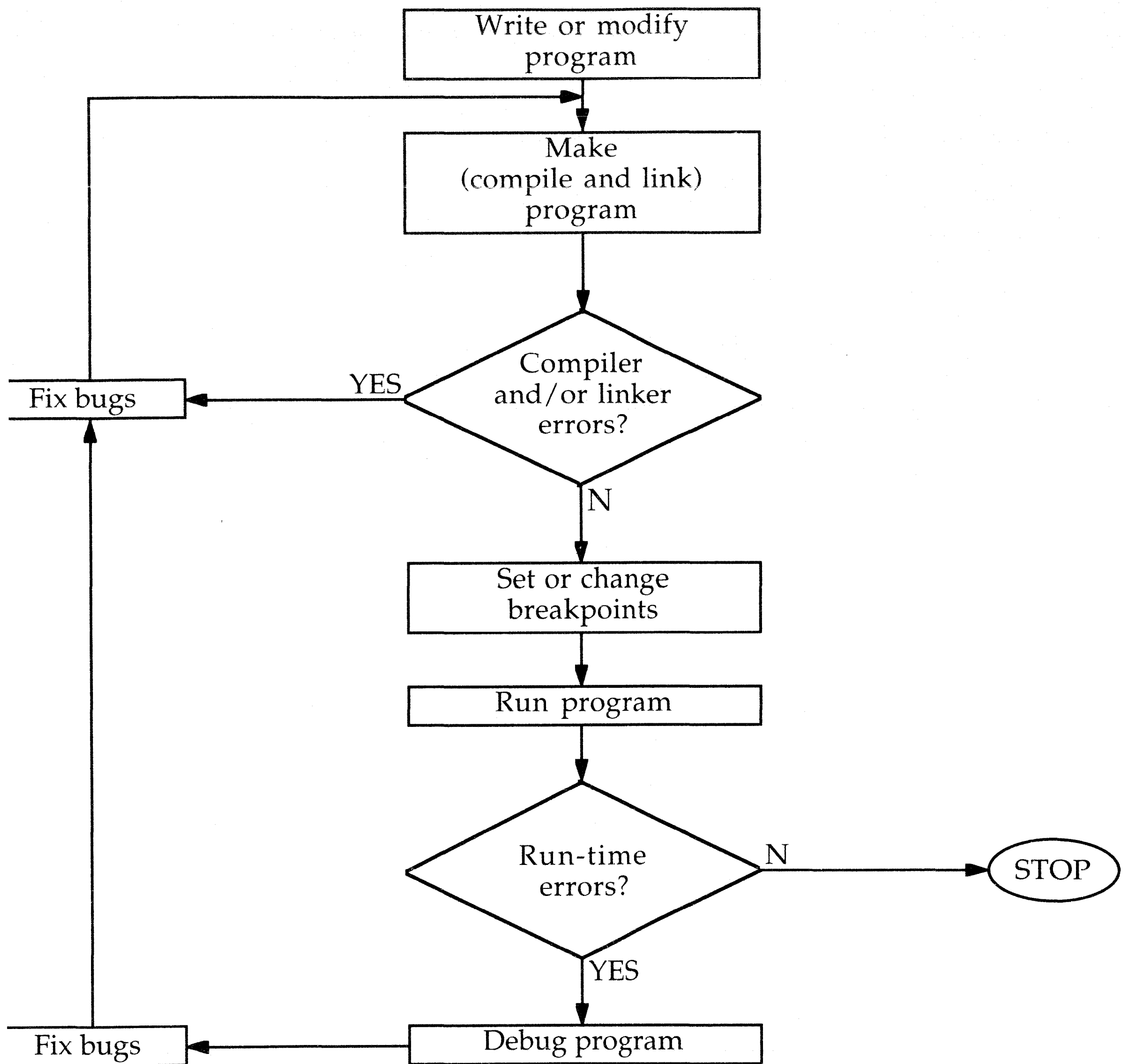


Figure 4.1: Typical Steps in the Debugging Process

Example 1: Debugging a Simple Program

For your first debugging experience with Turbo C, you'll use the program shown in the following example. We'll refer to this program as WORDCNT. It is meant to display the contents of a text file and tabulate word lengths; that is, report how many words are one character long, how many are two characters long, and so forth. Unfortunately, WORDCNT contains several bugs. You're going to use the debugger to find them.

You'll find WORDCNT in the file WORDCNT.C on one of the distribution disks. Copy it into your Turbo C directory so you'll be sure to have a fresh, buggy version!

If you're working in a directory other than the one that contains Turbo C, make a working copy of WORDCNT.C. Also make working copies of the project file WORDCNT.PRJ and the data file WORDCNT.DAT. All three of these files are on the distribution disks and in your Turbo C directory.

```
/*****
*  Read a text file; count the number of words of length 1, 2, 3, etc.
*
*  NOTE: This program is for use with the debugging tutorial
*        in the debugging chapter of the User's Guide. It
*        intentionally contains bugs.
*****/

#include <stdio.h>
#include <ctype.h>

#define MAXWORDLEN      16
#define NUL             ((char)0)
#define SPACE           ((char)0x20)

/*****
*  Find the next word in the line buffer.
*  IN:      wordptr points to the first character of a word or a preceding
*           space.
*  RETURN:  A pointer to the first character of the word. If there are no
*           more words, a pointer to the terminating NUL.
*****/
char *nextword(char *wordptr)
{
/* Advance to the first non-space. */
   while ( *wordptr==SPACE )
      wordptr++;
   return(wordptr);
}
```

```
/*****
*  Find the length of a word. A word is defined as a sequence of characters
*  terminated by a space or a NUL.
*  IN:      wordptr points to a word.
*  RETURN:  The length of the word.
*****/
int wordlen(char *wordptr)
{
   char *wordlimit;

   wordlimit = wordptr;
   while ( *wordlimit & *wordlimit!=SPACE )
      wordlimit++;
   return( wordlimit-wordptr );
}

/*****
*  The main function.
*****/
void main(void)
{
   FILE      *infile;         /* Input file. */
   char       linebfr[1024],  /* Input line buffer, very long for safety. */
          *   wordptr;        /* Pointer to next word in linebfr. */
   int     i;                 /* Scratch variable. */
   static int wordlencnt[MAXWORDLEN],
                              /* Word lengths are counted in elements 1 to
                                 MAXWORDLEN. Element 0 isn't used. The array is
                                 static so that the elements need not be set to
                                 zero at run time. */
              overlencnt;     /* Overlength words are counted here. */

   printf(" WARNING: This is an example program for the practice\n");
   printf(" debugging session. If you are not running this under the.\n");
   printf("Integrated Development Environment, press control-break now.\n");
   printf("See the debugging chapter of the User's Guide for details.\n\n");

   printf( "Enter the input file's name: " );
   gets(linebfr);
   if ( !strlen(linebfr) ) {
      printf( "You must specify an input file name!\n" );
      exit();
   }

   infile = fopen( linebfr, "r" );
   if ( !infile ) {
      printf( "Can't open input file!\n" );
      exit();
   }
```

```
/* Each loop processes one line. NOTE: If a line is longer than the
   input buffer,the program may produce invalid results. The very large
   buffer makes this unlikely. */
   while (   fgets( linebfr, sizeof(linebfr), infile )   )   {
      printf("%s\n",linebfr);
   /* Check for buffer overflow & remove the trailing newline. */
      i = strlen(linebfr);
      if ( linebfr[i-1] != '\n' )
         printf( "Overlength line beginning\n\t%70s\n", linebfr );
      else
         linebfr[i-1] = NUL;

   /* lineptr points to the 1st word in linebfr (past leading spaces). */
      wordptr = nextword(linebfr);

   /* Each loop processes one word. The loop ends when [nextword]
      returns NULL, indicating there are no more words. */
      while (*wordptr) {
      /* Find the length of this word, increment the proper element of the
         length count array, and point to the space following the word. */
         i = wordlen(wordptr);
         if ( i > MAXWORDLEN )
            overlencnt++;
         else
            ;
            wordlencnt[i]++;
         wordptr += i;

      /* Find the next word (if any). */
         wordptr = nextword(wordptr);
      }
   }

/* Print the word length counts. Each loop prints one. */
   printf( "  Length Count\n" );
   for ( i=1; i<MAXWORDLEN; i++ )
      printf( "  %5d %5d\n", i, wordlencnt[i] );
   printf( "Greater %5d\n", overlencnt );

/* Close the file and quit. */
   fclose(infile);
}
```

Be sure the **Debug/Source Debugging** and **O/C/C/OBJ Debug Information** toggles are both set to On. Start up TC by typing

```
TC WORDCNT
```

at the DOS prompt. Build the program (choose **Compile/Build All**). Turbo C will compile and link WORDCNT, prepare it for execution, and then halt. Now choose **Run/Trace Into** (or press *F7*).

At this point, the debugger has scrolled the beginning of **main** into the Edit window. It has highlighted the line containing the declaration of the **main** function (`void main (void)`), indicating that this will be the first line to be executed when you run WORDCNT. This highlight is called the *execution bar,* and it marks the *execution position*: the line containing the next statement to be executed.

To make WORDCNT run, choose **R**un/**R**un. WORDCNT prompts you to enter the name of an input file. Type `WORDCNT.DAT` and press *Enter.* WORDCNT should read and display the first line of the data file, then lock up your computer, because of the bugs in the program. Enter *Ctrl-Break* to unlock it, and press *Esc* to verify. Then choose **R**un/**P**rogram Reset (or press the hot key, *Ctrl-F2*) to end the debug session and press *F7* (**R**un/**T**race Into) to start a new one.

Setting and Using a Breakpoint

As the source file comments explain, the first part of **main** prompts for the input file's name and opens the file. The fact that WORDCNT read and displayed the first line of the input file suggests strongly this part of the code is working—well enough, at least, not to be responsible for the malfunction we've observed. Therefore, you can run through the first part of **main** and stop when you reach the suspicious part. To accomplish this, you must set a *breakpoint* on the line where you want to stop.

Use *PgDn* and *Down arrow* to move the cursor to the line that begins

```
while ( fgets( ....
```

(It's just below the long comment that begins `Each loop processes....`) Notice that the execution bar doesn't move. That's because you aren't running statements in the program; you're just moving the editor's cursor.

To set a breakpoint on the line at the cursor, choose **B**reak/Watch/**T**oggle Breakpoint (or its hot key combination, *Ctrl-F8*). The debugger will highlight the line to indicate that a breakpoint is set there. Notice that the appearance of this breakpoint highlight is different from the appearance of the execution bar.

Now choose **R**un/**R**un (or its hot key combination, *Ctrl-F9*). WORDCNT will continue (in this case, start) running.

After WORDCNT prompts you to enter the name of an input file, it will halt and wait for keyboard input. The debugger shows you what WORDCNT wants by displaying the Execution screen, which shows the

program's output just as if the program were running wthout the debugger. Type `WORDCNT.DAT` and press *Enter*. WORDCNT will continue running until it encounters the breakpoint; then it will stop and the Edit window will reappear. The cursor and the execution bar are now on the **while** statement.

Note that the breakpoint you set on the **while** statement is still there. You can't see it because the execution bar obscures it, but it will reappear when the execution bar moves on. WORDCNT will stop every time it reaches this breakpoint until you toggle the breakpoint off.

Using *Ctrl-Break*

In addition to any breakpoints you might set, you also have an "instant" breakpoint during execution: pression *Ctrl-Break*. This means that, barring a major crash, you can interrupt your program at any time. When you press *Ctrl-Break*, you drop out of your program and back into the TC Editor, with the execution bar on the next line to execute and ready to step through the rest of the program.

What actually happens is that the debugger is in contact with DOS, the BIOS, and other services, and so it knows whether or not the code currently executing is a DOS routine, a BIOS routine, or the program itself. When you press *Ctrl-Break*, the debugger waits until the program itself is executing. Then it starts stepping every machine-level instruction until it comes to one that is at the beginning of a C source code line. At that point it breaks, so that the execution bar is on that line of code.

If a second *Ctrl-Break* is detected before the debugger locates and displays the source code line, the debugger terminates the program immediately and doesn't try to find the line of source code. In such a case, the program terminates without flushing any output or calling any exit functions. (This is similar to terminating via the **_exit** function.) Therefore, you should press *Ctrl-Break* twice *only* when your program is stuck in an infinite loop and one *Ctrl-Break* doesn't abort it.

Stepping Over Function Calls

Now that you've reached a part of WORDCNT where a bug may be, you must proceed more cautiously. Run the next part of **main** one line at a time, pausing after each line to see whether it has had the desired effect.

To run one line of **main**, choose Run/Step Over. The debugger will run the **while** statement and read the first line of input. Then it will move the

highlight down to the next line containing executable statements. Choose **Run/Step Over** again to run the following call to **printf**.

Did the **while** statement have the desired effect? To find out, select **User Screen** from the **Run** menu, or press *Alt-F5*. This command displays the Execution screen. You can see the first line of input on the screen, so you may conclude that both the **while** and the **printf** worked correctly. Press any typing key to redisplay the Edit window.

The debugger has a hot key for **Run/Step Over**; it is *F8*. Press *F8* now to execute the next statement, which computes the length of the input line and assigns it to the variable *i*.

Warning! Tracing into or stepping over a call to the library function **longjmp** will not stop at the next line (because that function never returns). It will cause your program to run to the next breakpoint or to completion.

Evaluating an Expression

Did the assignment statement have the desired effect? You can find out by displaying the value of *i*.

Choose **Debug/Evaluate**. The debugger opens a pop-up window containing three fields. We'll refer to these fields by their functions:

1. The Evaluate field: Here you enter the expression you want to evaluate and possibly modify. (**Note:** If the expression you enter is too long for the Expression field box, you can scroll the expression using *Right arrow, Left arrow, Home,* and *End.*)
2. The Result field: Here the debugger displays the expression's value.
3. The New Value field: Here you enter the new value you want the expression to have (optional).

Notice that the Evaluate field contains a word. This is the field's default value, copied from the word at the cursor in the Edit window. We'll consider its uses in a moment. For the present, type the expression *i* in its place, and press *Enter*. The debugger displays the value of *i* in the Result field. It is 43, which is correct. Press *F10* to escape from the menu system.

The next group of statements checks whether the line read by **fgets** ends with a newline character. If it doesn't, the input line was too long to fit in the buffer, and the program displays a warning message. If the line does end with a newline character that's OK; the program removes the newline from the string so that it won't be counted as a character in the last word.

Before you execute the **if** statement, it will be helpful to display the input line to see what result you should expect. Move the cursor to an occurrence of the word *linebfr* in the Edit window, and choose **Debug/Evaluate** (or its hot key combination *Ctrl-F4*) again. The debugger displays *linebfr* in the **Debug/Evaluate** window's Evaluate field. Press *Enter.*

The debugger should display

```
To be or not to be: that is the question.\n
```

`\n` represents a newline character, just as it does in a C source program.

Now run the **if** statement by choosing *F8.* The execution bar moves to the **else** clause, so the **if** statement did the right thing. Run the assignment statement in the **else** clause, then evaluate *linebfr* again. (You can use the hot key *Ctrl-F4* for **Debug/Evaluate.**) The newline character (`\n`) has been removed. So far, so good.

The nextword and wordlen Functions

The next statement calls **nextword**, a function that locates the next (in this case, the first) word in a string. Run this statement using *F8* and evaluate *wordptr* to see what value **nextword** returned. You should find that *wordptr* points to the *T* in `To be or not to be: that is the question`. If it does, **nextword** is functioning correctly, at least in this simple case.

Next the program enters a **while** loop. Each iteration of the loop should process one word in *linebfr*, then advance *wordptr* to the next word. After the loop has processed the last word, *wordptr* should be pointing to the null character that terminates the line, and the loop should end.

Run the **while** statement. The execution bar moves to the first statement in the body of the loop, which calls the function **wordlen**. This function determines the length of the word at *wordptr*. Run the statement and evaluate *i*. The value of *i* is 0, which is not correct; the length of the first word on the line should be 2. We've found a bug!

Stop and Think

Before you rush off to fix the bug, though, it's worth your while to consider its effect on the program. An erroneous word length of 0 will have two effects. First, it will increment the wrong element of *wordlencnt*, element 0.

Second, it will cause the statement `wordptr += i` to leave the value of *wordptr* unchanged. That, in turn, will make the second iteration of the **while** loop begin with the same value of *wordptr* as the first. Since **wordlen** returned 0 the first time it was called, it presumably will return 0 the second time, too. Thus the value of *wordptr* will be the same the third time through the **while** loop, and the fourth time, and so on, forever. That fits the observed behavior of WORDCNT perfectly. This bug is the one that makes WORDCNT lock up your computer.

What was the point of this exploration? You might have found that the bug explained only part of WORDCNT's misbehavior, or that it wasn't related to the misbehavior at all. In either case, you'd probably want to run more of the program to see what would happen next. As it is, you can concentrate on fixing the bug you've found with a high degree of confidence that the program's behavior will be corrected, or at least improved.

What You've Accomplished

You've found that WORDCNT is misbehaving because of a bug in **wordlen**. You still must find out exactly what is wrong with **wordlen**. We'll return to that in a moment. First, we'll take some time out to review the debugger commands you've used, and to learn more about them.

In your first effort at debugging WORDCNT, you have done the following things:

- ensured that the **D**ebug/**S**ource Debugging and **O**/**C**/**C**/**O**BJ Debug Information toggles are set to On
- selected **C**ompile/**B**uild All to prepare WORDCNT for debugging
- used editor commands to move the cursor to the suspicious part of WORDCNT; selected **B**reak/Watch/**T**oggle Breakpoint to set a breakpoint there; selected **R**un/**R**un (or its hot key combination *Ctrl-F9*) to run WORDCNT up to the breakpoint
- used **R**un/**U**ser Screen or its hot key *Alt-F5* to inspect your program output on the User screen.
- selected **R**un/**S**tep Over (or pressed its hot key, *F8*) to run the statements in **main**, one line at a time
- selected **D**ebug/**E**valuate (or pressed its hot key command, *Ctrl-F4*) to display the values of several variables
- thought about the bug you found, and concluded that it explains WORDCNT's observed misbehavior and so warrants immediate correction

The Default Expression in the Evaluate Window

Recall that **Debug/Evaluate** copies the word at the cursor in the Edit window into the Evaluate field. You can often save work by putting the cursor at a variable you want to evaluate before you choose **Debug/Evaluate**. Even if the expression you want is different, you may be able to enter it more quickly by editing the default expression than by typing it in from scratch. Furthermore, you can copy more text from the Edit window to the default expression by pressing *Right arrow*. Each time you press *Right arrow*, it copies one additional character.

For example, suppose you want to evaluate the expression *linebfr*[*i*-1], which appears in the source file in the line

```
if ( linebfr[i-1] != '\n' )
```

Move the cursor to *linebfr* and choose **Debug/Evaluate**. The Evaluate field displays the default expression *linebfr*. Press *Right arrow* five times to append `[i-1]` to the expression. Then press *Enter*.

Changing the Value of an Evaluated Expression

Debug/Evaluate can change the values of some types of expressions. It can change the value of any expression that represents a single data element, such as *i*, *linebfr*[*i*], or *(*linebfr*+*i*).

Try evaluating the variable *i* and then changing its value. When you press *Enter* to evaluate *i*, the debugger displays the value of *i* in the Result field. Go to the New Value field (use *Down arrow*) and specify the value you want to assign to *i*. For example, you can edit the New Value field to say `i+1` (to increment *i* by 1), or type in 17. When you press *Enter*, the debugger evaluates what you entered, changes the value of *i*, and displays the new value in the Result field. (**Note:** Remember, once you have pressed *Enter* in the New Value box, the value of the variable being evaluated changes, and pressing *Esc* will not undo the change.) Press *Esc* to leave **Debug/Evaluate**, invoke the command again, and redisplay the value of *i* to confirm that it has been changed. Then change it back and leave **Debug/Evaluate** again.

You can modify the value of an expression to correct the effects of a bug, letting you run your program somewhat further in order to find additional bugs. You can also use it to gain insight into your program's behavior. For example, suppose you want to see how a certain function behaves when it is passed an invalid parameter value. It may be hard to make the program pass the function that particular value, but you can often get the same

result by changing the value of some variable just before the program calls the function.

If you leave the New Value field by pressing *Esc* instead of *Enter*, the debugger does not change the expression's value. Press *Esc* if you modify the new value by accident, or modify it and then change your mind.

You can evaluate any legal C expression, provided it doesn't contain

- function calls
- defined or type-defined symbols or macros (`*wordptr == 0x20` is OK; `*wordptr == SPACE` is not OK, since SPACE is defined)
- local or static variables not in the scope of the function being executed, unless they are fully qualified

Qualifying Variable Names

There are two typical situations in which it is desirable to qualify fully a variable name used in an expression.

- When you want to examine static variables in various modules
- When you want to look at an auto (local) or static variable in a function.

To qualify a variable name fully, use the following syntax:

```
.<module name>.<function name>.<variable name>
```

Note that either the module name or the function name may be left out in certain cases. For example, if you are stepping through your main function and wish to view a static variable named *myvar* in a different module named **mysubs**, you can type in `.mysubs.myvar`. However, if *myvar* appears in the function **myfunc** in the module **mysubs**, you will have to use `.mysubs.myfunc.myvar`. On the other hand, if **myfunc** is in the same module with the function **main**, you will need to type only `.myfunc.myvar`.

Format Specifiers

To control exactly how information is displayed in the Debug/Evaluate window, Turbo C allows you to add optional *format specifiers* to expressions in the Evaluate field (the same is true for the Watch window). A format specifier follows the expression, separated from it by a single comma. It may be either upper or lowercase.

A format specifier consists of an optional *repeat count* (an integer), followed by zero or more *format characters*; no spaces are required between the repeat count and the format characters. Table 4.1 lists the available format characters, and describes their effect.

The repeat count is used to display consecutive variables, a typical example of which is the elements of an array. For example, if *list* is an array of 10 integers, the expression `list` would display:

```
list: { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 }
```

If you want to look at a particular range of the array, you can specify the index of the first element you want to examine, and add a repeat count:

```
list[5],3: 60, 70, 80
```

This technique is particularly useful for dealing with arrays that are too large to be displayed completely on a single line.

Repeat counts aren't limited to arrays; any variable may be followed by a repeat count. The general syntax `var,<n>` simply displays *n* consecutive variables of the same type as *var*, starting at the address of *var*. Note however, that the repeat count is ignored if your expression is not equivalent to a variable. A good rule of thumb is that a given construct is a variable if it can legally appear on the left hand side of an assignment statement, or if it can be used as an argument to a function.

Table 4.1: Debugging Format Specifiers

Character	Function
C	**Character**. Shows special display characters for control characters (ASCII 0 through 31); for example, a ^C would be displayed as a Happy Face. Affects characters and strings.
S	**String**. Shows control characters (ASCII 0 through 31) as ASCII values using the appropriate C escape sequences. Since this is the default character and string display format, the S specifier is only useful in conjunction with the M specifier.
D	**Decimal**. All integer values are displayed in decimal. Affects simple integer expressions as well as arrays and structures containing integers.
H or X	**Hexadecimal**. All integer values are displayed in hexadecimal with the `0x` prefix. Affects simple integer expressions as well as arrays and structures containing integers.

F<*n*> — **Floating-point.** *n* is an integer between 2 and 18 specifying the number of significant digits to display. Affects only floating-point values.

M — **Memory dump**. Displays a memory dump, starting with the address of the indicated expression. The expression must be a construct that would be valid on the lefthand side of an assignment statement, i.e. a construct that denotes a memory address; otherwise, the M specifier is ignored. By default, each byte of the variable is shown as two hexadecimal digits. Adding a D specifier with the M causes the bytes to be displayed in decimal, and adding an H or X specifier causes the bytes to be displayed in hexadecimal. A C or an S specifier causes the variable to be displayed as a string (with or without special characters). The default number of bytes displayed corresponds to the size of the variable, but a repeat count may be used to specify an exact number of bytes.

P — **Pointer**. Displays pointers in *seg:ofs* format with additional information about the address pointed to, rather than the default hardware-oriented *seg:ofs* format. Specifically, it tells you the region of memory in which the segment is located, and the name of the variable at the offset address, if that is appropriate. The memory regions are as follows:

Memory Region	Evaluate Message
0000:0000 – 0000:03FF	Interrupt vector table
0000:0400 – 0000:04FF	BIOS data area
0000:0500 – Turbo C	MSDOS/TSR's
Turbo C – User Program PSP	Turbo C
User Program PSP	User Process PSP
User Program – top of RAM	Name of a static user variable if its address falls inside the variable's allocated memory; otherwise nothing

Table 4.1: Debugging Format Specifiers (continued)

	A000:0000 – AFFF:FFFF	EGA Video RAM
	B000:0000 – B7FF:FFFF	Monochrome Display RAM
	B800:0000 – BFFF:FFFF	Color Display RAM
	C000:0000 – EFFF:FFFF	EMS Pages/Adaptor BIOS ROM's
	F000:0000 – FFFF:FFFF	BIOS ROM's
R	**Structure/Union.** Displays field names as well as values, such as `{ X:1, Y:10, Z:5 }`. Affects only structures and unions.	

Here are some general rules that apply to format specifiers:

1. A format specifier takes effect only if it is appended to a variable of the appropriate type. Otherwise it is ignored.

   Note that if the expression being evaluated causes multiple objects to be displayed (for example, as in a structure), and you suffix the expression with more than one format specifier, the appropriate format specifier will be applied to each object. For example, if you enter `struct,F5H` to display a structure containing integers and reals, integers will be displayed in hexadecimal (H) and reals in floating-point format to five significant digits (F5).
2. If you enter more than one format specifier of the same type for an expression, type-dependent priority determines which of the conflicting specifiers will be used: the one with the highest priority is chosen. This whole issue, of course, is really relevant only for structures and unions. For simple variables and arrays of simple variables, you will typically enter only one format specifier.

Table 4.2: Priority and Defaults in Format Specifier Classes

Type	Specifiers in Order of Priority	Default
char	C S H D	S
unsigned char	C S H D	S
int	H D C* S*	D
unsigned int	H D C* S*	D
long	H D C* S*	D
unsigned long	H D C* S*	D
char ptr	C S P H	S
other **ptr**	P H	P
enum	H D C* S*	D (followed by member name)
float	F*n*	F7
double	F*n*	F15
long double	F*n*	F18
array of **char**	C S H D	S
other array	elements enclosed in braces {} and separated by commas	
structure	R	——

* character type format specifiers will be accepted only for values that fall within the appropriate range (-128 to 127 for **signed** types and 0 to 255 for **unsigned** types).

Note: The H format specifier used by itself with a pointer variable displays the pointer as a hexadecimal integer value.

To demonstrate the use of format specifiers, assume that the following structure and variables have been declared:

```
struct {
   int    account;
   char   name[10];
} client = { 5000, "Jones" }

int       list[5] = {0,10,20,30,40};
char      *ptr = list;

void main()
{
}
```

Then entering the following expressions in the Evaluate field will produce the corresponding evaluation in the Result field:

Evaluate	Result
`list`	`{ 0, 10, 20, 30, 40 }`
`list[2],3`	`20, 30, 40`
`list[2],3x`	`0x14, 0x1E, 0x28`
`list,m`	`00 00 0A 00 00 14 00 1E 00 28 00`
`ptr`	`DS:0198`
`ptr,p`	`DS:0198 [_list]`
`*ptr,3`	`0, 10, 20`
`client`	`{ 5000, "Jones\0\0\0\0\0" }`
`client,r`	`{ account:5000, name:"Jones\0\0\0\0\0" }`

Exercise 2: Finding the Bug in wordlen

Now let's return to WORDCNT and find out what's wrong with the **wordlen** function.

It's generally a good idea to see if you can figure out what's wrong just by studying the code. In fact, it's often the quickest way to find a bug once your attention has been drawn to the right place. But this time, please resist the temptation and play along with these exercises.

If you're still in the debugging session you started in Exercise 1, cancel it by choosing **R**un/**P**rogram Reset (or pressing its hot key, *Ctrl-F2*). This makes Turbo C release any memory WORDCNT allocated, close the file it opened, and end the current run of WORDCNT.

On the other hand, if you've left the integrated environment or used it to run other programs since you completed Exercise 1, start it again and choose the project file WORDCNT.PRJ. Then set a breakpoint on the `while (fgets(... statement` again.

Now choose **R**un/**R**un to start a new debugging session. Turbo C will prepare WORDCNT for execution again. Turbo C lets the program run until it stops at the breakpoint.

At the filename prompt, type in `WORDCNT.DAT` and press *Enter*.

Choose **R**un/**S**tep Over to run WORDCNT up to the statement that calls **wordlen**. Then choose **R**un/**T**race Into (or use its hot key, *F7*) to make the debugger go into **wordlen**, rather than run the statement that calls it.

Run/Trace Into runs a program one line at a time, like **Run/Step** Over, but it steps into function calls instead of over them. In this case, it leaves the execution bar at the declaration of **wordlen**.

Study **wordlen**'s logic a moment. The function expects one parameter, a pointer to a word in a line buffer, which it names *wordptr*. It assigns the value of *wordptr* to a local variable, *wordlimit*. Then a **while** loop advances *wordlimit* until it points at a space ending the word or the null character ending the line. It returns the difference between *wordlimit* and *wordptr* as the length of the word.

Run two lines to execute **wordlen**'s definition through the assignment statement. You may use either **Run/Trace** Into or **Run/Step** Over; since **wordlen** calls no lower functions, the two commands will have the same effect.

Evaluate the string that *wordlimit* points to; it's the first line of input, as it should be. Run the **while** statement. The execution bar *should* move to the following line, which increments *wordlimit*. Instead it moves to the line after that, which contains a return statement. This may be the bug you're looking for.

Note: You have probably been pressing *Esc* twice at the end of an evaluation, in order to return to the TC Editor. Instead, try the hot key for this operation, *F10*.

Consider what will happen next. Since *wordlimit* has not been incremented, the difference between *wordlimit* and *wordptr* is 0, and **wordlen** will return 0. This is the bug you're looking for. You've narrowed down the scope of the bug from "something in **wordlen**" to "something in the expression of **wordlen**'s **while** statement." Once again, it's worthwhile to look at the code and see if you can deduce what's wrong.

If you can't, try evaluating the parts of the guilty expression to see how they work together. You'll find that the value of **wordlimit* is ASCII *T*. You can't evaluate **wordlimit* != SPACE, since SPACE is a #defined symbol; but you can evaluate **wordlimit* != ' ' (the value of SPACE is ' '), and its value is 1 (true). The value of the whole expression ought to be true, and it's false. Something's wrong with the & operator.

In fact, & is the wrong operator. It is C's *bitwise and operator*, which ands each bit of one operand with the corresponding bit of the other. Since **wordlimit* != SPACE is always 0 or 1, **wordlimit* & **wordlimit* != SPACE is 0 whenever **wordlimit* is even. The operator should be &&, which gives a result of 1 whenever both of its operands are nonzero. (Try evaluating the whole expression with one & and then with two to see the difference.)

(If you didn't figure out the problem yourself, don't feel bad. Confusing & with && and | with || is one of the most common errors that novice C programmers make. After you've found this error in your own code a few times, you'll learn to recognize it readily.)

Fixing the Bug

To fix the bug, all you need to do is change the & to &&. Save the corrected program file (press *F2*) to protect yourself from losing the change you made if your program should crash during a subsequent debugging session. Then enter Run/Run again. Since you have modified the source code, you will be asked if you want to rebuild. Type `Y`, and the program will compile and link. Presto, you're ready to debug the corrected program.

We'll go hunting for more bugs after another brief time-out to summarize what you've learned and take a closer look at some of the features you've encountered.

What You've Accomplished

You have cancelled your first debugging session with Run/Program Reset or its hot key, *Ctrl-F2*. Then you ran WORDCNT up to the call to **wordlen** by setting a breakpoint there and choosing Run/Run (or its hot key combination *Ctrl-F9*).

You traced into the call to **wordlen** with Run/Trace Into (hot key *F7*). That let you step through **wordlen** and find the bug. You fixed the bug, saved the source file, and prepared the updated program for debugging with Run/Run.

More About Breakpoints

If you didn't leave Turbo C after working through the first exercise, the breakpoint you set at `while ( fgets(...` was still set when you started the second one. That's why Run/Run let WORDCNT run to the breakpoint instead of running to the end of the program. As you can see, breakpoints "stick" from one debugging session to the next. This is so even if you edit and remake your program in between. Turbo C moves each breakpoint up or down to keep it on the right statement.

Breakpoints stick even if you set them in a program file, save the file, and edit other files. You lose your breakpoints only when you:

- leave the integrated environment
- delete the lines they are set on in the source file
- clear them all by choosing **Break/Watch/Clear** All Breakpoints

However, Turbo C can lose track of its breakpoints in two cases:

- *If you edit a file that contains breakpoints, then abandon (fail to save) the edited version of the file.* Turbo C cannot remember where the breakpoints were set in the original version of the file, and so will display them on the wrong lines.

  If you must abandon the edited version of a source file, clear all breakpoints (choose **Break/Watch/Clear** All Breakpoints) and recreate the ones you need. Note that **Break/Watch/Clear** All Breakpoints clears all the breakpoints in your programs, not just those in the source file you are editing.
- *If you edit a file and then continue the current debugging session, or start a new session without recompiling and relinking.* The breakpoints are actually set in the right places, but because the source file no longer matches the executable program, the breakpoint highlights appear on the wrong lines. (The execution bar also appears on the wrong lines.)

  You aren't likely to get into this situation by accident because Turbo C displays the warning prompt "Source modified, rebuild?" when you try to continue or restart the debugging session.

Before you compile a source file you can set breakpoints on lines that contain no executable statements, such as comments and blank lines. In that case, Turbo C will warn you before it runs the program that the breakpoints are set on source lines that contain no executable code. Once you have compiled a file, Turbo C knows which lines contain executable statements, and will warn you if you try to set breakpoints on those lines.

You can move the cursor to the next breakpoint in a file by choosing **Break/Watch/View** Next Breakpoint. Note that this command moves the cursor to the next breakpoint in the current project, not the next breakpoint that will be executed when the program is run. You can use **Break/Watch/N**ext Breakpoint to review the breakpoints in a program when you want to clear some but not all of them.

Exercise 3: Back to the Program

Let's see if the change you made fixed the bug.

If you have left the integrated environment or run other programs since completing Exercise 2, start the integrated environment again and make WORDCNT.PRJ the current project. Choose **R**un/**R**un to start a new debugging session. Run WORDCNT up to the call to **wordlen**; trace into it and step through it. This time it should work correctly and return a value of 2. Success!

Your work is not over, though. Much of this program has not yet been tested, and it may contain more bugs.The next thing you should test is the inner **while** loop in **main**. The most thorough approach is to step through **main** until the entire first line of input has been processed and control leaves the loop. Verify that each step does the right thing and that control leaves the loop at the right point.

Does that sound like a great deal of work? The debugger's Watch window makes it a lot easier. The Watch window displays one or more expressions and their current values. Each time the debugger halts, it re-evaluates each expression in the Watch window. Thus you can watch the expressions' values change as you run your program.

You're going to create watch expressions for all the data elements that are important in the inner **while** loop: the variable *i*, the string beginning at wordptr, and the first few elements of *wordlencnt*.

The Watch window normally occupies the lower part of the screen during a debugging session, just as the Message window does during a compile and link. It is initially one line high and is empty. If the Watch window is invisible, it's because the Edit window is zoomed to full screen; press *F5* to return to a split-screen environment and reveal the Watch window.

Choose **B**reak/Watch/**A**dd Watch (or press the hot key combination, *Ctrl-F7*). The debugger opens a window and prompts you to enter a watch expression. Like **D**ebug/**E**valuate, **B**reak/Watch/**A**dd Watch uses the word at the Edit window's cursor as a default expression. If the default expression doesn't happen to be *i*, type *i*; press *Enter*. The debugger adds the expression *i* and its current value to the Watch window.

Repeat this procedure for the five expressions *wordptr* and *wordlencnt*[1] through *wordlencnt*[4]. As you add each watch expression, the Watch window grows to accommodate it. Now step through the **while** loop until WORDCNT has processed the entire first line of input. As you go, you can watch WORDCNT advance *wordptr* a word at a time and increment the

appropriate elements of *wordlencnt*. Stop when *wordptr* reaches the end of the line and the execution bar leaves the inner **while** loop. The loop appears to be working correctly.

Editing and Deleting Watch Expressions

You can edit and delete watch expressions in the Watch window, as well as add them.

To edit and delete watch expressions, begin with the Watch window active. If you're in the menu system, press *F10* to leave it. Press *F6* to switch from the Edit window to the Watch window. The watch editor highlights the expression about to be edited or deleted; you can move the highlight with the *Home, End, Up arrow,* and *Down arrow* keys.

First let's edit a watch expression. Move the highlight to *wordlencnt*[4] and choose **B**reak/**W**atch/**E**dit Watch (or press *Enter*). The debugger opens a window containing the watch expression, and prompts you to edit it.

Change the array index from 4 to 6 and press *Enter*. The debugger changes the watch expression in the Watch window and displays the value of the new expression.

Move the highlight to the expression *wordlencnt*[3] and choose **B**reak/ **W**atch/**D**elete Watch (or press *Del* or *Ctrl-Y*). The debugger deletes the watch expression.

Press *F6* to activate the Edit window. Notice the diamond-shaped symbol that appears before the highlighted watch expression to mark it when the Watch window is deactivated. Press *F6* again to reactivate the Watch window.

You can delete all the watch expressions at once by choosing **B**reak/**W**atch/ **R**emove All Watches. (You needn't make the Watch window active to do this). Delete all the watch expressions now. The Watch window returns to its original empty state.

Press *F6* again to reactivate the Edit window. Each time you enter this command, it switches the active window from the Edit window to the Watch window, or vice versa.

Zooming and Switching Windows

The rules for zooming and switching windows in the debugger are an extension of the rules you have already learned for the editor, compiler, and linker.

The screen is normally split between the Edit window and the Watch window during a debugging session, just as it is normally split between the Edit window and the Message window when you're compiling and linking.

To switch between the two visible windows—Edit and Message, or Edit and Watch—press *F6*.

To zoom the active window to full screen, press *F5*. When the active window is already zoomed, pressing *F5* returns the screen to the split-screen display. Try this now with the Edit window and the Watch window.

To display the Execution screen, select **Run/User** Screen, or press *Alt-F5*. Try this now. Press any typing key to return to the prior display.

Use *Alt-F6* to change the contents of a window:

- When the Edit window is active, the file loaded just before the current file is reloaded.
- If the Watch window or Message window is active, pressing *Alt-F6* toggles between the Watch editor and the Message tracking window.

Scrolling Watch Expressions

As you add expressions to the Watch window, it will grow to fill about half of your screen. If you add more expressions, some will scroll out of the window. You can get them back by scrolling the window display with the *PgUp, PgDn, Up arrow,* and *Down arrow* keys.

If an individual watch expression is too long to fit in the window, you can see its beginning and end by scrolling it with the *Home, End, Left arrow,* and *Right arrow* keys (in the same way **Debug/Evaluate** lets you scroll long expressions and values).

Exercise 4: Debugging the Print Loop

You've now debugged the whole of the **while** loop that reads and processes the input file. There are a few places in this loop where bugs might still lurk, but to reach them you'll have to feed WORDCNT specific types of

input, such as empty lines. Therefore, we'll defer searching for such bugs and go on to the **for** loop that prints the results.

You need to run WORDCNT up to the line that begins the **for** loop (it's around Line 117). As you know, you can do that by moving the cursor to the line, setting a breakpoint, and choosing **R**un/**R**un. This time you're going to use another technique that's more convenient if you expect to stop at a certain point only once.

Move the cursor to the line that begins the **for** loop, and choose **R**un/**G**o to Cursor. (You can also use the hot key, *F4*.) The debugger runs WORDCNT and stops on the line at the cursor. (It would stop at a breakpoint if it encountered one first, but none are set in this part of the code.)

Run one statement, moving the execution bar to the **printf** in the **for** loop. Notice that there is no breakpoint highlight on the **for** line. **R**un/**G**o to Cursor is a one-time operation; it does not set a permanent breakpoint.

Start stepping through the **for** loop with **R**un/**T**race Into. Notice that **R**un/**T**race Into does not trace into the **printf** function. That's because **printf** isn't defined in a source file compiled with debug information. The debugger can run such functions, but it can't trace into them. (In this case, there's another reason why you can't trace into the function: The source file for **printf** isn't on your disk.)

Each time you run a line that contains a **printf** call, the debugger swaps to the Execution screen. This permits **printf** to perform its output in the proper context. The debugger displays the Execution screen every time you run a statement containing a function, since it can't tell which functions will or won't write to the screen.

You probably won't be able to read the program's output in the brief time the Execution screen is visible, so redisplay it with **R**un/**U**ser Screen or *Alt-F5* to see what WORDCNT is doing to the screen.

Note: On the other hand, if you notice that output from your program is overwriting your source code in the Edit window, you will want to get rid of it by choosing **D**ebug/**R**efresh Display to refresh the screen. Then check to make sure that the **D**ebug/**D**isplay Swapping option is set to either **S**mart or **A**lways.

(For more information on these options, see the discussion in Chapter 5.)

This completes the fourth exercise. We'll leave it to you to decide whether there's a bug in the **for** loop, and if so, how to fix it.

Note: We promise that WORDCNT contains several more bugs. Can you find them? Each of them is mentioned somewhere in the section "Guidelines for Effective Software Testing," later in this chapter.

Exercise 5: Working with Large Programs

The debugger has several features to help you work with large source files and multi-file programs. In the next few sections, we'll demonstrate them.

To use one of the features we're going to demonstrate, you must compile your program with the **Options/Compiler/Code Generation/Standard Stack Frame** option on. Check this option now. If it's off, turn it on, then make sure that WORDCNT is loaded into the Edit window and recompile by pressing *Alt-F9*.

Start a new debugging session by choosing **Run/Step Over** or pressing the corresponding hot key, *F8*.

Finding the Definition of a Function

Debug/Find Function scrolls a function's definition into the Edit window. It can find any function in your program that was compiled with the **Debug/Source Debugging** and **O/C/C/OBJ Debug Information** toggles set to On.

Debug/Find Function is useful if you discover a bug while you are working in one part of a program, but must fix it by changing another part. Or it can show you a function's code or comments to help you remember how the function works.

To try out this command, move the cursor to that infamous line in **main** that calls **wordlen**. Choose **Run/Go to Cursor**; then move the cursor to the name **wordlen** and choose **Debug/Find Function**. The debugger opens a window and prompts you to enter a function name; Since **wordlen** is the function you want to find, type it in, then press *Enter*. The debugger displays the definition of **wordlen** in the Edit window.

The Call Stack

When you're debugging a program that calls many levels of functions, you sometimes need to see the call stack, which tells you what functions the program has called, and in which order, to reach the current execution

position. This is the feature that requires the Options/Compiler/Code Generation/Standard Stack Frame option to be on.

Trace into **wordlen**, then choose Debug/Call Stack, or press the hot key for Debug/Call Stack, *Ctrl-F3*. The debugger displays the call stack in a pop-up window. The function now being executed is at the top of the stack, and **main** is at the bottom. Notice that the call stack displays not only the names of the functions the program has called, but the values of their parameters.

You can display the currently executing line of any function on the call stack by moving the call stack window's highlight to that function and pressing *Enter*. For example, to display the currently executing line of **main**, move the highlight to the call stack entry for main and press *Enter*. The debugger scrolls the part of **main** containing the call to **wordlen** into the Edit window. If there were additional entries on the call stack, you could display the currently executing line of any other function by choosing Debug/Call Stack again and choosing another call stack entry.

Returning to the Execution Position

You can also use the Debug/Call Stack option to return to the execution bar after looking at another part of your program. To scroll the line with the execution bar back into the Edit window, just choose the topmost function on the call stack. Try it now.

This ends the exercises relating to the debugger.

About Multiple Source Files

All the debugger's commands work with programs that consist of multiple source files. For example, if you choose Debug/Find Function to find a function that is defined in a source file other than the one in the Edit window, the debugger loads the appropriate file into the window. If you have made changes to the file currently in the window, the debugger asks whether you want to save them before loading the new file.

Similarly, when you choose a function on the call stack (using Debug/Call Stack), the debugger reloads the Edit window with the source file containing the execution position (that is, the next line to be executed in the current function). If you have made changes to the other file, the debugger asks whether you want to save them before reloading the original file. If your program consists of many source files, it is wise to debug only a few at

a time. It is easier to keep control of your work if only a few parts of the program are changing at once.

Survey of Debugger Commands and Hot Keys

This tutorial has presented the integrated debugger's most often-used commands. There are more commands that you may want to learn when you have gained some mastery of the debugger. To find them, refer to the two tables below.

Many debugger commands and other menu commands can be invoked by using hot keys or key combinations. To avoid presenting a confusing amount of detail, we've mentioned only the most important ones. Table 4.3 shows all the hot keys for the debugging commands you have learned.

Table 4.3: Debugger Commands and Hot Keys

Hot Key	Menu Command	Description
F4	Run/Go to Cursor	Runs program, stopping on line at the cursor. Will initiate a debugging session.
Ctrl-F2	Run/Program Reset	Cancels current debugging session, releases allocated memory, and closes files. Valid only in debugging sessions.
F7	Run/Trace Into	Runs next statement in the current function. If it calls a lower-level function compiled with Debug/Source Debugging and O/C/C/OBJ Debug Information set to On, traces into that function. Will initiate a debugging session.
F8	Run/Step Over	Runs next statement in the current function. Does not trace into called functions. Will initiate a debugging session.
	O/C/C/Standard Stack Frame	Toggles the Options/Compiler/Code Generation/Standard Stack Frame option. This option must be set to On when a program is compiled if the Debug/Call Stack Option is to work correctly.
	O/C/C/OBJ Debug Information	Toggles the O/C/C/OBJ Debug Information option. Only source

Hot Key	Menu Command	Description
		files compiled and linked with this option set to On can be debugged.
Ctrl-F4	Debug/Evaluate	Evaluates a C expression; allows you to modify the value of a variable.
	Debug/Find Function	Finds a function's definition and displays it in the Edit window. Valid only in debugging sessions.
Ctrl-F3	Debug/Call Stack	Displays call stack. You can display the currently executing line of a function by choosing that function's name from the call stack. Valid only in debugging sessions.
	Debug/Source Debugging	Controls whether debugging is allowed. When it is set to On, both integrated and standalone debugging are possible. When it is set to Standalone, you can debug programs only with the standalone debugger, although you can still run them in TC. When it is set to None, no debugging information is placed in the .EXE file, so you cannot debug your program with either debugger.
Ctrl-F7	Break/Watch/Add Watch	Adds a watch expression.
	Break/Watch/Delete Watch	Deletes a watch expression.
	Break/Watch/Edit Watch	Lets you edit a watch expression.
	Break/Watch/Remove All Watches	Deletes all watch expressions.
Ctrl-F8	Break/Watch/Toggle Breakpoint	Sets or clears a breakpoint on the line at the cursor position.
	Break/Watch/Clear Breakpoints	Clears all breakpoints in the program.
	Break/Watch/Next Breakpoint	Displays next breakpoint.

Table 4.4 shows other menu commands often used with the integrated debugger. (Refer to Chapter 5 to learn hot keys for the rest of the menu commands.)

Table 4.4: Menu Commands and Hot Keys Used with the Debugger

Hot Key	Menu Command	Description
F5		Zooms and unzooms the active window between full-screen and split-screen modes.
Alt-F5		Switches the display to the User screen. Press any key to return to the integrated environment.
F6		Switches active window between the Edit window and the Watch or Message window.
Alt-F6		If Edit window is active, switches to the last file loaded into the Editor. If lower window is active, switches between Watch window and Message window.
Ctrl-F9	Run/Run	Runs a program, with or without the debugger. Compiles source file(s) and links program if necessary. When the program has been compiled and linked with Debug/Source Debugging and O/C/C/OBJ Debug Information set to On, runs program to a breakpoint or to the end of the program.
	Project/Remove Messages	Deletes contents of the Message window.

Guidelines for Effective Software Testing

There's a lot more to testing software than knowing how to use a debugger. Figuring out whether and why a program is misbehaving is one of the most challenging phases of programming.

The rest of this chapter will suggest some techniques that can make your testing work easier.

Develop a Standard Approach

Evolve a standard approach to software testing: a checklist of steps that your experience shows will lead you to a reliable program.

There is no one "right" way to test a program; your checklist will depend on the types of programs you write, your strengths and weaknesses as a programmer, and your personal style. The following checklist may serve as a starting point:

- Feed the program some input that is simple but not trivial. Trace into the code, using Debug/Evaluate and watch expressions liberally to check the values of data items. Correct the bugs you find, one or a few at a time.
- Feed the program other sets of data that will let you exercise the parts you couldn't test in the preceding step.
- Test every statement in your program. You may find bugs where you didn't suspect they could exist. In WORDCNT, for example, testing every statement would reveal a vagrant semicolon after an else that has disasterous effects when the program encounters a word more than MAXWORDLEN characters long. (The semicolon is normally beyond the right edge of the Edit window, and so you'd be unlikely to notice it any other way.)
- Be alert for individual statements or expressions that must be tested more than one way, like these:

  ```
  if ( strcmp(a,b) ) . . .
  ```

  strcmp can return three values, 0 (*a* equals *b*), –1 (*a* is less than *b*), or +1 (*a* is greater than *b*). This suggests that you should test the statement with three sets of input values to verify that **strcmp** makes it do the right thing in each case.

  ```
  x = (x>0) ? func(x) : 0 ;
  ```

  This statement contains an "implicit if" that can produce two different results.
- Give special attention to boundary conditions: Conditions that make a program escape from a loop, fill an array, and so on. Bugs are especially likely to be manifested as failures to handle boundary conditions correctly.

  WORDCNT contains two related examples of boundary condition bugs. First, the final **for** loop fails to print the element of *wordlencnt* that represents words MAXWORDLEN characters long. Second, the definition of *wordlencnt* allocates one element too few, so that that element doesn't even exist.

- Put aside the debugger and test the entire program for correct behavior. If the program will be used by other people who will expect it to be well-behaved, test its response to every type of error it could possibly encounter. A program that handles most types of errors well is said to be robust.
- If other people will be using your program, have at least one other person test it. Choose someone whose skills and needs are typical of your program's intended users, but who has enough persistence and enthusiasm to dig out obscure bugs and report them accurately. Don't choose a programmer unless your program is meant to be used by programmers.

Test Modifications Thoroughly

When you modify a program, retest the affected parts thoroughly. You may have to retest parts that haven't changed but are affected by the changes.

If the program is complex, keep a record of the tests you have performed. When you modify the program, this record will help you repeat all the tests whose results could possibly be affected by the change. If the tests involve particular input files, save the files.

Design Defensively

You can avoid bugs by designing your program defensively, just as you can avoid accidents by driving your car defensively.

Write your code cleanly, with consistent indention, liberal comments, and descriptive variable names.

Keep your code simple. Express complicated operations in many simple statements rather than a few complex ones. Turbo C's code optimization will make your code reasonably efficient, and it will be much easier to debug, read, and modify.

Try to build up your program from functions whose purposes are simple and well-defined. This will make it easier to set up test cases and analyze their results. It will also make your program easier to read and modify.

Try to minimize the number of data elements each function requires and the number of elements it changes. This too will make it easier to set up test cases and analyze their results, and to read and modify your program. It will also tend to limit the amount of havoc a misbehaving function can

cause, allowing you to run the function several times in a single debugging session. A program designed this way is said to be loosely coupled.

Don't try to squeeze the last bit of efficiency out of your program when you write it. When you try to make code as efficient as it can be, it also tends to become hard to read and debug. If your program turns out to be too slow when it's done, that's the time to decide which parts are worth speeding up, and how best to do it.

Be alert for opportunities to write functions that can be used more than one way in your program, or can be reused in other programs. Writing and debugging one generalized function is usually easier than writing two or more specialized ones.

Debug from the Bottom Up

As far as possible, concentrate on debugging your program's lowest-level functions (the ones that don't call other functions) first. Then work upward toward **main**. In this way, you'll get a foundation of reliable functions that you can step over when they are called in other parts of your code.

Look for Classes of Bugs

When you find a bug, look carefully for other bugs of the same kind and/or in the same part of the program. For example, if you find a function call that says

```
fp = fopen("rb",filename);
```

which should be

```
fp = fopen(filename,"rb");
```

check your code for other calls to **fopen** and similar functions that exhibit the same error.

Debugging Inline Assembly Code

If you are using the integrated development environment in conjunction with Borland's TASM assembler, you may step through assembly level code without having to use an external debugger. (For full-featured assembly level debugging, however, you should get Borland's standalone debugger.)

When an assembly language module is assembled with TASM's `-zi` switch, TC can recognize lines and symbols from the assembly source. If you step into an assembly level function, TC will display the assembly language source code for that function. You may use normal TC debugging commands such as **Debug/Go** to cursor (F4), **Debug/Trace** Into (F7), and **Debug/Step** Over (F8) with your assembly language source code.

Most symbols defined in your assembly language source code will be available for you to use in evaluating expressions and watch expressions. In addition, you have access to the pseudo-registers (_AX, _BX, etc.) and a special _FLAGS variable that reflects the state of the CPU's flags register. However, since TC is a C development environment, it will not recognize every assembler construct or expression.

C H A P T E R

5

The Turbo C Integrated Development Environment

TC, the Turbo C integrated development environment, is much more than just a fast C compiler; it is a fast, efficient C compiler with a built-in editor, debugger, and other utilities that are easy to learn and easy to use. With TC, you don't need to use a separate editor, debugger, compiler, linker, and Make utility in order to create and run your C programs. All these features are built right into TC, and they are all accessible from one clear and simple display—the main TC screen.

In This Chapter ...

This chapter is divided into four sections: Part I, "Using TC;" Part II, "The Menu Commands;" Part III, "More about Configuration and Pick Files;" and Part IV, "Additional Features and Editing Commands."

In Part I, "Using TC," we

- introduce the TC command-line switches and hot keys
- describe the components of the main TC screen
- explain how to use the TC main menu choices
- demonstrate how to get into the Edit window and use the TC Editor (editing commands are covered in Appendix A in the *Turbo C Reference Guide*, "The Turbo C Interactive Editor"

- introduce you to the TC integrated debugger (for detailed information on how to use the debugger, see Chapter 4, "Debugging Your Program").

In Part II, "The Menu Commands," we

- examine and explain each menu item's function
- summarize the compile-time options

In Part III, "More about Configuration and Pick Files," we

- discuss what a configuration file is and how you create and use it in TC
- discuss how to create and use a pick file in TC

In Part IV, "Additional Features and Editing Commands," we

- discuss editing features not available through the menu system
- explain how to customize your editing keys with the Turbo C customization program TCINST.EXE.

What You Should Read

If you are not familiar with Borland's integrated development environments, you will want to read Parts I and II first. If you are well-versed in working with menu-driven products such as SideKick or Turbo Pascal, you may want to skim these sections before reading Parts III and IV.

How to Get Help

Turbo C, like other Borland products, gives context-sensitive onscreen help at the touch of a single key. You can get help at any point within any TC menu.

To call up the Help system, press *F1*. The Help window explains the functions of the item on which you're currently positioned. Any help screen may contain a *keyword* (a highlighted item) that you can choose to get more information. Use the arrow keys to move to any keyword and press *Enter* to get more detailed help on the chosen item. You can use the *Home* and *End* keys to go to the first and last keywords on the screen, respectively.

If you want to return to a previous help screen, you can press *Alt-F1* whether you are inside or outside the Help system. TC lets you back up through 20 previous help screens.

To get to the help index, press *F1* again once you're in the Help system.

You may find that while you are in the TC Editor you need help on various library functions. If you position the cursor under the function name (such as **printf**) you want information on, then press *Ctrl-F1*, you can bring up a help screen with the information you want.

To exit from the Help system and return to your menu choice, press *Esc* or any of the *hot keys* described in the next section.

Part I: Using TC

To load the Turbo C integrated development environment (TC), type `TC` and press *Enter* at the DOS prompt.

The startup screen that appears includes the main TC screen and a box containing product version information. When you press any key, the version information disappears (pressing *Shift-F10* any time will display it again), but the main screen remains (see Figure 5.1).screens, main TC

Look closely at the main TC screen; it consists of four parts: the main menu, the Edit window, the Message window, and the Quick Reference (Quick-Ref) Line.

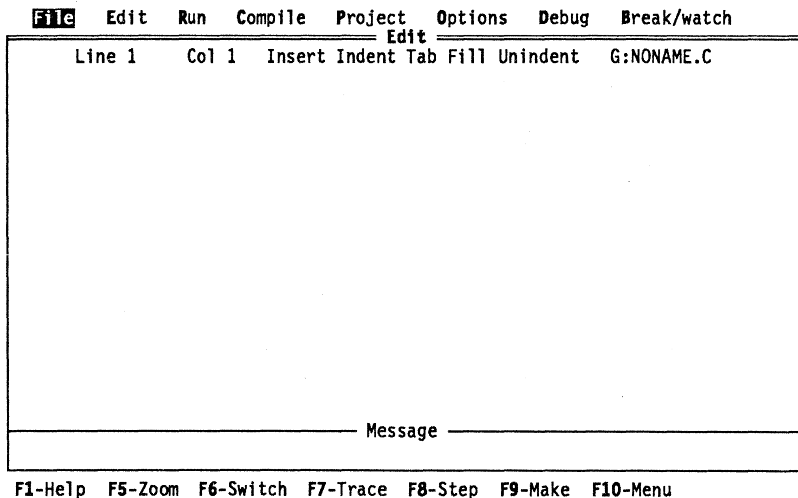


Figure 5.1: The Main TC Screen

TC Command-Line Switches

The Turbo C integrated development environment accepts the following command-line switches:

- The **/c switch** causes a configuration file to be loaded. Enter the `tc` command, followed by `/c` and the configuration file name, with no space between the two:

```
tc /cmyconfig.tc
```

(See Part III of this chapter for more on configuration files and pick files.)

- The **/b switch** causes TC to recompile and link all the files in your project, print the compiler messages to the standard output device, and then return to DOS. This switch allows you to invoke TC from a batch file, so you can automate builds of projects. Before the build, TC will load either a default configuration file or one given specially with the `/c` switch. TC determines what .EXE to build based on the project file, the primary file, or the file currently loaded into the TC Editor, in that order of precedence. Enter the `tc` command with either `/b` alone or `/c` and the configuration file name followed by `/b`.

```
tc /cmyconfig.tc /b
tc /b
```

  Unless the loaded configuration file specifies a project or primary file, you can specify the name of a program to be compiled and linked on the command line. Type in the program name after the `tc` command, followed by `/b`.

```
tc myprog /b
```

- The **/m switch** lets you do a make rather than a build (that is, only outdated source files in your project are recompiled and linked). Follow the instructions for the `/b` switch, but use `/m` instead.
- The **/d switch** causes TC to work in dual monitor mode, if it detects appropriate hardware. Otherwise, the `/d` switch is ignored. Dual monitor mode is used when you are running or debugging a program, or shelling to DOS (File/OS Shell).

  When you type in the `/d` switch on TC's command line, it enables dual monitor mode, as long as the required hardware is present (for example, a monochrome card and a color card). If your system has two monitors, DOS treats one monitor as the active monitor. Use the DOS MODE command to switch between the two monitors (`MODE CO80`, for example, or `MODE MONO`). In dual monitor mode, the normal TC screen will appear on the inactive monitor, and program output will go to the active monitor. Thus, when you type `tc /d` at the DOS prompt on one monitor, TC will come up on the other monitor. When you want to test your program on a particular monitor, you must exit TC, switch the active monitor to the one you want to test with, then issue the `tc /d` command again. Program output will then go to the monitor where you typed the `tc` command.

  WARNING:

  - Do not change the active monitor (by using the DOS MODE command, for example) while you are in a DOS shell (File/OS Shell).

- User programs which directly access ports on the inactive monitor's video card are not supported, and can cause unpredictable results.
- When you run or debug programs that explicitly make use of dual monitors, do not use the TC dual monitor switch (`/d`).

Finding Your Way around TC

To help you gain familiarity with the Turbo C integrated development environment, here are some navigating basics:

From anywhere in TC:

- Press *F1* to get information about your current position (help on running, compiling, and so on).
- Press *F5* to zoom/unzoom active window.
- Press *F6* to switch windows.
- Press *F10* to toggle between the menus and the active window.
- Press *Alt-F6* to change the contents of a window (switch from Message window to Watch window and back, or toggle between the current file and the previous file).
- Press *Alt* plus the first letter of any main menu command (*F*, *E*, *R*, *C*, *P*, *O*, *D*, or *B*) to invoke the specified command. For example, pressing *Alt-E* from anywhere in the system will take you to the **E**dit window; *Alt-F* takes you to the **F**ile menu.

From within a menu:

- Use the highlighted capital letter to choose a menu item, or use the arrow keys to move to the option, then press *Enter*.
- Press *Esc* to exit a menu.
- In the main menu, or in one of the pull-down menus invoked from the main menu, pressing *Esc* will take you directly back to the active window. (When it is active, the window will have a double bar at its top and its name will be highlighted.)
- Press *F10* to get from any menu level to the previously active window.
- Use the *Right* and *Left arrow* keys to move from one pull-down menu to another.

To exit TC and return to DOS:

Go to the **F**ile menu and choose **Q**uit (press *Q*, or move the highlight bar to **Q**uit and press *Enter*). If you choose **Q**uit without saving your current work

file, the Editor will query whether you want to save it. (You can also use the hot key *Alt-X* to quit and return to DOS.)

The TC Hot Keys

Before we describe the various menu options available to you, there are a number of *hot keys* (shortcuts) you should be aware of. Hot keys are keys set up to perform a certain function. For example, as discussed previously, pressing *Alt* and the first letter of a main menu command will take you to the specified option's menu or perform an action. The only other *Alt*/first-letter command is *Alt-X*, which is really just a shortcut for **F**ile/**Q**uit.

In addition to these *Alt*-first-letter commands, TC has a special User screen hot key, *Alt-F5*, that you use to switch from the main TC screen and a *User screen* where your program output is displayed. It is equivalent to the menu command **U**ser Screen on the **R**un menu.

When you are using TC, you see one of two screens—the main TC screen or the User screen. The main TC screen is what you see when you edit, compile, link, and debug your programs. The User screen is what you see when you run a Turbo C executable program or temporarily exit to DOS through the **F**ile/**O**S Shell menu command. When you are using the integrated debugger, you will often swap the TC and User screens. TC is able to preserve the contents of the latter screen continuously in a saved User screen buffer, updating it each time you choose a run command (like **R**un, **T**race Into, or **S**tep Over) or **F**ile/**O**S Shell. To view this saved screen, select **U**ser Screen from the **R**un menu, or press *Alt-F5*.

Note: In dual monitor mode, the User screen is already displayed on one of the two monitors in the system. Thus, the **R**un/**U**ser Screen command and *Alt-F5* will be disabled.

How TC determines whether it needs to clear the User screen depends on the video mode. When TC is invoked from DOS, or when you return to it from the DOS shell, it remembers the video mode and cursor type. These two states are reset independently of one another whenever you shell to DOS (**F**ile/**O**S Shell) or exit the integrated environment (**F**ile/**Q**uit), if the current state is found to be different than the remembered state. There is one exception to this: If you shell to DOS during a debugging session (when a program is running), the mode and cursor type are left in the state your program put them.

Table 5.1 lists all the hot keys you can use in TC. Remember that, when these keys are pressed, their specific function is carried out no matter

where you are in the TC environment. There is one exception: Whenever you are presented with a dialog box that requests specific keys to be pressed, the hot keys are disabled until you have pressed the requested key.

Table 5.1: Turbo C Hot Keys

Key(s)	Function
F1	Brings up a Help window with information about your current position
F2	Saves the file currently in the Editor
F3	Lets you load a file (an input box will appear)
F4	Runs program to line the cursor is on
F5	Zooms and unzooms the active window
F6	Switches active windows
F7	Runs program in debug mode, tracing into functions
F8	Runs program in debug mode, stepping over function calls
F9	Performs a "make"
F10	Toggles between the menus and the active window
Ctrl-F1	Calls up context help on functions (TC Editor only)
Ctrl-F2	Resets running program
Ctrl-F3	Brings up call stack
Ctrl-F4	Evaluates an expression
Ctrl-F7	Adds a watch expression
Ctrl-F8	Toggles breakpoints On and Off
Ctrl-F9	Runs program
Shift-F10	Displays the version screen
Alt-F1	Brings up the last help screen you referenced
Alt-F3	Lets you pick a file to load
Alt-F5	Switches between main TC screen and User screen
Alt-F6	Switches contents of active window
Alt-F7	Takes you to previous error
Alt-F8	Takes you to next error
Alt-F9	Compiles to .OBJ the file loaded in the TC Editor
Alt-B	Takes you to the Break/Watch menu

Table 5.1: Turbo C Hot Keys (continued)

Alt-C	Takes you to the Compile menu
Alt-D	Takes you to the Debug menu
Alt-E	Puts you in the Editor
Alt-F	Takes you to the File menu
Alt-O	Takes you to the Options menu
Alt-P	Takes you to the Project menu
Alt-R	Takes you to the Run menu
Alt-X	Quits TC and returns you to DOS

Menu Structure

Figure 5.2 shows the complete structure of TC's main menu and its successive pull-down menus. There are three general types of items on the TC menus: commands, toggles and settings.

Commands Perform a task (running, compiling, storing options, and so on).

Toggles Switch a TC feature On or Off (**A**uto Dependencies, **T**est Stack Overflow, and so on) or let you cycle through and choose one of several options by repeatedly pressing the *Enter* key until you reach the item desired (such as **M**essage Tracking or **F**loating Point).

Settings Allow you to specify certain compile-time and run-time information to the compiler, such as directory locations, names of files, macro definitions, and so on.

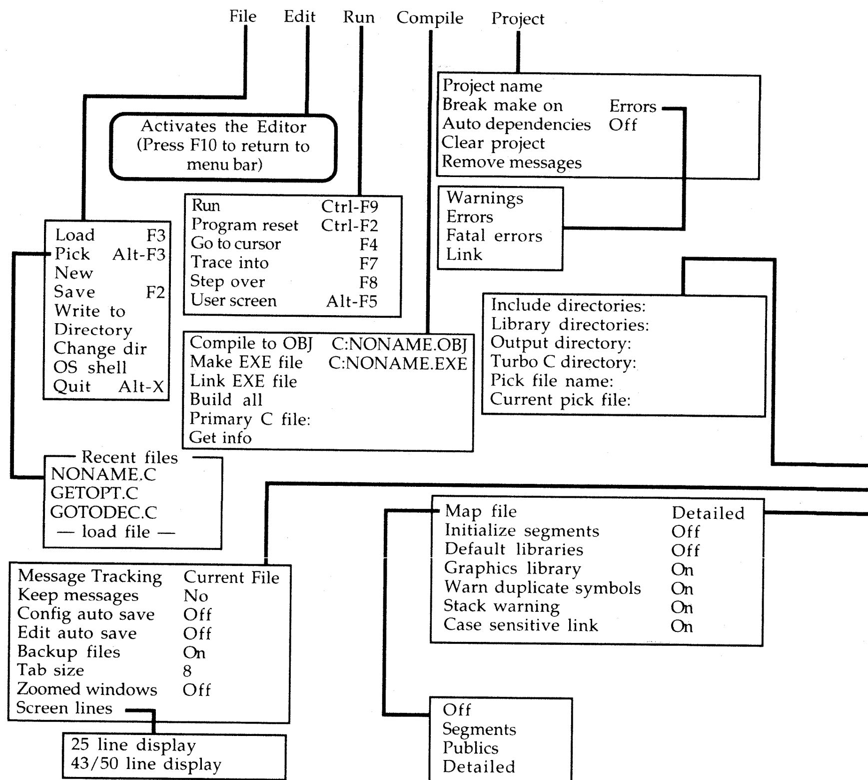


Figure 5.2: The TC Menu Structure

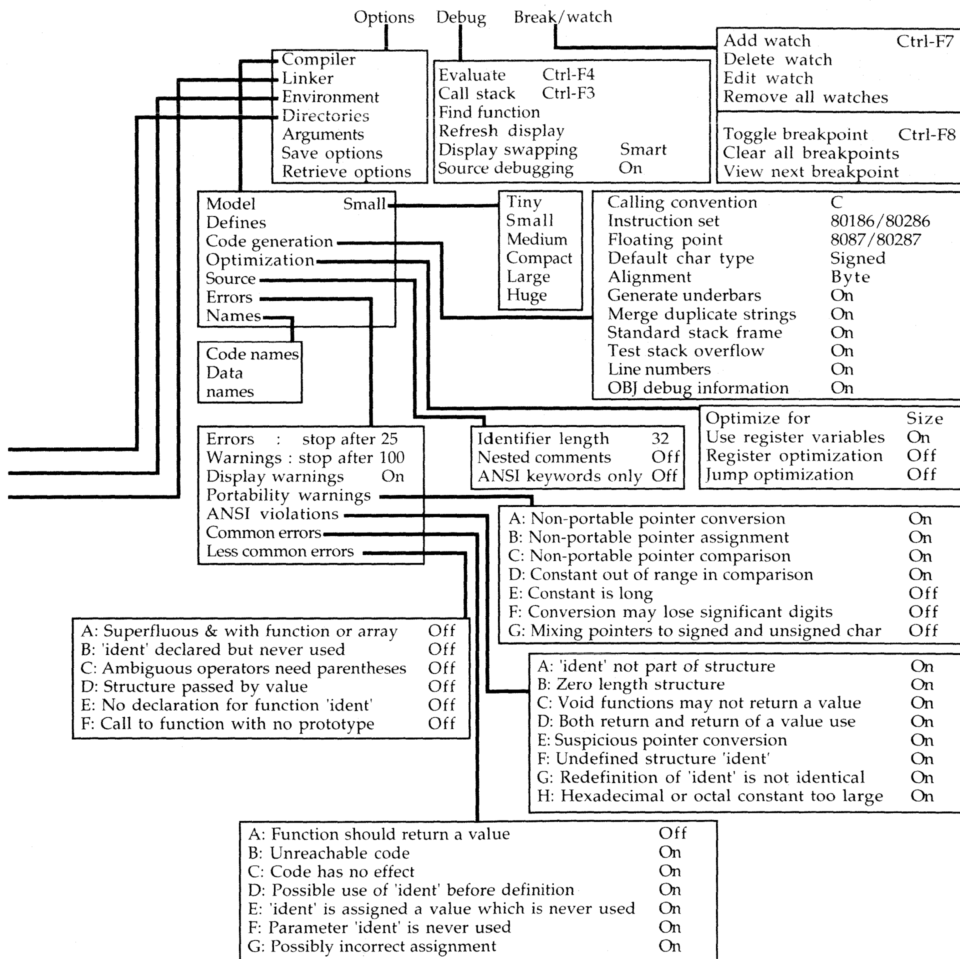


Figure 5.2: The TC Menu Structure, continued

Menu-Naming Conventions

In this book, we often refer to menu items by an abbreviated name. The abbreviated name for a given menu item is represented by the sequence of letters you press to choose that item from the main menu. For example:

- At the main menu, the menu offering compile-time options related to error messages is Options/Compiler/Errors; it may also be referred to as O/C/Errors (press *O C E*, in that order).
- At the main menu, the menu for specifying the name of the include directories is Options/Directories/Include Directories; it may be referred to as O/D/Include Directories (press *O D I*, in that order).

The Main Menu

```
File  Edit  Run  Compile  Project  Options  Debug  Break/watch
```

Figure 5.3: The TC Main Menu Bar

At the top of the main TC screen is the TC main menu bar (see Figure 5.3), which offers eight choices:

File Handles files (loading, saving, picking, creating, writing to disk), manipulates directories (listing, changing), quits the program, and invokes DOS.

Edit Lets you create and edit source files.

Run Controls a running program. If you have compiled and linked your program with the Debug/Source Debugging and O/C/C/OBJ Debug Information toggles set to On, you can also initiate a debugging session from this menu.

Compile Compiles and makes your programs into object and executable files.

Project Allows you to specify what files are in your program and manage your project.

Options Allows you to choose compiler options (such as memory model, compile-time options, diagnostics, and linker options) and define macros. Also records the Include, Output, and Library file directories, saves

compiler options, and loads options from the configuration file.

Debug Allows you to check or alter the value of variables, locate any function, and inspect the call stack while your program is running. Also lets you choose whether your program will compile with debugging information in the executable code.

Break/watch Lets you add, delete, and edit watch expressions and set, clear, and go to breakpoints.

Note that one main menu item is a command: Edit simply takes you into the Editor. The other menu items invoke pull-down menus with many options and/or subsequent menus.

The Quick-Ref Lines

Whether you're in one of the windows or one of the menus, the default *Quick-Ref Line* appears at the bottom of the screen. This line provides an at-a-glance function-key reference for your current position.

When you first enter TC, the default Quick-Ref Line looks like this:

```
F1-Help F5-Zoom F6-Switch F7-Trace F8-Step F9-Make F10-Menu
```

Now hold down the *Alt* key for a few seconds. The Quick-Ref Line will change to describe what function will be performed when you combine other keys with the *Alt* key. It looks like this:

```
Alt: F1-Last help F3-Pick F6-Swap F7/F8-Prev/Next error F9-Compile
```

The Edit Window

In this section, we describe the components of the main TC screen and explain how to work in the TC Edit window.

First, to get into the Edit window, press *F10* to invoke the main menu, then either move the highlight to the Edit option and press *Enter* or press *E* from anywhere in the main menu. To get into the Edit window from anywhere in the system, just press *Alt-E*. Once you're in the Edit window, notice that there are double lines at the top of it and its name is highlighted—that means it's the active window.

Besides the Edit window, where you can see and edit several lines of your source file, the TC screen has two information lines you should note: the *status line* and the Quick-Ref Line.

The status line at the top of the TC screen gives information about the file you are editing, where in the file the cursor is located, and which *editing modes* are activated. It looks like this:

```
Line  Col  Insert  Indent  Tab  Fill  Unindent  *  C:FILENAME.EXT
```

`Line n`	Cursor is on file line number *n*.
`Col n`	Cursor is on file column number *n*.
`Insert`	Insert mode is On; toggle Insert mode On and Off with *Insert* or *Ctrl-V*. See Appendix A in the *Turbo C Reference Guide* for an explanation of Insert and Overwrite modes.
`Indent`	Autoindent is On. Toggle it Off and On with *Ctrl-O I*. See Appendix A in the *Turbo C Reference Guide* for an explanation of Autoindent mode.
`Tab`	Tab mode is On. Toggle it On and Off with *Ctrl-O T*.
`Fill`	When Tab mode is on, the Editor will fill the beginning of each line optimally with tabs and spaces. This option is toggled with *Ctrl-O F*. See Appendix A in the *Turbo C Reference Guide*.
`Unindent`	The backspace will outdent a level whenever the cursor is on the first nonblank character of a line or on a blank line. This option is toggled with *Ctrl-O U*. See Appendix A in the *Turbo C Reference Guide*.
`*`	The asterisk appears before the file name whenever the file has been modified and has not yet been saved.
`C:FILENAME.EXT`	The drive (C:), name (FILENAME), and extension (.EXT) of the file you are editing.

The Quick-Ref Line at the bottom of the TC screen displays which hot key performs what action:

```
F1-Help F5-Zoom F6-Switch F7-Trace F8-Step F9-Make F10-Menu
```

To choose one of these functions, press the indicated function key:

F1	Opens a Help window that provides information about the TC editing commands.
F5	Expands the active window (in this case, the Edit window) to full screen. Press *F5* again to get back to the split-screen environment.
F6	Switches you from one active window to another (Edit vs Message/Watch).
`F7-Trace`	Lets you run your program one line at a time in debugging mode, tracing into functions as they are called.
`F8-Step`	Lets you run your program one line at a time in debugging mode, stepping over function calls.
`F9-Make`	Makes (compiles and links) your .EXE file.
`F10-Menu`	Takes you from the Edit window to the main menu, and from any menu to the Edit window.

The TC Editor uses a command structure similar to that of SideKick's Notepad and Turbo Pascal's editor; if you're unfamiliar with the editor these products use, Appendix A in the *Turbo C Reference Guide* describes the editing commands in detail. The most commonly used commands are listed below.

If you're entering code in the Edit window while you're in Insert mode, you can press *Enter* to end a line (the TC Editor has no wordwrap). The maximum line width is 248 characters; the Edit window is 77 columns wide. If you type past column 77, the window scrolls as you type. The TC screen's status line gives the cursor's location in the file by line and column.

After you've entered your code into the Edit window, press *F10* to invoke the main menu. Your file will remain onscreen; you need only press *E* (for Edit) at the main menu to return to it.

Quick Guide to Editing Commands

Here is a summary of the TC Editor commands you will use most often:

- Scroll the cursor through your text with the *Up/Down arrow, Left/Right arrow,* and *PgUp/PgDn* keys.
- Delete a line with *Ctrl-Y*.

- Delete a word with *Ctrl-T*.
- Mark a block with *Ctrl-K B* (beginning) and *Ctrl-K K* (end).
- Move a block with *Ctrl-K V*.
- Copy a block with *Ctrl-K C*.
- Delete a block with *Ctrl-K Y*.

See Appendix A in the *Turbo C Reference Guide* for a more detailed explanation of the Editor commands.

How to Work with Source Files in the Edit Window

When you invoke the Edit window before loading a particular file, the TC Editor automatically names the file NONAME.C. At this point you have all the features of the Editor at your fingertips. You can

- Create a new source file either as NONAME.C or another file name.
- Load and edit an existing file.
- Pick a file from a list of source files and load it into the Edit window.
- Save the file seen in the Edit window.
- Write the file in the Editor to a new file name.
- Alternate between the Edit window and the Message window for finding and correcting compile-time errors.

While you are creating or editing a source file, but before you have compiled it, you do not need the Message window. So you can press *F5* to zoom the Edit window to full screen. Press *F5* again to unzoom the Edit window (return to split-screen mode).

Creating a New Source File

To create a new file, choose either of the following methods:

- At the main menu, choose File/New, then press *Enter*. This opens the Edit window with a file named NONAME.C.
- At the main menu, choose File/Load. The Load File Name prompt box opens; type in the name of your new source file. (Pressing the hot key *F3* anywhere within TC will accomplish the same thing.)

Loading an Existing Source File

To load and edit an existing file, you can choose two options: **File/Load** or **File/Pick**.

If you choose **File/Load** at the main menu, you can

- Type in the name of the file you want to edit; paths are accepted—for example, C:\TURBOC\TESTFILE.C.
- Enter a mask in the Load File Name prompt box (using the DOS wildcards * and ?), and press *Enter*. Entering *.* will display all the files in the current directory as well as any other directories. Directory names are followed by a backslash (\). Choosing a directory displays the files in that directory. Entering C:*.C, for example, will bring up *only* the files with that extension in the root directory.

  Press the *Up/Down* and *Left/Right arrow* keys to highlight the file name you want to choose. Then press *Enter* to load the chosen file; you are placed in the Edit window.

If you choose **File/Pick** or press *Alt-F3* (see the discussion of the **File/Pick** menu later in this chapter), you can quickly pick the name of a previously loaded file.

There is an additional hot key to reload the previously loaded file. Press *Alt-F6* (change window contents) to switch between the file currently in the editor and the previously loaded file.

Saving a Source File

- From anywhere in the system, press *F2*.
- From the main menu, choose **File/Save**.

Writing an Output File

You can write the file in the Editor to a new file or overwrite an existing file. You can write to the current (default) directory or specify a different drive and directory.

At the main menu, choose **File/Write To**. Then, in the New Name prompt box, enter the full path name of the new file name; for example,

```
C:\DIR\SUBDIR\FILENAME.EXT
```

and press *Enter*.

If the file already exists, the Editor will verify that you want to overwrite the existing file before proceeding.

Press *Esc* to return to the active window (the Edit window). You can also press *Alt-E* or *F10*.

Note: For a comprehensive explanation of the TC Editor, refer to Appendix A in the *Turbo C Reference Guide.*

The Message Window

You will use the Message window to view diagnostic messages when you compile and debug your source files. TC's unique error-tracking feature lists each compiled file's warnings and error messages in the Message window and simultaneously highlights the corresponding position of the appropriate source file in the Edit window (depending upon the setting of the Message Tracking command on the Option/Environment menu).

When the cursor is in the Message window, the Quick-Ref Line hot keys perform the following functions:

`F1-Help`	Opens a Help window that summarizes the TC error-tracking feature.
`F5-Zoom`	Expands the Message window to full screen.
`F6-Switch`	Makes the Edit window the active window.
`F7-Trace`	Lets you run your program one line at a time in source debug mode, tracing into functions as they are called.
`F8-Step`	Lets you run your program one line at a time in source debug mode, stepping over function calls.
`F9-Make`	Makes the .EXE file.
`F10-Menu`	Takes you from the active window to the main menu, and from any menu to the active window.

The Watch Window

The Watch window replaces the Message window when you are running your program with the integrated debugger. It contains *watch expressions* (expressions you insert into the Watch window from your program) and the current value of each expression. A watch expression is reevaluated after each step or run, since its value may have changed. The Watch

window enables you to keep track of the value of important expressions while your program is running.

As you add expressions to the Watch window, the window expands until it reaches the size specified by the TCINST Resize Windows option. After that you can still add expressions, but you will have to scroll the window to see them all, using the *PgUp, PgDn, Up arrow,* and *Down arrow* keys.

The current expression in the Watch window is marked by a highlight bar when the window is active, and by a bullet in the left margin when it is not.

To edit expressions in the Watch window, you can generally use the same edit commands that you use in the Edit window. For example, *Ctrl-Y* deletes a watch expression, and *Ctrl-N* inserts a watch expression. The basic Watch window editing commands are listed in the following table.

Table 0.1: Watch Window Editing Commands

Key(s)	Function
Ctrl-E or *Up arrow*	Move cursor up
Ctrl-X or *Down arrow*	Move cursor down
Ctrl-S or *Left arrow*	Scroll watch expression right
Ctrl-D or *Right arrow*	Scroll watch expression left
Enter	Edit watch expression
Ctrl-N or *Ins*	Insert watch expression
Ctrl-Y, Del, or *Ctrl-G*	Delete watch expression

When the cursor is in the Watch window, the Quick-Ref Line hot keys perform the following functions:

F1 Opens a Help window.

F5 Expands the Watch window to full screen.

F6 Makes the Edit window the active window.

Ins Lets you add a watch expression to the Watch window.

Del Lets you delete a watch expression from the Watch window.

Enter Lets you edit the current watch expression in the Watch window.

The Turbo C integrated development environment includes a special built-in debugging feature called the integrated debugger to help you find errors ("bugs") in your programs. For a detailed description of how to use the integrated debugger, refer to Chapter 4. This chapter will introduce you to the menu features you need to run a debugging session.

The debugger operates by allowing you to stop your program at any point as it is executing, so you can check or even alter the value of variables.

Controlling the Debugger

The parts of the program you want to debug must be compiled with the O/C/C/OBJ Debug Information toggle and the Debug/Source Debugging toggle both set to On. The integrated environment then invokes the integrated debugger automatically when you run the program.

When you start a debugging session with Run/Run, Turbo C compiles the source file(s) (if necessary), links the program (if necessary), and prepares the program to run. Then it runs the program until it reaches either a breakpoint or the end of the program.

To start a debugging session when no breakpoints have been set, press *F8* (Run/Step Over). The debugger will stop on the declaration of the function **main**.

Once Turbo C has prepared the program to run, you are in a debugging session, and you can use any other feature of Turbo C.

You can run your program

- One line at a time, either skipping over function calls or stepping through the function.
- From your current position to a pre-established breakpoint.
- From your current position to wherever you have positioned the cursor.

You can use any of these methods or all of them, in combination, and in any order.

It is generally unwise to continue running the program after you have modified any of the source files you are debugging. Instead, recompile your program by choosing Compile/Make EXE File. In fact, if you have made changes to your source file, Turbo C will ask if you want to rebuild your .EXE file when you issue a run command like Step Over or Trace Into.

Once the rebuild has been made, TC will not ask you again until a further change has been made in your source files.

The Debugger Screen Display

The debugger screen display consists of the Edit window on top and the Watch window on the bottom. You can toggle between these windows by pressing *F6*.

As watch expressions are added to the Watch window, it grows to its maximum size (controlled by the TCINST utility's Resize Windows option), and then scrolls.

Your current position in the program is called the *execution position*. It is indicated in the Edit window by a highlight bar called the *execution bar*.

Debugging Menu Commands and Hot Keys

Table 5.3 shows the special debugging menu commands.

Table 5.3: Debugger Commands and Hot Keys

Hot Key	Menu Command	Description
F4	Run/Go to Cursor	Runs program, stopping on line at the cursor. Will initiate a debugging session.
Ctrl-F2	Run/Program Reset	Cancels current debugging session, releases allocated memory, and closes files. Valid only in debugging sessions.
F7	Run/Trace Into	Runs next statement in the current function. If it calls a lower-level function compiled with Debug/Source Debugging and O/C/C/OBJ Debug Information toggles set to On, traces into that function. Will initiate a debugging session.
F8	Run/Step Over	Runs next statement in the current function. Does not trace into called functions. Will initiate a debugging session.

Table 5.3: Debugger Commands and Hot Keys (continued)

Hot Key	Command	Description
	O/C/C/Standard Stack Frame	Toggles the Options/Compiler/Code Generation/Standard Stack Frame option. This option must be set to On when a program is compiled if the Debug/Call Stack option is to work correctly.
	O/C/C/OBJ Debug Information	Toggles the O/C/C/OBJ Debug Information option. Only source files compiled and linked with this option set to On can be debugged.
Ctrl-F4	Debug/Evaluate	Evaluates a C expression; allows you to modify the value of a variable.
	Debug/Find Function	Finds a function's definition and displays it in the Edit window. Valid only in debugging sessions.
Ctrl-F3	Debug/Call Stack	Displays call stack. You can display the currently executing line of a function by choosing that function's name from the call stack. Valid only in debugging sessions.
	Debug/Source Debugging	Controls whether debugging is allowed. When it is set to On, both integrated and standalone debugging are possible. When it is set to Standalone, you can debug programs only with the standalone debugger, although you can still run them in TC. When it is set to None, no debugging information is placed in the .EXE file, so you cannot debug your program with either debugger.
Ctrl-F7	Break/Watch/Add Watch	Adds a watch expression.
	Break/Watch/Delete Watch	Deletes a watch expression.
	Break/Watch/Edit Watch	Lets you edit a watch expression.
	Break/Watch/Remove All Watches	Deletes all watch expressions.
Ctrl-F8	Break/Watch/Toggle Breakpoint	Sets or clears a breakpoint on the line at the cursor position.
	Break/Watch/Clear Breakpoints	Clears all breakpoints in the program.
	Break/Watch/View Next Breakpoint	Displays next breakpoint.

Table 5.4 shows other menu commands that are often used when you are running the debugger.

Table 5.4: Menu Commands and Hot Keys Used with the Debugger

Hot Key	Menu Command	Description
F5		Zooms and unzooms the active window between full-screen and split-screen modes.
Alt-F5		Switches the display to the User screen. Press any key to return to the integrated environment.
F6		Switches active window between the Edit window and the Watch or Message window.
Alt-F6		If Edit window is active, switches to the last file loaded into the Editor. If lower window is active, switches between Watch window and Message window.
Ctrl-F9	Run/Run	Runs a program, with or without the debugger. Compiles source file(s) and links program if necessary. When the program has been compiled and linked with Debug/Source Debugging and O/C/C/OBJ Debug Information set to On, runs program to a breakpoint or to the end of the program.
	Project/Remove Messages	Deletes contents of the Message window.

Part II: The Menu Commands

The main menu contains the major choices you'll use to load, edit, compile, link, debug, and run Turbo C programs. The eight menu choices include File, Edit, Run, Compile, Project, Options, Debug, and Break/Watch, each of which will be described here. A few of the options within the main menu pull-downs are actually for use in advanced programming; they are described in more detail in Chapter 3.

Note: The references to "make" in this chapter refer to Project-Make, not to the stand-alone MAKE utility. Project-Make is a program building tool similar to MAKE; refer to Chapter 3 for more on Project-Make. The MAKE utility is described in Appendix D in the *Turbo C Reference Guide*.

The File Menu

The File pull-down menu offers various choices for loading existing files, creating new files, and saving files. When you load a file, it is placed in the Editor. When you finish with a file, you can save it to any directory or file name. In addition, from the File menu you can change to another directory, temporarily go to a DOS shell, or exit Turbo C.

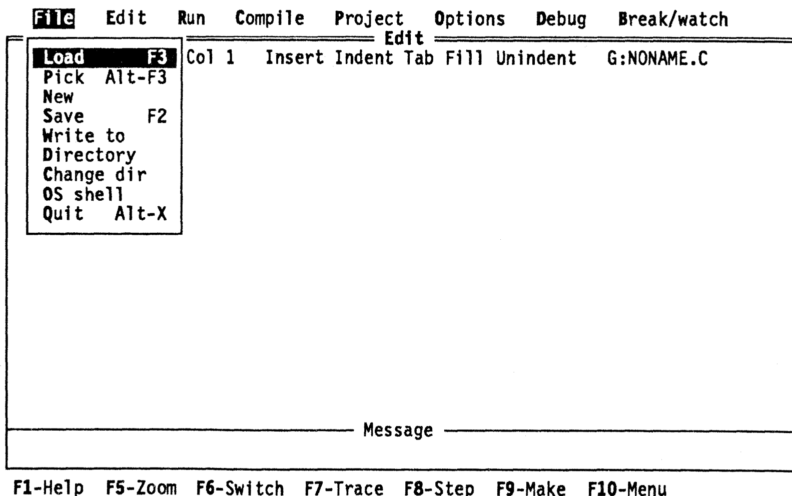


Figure 5.4: The File Menu

Load

Loads a file. You can use DOS-style masks (for example, `*.C`) to get a listing of file choices, or you can load a specific file. Simply type in the name of the file you want to load.

Note: If you enter an incorrect drive or directory, you'll get an error box onscreen. You'll get a verify box if you have an unsaved, modified file in the Editor while you're trying to load another file. In either case, the hot keys are disabled until you press the key specified in the error or verify box.

Pick

Lets you pick a file from a *pick list* of the previous eight files loaded into the Edit window. The file chosen is then loaded into the Editor and the cursor is positioned at the location where you last edited that file. If you choose the "`--load file--`" item from the pick list, you'll get a Load File Name prompt box exactly as if you had chosen **File/Load** or pressed *F3*. *Alt-F3* is a shortcut for the **File/Pick** command. The integrated environment can save this list of file names from one editing session to another, if you create a *pick file* to hold it.

See the section on the **Options/Directories/Pick File Name** command (page 132) for details on how to create a pick file.

New

Specifies that the file is to be a new one. You are placed in the Editor; by default, this file is called NONAME.C. (You can change this name later when you save the file.)

Save

Saves the file in the Editor to disk. If your file is named NONAME.C and you go to save it, the Editor will ask if you want to rename it. From anywhere in the system, pressing the *F2* hot key will save your file.

Write To

Prompts for a file name and writes the contents of the Editor to that file. If a file by that name already exists, this command causes it to be overwritten.

Directory

Displays the directory and file set you want (to get the current directory, just press *Enter*). *F4* allows you to change the wildcard mask. Choose a file name to load that file into the Editor.

Change Dir

Displays the current directory and allows you to change to a specified drive and directory.

OS Shell

Leaves Turbo C temporarily and takes you to the DOS prompt. To return to Turbo C, type EXIT. This is useful when you want to run a DOS command without quitting Turbo C.

Note: In dual monitor mode, the DOS shell will come up on the TC screen rather than the User screen. This allows you to shell to DOS without disturbing the output of your program. Since your program output is available on one monitor in the system, **Run/User** Screen and *Alt-F5* will be disabled.

Quit

Quits Turbo C and returns you to the DOS prompt.

The hot key for this command is *Alt-X*.

The Edit Command

The **Edit** command invokes the built-in screen Editor.

You can invoke the main menu from the Editor by pressing *F10* (or *Alt* and the first letter of the main menu command you desire). Your source text remains displayed on the screen; you need only press *Esc* or *E* at the main menu to return to it (or press *Alt-E* from anywhere).

The Run Menu

The **Run** menu's commands run your program, and also start and end debugging sessions. In order to use any of the Run commands except **Run/Run**, you must have compiled and linked your program with the Debug/Source Debugging toggle set to On.

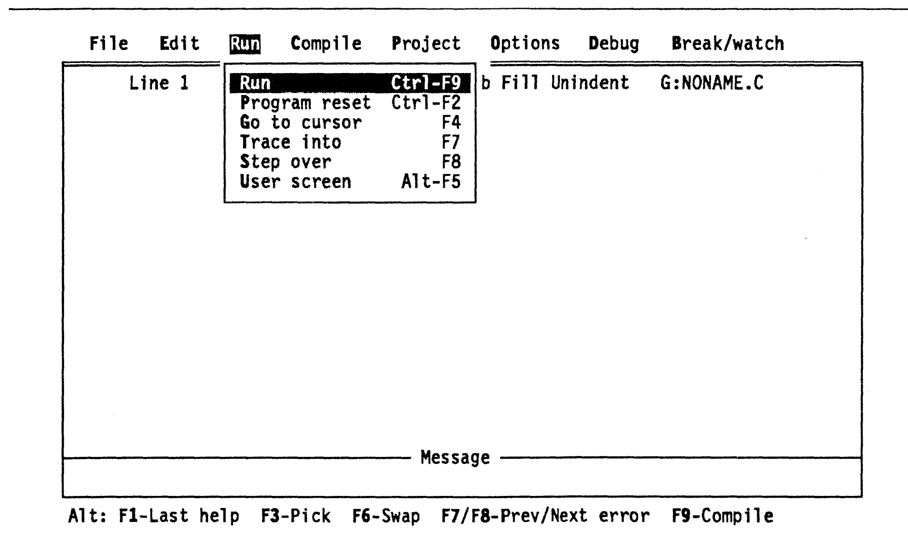


Figure 5.5: The Run Menu

Run

Run/Run runs your program, using the arguments you pass to it with **Options/Arguments**. If the source code has been modified since the last compilation, it will also invoke Project-Make to recompile and link your program. (Project-Make is a program building tool incorporated into the integrated environment; see Chapter 3 for more on this feature.)

If you don't want to debug your program, compile and link it with the Debug/Source Debugging toggle set to None or to Standalone. If you compile your program with this toggle set to On, the resulting executable code will contain debugging information that will affect the behavior of the Run/Run command in the following ways:

If you have *not* modified your source code since the last compilation:

- The Run/Run command will cause your program to run to the next breakpoint, or to the end if no breakpoints have been set.

If you have modified your source code since the last compilation:

- If you are already stepping through your program using Run/Step Over (*F8*) or Run/Trace Into (*F7*), Run/Run will cause a prompt to appear onscreen asking whether you want to rebuild your program.
  - If you press *Y*, Project-Make will recompile and link your program, and set it to run from the beginning.
  - If you press *N*, your program will run to the next breakpoint, or to the end if no breakpoints are set.
- If you are not yet stepping through your program, Project-Make will recompile your program and set it to run from the beginning.

The hot key for the Run/Run command is *Ctrl-F9*.

Program Reset

Run/Program Reset cancels the current debugging session. It releases memory your program has allocated, and closes any open files. The hot key for Run/Program Reset is *Ctrl-F2*.

Go to Cursor

Run/Go to Cursor runs your program from the execution bar to the line the Edit window cursor is on. If the cursor is at a line that does not contain an executable statement, the command warns you by bringing up an *Esc* box. Run/Go to cursor can also initiate a debug session.

Go to Cursor does not set a permanent breakpoint, but does allow the program to stop at a permanent breakpoint if it encounters one before the line the cursor is on. If this occurs, you must reissue the Go to Cursor command.

Use **Go to Cursor** to advance the execution bar to the part of your program you want to debug. If you want your program to stop at a certain statement every time it reaches that point, set a breakpoint on that line.

The hot key for **Run/Go to Cursor** is *F4*.

Trace Into

Run/Trace Into runs the next statement in the current function. If the statement contains no calls to functions accessible to the debugger, **Trace Into** halts at the next executable statement.

If the statement does contain a call to a function accessible to the debugger, **Trace Into** halts at the beginning of the function's definition. Subsequent **Trace Into** or **Step Over** commands will run the statements in the function's definition. When the debugger leaves the function, it will resume evaluating the statement that contains the call.

A function is accessible to the debugger if it is defined in a source file that was compiled with both the **O/C/C/OBJ Debug Information** and the **Debug/Source Debugging** toggles set to On, and the debugger can find the source file on your disk.

Use **Trace Into** to move the execution position into a function called by the function you are now debugging.

The hot key for the **Run/Trace Into** command is *F7*.

Step Over

Run/Step Over executes the next statement in the current function. It does not trace into calls to lower-level functions, even if they are accessible to the debugger.

Use **Step Over** to run the function you are now debugging, one statement at a time.

Here is an example of the difference between **Run/Trace into** and **Run/Step over**. These are the first twelve lines of a program loaded into the Editor:

```
int findit(void)                                        /* Line 1 */
{
 return(2);
}
```

```
void main(void)                                    /* Line 6 */
{
 int i, j;

 i = findit();                                     /* Line 10 */
 printf("%d\n", i);                                /* Line 11 */
 j = 0; . . .                                      /* Line 12 */
```

findit is a user-defined function in a module that has been compiled with debugging information. Let's say that the execution bar is on Line 10 of your program.

- If you now select **Run/Trace into**, the execution bar will move to the first line of the **findit** function (Line 1 of your program), allowing you to step through the function.
- If you select **Run/Step over**, the **findit** function will execute and the return value will be assigned to *i*. Then the execution bar will move to Line 11.

If the execution bar had been on Line 11 of your program, it would have made no difference which command you selected; **Run/Trace into** and **Run/Step over** would both have executed the **printf** function and moved the execution bar to Line 12. This is because the **printf** function does not contain debug information.

The hot key for the **Run/Step Over** command is *F8*.

The Compile Menu

You use the items on the Compile menu to compile to an .OBJ file (Compile to OBJ), to make an .EXE file (Make EXE File), to link an .EXE file (Link EXE File), to Build All, to set a Primary C File, and to get information about the last compilation or run (Get Info).

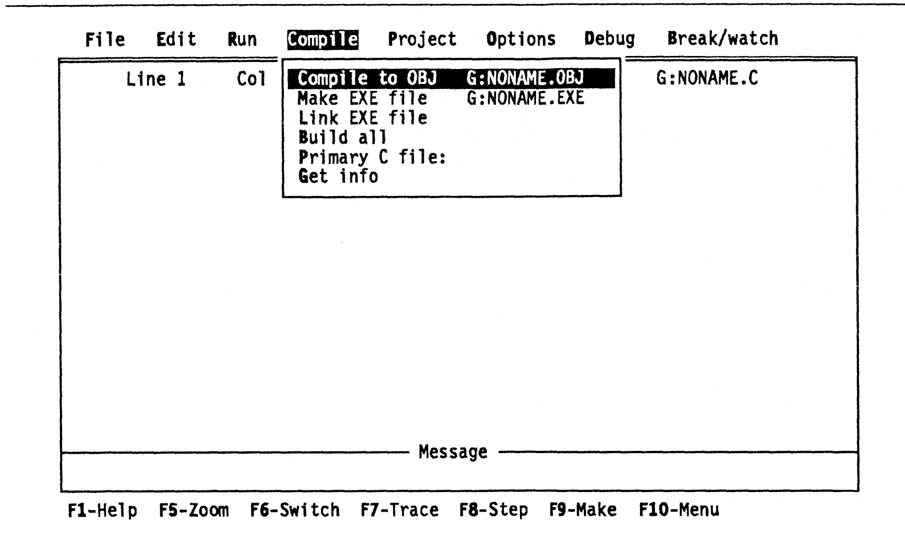


Figure 5.6: The Compile Menu

Compile to OBJ

This command compiles a .C file to an .OBJ file. It always displays the name of the file to be produced; for example, `C:EXAMPLE.OBJ`. The .OBJ file name listed is derived from one of two names, in the following order:

- the primary .C file name, or, if none is specified,
- the name of the last file you loaded into the Edit window

When Turbo C is compiling, a window pops up to display the compilation results. When compiling/making is complete, press any key to remove this compiling window. If any errors occurred, you are automatically placed in the Message window at the first error (which is highlighted). This Compile command and its options are explained in more detail in Chapter 3.

The hot key for this command is *Alt-F9*.

Make EXE File

This command invokes Project-Make to make an .EXE file. It always displays the name of the .EXE file to be produced; for example,

C:EXAMPLE.EXE.

The .EXE file name listed is derived from one of three names in the following order:

- the project file (.PRJ) specified with Project/Project Name, or, if none is specified,
- the primary C file name specified with Primary C File, or, if none is specified,
- the name of the last file you loaded into the Edit window.

The hot key for this command is *F9*.

Link EXE File

Takes the current .OBJ and .LIB files (either the defaults or those defined in the current project file) and links them without doing a make; this produces a new .EXE file.

Build All

Rebuilds all the files in your project regardless of whether they are out of date. This option is similar to Compile/Make EXE File, except that it is unconditional; Compile/Make EXE File rebuilds only the files that aren't current. This command first sets the date and time of all the project's .OBJ files to zero, then does a make. (Thus, if you break a Build All command with *Ctrl-Break*, you can cause it to pick up where it left off simply by choosing C/Make EXE File.)

Primary C File

The Primary C File command is useful (but not required) when you're compiling a single .C file that includes multiple header (.H) files. If an error is found during compilation, the file containing the error (which might be a .C file or a .H file) is automatically loaded into the Editor so you can correct it. (Note that the .H file is *only* automatically loaded if you have changed the default setting of Options/Environment/Message Tracking to All Files; using the default settings will not cause automatic loading of the .H file.) The primary .C file is then recompiled when you press *Alt-F9*, even if it is not in the Editor.

Get Info

Compile/Get Info calls up a window that gives you information on

- primary file
- object file name associated with the current file
- name of current source file
- size in bytes of current source file
- program exit code
- available memory

```
  File   Edit   Run   Compile   Project   Options   Debug   Break/watch
         Line 1     Col  | Compile to OBJ   G:HELLO.OBJ  |  * G:HELLO.C
/*                ═══════════ Information ═══════════
#include    Current directory : G:\PUBLIC\NETFILES\C\C2\
            Current file      : G:\PUBLIC\NETFILES\C\C2\HELLO.C
main()      File size         : 104 (Max: 64605)
{           EMS usage         : OK

}           Lines compiled: 0            No program running.
            Total warnings: 0            Program exit code
            Total errors  : 0            Available memory: 112K
                             Press any key

──────────────────────────── Message ────────────────────────────

F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu
```

Figure 5.7: The Compile/Get Info screen

The Project Menu

The commands on the Project menu allow you to combine multiple source and object files to create finished programs.

For more information on Project, refer to Chapter 3.

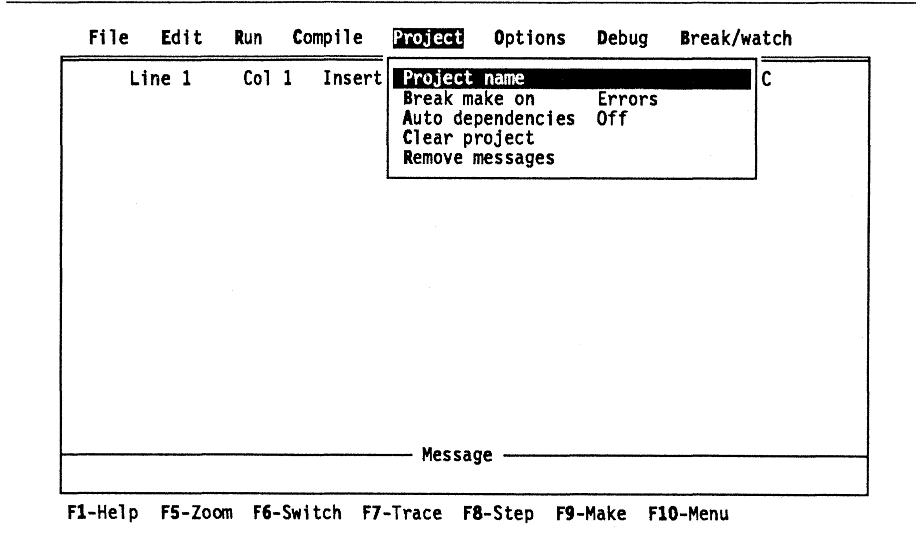


Figure 5.8: The Project Menu

Project Name

Chooses a project file containing the names of the files to be compiled and/or linked. The project name is given to the .EXE and .MAP files when they are created. A typical project file has the extension .PRJ.

Break Make On

This menu lets you specify the default condition for stopping a make: if the file has Warnings, Errors, or Fatal Errors, or before linking (Link).

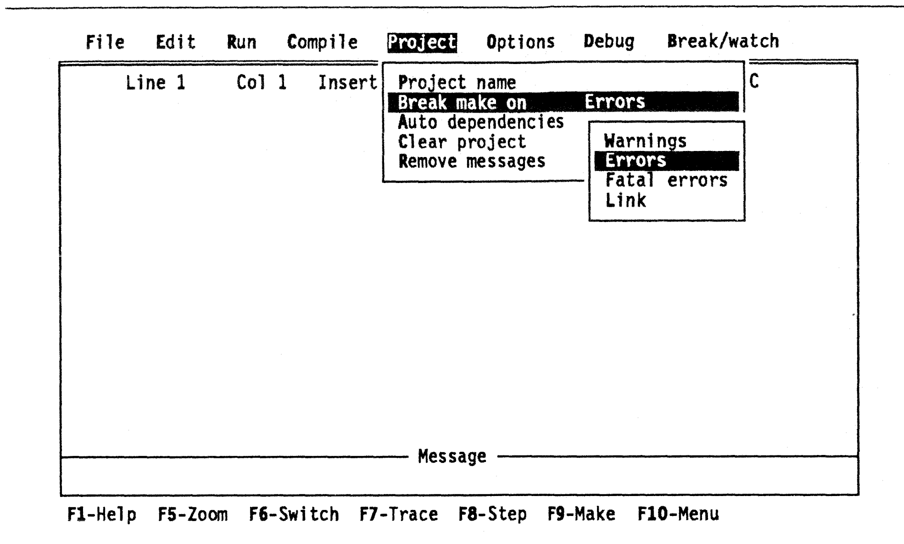


Figure 5.9: The Project/Break Make On Menu

Auto Dependencies

This option is a toggle. If you set it to On, Project-Make will automatically check dependencies for every .OBJ file on disk that has a corresponding .C source file in the project list.

Project-Make opens the .OBJ file and looks for information about files included in the source code. This information is always placed in the .OBJ file by both TC and TCC when the source module is compiled. Then every file that was used to build the .OBJ file is checked for time and date against the time/date information in the .OBJ file. The .C source file is recompiled if the dates are different.

This is called an *autodependency check.*

If the Auto Dependencies option is toggled to Off, no such file checking will be done.

Clear Project

This command clears the project name and resets the Message window.

Remove Messages

This command clears the error messages from the Message window.

The Options Menu

The Options menu contains settings that determine how the integrated environment works. These settings affect things like compiler and linker options, library and include directories, program run-time arguments, and so on. The items on this menu call up more menus, one setting, and two commands that perform managerial tasks, as follows:

- Compiler (calls up more menus)
- Linker (calls up more menus)
- Environment (calls up more menus)
- Directories (calls up more menus)
- Arguments (setting)
- Save Options (performs task)
- Retrieve Options (performs task)

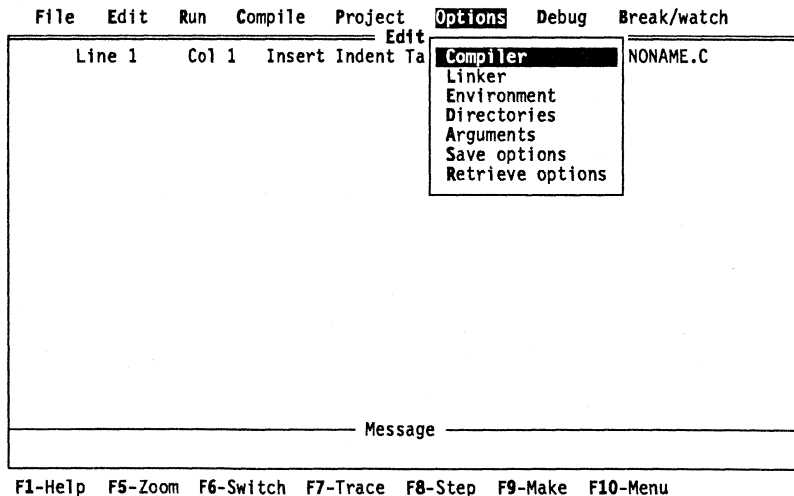


Figure 5.10: The Options Menu

Compiler

The options on this menu allow you to specify particular hardware configurations, memory models, debug techniques, code optimizations, diagnostic message control, and macro definitions. The items in this menu, described in the next several pages, are as follows:

- Model
- Defines
- Code Generation
- Optimization
- Source
- Errors
- Names

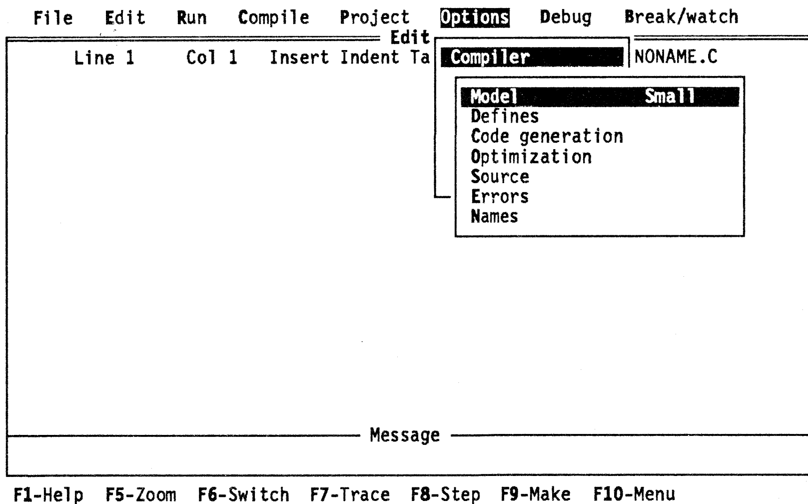


Figure 5.11: The Options/Compiler Menu

The Model menu

These commands are the different memory model switches available in Turbo C. The memory model chosen determines the default method of memory addressing. The options are **Tiny**, **Small**, **Compact**, **Medium**,

Large, and Huge. The default memory model is Small, so normally the word "Small" appears to the right of the menu choice Model. Refer to Chapter 12 for more information about these memory models.

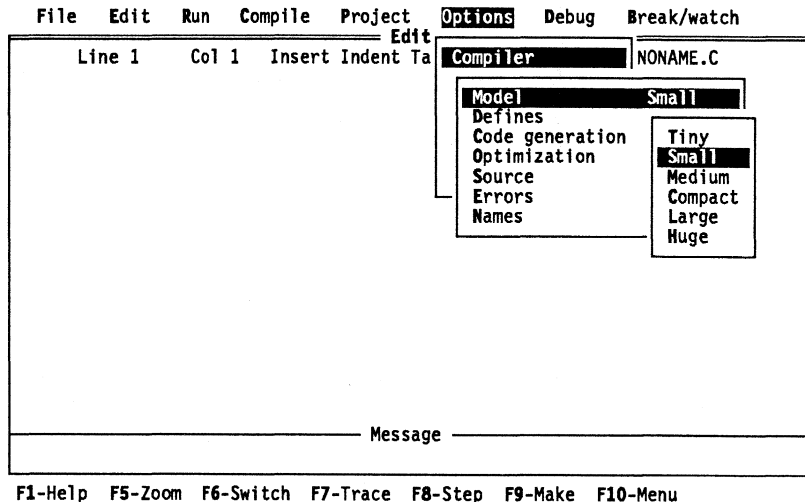


Figure 5.12: The O/C/Model Menu

Defines

Choosing Defines opens up a macro definition box in which you can pass macro definitions to the preprocessor. Multiple "defines" can be separated by semicolons (;). Values can be assigned optionally with an equal sign (=).

Leading and trailing spaces are stripped, but embedded spaces are left intact. If you want to include a semicolon in a macro, you must place a backslash (\) in front of it.

Here's a macro that defines the symbol *BETA_TEST*, sets *ONE* to 1, and *COMPILER* equal to the string *TURBOC*:

```
BETA_TEST; ONE = 1; COMPILER = TURBOC
```

The Code Generation Menu

These options tell the compiler to prepare the object code in various ways.

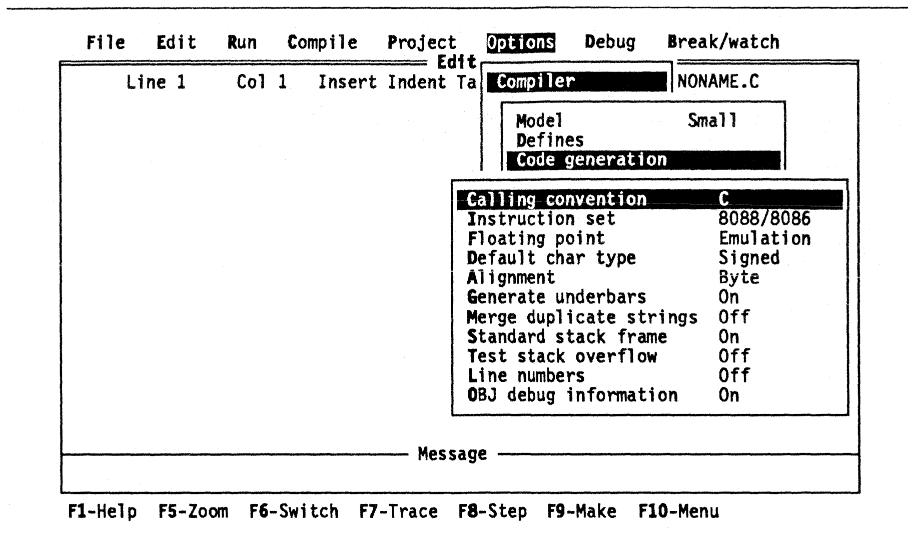


Figure 5.13: The O/C/Code Generation Menu

Calling Convention:

Causes the compiler to generate either a C calling sequence or a Pascal (fast) calling sequence for function calls. The differences between C and Pascal calling conventions are in the way each handles stack cleanup, number and order of parameters, case and prefix (underbar) of external identifiers.

Do not change this option unless you're an expert and have read Chapter 12 on advanced programming techniques.

Instruction Set:

Permits you to specify a different target CPU; this is a toggle between an 8088/8086 instruction set and an 80x86 instruction set. The default generates 80x86 code. Turbo C can generate extended 80x86 instructions. You will also use this option to generate 80x86 programs running in the real mode, such as with the IBM PC AT under MS-DOS 3.x.

Floating Point:
This toggle allows for three options:

- 8087/80287, which generates direct 8087/80287 inline code.
- Emulation, which detects whether you have an 8087/80287 and uses it if you do—otherwise, it emulates the 8087/80287 just as accurately, but at a slower pace.
- None, which assumes you're not using floating point. (If None is chosen and you use floating-point calculations in your program, you will get link errors.)

Default Char Type:

Toggles between Signed and Unsigned **char** declarations. If you choose Signed, the compiler will treat all **char** declarations as if they were **signed char** type; and vice versa for choosing Unsigned. The default value is Signed.

Alignment:

This allows you to toggle between word-aligning and byte-aligning. With word-aligning, noncharacter data aligns at even addresses. With byte-aligning, data can be aligned at either odd or even addresses, depending on which is the next available address. Word-alignment increases the speed with which 8086 and 80286 processors fetch and store the data.

Generate Underbars:

By default, this option is toggled to On.

Don't change this unless you're an expert and have read Chapter 12 on advanced programming techniques.

Merge Duplicate Strings:

This optimization merges strings when one string matches another, producing smaller programs. The default is Off.

Standard Stack Frame:

Generates a standard stack frame (standard function entry and exit code). This is helpful when you use a debugger—it simplifies the process of tracing back through the stack of called subroutines. The default is On.

The Standard Stack Frame option is a toggle. If a source file is compiled with this option set to Off, any function that does not use local variables and has no parameters is compiled with abbreviated entry and return code. This makes the code shorter and faster, but prevents Debug/Call Stack

from "seeing" the function. Thus, the toggle should always be set to On when a source file is compiled for debugging.

Test Stack Overflow:

Generates code to check for a stack overflow at run time. Although this costs space and time in a program, it can be a real lifesaver; a stack overflow can be a difficult bug to track down. The default is Off.

Line Numbers:

Includes line numbers in the object map file (for use by a symbolic debugger). This increases the size of the object and map files but will not affect speed of the executable program. (The size of the executable program will increase if the **Debug/Source Debugging** toggle is set to On, and you are linking in object files that were created with the **O/C/C/Line** numbers switch toggled on; the additional size is due to the debugging information.) The default is Off.

Since the compiler may group together common code from multiple lines of source text during jump optimization, or may reorder lines (which makes line-number tracking difficult), we recommend setting **Options/Compiler/Optimization/Jump Optimization** to Off when this option is On.

OBJ Debug Information:

Controls whether debugging information is included in object (.OBJ) files. This toggle defaults to On, which allows both integrated debugging and debugging with the standalone Turbo Debugger.

The Optimization Menu

The options in this menu allow you to optimize your code to your own programming needs.

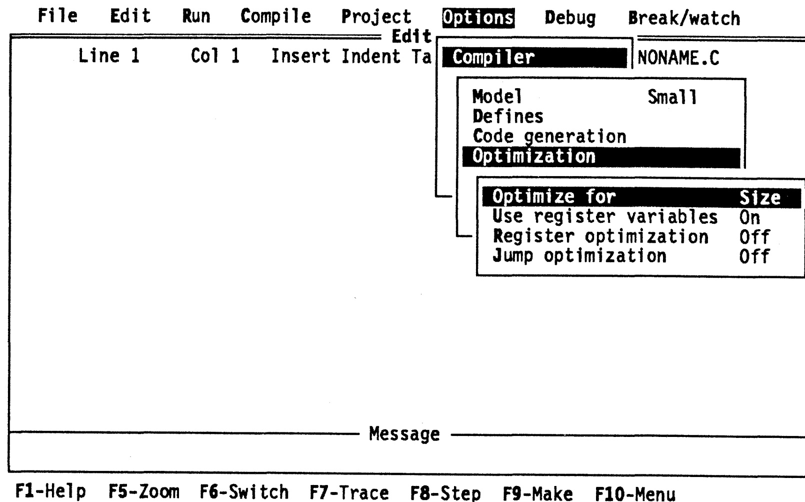


Figure 5.14: The O/C/Optimization Option

Optimize For:

Changes Turbo C's code generation strategy. Normally the compiler uses **O**ptimize for...Size, choosing the smallest code sequence possible. With this item toggled to **O**ptimize for...Speed, the compiler will choose the *fastest* sequence for a given task.

Use Register Variables:

Suppresses or enables the use of register variables. With this option set to On, register variables are automatically assigned for you. With this option set to Off, the compiler does not use register variables even if you have used the **register** keyword (see Appendix C in the *Turbo C Reference Guide* for more details).

Generally, you can keep this option set to On unless you are interfacing with preexisting assembly code that does not support register variables.

Register Optimization:

Suppresses redundant load operations by remembering the contents of registers and reusing them as often as possible.

Note: You should exercise caution when using this option because the compiler cannot detect whether a value has been modified indirectly by a pointer. Refer to Appendix C in the *Turbo C Reference Guide* for a detailed explanation of this limitation.

Jump Optimization:

Reduces the code size by eliminating redundant jumps and reorganizing loops and switch statements. The loop reorganizations can speed up tight inner loops.

Note: When this switch is set to On, the sequences of tracing and stepping in the integrated debugger can be confusing, since there may be multiple lines of source code associated with a particular generated code sequence. For best results, turn this switch Off while you are debugging.

The Source Menu

The items on this menu govern how the compiler treats your source code during the initial phases of the compilation.

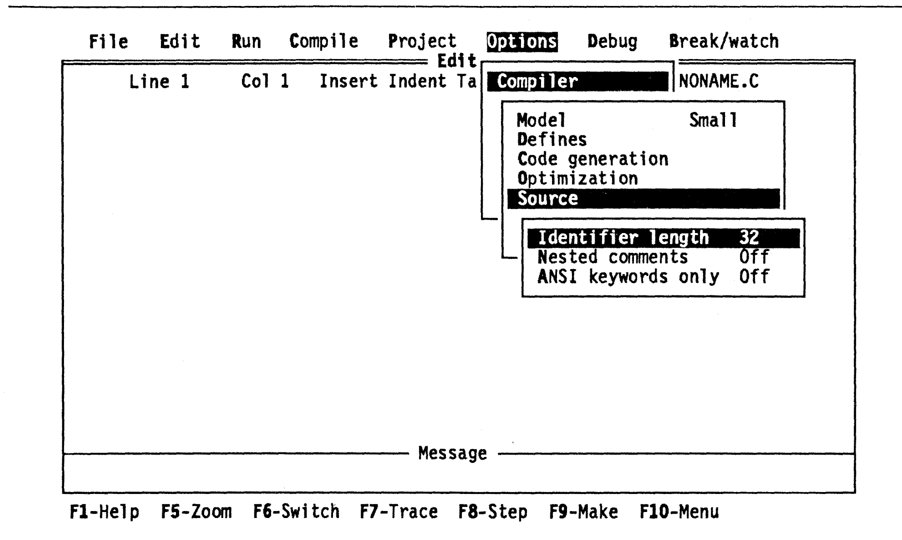


Figure 5.15: The O/C/Source Menu

Identifier Length:

Specifies the number of significant characters in an identifier. All identifiers are treated as distinct only if their first *N* characters are distinct. This includes variables, preprocessor macro names, and structure member names. The number given can be any value from 1 to 32; the default is 32.

Nested Comments:

Allows you to nest comments in Turbo C source files. Nested comments are not normally allowed in C implementations, and they are not portable.

ANSI Keywords Only:

Toggle to On when you want the compiler to recognize only ANSI keywords and treat any Turbo C extension keywords as normal identifiers. These keywords include **near**, **far**, **huge**, **asm**, **cdecl**, **pascal**, **interrupt**, **_es**, **_ds**, **_cs**, **_ss**, and the register pseudo-variables (_AX, _BX, ...). This option also defines the symbol *__STDC__* during compiles.

The Errors Menu

With the commands on this menu, you govern how the Turbo C compiler deals with and responds to diagnostic messages.

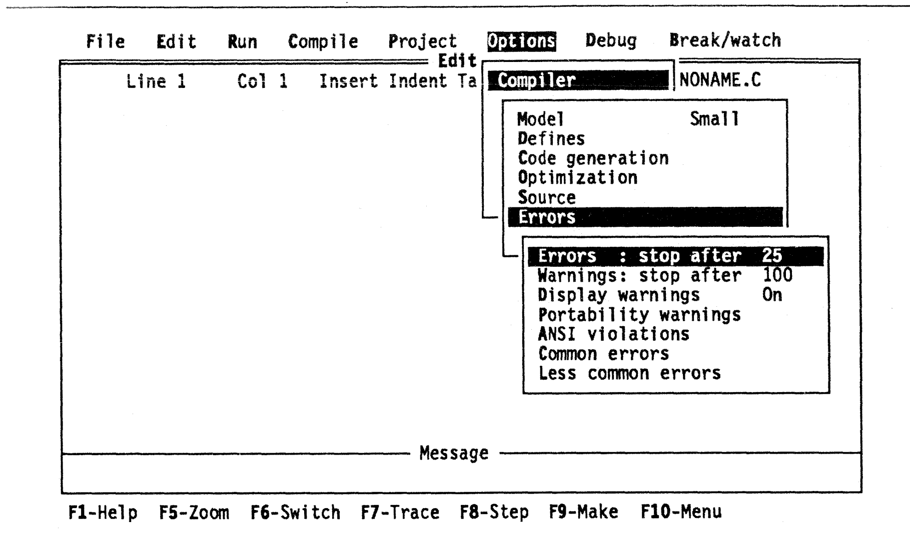


Figure 5.16: The O/C/Errors Menu

Errors: Stop After:

This option causes compilation to stop after a specified number of errors have been detected. The default is 25; however, you can enter any number from 0 to 255. (Entering 0 will cause compilation to continue indefinitely.)

Warnings: Stop After:

Choosing this option causes the compilation to stop after 100 warnings have been detected. However, 100 is only the default; the legal range is 0 to 255, where entering 0 will cause compilation to continue indefinitely or until the error limit has been reached.

Display Warnings:

By default, this is set to On, which means that any or all of the following warning types can be displayed if chosen:

- Portability Warnings
- ANSI Violations
- Common Errors
- Less Common Errors

When this item is set to Off, none of the warnings will be displayed. These warning messages are discussed in more detail in Appendixes B and C in the *Turbo C Reference Guide.*

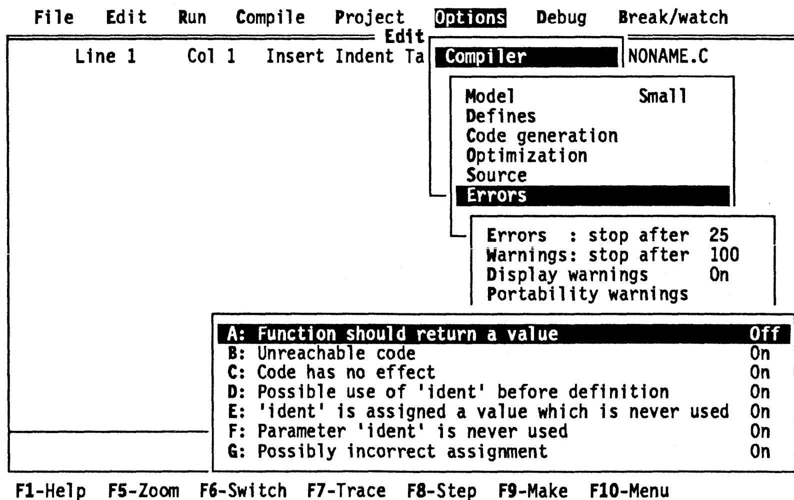


Figure 5.17: Displaying the Common Errors

The Names Menu

With the items in this menu, you can change the default segment, group, and class names for code, data, and BSS sections.

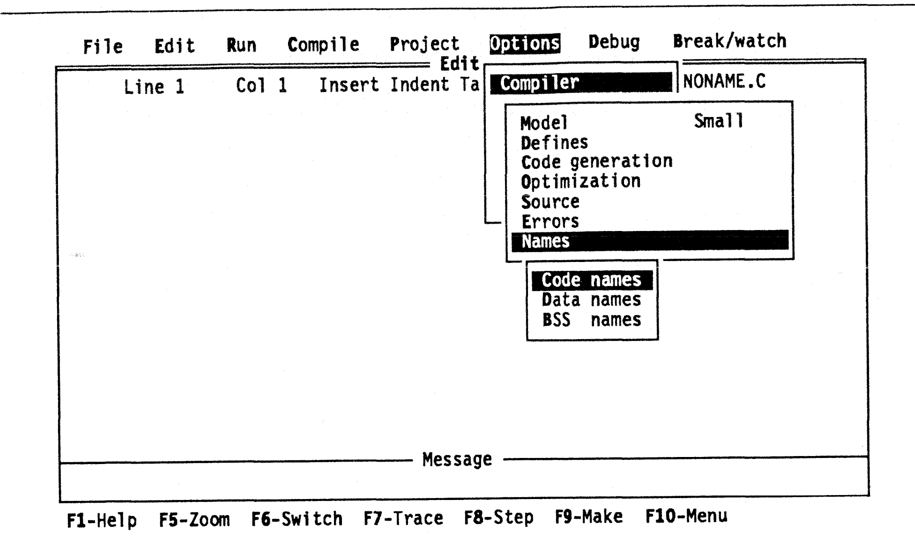


Figure 5.18: The O/C/Names Option

When you choose one of these items, the asterisk (*) on the next menu that appears tells the compiler to use the default names.

Don't change this option unless you are an expert and have read Chapter 12 on advanced programming techniques.

Linker

The items on this menu deal with setting options for the linker. Refer to Appendix D in the *Turbo C Reference Guide* for more information about these settings.

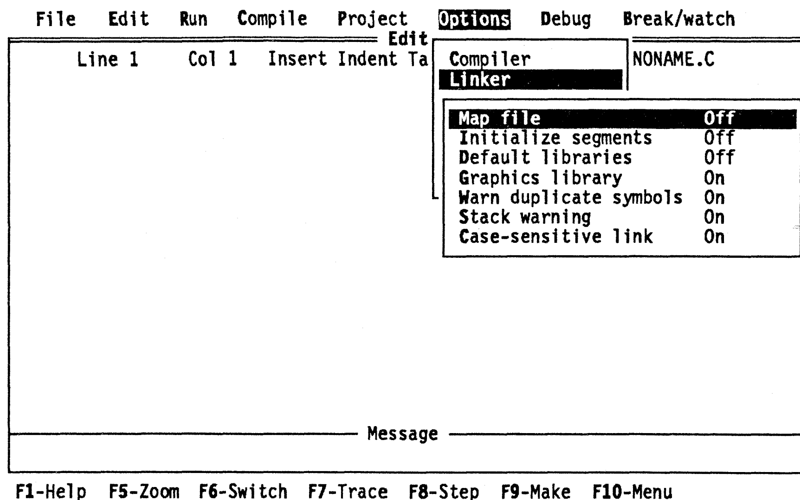


Figure 5.19: The Options/Linker Menu

The Map File Menu

Chooses the type of map file to be produced. For values other than Off, the map file is placed in the output directory defined with Options/ Directories/Output Directory. By default, this is set to the Off option; your other choices are Segments, Publics, and Detailed.

Initialize Segments

Tells the linker to initialize uninitialized segments. (This is normally not needed, and will make your .EXE files larger than necessary.)

Default Libraries

When you're linking with modules created by a compiler other than Turbo C, the other compiler may have placed a list of default libraries in the object file.

If this option is set to On, the linker will try to find any undefined routines in these libraries as well as in the default libraries supplied by Turbo C.

If this option is set to Off, only the default libraries supplied by Turbo C will be searched; any defaults in .OBJ files will be ignored.

Graphics Libraries

Turns on and off the Automatic searching of the BGI graphics library. When this toggle is set to On, it is possible for you to build and run single-file graphics programs without using a project file. Turning the toggle Off speeds up the link step, because the linker does not have to link in the BGI graphics library file. The default is On

Note: You can set this toggle to Off and still build programs that use BGI graphics, provided you name the BGI graphics library in your project file.

Warn Duplicate Symbols

Turns On and Off the linker warning for duplicate symbols in object and library files. The default is Off.

Stack Warning

Disables the `No stack` message generated by the linker. (It is normal for a program generated under the tiny model to generate this message if the message is not disabled.)

Case-sensitive Link

Turns case sensitivity On and Off during linking. Normally, this option will be set to On, since C is a case-sensitive language.

Environment

This menu's entries let you automatically back up your source file in the Editor and tailor the Turbo C working environment to suit your programming needs.

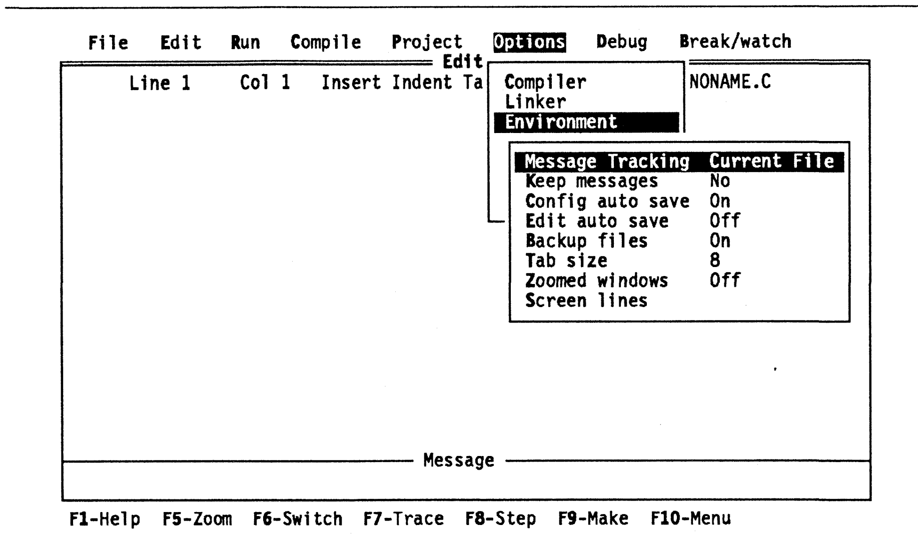


Figure 5.20: The Options/Environment Menu

Message Tracking

Turbo C will track syntax errors in the Editor when you scroll through the error messages in the Message window. This three-way toggle tells Turbo C which files to track in.

The default (Track...Current file) will track errors only in the file currently in the Editor. Track...All Files will load and track in every file for which there is a message. You can also turn tracking Off.

Keep Messages

This is a toggle; when it is set to On, Turbo C saves the error messages currently in the Message window, appending any messages from further compiles to the window. When a file is compiled, any messages for that file are removed from the Message window and new messages are added to the end. When this toggle is set to Off, messages are automatically cleared before a compile or make.

Config Auto Save

Normally, Turbo C saves the current configuration (writes it to disk) only when you choose the Options/Save Options command. With Config Auto Save toggled to On, Turbo C also saves the file whenever you choose Run/Run or File/OS Shell, or when you exit the integrated environment, if the configuration file has never been saved or has been at all modified since it was last saved.

With Config Auto Save set to On, if the configuration file has not yet been saved, Turbo C chooses a file name for the automatically saved file. This is the name of the last configuration file you stored or retrieved, or TCCONFIG.TC (in the current directory) if you haven't yet loaded, retrieved, or saved a configuration file.

Edit Auto Save

With this feature toggled to On, Turbo C automatically saves the source file in the Editor whenever you use the Run/Run or File/OS Shell command, if the file has been modified since the last time you saved it.

Backup Files

By default, Turbo C automatically creates a backup of the source file in the Editor when you do use File/Save; the backup file is given the extension .BAK. You can toggle this backup feature On and Off with this option.

Tab Size

When the Editor Tab mode is On and you press the *Tab* key, the Editor inserts a tab character in the file and the cursor jumps to the next tab stop. This menu item allows you to dictate how far apart the tab stops are; any number in the range 2 through 16 is allowed. The default is 8.

To change the way tabs are displayed in a file, just change Tab Size to the size you prefer, and the Editor redisplays all tabs in that file in the size you chose. You can save this new tab size in your configuration file (choose Save Options from the Options menu).

Zoomed Windows

If your Turbo C integrated environment screen is set up with the Edit window and Message window both showing, choosing Zoomed Windows...On zooms both windows to full screen, with the active window visible.

Use *F6* to switch from one window to the other, just as you do when both windows are showing.

To "unzoom" the windows (return to the setup where both windows are showing) just choose Zoomed Windows...Off.

The Screen Size Menu

When you choose Screen Size, another menu appears; the items on this Screen Size menu allow you to specify whether your integrated environment screen displays text in 25 lines or 43/50 lines. One or two of these items will be enabled, depending on the type of video adapter in your PC.

- **25 Lines**

  This is the standard PC display: 25 lines by 80 columns. This menu item is always enabled; it's the only screen size available to systems with a Monochrome Display Adapter (MDA) or Color Graphics Adapter (CGA).

- **43/50 Lines**

  If your PC is equipped with an EGA or VGA, this menu item is enabled, as well as 25 line standard display. Select it to transform your text to 43 lines by 80 columns if you have an EGA, or 50 lines by 80 columns if you have a VGA.

Directories

The entries in this menu tell Turbo C where to find the files it needs to compile, link, and output executable files, and where to find the configuration file, pick file, and help file.

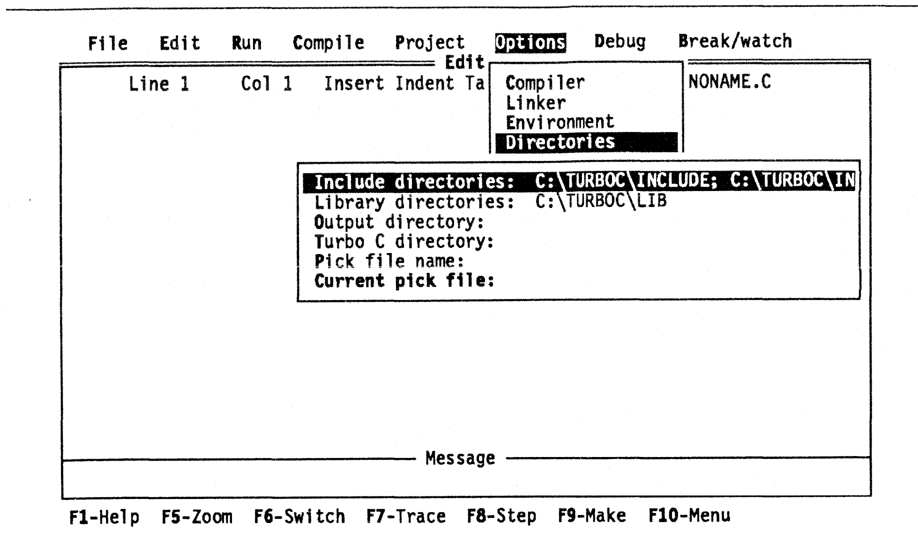


Figure 5.21: The Options/Directories Menu

Include Directories

Specifies the directories that contain your standard include files. Standard include files are those given in angle brackets (<>) in an `#include` statement (for example, `#include` <*myfile.h*>). Multiple directories are separated by semicolons (;). See Chapter 3 for more information about this option.

Library Directories

Specifies the directories that contain your Turbo C startup object files (C0?.OBJ) and run-time library files (.LIB files). Lets you list multiple library directories, up to a maximum of 127 characters (including white-space).

Use the following guidelines when entering library directories:

- You must separate multiple directory path names with a semicolon (;).
- Whitespace before and after the semicolon is allowed, but not required.
- Relative and absolute path names are allowed, including path names relative to the logged position in drives other than the current one.

For example,

```
C:\TURBOC\LIB; C:\TURBOC\MYLIBS; A:NEWTURBO\MATHLIBS; A:..\VIDLIBS
```

Output Directory

Your .OBJ, .EXE, and .MAP files are stored here; Turbo C looks for them here when doing a Make or Run. If the entry is blank, the files are stored in the current directory.

Turbo C Directory

This is used by the Turbo C system to find the configuration file (.TC) and the help file (TCHELP.TCH). For Turbo C to find your default configuration file (TCCONFIG.TC) at startup (if it's not in your current directory), you must install this path with TCINST, the external customization program.

Pick File Name

This item defines the name of a pick file to load. Entering a name here loads that pick file (if it exists) and defines where Turbo C will save the pick file when you exit. When you change the pick file name, Turbo C saves the current pick file before loading the new one.

If no pick file name is listed here, then Turbo C only writes a pick file if the Options/Directories/Current Pick File setting contains a file name.

To create a pick file, you must define a pick file name. You do this by entering a file name in the prompt box called up by the Options/Directories/Pick File Name setting. Once you have defined a pick file name, Turbo C will update that pick file on disk whenever you exit the integrated environment. This pick file name will be saved in your configuration file when you choose Options/Save Options.

Current Pick File

This menu item shows the file name and location of the current pick file, if there is one. This item is always disabled; it is for information only. Current Pick File shows a file name when a default pick file is loaded or when you enter one with the Pick File Name command. If you change the pick file

name or exit the integrated environment, Turbo C stores the current pick list information in this listed pick file.

Arguments

This setting allows you to give your running programs command-line arguments exactly as if you had typed them on the DOS command line(redirection is not supported). It is only necessary to give the arguments here; omit the program name.

Save Options

Saves all your chosen **C**ompiler, **L**inker, **E**nvironment, **D**ebug, and **P**roject options in a configuration file (the default file is TCCONFIG.TC). On start-up, Turbo C looks in the current directory for TCCONFIG.TC; if it doesn't find the file, then Turbo C looks in the Turbo C directory for the same file.

Retrieve Options

Loads a configuration file previously saved with the **O**ptions/**S**ave Options command.

The Debug Menu

The **D**ebug menu's commands control features of the integrated debugger other than breakpoints and watch expressions (which are on the **B**reak/ Watch menu.)

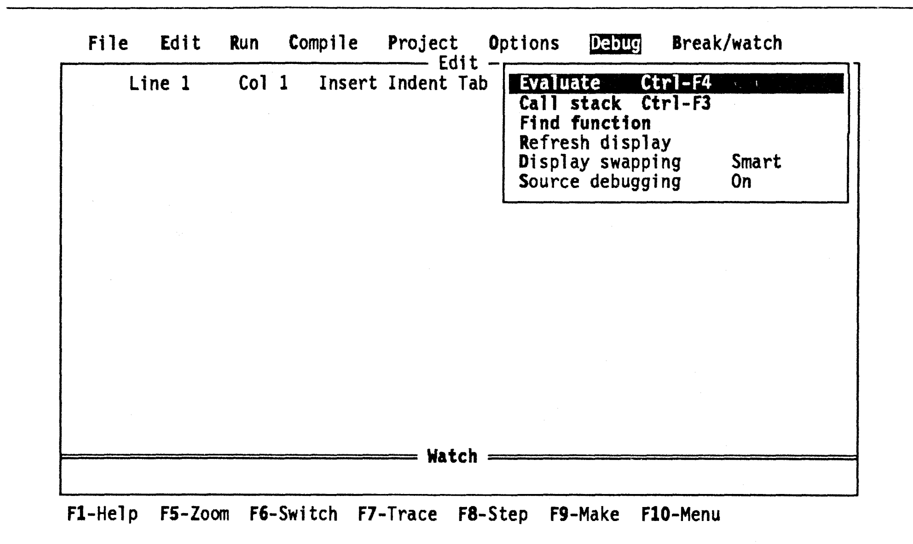


Figure 5.22: The Debug Menu

Evaluate

Evaluate evaluates a variable or expression, displays its value, and, if appropriate, lets you modify the value.

The command opens a pop-up window containing three fields: the Evaluate field, the Result field, and the New Value field. It fills the Evaluate field with a default expression consisting of the word at the cursor in the Edit window. You may evaluate the default expression by pressing *Enter*, or you may edit or replace it first. You can also extend the default expression by copying additional characters from the Edit window with the *Right arrow* key.

You may evaluate any valid C expression that doesn't contain

- function calls
- symbols or macros defined with **#define** or **typedef**, or
- local or static variables not in the scope of the function being executed, unless they are fully qualified.

If the debugger can evaluate the expression, it displays the value in the Result field.

If the expression refers to a variable or simple data element, you may move the cursor down to the New Value field and enter an expression as the new value.

If it is meaningful to modify the expression's value, but you do not want to do so, press *Esc* to close the window. If you have changed the contents of the New Value field and have not pressed *Enter*, the debugger will ignore the change you have made when you exit the window.

Debug/Evaluate displays each type of value in an appropriate format. For example, it displays an **int** as an integer in base 10, and an array as a pointer in base 16. To get a different display format, suffix the expression with a comma followed by one of the format specifiers shown in Table 5.5.

Use a repeat expression to display the values of consecutive data elements. For example, for an array of integers named *xarray*,

`xarray[0],5`	displays 5 consecutive integers in decimal
`xarray[0],5x`	displays 5 consecutive integers in hexadecimal

An expression used with a repeat count must represent a single data element. The debugger views the data element as the first element of an array if it isn't a pointer, or as a pointer to an array if it is.

Table 5.5: Format Specifiers Recognized in Debugger Expressions

Character	Function
C	**Character**. Shows special display characters for control characters (ASCII 0 through 31); by default, such characters are shown using the appropriate C escape sequences (`\n`, `\t`, etc). Affects characters and strings.
S	**String**. Shows control characters (ASCII 0 through 31) as ASCII values using the appropriate C escape sequences. Since this is the default character and string display format, the S specifier is only useful in conjunction with the M specifier.
D	**Decimal**. All integer values are displayed in decimal. Affects simple integer expressions as well as arrays and structures containing integers.
H or X	**Hexadecimal**. All integer values are displayed in hexadecimal with the `0x` prefix. Affects simple integer expressions as well as arrays and structures containing integers.

F<*n*>	**Floating-point.** *n* is an integer between 2 and 18 specifying the number of significant digits to display. The default value is 7. Affects only floating-point values.
M	**Memory dump**. Displays a memory dump, starting with the address of the indicated expression. The expression must be a construct that would be valid on the lefthand side of an assignment statement, i.e. a construct that denotes a memory address; otherwise, the M specifier is ignored. By default, each byte of the variable is shown as two hexadecimal digits. Adding a D specifier with the M causes the bytes to be displayed in decimal, and adding an H or X specifier causes the bytes to be displayed in hexadecimal. A C or an S specifier causes the variable to be displayed as a string (with or without special characters). The default number of bytes displayed corresponds to the size of the variable, but a repeat count may be used to specify an exact number of bytes.
P	**Pointer**. Displays pointers in *seg:ofs* format with additional information about the address pointed to, rather than the default hardware-oriented *seg:ofs* format. Specifically, it tells you the region of memory in which the segment is located, and the name of the variable at the offset address, if that is appropriate. The memory regions are as follows:

Memory Region	Evaluate Message
0000:0000 – 0000:03FF	Interrupt vector table
0000:0400 – 0000:04FF	BIOS data area
0000:0500 – Turbo C	MSDOS/TSR's
Turbo C – User Program PSP	Turbo C
User Program PSP	User Process PSP
User Program – top of RAM	Name of a static user variable if its address falls inside the variable's allocated memory; otherwise nothing

Table 5.5: Format Specifiers Recognized in Debugger Expressions (continued)

	A000:0000 – AFFF:FFFF	EGA Video RAM
	B000:0000 – B7FF:FFFF	Monochrome Display RAM
	B800:0000 – BFFF:FFFF	Color Display RAM
	C000:0000 – EFFF:FFFF	EMS Pages/Adaptor BIOS ROM's
	F000:0000 – FFFF:FFFF	BIOS ROM's
R	**Structure/Union.** Displays field names as well as values, such as `{ X:1, Y:10, Z:5 }`. Affects only structures and unions.	

The hot key for Debug/Evaluate is *Ctrl-F4*.

Find Function

Find Function displays the definition of a function in the Edit window. The command can find any function in your program that was compiled with the Debug/Source Debugging and O/C/C/OBJ Debug Information options set to On, and whose source file is available. If the function is not in the currently displayed file, the command automatically loads the proper file.

You must be in a debugging session to use Find Function.

Call Stack

Call Stack displays a pop-up window containing the call stack. The call stack shows the sequence of functions your program called to reach the function now running. **main** is at the bottom of the stack; the function now running is at the top.

Each entry on the call stack displays the name of the function called and the values of the parameters passed to it.

Initially the entry at the top of the stack is highlighted. To display the current line of any other function on the call stack, move the highlight to that function's name and press *Enter*. The cursor will be positioned on the line containing the call to the function next above it on the stack. For example, if the call stack looked like this:

```
func2()
func1()
main()
```

(that is, **main** calls **func1**, and **func1** calls **func2**), and you wanted to see the currently executing line of **func1**, you would place the highlight on `func1` in the call stack and press *Enter*. The code for **func1** would appear in the Edit window, with the cursor positioned on the call to **func2**.

To return to the current line of the function now being run (that is, to the execution position), highlight the topmost function in the call stack and press *Enter*.

Some functions may be omitted from the call stack if your program is compiled with the O/C/C/Standard Stack Frame option set to Off. See the description of O/C/C/Standard Stack Frame for more information.

The hot key for O/C/C/Stack Frame is *Ctrl-F3*.

Source Debugging

The Debug/Source Debugging option is a three-way toggle you can set to On, Standalone, or None.

Programs linked with this toggle set to On can be debugged with either the TC integrated debugger or the standalone Turbo Debugger. When it is set to Standalone, programs can be debugged only with Turbo Debugger, although they can still be run in TC. When the toggle is set to None, programs cannot be debugged with either debugger, because no debugging information has been placed in the .EXE file.

Display Swapping

Debug/Display Swapping is a three-way toggle that can be set to Smart (the default setting), Always, or None.

When you run your program in debug mode with the default Smart setting, the debugger looks at the code being executed to see whether the code will generate output to the screen. If the code does output to the screen (or if it calls a function), the screen is swapped from the Edit screen to the User screen long enough for output to take place, then is swapped back. Otherwise, no swapping occurs.

Note: The Smart default setting is not particularly smart in the following respects:

- It swaps on any function call, even if the function does no screen output.
- In some situations, the Editor screen may be modified without being swapped; for example, if a timer interrupt routine writes to the screen.

The **Always** setting causes the screen to be swapped every time a statement executes. You should choose this setting any time the Editor screen is likely to be overwritten by your running program.

The **None** setting causes the debugger not to swap the screen at all. It should be used for debugging sections of code that you are certain do not output to the screen.

Note: If you are debugging in dual monitor mode (that is, you used the TC command-line /d switch) you can see your program's output on one monitor and the TC screen on the other. Therefore, TC never swaps screens and the **Debug/Screen Swapping** setting has no effect.

Refresh Display

If the Editor screen should accidentally be overwritten, you can use this option to restore its previous contents.

The Break/Watch Menu

The **Break/Watch** menu's commands control breakpoints and watch expressions.

A *breakpoint* is a location in a program where execution should halt, to give you time to examine the value of critical variables and expressions and otherwise make sure that your program is behaving as it should.

A breakpoint is marked by a *breakpoint highlight*. A breakpoint's highlight is obscured by the execution bar when the program halts at that breakpoint, but reappears when the execution bar moves on.

A *watch expression* is an expression whose value is displayed in the Watch window, and is re-evaluated whenever the program halts. The rules for entering a valid watch expression are the same as those for entering a valid expression in **Debug/Evaluate**, except that watch expressions may not contain side effects such as `i++`. Conversion-type characters and repeat counts may be used in watch expressions, as in **Debug/Evaluate**; for example,

```
i,x
```

displays the contents of integer *i* in hexadecimal format.

As you add expressions to the Watch window, it will grow to its maximum size as specified by the TCINST utility's **R**esize Windows option (initially about half the screen). If you add more expressions, some expressions will scroll out of the window. You can get them back by scrolling the Watch window display with the *PgUp, PgDn, Up arrow,* and *Down arrow* keys.

The current watch expression in the Watch window is marked by a highlight bar when the Watch window is active, and by a bullet (♦) in the left column when it is not.

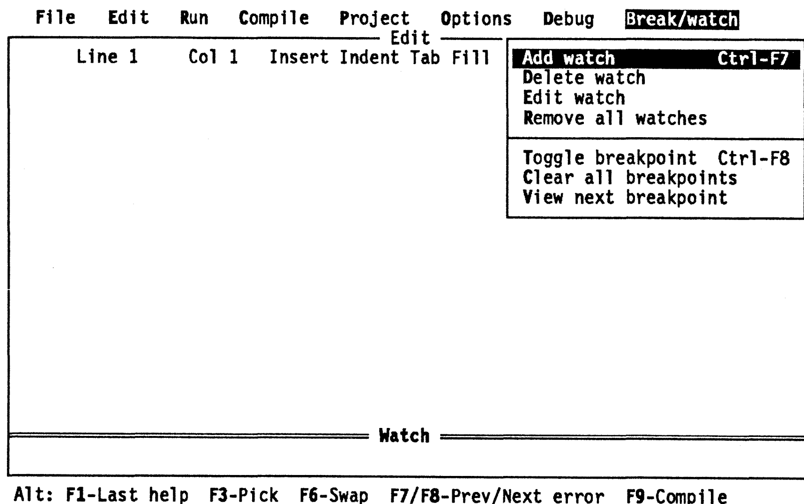


Figure 5.23: The Break/Watch Menu

Add Watch

Add Watch inserts a watch expression into the Watch window. When you choose this command, the debugger opens a pop-up window and prompts you to enter a watch expression. The default expression is the word at the cursor in the Edit window. When you type a valid expression and press *Enter,* the debugger adds the expression and its current value to the Watch window.

The hot key for this command is *Ctrl-F7*. Additionally, if the Watch window is the active window, you can insert a new watch expression by pressing *Ins* or *Ctrl-N*.

Delete Watch

Delete Watch deletes the current watch expression from the Watch window.

The Watch window must be visible (that is, the Edit window cannot be zoomed) in order for you to use this command. The current watch expression is marked by a highlight bar if you are in the Watch window, and by a bullet in the left margin if you are in the Edit window.

To delete the watch expression marked with the bullet when you are in the Edit window, choose **B**reak/Watch/**D**elete Watch. To delete a watch expression that is not current, you must move to the Watch window, position the highlight bar on the desired watch expression, and press either *Del* or *Ctrl-Y*.

Edit Watch

Edit Watch allows you to edit the current watch expression in the Watch window.

When you choose **B**reak/Watch/**E**dit Watch, the debugger opens a pop-up window containing a copy of the current watch expression. Edit the expression and press *Enter*. The debugger replaces the original version of the expression with the edited one. You can also edit a watch expression from inside the Watch window by positioning the highlight bar on it and pressing *Enter*.

Remove All Watches

Remove All Watches deletes all watch expressions from the Watch window.

Toggle Breakpoint

Toggle Breakpoint sets or clears a breakpoint on the line where the cursor is positioned. When a breakpoint is set, it is marked by a breakpoint highlight.

The hot key for this command is *Ctrl-F8*.

Your program will halt whenever it encounters a breakpoint in the course of running. When the program halts, the execution bar is on the line where the breakpoint is set. The breakpoint highlight is obscured by the execution bar, but reappears when the execution bar moves on.

When a source file is edited, each breakpoint "sticks" to the line where it is set. It is lost only when you leave the integrated environment, when you delete the source line it is set on, or when when you clear it with the **B**reak/Watch/**T**oggle Breakpoint command or the **B**reak/Watch/Clear All Breakpoints command.

Turbo C will "lose track" of its breakpoints in two cases:

- If you edit a file containing breakpoints and then abandon the edited version of the file. (Turbo C cannot remember where the breakpoints were set before the file was edited, and so will display them on the wrong lines.)
- If you edit a file containing breakpoints and then continue the current debugging session, without remaking the program. (Turbo C displays the warning prompt `Source modified, rebuild?`)

Before you compile a source file, you can set a breakpoint on any line, even a blank line or a comment. When you compile and run the file, Turbo C validates any breakpoints that are set and gives you a chance to remove, ignore, or change invalid breakpoints. When you are debugging the file, Turbo C knows which lines contain executable statements, and will warn you if you try to set invalid breakpoints.

Clear All Breakpoints

Clear All Breakpoints removes all breakpoints from your program.

View Next Breakpoint

View Next Breakpoint moves the cursor to the next breakpoint in the program. Note that it moves the cursor to the next breakpoint *in the order that the breakpoints were set*, not the order in which your program will encounter breakpoints. This command does not run your code; it only positions active breakpoints in the Editor window.

Part III: More about Configuration and Pick Files

What Is a Configuration File?

Basically, a configuration file is a file that contains information pertinent to Turbo C. In it, you store such information as your chosen compiler options, your linker options, and various directories that Turbo C will need to search when compiling and linking your programs.

There are two types of Turbo C configuration files: one you use with TCC.EXE (command-line Turbo C), and the other you use with TC.EXE (the Turbo C integrated environment). There is only one command-line configuration file; it must be named TURBOC.CFG. The integrated environment configuration file can have any file name. The file TCCONFIG.TC is the default (assumed) integrated development environment configuration file.

In this section we cover the integrated environment configuration files in detail. If you want to know more about how to use TURBOC.CFG, refer to "The TURBOC.CFG File" in Chapter 3.

The TC Configuration Files

When you enter the Turbo C integrated environment for the first time, there is no configuration file. TC.EXE will start up with all the menu toggles and settings set to their internal defaults (**O**ptions/**C**ompiler/**M**odel will be set to **S**mall, **O**ptions/**C**ompiler/**C**alling Convention set to C, **O**ptions/**E**nvironment/**K**eep Messages set to No, and so on). In the course of using the integrated environment, you will probably want to change some of the menu toggles and settings.

If you exit Turbo C without saving the new settings in a configuration file, the next time you invoke the integrated environment it will again start up with all the menu items set to their previous defaults. But if you save the new settings to a configuration file, the next time the integrated environment starts up the menu items will be set to the values you chose. You won't have to go through the process of resetting them.

TCCONFIG.TC

When you start up TC.EXE, it looks for a configuration file named TCCONFIG.TC. It looks for that file in certain locations (we'll explain exactly where it looks later); if TC.EXE can't find a TCCONFIG.TC file, the integrated environment starts up using the default settings that are built into TC.EXE.

Other TC Configuration Files

You can also start up TC.EXE at the DOS prompt with a request for a specific configuration file, using the /c switch (refer to the section on "TC Command-Line Switches," beginning on page 82, for more information). For example, if you type

```
tc /cmyconfig
```

at the DOS prompt, Turbo C will look for a configuration file named MYCONFIG.TC in the current directory (if you give no extension, Turbo C assumes the extension .TC).

If Turbo C can't find the configuration file you named, it will issue a warning message to that effect. It won't look for any other configuration file, but it will still start up, using the built-in default settings.

What Is Stored in TC Configuration Files?

The information stored in the TC configuration files can be broken down into two categories: compiler-linker options and TC.EXE-specific values.

The compiler-linker options govern the compiler and linker, and they all have corresponding options in the command-line version of Turbo C. The TC.EXE-specific values are related to the integrated environment itself. Some examples of these values specific to the integrated environment are Project/Project Name, Options/Directories/Pick File Name, and the Option/Environment menu options.

Creating a TC Configuration File

How do you create a TC configuration file? Unlike the command-line configuration file (TURBOC.CFG), the integrated environment configuration file is not one you can create or modify with an Editor. Instead, choose the Options/Save Options command from the Options menu, and the integrated environment will create the configuration file for you.

If you set Options/Environment/Config Auto Save to On, your current settings will be saved in the default TC configuration file (TCCONFIG.TC) whenever you exit the integrated environment.

Changing Configuration Files Midstream

It's easy to change to a different .TC configuration file from within the integrated environment. To do this:

- Choose Options/Retrieve Options from the Options menu. A pop-up box will appear, displaying the last configuration file name you entered (it defaults to *.TC the first time).
- You can type in a mask (like `*.tc` or `??config.*`), then press *Enter* to bring up a directory listing of .TC files. You then choose a file from the directory list, or you can type in a specific configuration file name, then press *Enter* to load that file.

Where Does TC.EXE Look for TCCONFIG.TC?

There are two places TC.EXE will look for the default configuration file TCCONFIG.TC. First, it will search the default (current working) directory. If it does not find TCCONFIG.TC there, it will then search the Turbo C directory, *if you have previously set the Turbo C directory using TCINST*.

To find out more about the Turbo C directory and TCINST, read Appendix F, "Customizing Turbo C," in the *Turbo C Reference Guide*.

TCINST vs. the Configuration File: Who's the Boss?

You can use TCINST to set any of the items found on Turbo C's menus, then store those settings directly in TC.EXE. If there is no TC configuration file to be found when you start up your customized TC.EXE, those settings you customized will be the defaults.

However, if TC.EXE starts up and finds a TCCONFIG.TC file in the default directory (or in the Turbo C directory), that configuration file's settings will take precedence over any default settings you installed with TCINST.

Furthermore, if you invoke TC.EXE using the /c switch, and Turbo C finds the configuration file you specified, that file's settings will take precedence over the TCINST-installed defaults.

What Does Options/Environment/Config Auto Save Do?

Normally, Turbo C will save the current configuration file (write it to disk) only when you give the Options/Save Options command. However, you can direct Turbo C to save the configuration file automatically under certain circumstances.

Just set the Options/Environment/Config Auto Save toggle to On. With Config Auto Save set to On, Turbo C will also save the file whenever you choose Run/Run or File/OS Shell, or when choose File/Quit to exit the integrated environment—if the configuration file has never been saved, or if it has been at all modified since it was last saved. If the configuration file has not yet been saved, Turbo C will choose a file name for the automatically saved file from two possibilities:

- the name of the last configuration file you stored or retrieved
- TCCONFIG.TC (in the current directory), if you haven't yet loaded, stored, or retrieved a configuration file

What Are Pick Lists and Pick Files?

The *pick list* and *pick file* are two features of the Turbo C integrated environment that work together to save the state of your editing sessions. The pick list remembers what files you are editing *while you are in* the integrated environment. The pick file remembers what files you were editing *after you leave* the integrated environment or *after you change contexts within* the integrated environment. (Changing contexts means loading a new configuration file or defining a new pick file name.)

The Pick List

You call up the pick list by choosing File/Pick or pressing the *Alt-F3* hot key. File/Pick provides a list of the eight files most recently loaded into the Editor. The top file listed is the file currently in the Editor. If there is more than one file name in the pick list, the second file name listed is highlighted; this is the file previously loaded into the Editor.

To load a file from the pick list into the Editor, use the arrow keys to move the highlight bar to the appropriate file name, then press *Enter*. When you do this, Turbo C will load the chosen file into the Editor, and the Editor will

position the cursor in that newly loaded file at the location you last left it. In addition, any marked block and markers in the file will be exactly as you left them.

The pick list is a handy tool for moving back and forth among your files as you develop your program. By pressing *Alt-F3* and then *Enter*, you can alternate files (this is the same as pressing *Alt-F6* when you are in the Editor).

If the file you want is not on the pick list, you can choose `--load file--` (the last entry on the pick list menu). This will bring up a *Load File Name* input box, and you can type in the name of the file you want (using DOS-style wildcards if appropriate). You can also press the *F3* hot key to choose **F**ile/**L**oad automatically.

The Pick File

The pick file stores file-related information, including the contents of the pick list. For each entry (file) in the pick list, Turbo C stores the file name, cursor position, marked block, and markers.

In addition to information about each file, the pick file contains data on the state of the Editor when you last exited. This includes the most recent search-and-replace strings and search options.

To create a pick file, you must define a pick file name. You can do this by choosing **O**ptions/**D**irectories/**P**ick File Name and entering a file name. Once you have defined a pick File Name, Turbo C updates that pick file on disk whenever you exit the integrated environment.

When and How Do You Get a Pick File?

There are two menu items you can look at for information about the pick file: **O**ptions/**D**irectories/**P**ick File Name and **O**ptions/**D**irectories/**C**urrent Pick File.

Q: How do you know whether you already have a pick file?
A: You have a pick file if the **O**ptions/**D**irectories/**C**urrent Pick File menu setting is not blank (contains a file name).

Q: Why does that file name appear in **O**ptions/**D**irectories/**C**urrent Pick File?

A: Either a file name has been listed explicitly in Options/Directories/Pick File Name, or (if the Options/Directories/Pick File Name setting is blank) you have loaded a default pick file.

Q: Suppose the Options/Directories/Pick File Name setting explicitly lists a file name. How did that file name get there?

A: You get a file name in Pick File Name by

- entering it yourself in the current session
- entering it in a previous session, saving the configuration file, then using that configuration file in the current session
- installing it with TCINST

Q: Suppose Options/Directories/Pick File Name is blank, but Options/Directories/Current Pick File contains a file name. How did that default pick file get loaded?

A: There was a default pick file, TCPICK.TCP, in the current directory or (if not there) in the Turbo C directory, and Turbo C loaded it automatically on startup.

Once a pick file is loaded, the integrated Environment remembers the full path name. This information is displayed in the Options/Directories/Current Pick File setting.

When Does Turbo C Save Pick Files?

Turbo C saves the file named in Options/Directories/Current Pick File whenever you exit the integrated environment. In addition, any time the pick file name is changed (either directly by entering a new name from the menu, or indirectly, by loading a configuration file that contains a different pick file name) Turbo C first saves the existing pick file.

Turbo C will *not* save a pick file to disk when you exit TC if the Options/Directories/Current Pick File setting is blank.

Part IV: Additional Features and Editing Commands

Some useful Turbo C editing features available in the integrated environment cannot be accessed from the menu system. This section contains information on how to take advantage of these features when you are editing source code.

More on Tabs

- When the Editor's Tab mode is off, pressing the *Tab* key inserts enough space characters to move the cursor to the next "soft" tab stop. Soft tab stops align with the first letter of each word in the line of text immediately above the current line. (When Tab mode is On, of course, pressing the *Tab* key inserts enough spaces to move the cursor to the next tab stop, as designated by Options/Environment/Tab Size—the default is 8.)
- When you send a marked block of text from the Editor to a file (or to PRN) with the *Ctrl-K W* command, the Editor treats all tab characters as *hardware tabs* and writes (or prints) them "as is." This generally yields tab stops at every eighth column. However, when you send text from the Editor to the printer with the *Ctrl-K P* command, the Editor treats tab characters as *software tabs* and prints them as the appropriate number of space characters (equal to the tab size you chose with Options/Environment/Tab Size).

Autoindent, Unindent, and Optimal Fill

Autoindent is an editing feature that, following a hard return, positions the cursor under the first nonblank character in the preceding nonblank line.

When you first run TCINST (the Turbo C Customization program), Autoindent mode will be turned On automatically, since the default for the TCINST Options/Environment/Options for Editor/Autoindent toggle is On.

Thereafter, in the Edit window, you can toggle Autoindent On or Off by pressing *Ctrl-O I* or *Ctrl-Q I*. (That is, hold down the Control key *and* press *O* or *Q*, then press *I*.)

Unindent is an editing feature that *outdents* the cursor; that is, it moves the cursor one or more spaces to the left to line up with the previous indentation level.

```
a  =  3;
i  =  1;
while  (i  <=  25)
     {
    product  =  a  *  i;
    printf("%d.   %d",  i,  product);
     ++i;
    }
```

← Autoindent will return you to this column.
← Pressing the backspace once to outdent will step you back to this column.

Figure 5.24: How Unindent Works

To use Unindent, position the cursor on the first nonblank character of a line, or on a blank line. Press *Backspace*. The cursor will be moved to the same column as the previous indentation level; in this case, *Backspace* may move back more than one space.

Optimal fill mode has no effect unless Tab mode is also set to On. When both these modes are enabled, the beginning of every autoindented and unindented line is filled optimally with tabs and spaces. This produces lines with a minimum number of characters. *Ctrl-O F* toggles Optimal fill On and Off.

Examples

- Options/Environment/Tab Size is set to 8 (tab stops are in columns 1, 9, 17, 25, ...); Autoindent, Tab, and Insert modes are On; and the cursor is at the end of a line that begins at column 27.
  - Press *Enter* to insert a new line; the Editor positions the cursor at column 27 in that new line.
  - Without moving the cursor, type a character on the new line.

- The Editor fills the beginning of the new line with three tab characters (to column 25) and two space characters (to column 27) for a total of five inserted fill characters.

- If, in this same example, Options/Evironment/Tab Size is set to 5 (tab stops in columns 1, 6, 11, 16, 21, 26, ...), the Editor fills with five tab characters (to column 26) and one space character.
- Or, if Tab Size is set to 6 (tab stops 1, 7, 13, 19, 25, ...), and you move the cursor to column 18 before typing your first characters, the Editor fills with two tab characters (to column 13) and five space characters (to column 18).

Pair Matching

There you are, debugging your source file that is full of functions, parenthesized expressions, nested comments, and a whole slew of other constructs that use delimiter pairs. In fact, your file is teeming with

- braces: { and }
- angle brackets: < and >
- parentheses: (and)
- square brackets: [and]
- comment markers: /* and */
- double quotes: "
- single quotes: '

Finding the match to a particular paired construct can be tricky. Suppose you have a complicated expression with a number of nested subexpressions, and you want to make sure all the parentheses are properly balanced. Or say you're at the beginning of a function that stretches over several screens, and you want to jump to the end of that function. With Turbo C's handy pair-matching commands, the solution is at your fingertips. Here's what you do:

1. Place the cursor on the delimiter in question (for example, the opening brace of some function that stretches for a couple of screens).
2. To locate the mate to this selected delimiter, simply press *Ctrl-Q Ctrl-[*. (In the example given, the mate should be at the end of the function.)
3. The Editor immediately moves the cursor to the delimiter that matches the one you selected. If it moves to the one you had intended to be the mate, you know that the intervening code contains no unmatched

delimiters of that type. If it moves to the wrong delimiter, you know there's trouble in River City; now all you need to do is track down the source of the problem.

A Few Details about Pair Matching

We've told you the basics of Turbo C's "Match Pair" commands; now you need some details about what you can and can't do with these commands, and notes about a few subtleties to keep in mind. This section covers the following points:

- There are actually two Match Pair editing commands: one for forward matching (*Ctrl-Q Ctrl-[*) and the other for backward matching (*Ctrl-Q Ctrl-]*).
- The way the Editor searches for comment delimiters (/* and */) is slightly different from the the way it performs the other searches.
- If there is no mate for the delimiter you've selected, the Editor doesn't move the cursor.

Directional and Nondirectional Matching

Two Match Pair commands are necessary because some delimiters are *directional*, while others are not.

For example, suppose you tell the Editor to find the match for an opening brace ({) or an opening square bracket ([ ). The Editor knows the matching delimiter can't be located *before* the one you've selected, so it searches forward for a match. Opening braces and opening square brackets are directional; the Editor knows in which direction to search for the mate, so it doesn't matter which Match Pair command you give. Given either command, the Editor still searches in the correct direction.

Similarly, if you tell the Editor to find the mate to a closing brace (}) or a closing parenthesis ( )), it knows that the mate can't be located *after* the selected delimiter, so it automatically searches backward for a match. Again, because these delimiters are directional, it doesn't matter which Match Pair command you give; the Editor always searches in the correct direction.

However, if you tell the Editor to find the match for a double quote (") or a single quote ('), it doesn't know automatically which way to go. You must specify the search direction by giving the correct Match Pair command. If you give the command *Ctrl-Q Ctrl-[*, the Editor searches forward for the

match; if you give the command *Ctrl-Q Ctrl-]*, it searches backward for the match.

The following table summarizes the delimiter pairs, whether they imply search direction, and whether they are nestable. (Nestable delimiters are explained after this table.)

Delimiter Pair	Direction Implied?	Are They Nestable?
{ }	Yes	Yes
( )	Yes	Yes
[ ]	Yes	Yes
< >	Yes	Yes
/* */	Yes	Yes and No
" "	No	No
' '	No	No

Nestable Delimiters

What does *nestable* mean? Simply that, when it is searching for the mate to a directional delimiter, the Editor keeps track of how many delimiter levels it enters and exits during the search.

This is best illustrated with some examples:

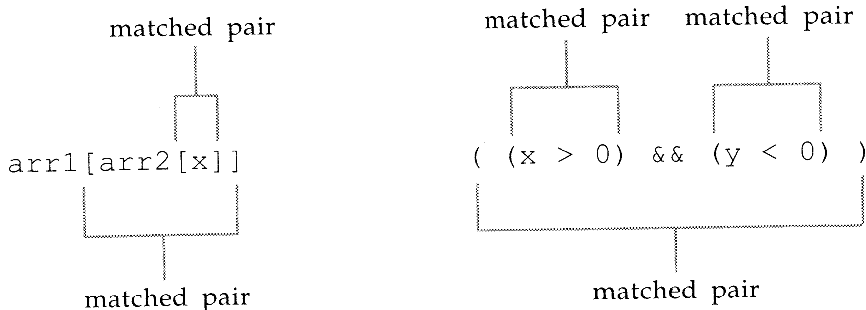


Figure 5.25: Search for Match to Square Bracket or Parenthesis

The Search for Comment Delimiters

Because comment delimiters are two-character delimiters, you must take care when you highlight one for a Match Pair search. In either case, the Editor recognizes only the *first* of the two characters: the slash (/) part of a /* comment delimiter, or the asterisk (*) part of a */ delimiter. If you place the cursor on the *second* character in either of these delimiters, the Editor won't know what you're looking for, so it won't do any searching at all.

Also, as shown in the preceding table, comment delimiters are sometimes nestable, sometimes not ("Yes and No"). This is not a vagary or an inability to decide: It is a test dependent on multiple conditions. ANSI-compatible C programs cannot contain nested comments, but Turbo C provides an optional **O/C/S/**Nested Comments feature that you can toggle On and Off. This feature affects the nestability of comment delimiters when it comes to pair matching.

- If **O**ptions/**C**ompiler/**S**ource/**N**ested Comments is toggled On, the Editor treats comment delimiters as nestable and keeps track of the delimiter levels it enters and exits in the search for a match.
- If **O**ptions/**C**ompiler/**S**ource/**N**ested Comments is toggled Off, the Editor does not treat comment delimiters as nestable; when a /* pair is selected, the first */ pair the Editor finds is the match (and vice versa).

Note: If unmatched delimiters of the same type in comments, quotes, or conditional compilation sections fall between the matched pair, this affects the search.

Here are some examples to illustrate these differences:

```
/* /* /* /* Here are some nested comments */ */ */ */
```

match level selected — match level found

Figure 5.26: Nested Comments Toggled On--Forward Search with ^Q ^[

Note: A backward search from the found */ will yield the selected /* when **O**ptions/**C**ompiler/**S**ource/**N**ested Comments is toggled On.

```
/* /* /* /* Here are some nested comments */ */ */ */
```

match level selected — match level found

Figure 5.27: Nested Comments Toggled Off--Forward Search with ^Q ^(

```
/* /* /* /* Here are some nested comments */ */ */ */
```

match level found — match level selected

Figure 5.28: Nested Comments Toggled Off--Backward Search with ^Q ^)

Editing Key Assignment

Turbo C's interactive Editor provides many editing functions that are assigned to certain keys or key combinations. These are explained in detail in Appendix A in the *Turbo C Reference Guide.*

TCINST is Turbo C's optional customization program. One of its menus allows you to assign the Turbo C editing functions to other keys, if you prefer. This is known as *rebinding the keys*.

Note: This feature is covered in detail in Appendix F in the *Turbo C Reference Guide*, so we'll cover just the basics here.

To change Turbo C's editing commands, follow this general procedure:

1. Load TCINST.EXE (at the DOS prompt, type `tcinst` and press *Enter*). From the main customization menu, choose the Editor commands

menu. The Install Editor screen will come up, displaying three columns of text:

- The first column (on the left) describes the editing functions available.
- The second column lists the *primary* keystrokes (what you press to invoke a particular editing function).
- The third column lists the *secondary* keystrokes (optional alternate keystrokes you can press to invoke the same editing function).

2. The bottom lines of text in the Install Editor screen summarize the keys you use to change entries in the primary and secondary keystroke columns. Press *Enter* to enter the keystroke-editing mode, then use the *Left* and *Right arrow* keys to move the highlight bar to either the primary or secondary column.
3. Use the *Up* and *Down arrow* keys to highlight the editing command you intend to rekey.
4. Press *Enter* to choose the highlighted editing command; the defined keystroke(s) for that command appears in a pop-up window.
5. Press *Backspace* to delete individual keystrokes from right to left in the pop-up window, or press *F3* to clear all defined keystrokes from the window.
6. Keystroke combinations come in three flavors: WordStar-like, Ignore Case, and Verbatim. Press *F4* to cycle through these until the one you want is highlighted on the bottom line of the screen. Refer to Appendix F in the *Turbo C Reference Guide* for more information about these three variations.
7. Type in the new defined keystrokes for that editing function (up to a maximum of six keystrokes). If you want to erase the last keystroke you assigned, press *Backspace.* If you want to abandon the new key assignments to that function, press *F2* to restore the originally assigned keys, or *Esc* to restore them *and* leave the keystroke-editing mode.
8. Once you're satisfied with the new (or restored) key assignment(s) to a given function, press *Enter* to accept them.
9. When you've finished assigning keys (you've accepted the last modification), press *Esc* to leave the Install Editor screen and return to TCINST's main menu.

Note: If you override a standard Turbo C key, you will not be able to use that Turbo C shortcut while you are in the Editor.

C H A P T E R

6

Programming in Turbo C

Have you ever programmed in C before? You may have heard various stories about how C is a difficult language to learn. Nonsense. It is true that some C programmers delight in writing obscure programs that are difficult to read and debug, but there's nothing that says you have to do the same. The basic elements of the C programming language are easy to understand and use.

In This Chapter ...

In this chapter, we will teach you the basic elements of the C language and show you how to use them in your programs. The next chapter, "More About Programming in Turbo C," teaches you more about C, and Chapter 12, "Advanced Programming in Turbo C," tells you all about memory models, interrupts, assembly language programming, and other advanced topics.

Of course, we can't teach you everything about programming in C in one or two chapters; there are entire books about that. See the bibliography in the back of the *Turbo C Reference Guide* for a list of books about the C language that you may want to refer to.

Before you work through this chapter, you should read Chapter 5, "The Turbo C Integrated Development Environment," and learn how to use the menus and text editor in Turbo C. You should also have made backup

copies of your Turbo C disks and installed Turbo C as described in Chapter 1.

Once you've done all that, then sit down, turn on your computer (if it isn't already on), and get ready to learn about programming in Turbo C.

The Seven Basic Elements of Programming

The purpose of most programs is to solve a problem. Programs solve problems by manipulating information. Your job is to

- get the information into the program
- arrange for someplace to keep it
- give the instructions to manipulate it
- get it back out of the program to the user

You can organize your instructions so that

- some are executed only when a specific condition (or set of conditions) is true
- some are repeated a number of times
- some are broken off into chunks that can be executed at different locations in your program

We've just described the seven basic elements of programming: *input*, *data types*, *operations*, *output*, *conditional execution*, *loops*, and *subroutines*. This list is not comprehensive, but it does describe those elements that programs usually have in common.

Most programming languages have all these; many, including C, have additional features as well. But to learn a new language quickly, it is usually most efficient to learn how that language implements these seven elements, then build from there. Here's a brief description of each element:

Input means reading values in from the keyboard, from a disk, or from an I/O port.

Data types are constants, variables, and structures that contain numbers (integer and real), text (characters and strings), or addresses (of variables and structures).

Operations assign one value to another, combine values (add, divide, etc.), and compares values (equal, not equal, etc.).

Output means writing information to the standard output (*stdout*), to a disk, or to an I/O port.

Conditional execution means that your program executes a set of instructions only if a specified condition is true (and skips them if it is false).

Loops (iterations) execute a set of instructions some fixed number of times or while some condition is true.

Subroutines are separately named sets of instructions that can be executed anywhere in the program just by a reference to the name.

Now we'll take a look at how to use these elements in Turbo C.

Output

It may seem funny to talk about output first, but a program that does not somehow output information isn't of much use. Output usually takes the form of information written to the screen (words and pictures), to a storage device (floppy or hard disk), or to an I/O port (serial port, printer port).

The printf Function

You've already used the most common output function in C: the **printf** routine. The purpose of **printf** is to write information to the screen. Its format is both simple and flexible:

```
printf(<format string>,<item>,<item>,...);
```

The Format String

The format string is just a string that begins and ends with double quotes ("like this"); **printf**'s purpose is to write that string to the standard output (*stdout*). First, though, **printf** substitutes in the string certain *items* listed after the string, according to the *format commands* found in the string itself. For example, your last program had the following **printf** statement:

```
printf("The sum is %d \n",sum);
```

The `%d` in the format string is one kind of format command called a *format specification*. All format specifications start with a percent sign (%) and are (usually) followed by a single letter, indicating the type of data to be inserted and how the data is to be formatted.

You should have exactly one item listed for each format specification. If the item is of a data type that doesn't directly correspond to the format specification, you will get unpredictable results. The items themselves can be variables, constants, expressions, function calls. In short, they can be anything that yields a value appropriate to the corresponding format specification.

The %d used in this specification says that it expects an integer. Here are some other commonly used format specifications:

%u	unsigned integer
%ld	long integer
%p	pointer value
%f	floating-point
%e	floating-point in exponential format
%c	character
%s	string
%x or %X	integer in hexadecimal format

You can set the width of the output field in which the data is printed out by placing a number between the % and the letter that follows; for example, to set an integer field to 4 spaces wide, you would type %4d. The value will be printed out right justified (with leading blanks), so that the total field width is 4 spaces.

If you need to print a percent sign, just insert %%.

The \n in the string isn't a format specification. It is known (for historical reasons) as an *escape sequence*, and it represents a special character being inserted into the string. In this case, the \n inserts a newline character. It causes the cursor to move to the start of a new line after the string is written out.

A complete list of all escape sequences can be found in Chapter 11. Here are a few of the more commonly used ones:

\f	formfeed
\t	tab
\b	backspace
\xhhh	insert the character represented by ASCII code *hhh*, where *hhh* equals 1 to 3 hexadecimal digits)

And if you need to print a backslash, just insert `\\`. If you want more detail on how **printf** works, turn to the **printf** entry in Chapter 2 of the *Turbo C Reference Guide*.

Other Output Functions: puts and putchar

There are two other output functions that you might be interested in: **puts** and **putchar**.

The function **puts** writes a string to the screen followed by a newline character.

For example, you could rewrite HELLO.C as

```
#include <stdio.h>

main()
{
   puts("Hello, world");
}
```

Note that we dropped the `\n` at the end of the string; it isn't needed, since **puts** adds one.

The function **putchar** writes a single character to the screen and does not add a `\n`. The statement `putchar(ch)` is equivalent to `printf("%c",ch)`.

Why might you want to use **puts** and/or **putchar** instead of **printf**? One good reason is that the routine that implements **printf** is rather large; unless you need it (for numeric output or special formatting), you can make your program both smaller and quicker by using **puts** and **putchar** instead. For example, the .EXE file created by compiling the version of HELLO.C that uses **puts** is much smaller than the .EXE file for the version that uses **printf**.

Data Types

When you write a program, you're working with some kind of information, most of which falls into one of four basic types: *integers*, *floating-point numbers*, *text*, and *pointers*.

Integers are the numbers you learned to count with (1, 5, –21, and 752, for example).

Floating-point numbers have fractional portions (3.14159) and exponents (2.579×10^{24}). These are also known as *real numbers*.

Text is made up of single characters (*a*, *Z*, *!*, 3) and strings ("This is only a test.").

Pointers don't hold information; instead, each one contains the address of some location in the computer's memory that does hold information.

Float Type

C supports these four basic data types in various forms. You've already used two of them: integers (**int**) and characters (**char**). Now you will modify your last program to use a third type: floating point (**float**).

Go into the Turbo C editor and change your program to look like this:

```
#include <stdio.h>

main()
{
   int    a,b;
   float  ratio;

   printf("Enter two numbers:  ");
   scanf("%d %d",&a,&b);
   ratio = a / b;
   printf("The ratio is %f \n",ratio);
}
```

Save this as RATIO.C by bringing up the menus and choosing the **File/Write To** command. Then press *Ctrl-F9* to compile and run the program. Enter two values (such as 10 and 3) and note the result (3.000000).

You were probably expecting an answer of 3.333333; why was the answer just 3? Because *a* and *b* are both of type **int**, so the result of *a/b* was of type **int**. That was converted to type **float** when you assigned it to *ratio*, but the conversion took place after the division, not before.

Go back and change the type of *a* and *b* to **float**; also change the format string `"%d %d"` in **scanf** to `"%f %f"`. Save the code (press *F2*), then compile and run. The result is now 3.333333, as you expected.

There are also two large versions of type **float**, known as **double** and **long double**. As you might have guessed, variables of type **double** are twice as large as variables of type **float**, and variables of type **long double** are even larger. This means that they have more significant digits and a larger range of exponents. The specific sizes and ranges of values for these types in Turbo C can be found in Chapter 11.

The Three ints

In addition to the type **int**, C supports **short int** and **long int**, usually abbreviated as **short** and **long**. The actual sizes of **short**, **int**, and **long** depend upon the implementation; all that C guarantees is that a variable of type **short** will not be larger (that is, will not take up more bytes) than one of type **long**. In Turbo C, these types occupy 16 bits (**short**), 16 bits (**int**), and 32 bits (**long**).

Unsigned

C allows you to declare certain types (**char**, **short**, **int**, **long**) to be **unsigned**. This means that instead of having negative values, those types only contain non-negative values (greater than or equal to zero).

Variables of those types can then hold larger values than signed types. For example, in Turbo C a variable of type **int** can contain values from –32768 to 32767; one of type **unsigned int** can contain values from 0 to 65535. Both take up exactly the same amount of space (16 bits, in this case); they just use it differently. Again, see Chapter 11 for specific details.

Defining a String

C does not support a separate string data type, but it does provide two slightly different approaches to defining strings. One is to use a *character array*; the other is to use a *character pointer*.

Using a Character Array

Choose the **Load** command from the **File** menu and bring your edited version of HELLO.C back in. Now edit it to appear as follows:

```
#include <stdio.h>
#include <string.h>

main()
{
   char  msg[30];

   strcpy(msg,"Hello, world\n");
   puts(msg);
}
```

The [30] after *msg* tells the compiler to set aside space for up to 29 characters, that is, an array of 29 **char** variables. (The 30th space will be filled by a *null character*—\0—often referred to in this user's guide as a *null terminator*.) The variable *msg* itself doesn't contain a character value; it holds the address (some location in memory) of the first of those 29 **char** variables.

When the compiler finds the statement

```
strcpy(msg,"Hello, world");
```

it does two things:

- It creates the string "Hello, world", followed by a null (\0) character (ASCII code 0) somewhere within the object code file.
- It generates the code to call a subroutine named **strcpy**, which copies the characters from that string, one at a time, into the memory location pointed to by *msg*. It does this until it copies the null character at the end of the "Hello, world" string.

When you call puts(msg), you pass the value in *msg*—the address of the first letter it points to—to **puts**. Then **puts** checks to see if the character at that address is the null character. If it is, then **puts** is finished; otherwise, **puts** prints that character, adds one (1) to the address, and checks for the null character again.

Because of this dependency on a null character, strings in C are known as being *null terminated*: a sequence of characters followed by the null character. This approach removes any arbitrary limit on the length of strings; instead, a string can be any length, as long as there is enough memory to hold it.

Using a Character Pointer

The second method you can use to define strings is a *character pointer*. Edit your program to look like this:

```
#include <stdio.h>
#include <string.h>

main()
{
   char   *msg;

   msg = "Hello, world\n";
   puts(msg);
}
```

The asterisk (*) in front of *msg* tells the compiler that *msg* is a pointer to a character; in other words, *msg* can hold the address of some character. However, the compiler sets aside no space to store characters and does not initialize *msg* to any particular value.

When the compiler finds the statement

```
msg = "Hello, world\n";
```

it does two things:

- As before, it creates the string `"Hello, world\n"`, followed by a null character somewhere within the object code file.
- It assigns the starting address of that string—the address of the character *H*—to *msg*.

The command `puts(msg)` works just as it did before, printing characters until it encounters the null character.

There are some subtle differences between the array and pointer methods for defining strings, which we'll talk about in the next chapter.

Identifiers

Up until now, we've been giving names to variables without worrying about whatever restrictions there might be. Let's talk about those restrictions now.

The names you give to constants, data types, variables, and functions are known as *identifiers*. Some of the identifiers used so far include:

char, int, float	predefined data types
main	main function of program
name, a, b, sum, msg, ratio	user-defined variables
scanf, printf, puts	predeclared functions

Turbo C has a few rules about identifiers; here's a quick summary:

- All identifiers must start with a letter (*a...z* or *A...Z*) or an underscore (_).
- The rest of an identifier can consist of letters, underscores, and/or digits (0...9). No other characters are allowed.
- Identifiers are *case-sensitive*. This means that lowercase letters (*a...z*) are *not* the same as uppercase letters (*A...Z*). For example, the identifiers *indx*, *Indx*, and *INDX* are different and distinct from one another.
- The first 32 characters of an identifier are significant.

Operations

Once you get that data into the program (and into your variables), what are you going to do with it? Probably manipulate it somehow, using the operators available. And C has lots and lots of operators.

The Assignment Operator

The most basic operation is *assignment*, as in `ratio = a/b` or `ch = getch()`. In C, assignment is a single equal sign (=); the *value* on the right of the equal sign is assigned to the *variable* on the left.

You can stack up assignments, such as `sum = a = b`. In a case like this, the order of evaluation is right to left, so that *b* would be assigned to *a*, which in turn would be assigned to *sum*, giving all three variables the same value (namely, *b*'s original value).

Unary and Binary Operators

C supports the usual set of binary arithmetic operators:

- multiplication (*)
- division (/)
- modulus (%)
- addition (+)
- subtraction (–)

Turbo C supports *unary minus* (`a + (-b)`), which performs an arithmetic negation. Turbo C also supports *unary plus*
(`a + (+b)`), as per the ANSI C standard.

Increment (++) and Decrement (--) Operators

C has some special unary and binary operators as well. The most well known unary operators are *increment* (++) and *decrement* (--). These allow you to use a single operator that *adds 1 to* or *subtracts 1 from* any value; the addition or subtraction can be done in the middle of an expression, and you can even decide whether you want it done before or after the expression is evaluated. Consider the following lines of code:

```
sum = a + b++;
sum = a + ++b;
```

The first says, "Add *a* and *b* together, assign the result to *sum*, and increment *b* by one." The second says, "Increment *b* by one, add *a* and *b* together, and assign the result to *sum*."

These are very powerful operators, but you have to be sure you understand them correctly before using them. Modify SUM.C as follows, then try to guess what its output will be before you run it.

```
#include <stdio.h>

main()
{
   int    a,b,sum;
   char   *format;

   format = "a = %d   b = %d   sum = %d \n";
   a = b = 5;
   sum = a   + b;  printf(format,a,b,sum);
   sum = a++ + b;  printf(format,a,b,sum);
   sum = ++a + b;  printf(format,a,b,sum);
   sum = --a + b;  printf(format,a,b,sum);
   sum = a-- + b;  printf(format,a,b,sum);
   sum = a   + b;  printf(format,a,b,sum);
}
```

Bitwise Operators

For bit-level operations, C has the following operators:

- shift left (<<)
- shift right (>>)
- AND (&)
- OR (|)
- XOR (^)
- NOT (~)

These allow you to perform very low-level operations on values. To see the effect of these operators, type in and run this program:

```
#include <stdio.h>

main()
{
   int   a,b,c;
   char  *format1,*format2;

   format1 = " %04X %s %04X = %04X\n";
   format2 = " %c%04X = %04X\n";
   a = 0x0FF0;  b = 0xFF00;
   c = a << 4;  printf(format1,a,"<<",4,c);
   c = a >> 4;  printf(format1,a,">>",4,c);
   c = a & b;   printf(format1,a,"& ",b,c);
   c = a | b;   printf(format1,a,"| ",b,c);
   c = a ^ b;   printf(format1,a,"^ ",b,c);
   c = ~a;      printf(format2,'~',a,c);
   c = -a;      printf(format2,'-',a,c);
}
```

Again, see if you can guess the output of this program before running it. Note that field-width specifiers have been used to nicely align the output; the `%04X` specifier says that we want the output to use leading zeros, to be four digits wide, and to be in hexadecimal (base 16).

Combined Operators

C allows you to use a little shorthand when writing expressions that contain multiple operators. You can combine the assignment operator (=) with the operators discussed so far (unary, binary, increment, decrement, and bitwise).

Just about any expression of the form

```
<variable> = <variable> <operator> <exp>;
```

can be replaced with

```
<variable> <operator> = <exp>;
```

Here are some examples of such expressions and how they can be condensed:

```
a = a + b;     is condensed to a += b;
a = a – b;     is condensed to a –= b;
a = a * b;     is condensed to a *= b;
a = a / b;     is condensed to a /= b;
a = a % b;     is condensed to a %= b;
a = a << b;    is condensed to a <<= b;
a = a >> b;    is condensed to a >>= b;
a = a & b;     is condensed to a &= b;
a = a | b;     is condensed to a |= b;
a = a ^ b;     is condensed to a ^= b;
```

Address Operators

C supports two special address operators: the *address-of* operator (&) and the *indirection* operator (*).

The & operator returns the address of a given variable; if *sum* is a variable of type **int**, then &*sum* is the address (memory location) of that variable. Likewise, if *msg* is a pointer to type **char**, then **msg* is the character to which *msg* points.

Type in the following program and see what you get.

```
#include <stdio.h>

main()
{
   int   sum;
   char  *msg;

   sum = 5 + 3;
   msg = "Hello, there\n";
   printf(" sum = %d  &sum = %p \n",sum,&sum);
   printf("*msg = %c   msg = %p \n",*msg,msg);
}
```

The first line prints out two values: the value of *sum* (8) and the address of *sum* (assigned by the compiler). The second line also prints out two values: the character to which *msg* points (*H*) and the value of *msg*, which is the address of that character (also assigned by the compiler).

Input

C has several input functions; some take input from a file, others from the keyboard. When you need detailed information about the Turbo C input functions, refer to the entries on **...scanf** and **read** in the *Reference Guide,* and Chapter 11 in this manual.

The scanf Function

For interactive input, you'll probably use **scanf** most of the time. **scanf** is the input analog to **printf**; its format is

```
scanf(<format string>,<addr>,<addr>,...)
```

scanf uses many of the same %<letter> formats that **printf** does: %d for integers, %f for floating-point values, %s for strings, and so on.

However, there is one important difference with **scanf**: The items following the format string must be addresses, not values. The program SUM.C contains the following call:

```
scanf("%d %d",&a,&b);
```

This call tells the program that it expects you to type in two decimal (integer) values separated by a space; the first will be stored at the address associated with *a* and the second at the address associated with *b*. Note that it uses the address-of (&) operator to pass the addresses of *a* and *b* to **scanf**.

Whitespace

The space between the two %d format commands actually represents more than just a space. It indicates that you can have any amount of *whitespace* between the values. What is whitespace? Any combination of blanks, tabs, and newlines. C compilers and programs typically ignore whitespace in most circumstances.

But what if you wanted to separate the numbers with a comma instead of a blank? Then you could change the line to read:

```
scanf("%d,%d",&a,&b);
```

This allows you to enter the values with a comma between them.

Passing an Address to scanf

What if you want to input a string? Type in and run the following program:

```
#include <stdio.h>

main()
{
   char   name[150];

   printf("What is your name:  ");
   scanf("%s",name);
   printf("Hello, %s\n",name);
}
```

Since *name* is an array of characters, the value of *name* is the address of the array itself. Because of that, you don't use the & operator in front of *name*; you simply say `scanf("%s",name)`.

Note that we used the array approach (`char name[150];`) rather than the pointer approach (`char *name;`). Why? Because the array declaration actually sets aside memory to hold the string, while the pointer declaration does not. If we wanted to use `char *name`, then we'd have to explicitly allocate memory for **name*.

Using gets and getch for Input

Using **scanf** to input strings introduces another problem, though. Run your program again, but this time type in your full name. Note that the program only uses your first name in its reply. Why? Because, to **scanf**, the blank you typed after your first name signaled the end of the string you were entering.

There are two possible solutions to this. Here's the first:

```
#include <stdio.h>

main()
{
   char   first[30],middle[30],last[30];

   printf("What is your name:  ");
   scanf("%s %s %s",first,middle,last);
   printf("Hello, Dr. %s, or should I say %s?\n",last,first);
}
```

This, of course, assumes that you have some middle name; in this example, **scanf** won't continue until you've actually typed in three strings. But what if you want to read in the entire name as a single string, blanks and all?

Here's the second solution:

```
#include <stdio.h>

main()
{
   char   name[150];

   printf("What is your name:  ");
   gets(name);
   printf("Hello, %s\n",name);
}
```

The function **gets** reads in everything you type until you press *Enter*. It does not store the *Enter* in the line; but it does stick a null character (\0) at the end.

Finally, there's the function **getch**. It reads a single character from the keyboard without echoing it to the screen (unlike **scanf** and **gets**). Note that it doesn't take a character parameter; instead **getch** is a function of type **char**, and its value can be assigned directly to *ch*.

Conditional Statements

There are some operators we haven't talked about yet: *relational* and *logical* operators. There are also some complexities about expressions that we saved for this discussion of conditional (true or false) statements.

Relational Operators

Relational operators allow you to compare two values, yielding a result based on whether the comparison is true or false. If the comparison is false, then the resulting value is 0; if true, then the value is 1. Here's a list of the relational operators in C:

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

Why would you care if something were true or false? Load and run the program RATIO.C and see what happens when you enter 0 for the second value. Your program prints a `Divide by zero` error message and halts.

Now make the following changes to your program and run it again.

```
#include <stdio.h>

main()
{
   float  a,b,ratio;

   printf("Enter two numbers:  ");
   scanf("%f %f",&a,&b);
   if (b == 0.0)
      printf("The ratio is undefined\n");
   else {
      ratio = a / b;
      printf("The ratio is %f \n",ratio);
   }
}
```

The statement on the two lines after the call to **scanf** is known as an *if statement*. You can read it as: "If the value of the expression ($b == 0.0$) is true, immediately call **printf**. If the value of the expression is False, assign a/b to *ratio*, then call **printf**."

Now if you enter 0 as the second value, your program prints the message

```
The ratio is undefined
```

in the Execution screen, then returns to Turbo C. If the second value is nonzero, the program calculates and prints out the ratio.

Logical Operators

There are also three logical operators: AND (&&), OR (||), and NOT (!). These are not to be confused with the bitwise operators (&, |, ~) previously described. These logical operators work with logical values (true and false), allowing you to combine relational expressions.

How do they differ from the corresponding bitwise operators?

- These logical operators always produce a result of either 0 (false) or 1 (true), while the bitwise operators do true bit-by-bit operations.
- The logical operators && and || will *short circuit*. Suppose you have the expression `exp1 && exp2`. If *exp1* is false, then the entire expression is false, so *exp2* will never be evaluated. Likewise, given the expression `exp1 || exp2`, *exp2* will never be evaluated if *exp1* is true.

More About Expressions

Before we go on to loops, we have a few more comments about expressions. Things like `(b == 0.0)` and `(a <= q*r)` are pretty straightforward. However, C allows you to make things more complicated than that.

Assignment Statements

Any assignment statement enclosed in parentheses is an expression that has the same value as that which was being assigned.

For example, the expression `(sum = 5+3)` has the value 8, so that the expression `((sum = 5+3) <= 10)` would always yield a value of true (1) (since 8 <= 10).

More exotic is this example:

```
if ((ch=getch()) == 'q')
   puts("Quitting, huh?\n");
else
   puts("Good move; we'll have another go at it\n");
```

Can you figure out what this does? When your program hits the expression `((ch=getch()) == 'q')`, it stops until you press a character, assigns that character to *ch*, then compares that same character to the letter *q*. If the character you pressed equals *q*, then the message `"Quitting, huh?"` is printed on the screen; otherwise, the other message (`"Good move..."`) is printed.

The Comma Operator

You can use the comma operator (,) to put multiple expressions inside a set of parentheses. The expressions are evaluated from left to right, and the entire expression assumes the value of the last one evaluated. For example, if *oldch* and *ch* are both of type **char**, then the expression

```
(oldch = ch, ch = getch())
```

assigns *ch* to *oldch*, gets a character from the keyboard and assigns it to *ch*, and then assumes the (assigned) value of *ch*.

For example,

```
ch = 'a';
if((oldch = ch, ch = 'b') == 'a')
   puts("aye");
else
   puts("bee");
```

The if...else Statement

Look again at the **if** statement in the previous examples. The **if** statement takes the following generic format:

```
if(value)
   statement1;
else
   statement2;
```

where *value* is any expression that resolves to (or can be converted to) an integer value. If *value* is nonzero (true), then *statement1* is executed; otherwise, *statement2* is executed.

We must explain two important points about **if...else** statements in general.

First, the `else statement2` portion is optional; in other words, this is a valid **if** statement:

```
if (value)
   statement1;
```

In this case, *statement1* is executed if and only if *value* is nonzero. If *value* is zero, then *statement1* is skipped, and the program continues.

Second, what if you want to execute more than one statement if a particular expression is true (or false)? Answer: use a *compound statement*. A compound statement consists of

- a left brace ({)
- some number of statements, each ending with a semicolon (;)
- a right brace (})

The ratio example uses a single statement for the **if** clause

```
if (b == 0.0)
   printf("The ratio is undefined\n");
```

and a compound statement for the **else** clause

```
else {
   ratio = a / b;
   printf("The ratio is %f \n",ratio);
}
```

You might also notice that the body of your program (the function **main**) is simply a compound statement.

Loops

Just as there are statements (or groups of statements) that you want to execute conditionally, there are other statements that you may want to execute repeatedly. This kind of construct is known as a *loop*.

There are three basic kinds of loops (though two are just special cases of the other one): the **while** loop, the **for** loop, and the **do...while** loop. We'll cover them in that order.

The while Loop

The **while** loop is the most general loop and can be used to replace the other two; in other words, a **while** loop is all you need, and the others are just there for your convenience. Load up HELLO.C and modify it as follows:

```
#include <stdio.h>

main()
{
   int   len;

   len = 0;
   puts("Type in a sentence, then press <Enter>");
   while (getchar() != '\n')
      len++;
   printf("\nYour sentence was %d characters long\n",len);
}
```

This program lets you type in a sentence and counts the number of keystrokes, until you press *Enter* (\n). It then tells you how many characters (not counting the *Enter*) you typed.

The format of the **while** statement is

```
while (expression)
   statement
```

where *expression* resolves to a zero or nonzero value, and *statement* is either a single or a compound statement.

The **while** loop evaluates *expression*. If it's true, then *statement* is executed, and *expression* is evaluated again. If *expression* isn't true, the **while** loop is finished and the program continues on.

Take a look at another example of the **while** loop, based on HELLO.C:

```
#include <stdio.h>

main()
{
   char   *msg;
   int    indx;

   msg = "Hello, world";
   indx = 1;
   while (indx <= 10) {
      printf("time #%2d:  %s\n",indx,msg);
      indx++;
   }
}
```

When you compile and run this program, it prints out the following lines:

```
time # 1: Hello, world
time # 2: Hello, world
time # 3: Hello, world
```

and so on, down to

```
time #10: Hello,  world
```

The **printf** statement was executed exactly ten times, with *indx* going from 1 to 10 during those ten executions.

If you think about it, you may see a way to write that loop a little tighter:

```
indx = 0;
while (indx++ < 10)
   printf("time #%2d:  %s\n",indx,msg);
```

Study this second **while** loop until you understand why it functions exactly the same way as the first version. Then go on and learn about the **for** loop.

The for Loop

The **for** loop is the one found in most major programming languages, including C. However, the C version of the **for** loop is very flexible and very powerful, as you'll see.

The basic idea is that you execute a set of statements some fixed number of times while a variable (known as the *index variable*) steps through a range of values.

For example, modify the previous program to read as follows:

```
#include <stdio.h>

main()
{
   char   *msg;
   int    indx;

   msg = "Hello, world";
   for (indx = 1; indx <= 10; indx++)
      printf("time #%2d:  %s\n",indx,msg);
}
```

As you can see when you run it, this does the same thing as both **while** loops already shown and, in fact, is precisely equivalent to the first one. Here's the generic format of the **for** loop statement:

```
for (exp1; exp2; exp3)
   statement
```

As with **while**, the **for** statement executes just one statement, but that statement can be a compound statement ({...}).

Note what's inside the parentheses following the word **for**; there are three sections separated by semicolons.

- *exp1* is usually an assignment to the index variable.
- *exp2* is a test for loop continuation.
- *exp3* is usually some modification of the index variable.

The generic **for** loop is equivalent to the following code:

```
exp1;
while (exp2) {
   statement;
   exp3;
}
```

You can leave out any or all of the expressions, though the semicolons must remain. If you leave out *exp2*, it is assumed to have a value of 1 (true), so the loop never terminates (this is known as an *infinite loop*).

On the other hand, you can use the comma operator to put in multiple expressions for each expression.

For example, try these modifications on HELLO.C:

```
#include <stdio.h>

main()
{
   char  *msg;
   int   up,down;

   msg = "Hello, world";
   for (up = 1, down = 9; up <= 10; up++, down--)
      printf("%s: %2d down, %2d to go\n",msg,up,down);
}
```

Note that the first and last expressions in this **for** loop have two expressions each, initializing and modifying the variables *up* and *down*. You can make these expressions arbitrarily complex. (Perhaps you have heard the legends of C hackers who crammed most of their programs into the three expressions of a **for** statement, leaving only a few statements for the loop to execute.)

The do...while Loop

The final loop is the **do...while** loop. Modify RATIO.C as follows:

```
#include <stdio.h>
#include <conio.h>

main()
{
   float  a,b,ratio;

   do {
      printf("\nEnter two numbers:  ");
      scanf("%f %f",&a,&b);
      if (b == 0.0)
         printf("The ratio is undefined\n");
      else {
         ratio = a / b;
         printf("The ratio is %f \n",ratio);
      }
      printf("Press 'q' to quit, any other key to continue");
```

```
    } while (getch() != 'q');
}
```

This program calculates a ratio, then asks you to press a key. If you press *q*, the expression at the bottom is False and the loop ends. If you press some key other than *q*, the expression is True and the loop repeats.

Here's the generic format for the **do...while** loop:

```
do statement while (exp);
```

The main difference between the **while** loop and the **do...while** loop is that the statements in the **do...while** loop always execute at least once. This is similar to the **repeat...until** loop in Pascal, with one major difference: The **repeat** loop executes *until* its condition is true; **do...while** executes *while* its condition is true.

Functions

You've learned how to execute code *conditionally* and *iteratively*. Now, what if you want to perform the same set of instructions on different sets of data or at different locations in your program? Answer: You put those statements into a *subroutine*, which you then call as needed.

In C, all subroutines are known as *functions*. In theory, every function returns some value. In practice, the values returned by many functions are ignored, and more recent definitions of C (including the draft ANSI C standard and Turbo C) allow you to declare functions of type **void**, which means they don't return values at all.

In C, you can both *declare* and *define* a function. When you declare a function, you let the rest of your program know about it so that other functions (including **main**) can call it. When you define a function, you give the actual code for the function itself. For example, consider this rewrite of RATIO.C:

```
#include <stdio.h>
#include <conio.h>

/* Function declarations */

void  get_parms(float *p1, float *p2);
float get_ratio(float dividend, float divisor);
void  put_ratio(float ratio);

const float    INFINITY  = 3.4E+38;

/*  Main function: starting point for program */
```

```
main()
{
   float  a,b,ratio;
   do {
      get_parms(&a,&b);                                    /* Get parameters */
      ratio = get_ratio(a,b);                              /* Calculate ratio */
      put_ratio(ratio);                                    /* Print answer out */
      printf("Press q to quit, any other key to continue ");
   } while (getch() != 'q');
}  /* End of main */

/* Function definitions */
void  get_parms(float *p1,float *p2)
{
   printf("\nEnter two numbers:  ");
   scanf("%f %f",p1,p2);
}

float get_ratio(float dividend, float divisor)
{
   if (divisor == 0.0)
      return (INFINITY);
   else
      return (dividend / divisor);
}

void put_ratio(float ratio)
{
   if  (ratio == INFINITY)
      printf("The ratio is undefined\n");
   else
      printf("The ratio is %f\n",ratio);
}
```

Breaking Down the Program

The first three lines of the program are the function declarations; their purpose is to declare the function type as well as the type and number of the parameters for purposes of error-checking.

The next line defines a floating-point constant called INFINITY (it is a C convention to name constants in uppercase). This constant has a very high positive value—about the highest you can have with type **float**—and is used to flag a divide-by-zero. Note that since it is declared outside of all functions, it is "visible" inside all of the functions (including **main**).

Next comes the function **main**, which is the main body of your program. Every C program has a function called **main**; when your program starts executing, **main** is called, and everything proceeds from there. Once **main** is through executing, your program is finished, and you return to Turbo C (or, if you executed from a DOS prompt, to DOS).

The function **main** can be placed anywhere in the program; often it's the first function, following any prototypes or other global declarations. That makes it easy to find and helps to document the function of the entire program.

After **main** come the actual definitions of the three functions declared in the prototypes: **get_parms**, **get_ratio**, and **put_ratio**. Let's take a look at each of these definitions.

The get_parms Function

The **get_parms** function doesn't return a value of a given type, so we've declared it to be of type **void**. Its purpose is to read in two values and store them somewhere. Where? We have to pass two parameters to **get_parms**; these parameters are the addresses where the values should be stored. Look carefully: The two parameters are not of type **float** but are pointers to type **float**. In other words, they are supposed to be addresses of **float** variables.

That's exactly what we pass: When we call **get_parms** in **main**, the parameters are *&a,&b* instead of just *a,b*. Notice also that when **scanf** is called inside of **get_parms**, there are no address-of operators in front of *p1* and *p2*. Why? Because *p1* and *p2* are addresses already; they're the addresses of *a* and *b*.

The get_ratio Function

The **get_ratio** function does return a value (of type **float**) calculated from the two **float** values passed to it (*dividend* and *divisor*). The value returned depends upon whether or not *divisor* is 0. If it is, **get_ratio** returns INFINITY. If *divisor* is not 0, **get_ratio** returns the actual ratio. Note the format of the **return** statement.

The put_ratio Function

The **put_ratio** function doesn't return a value, so it is of type **void**. It has just a single parameter—*ratio*—which is used to determine what to print to

the screen. If *ratio* equals INFINITY, then it is considered undefined; otherwise, *ratio* is printed out.

Global Declarations

Constants, data types, and variables declared outside of any function (including **main**) are considered to be *global* from that point on. This means that they can be used by any function in the entire program following their declaration. If you were to move the declaration of INFINITY to the end of the program, you would get two compiler errors, one in **get_ratio** and one in **put_ratio**, for using an undeclared identifier.

Function Declarations

You can use two different styles in declaring functions: the "classic" style and the "modern" style. The classic style, found in many C texts and programs, takes this form:

```
type funcname();
```

This specifies the function's name (*funcname*) and the type of data value it returns (*type*). It does not give any parameter information, so no error-checking can be done. If you rewrote the function declarations in RATIO.C using this style, they would look like this:

```
void  get_parms();
float get_ratio();
void  put_ratio();
```

The modern style uses a construct from the ANSI extensions known as a *function prototype*. This declaration adds parameter information:

```
type  funcname(pinfo,pinfo,etc.);
```

where *pinfo* takes one of the following formats:

```
type
type pname
...
```

In other words, for each formal parameter you can specify just the data type, or you can give it a name as well. If the function takes a variable number of parameters, then you can use the ellipsis (...) for the last parameter.

This is the preferred approach, since it allows the compiler to check the numbers and types of the parameters in actual calls to the function. This approach also allows the compiler to perform proper conversions when possible. The function declarations found in the previous version of RATIO.C are function prototypes. More information about function prototypes can be found in Chapters 11 and 12.

Function Definitions

As with function declarations, there are two styles of function definitions: *classic* and *modern*.

The classic format of a function definition is like this:

```
type  funcname(pnames)
parm definitions;
{
   local declarations;
   statements;
}
```

The modern format moves the parameter definitions into the parentheses following *funcname*:

```
type funcname(pinfo,pinfo,etc.)
```

In this example, however, the term *pinfo* represents all the information about a given parameter; its type modifiers *and* identifier name. This makes the first line of the function definition look just like the corresponding function prototype, with one important exception: There is no semicolon (;) following the definition, whereas a function prototype is always ended by a semicolon. For example, the function **get_parms** in the classic style looks like this:

```
void get_parms(p1, p2)
float *p1; float *p2;
{ ... }
```

and in the modern style it looks like this:

```
void get_parms (float *p1, float *p2)
{ ... }
```

Note that any declarations (constants, data types, variables) made within a given function (including **main**) are visible (that is, can be used and referenced) only within that function. Also note that C does not allow

nested functions; you can't declare one function inside another, the way you can in, say, Pascal.

Functions can also be placed in any order in the program and are considered global throughout the entire program, including within functions declared prior to those being used. Be careful using a function before it's defined or declared: When the compiler encounters a function it hasn't seen before, it assumes the function returns an **int**. If you later define it to return something else, say a **char***, you'll get an error.

Comments

Sometimes, you want to insert notes in your program to remind you (or inform someone else) of what certain variables mean, what certain functions or statements do, and so on. These notes are known as *comments*. C, like most other programming languages, allows you to insert comments into your program. The compiler does not compile a comment. It recognizes the special character sequences that begin and end it and ignores any text between them.

To start a comment, you put in the slash-star character sequence (/*). From then on, the compiler will skip over everything until it encounters the star-slash (*/) sequence that ends the comment.

Comments can even extend across multiple lines, like this:

```
/* This is a long
comment, extending
over several lines. */
```

Look in the expanded version of RATIO.C on page 180 for additional examples of comments.

Summary

In this chapter we have introduced you to the seven basic elements of programming and shown you how you use each of them in Turbo C.

In Chapter 7 we will have more to say about them, and about other Turbo C features.

C H A P T E R

7

More Programming in Turbo C

In the last chapter, we gave you a taste of working with Turbo C, just enough to whet your appetite. Now you're ready to dig into some of the more subtle and esoteric issues of C programming, and we're here to help you.

In This Chapter...

In this chapter, we cover the following:

- Data structures (including pointers, arrays, and structures)
- The **switch** statement
- Control flow commands, including **return**, **break**, **continue**, **goto**, and the conditional expression operator (? :)
- Streams and stream I/O: how to read from and write to a disk file or hardware device
- Programming style in C, especially with regards to some of the new C extensions
- Some common pitfalls for C programmers

A Survey of Data Structures

We covered basic data types in the last chapter—things such as integers, floating-point numbers, characters, and their variants. Now we are going to talk about how to use these elements to build data *structures*—collections of data elements. But first, we'll explore an important concept in C—*pointers*.

Pointers

Most variables you've looked at so far hold data, that is, the actual information your program is manipulating. But sometimes you want to keep track of where some data is, rather than just its value. For that, you probably need pointers.

If you feel shaky about the concepts of addresses and memory, here's a quick review. Your computer holds your program and the associated data in its memory (often called RAM, meaning **R**andom **A**ccess **M**emory). At its lowest level, your computer's memory is composed of *bits*, microscopic electronic circuits that can "remember" (while the computer's power is on) one of two values, which are usually interpreted as being 0 and 1.

Eight bits are grouped together into 1 *byte*. Groups of bytes are often given names as well; commonly, 2 bytes is considered a *word*; 4 bytes is considered a *longword*; and on the IBM PC, 16 bytes is considered a *paragraph*.

Each byte in your computer's memory has a unique address, much as does each house on a given street. But unlike most streets, consecutive bytes have consecutive addresses; if a given byte has an address of N, then the preceding byte has an address of N-1, and the following byte has an address of N+1.

A *pointer* is a variable that holds the address of some data, rather than the data itself. Why is this useful? First, you can use a pointer to point to different data and different data structures. By changing the address the pointer contains, you can manipulate (assign, retrieve, change) information in various locations. This allows you, for example, to traverse a linked list of structures with only one pointer.

Second, using pointers allows you to create new variables while your program is executing. C lets your program ask for some amount of memory (in bytes), returning an address that you can store in a pointer. This is known as *dynamic memory allocation*; using it, your program can adapt to how much (or little) memory is available on a given computer.

Third, you can use a pointer to access different locations in a data structure, such as an array, a string, or a structure. A pointer really points to just one location in memory (the sum of a segment and its offset); by indexing the pointer, you can access any succeeding byte(s).

You're undoubtedly convinced now that pointers are handy. So how do you use them in C? First, you have to declare them. Consider the following program:

```
main()
{
   int ivar,*iptr;

   iptr = &ivar;
   ivar = 421;
   printf("location of ivar: %p\n",&ivar);
   printf("contents of ivar: %d\n", ivar);
   printf("contents of iptr: %p\n", iptr);
   printf("value pointed to: %d\n",*iptr);
}
```

This **main** has declared two variables: *ivar* and *iptr*. The first, *ivar*, is an integer variable; that is, it holds a value of type **int**. The second, *iptr*, is a *pointer* to an integer variable; that is, it holds an *address* of a value of type **int**. You can tell that *iptr* is a pointer because it has an asterisk (*) in front of it when it is declared. In C, this asterisk is known as the *indirection operator*.

In **main**, these assignments are as follows:

- the address of *ivar* is assigned to *iptr*
- the integer value 421 is then assigned to *ivar*

The address-of operator (&) mentioned in the previous chapter gets the address of *ivar*.

Type in and run the preceding program; you'll get output that looks like this:

```
location of ivar: 166E
contents of ivar: 421
contents of iptr: 166E
value pointed to: 421
```

The first two lines show the address and contents of *ivar*. The third shows the address that *iptr* contains. As you can see, it's the address of the variable *ivar*, that is, the location in memory where your program decided to create *ivar*. The last value printed is the data stored at that address, the same data already assigned to *ivar*.

Note that the third call to **printf** used the expression *iptr* to get its contents, the address of *ivar*. Then the last **printf** call used the expression **iptr* to fetch the data stored at that address.

Here's a slight variation on the previous program.

```
main()
{
   int ivar,*iptr;

   iptr  = &ivar;
   *iptr = 421;
   printf("location of ivar: %p\n",&ivar);
   printf("contents of ivar: %d\n", ivar);
   printf("contents of iptr: %p\n", iptr);
   printf("value pointed to: %d\n",*iptr);
}
```

This still assigns the address of *ivar* to *iptr,* but instead of assigning 421 to *ivar,* **main** assigns it to **iptr*. The results? Exactly the same as the previous program. Why? Because the statement `*iptr = 421` is the same as the statement `ivar = 421`. And why is that so? Because *ivar* and **iptr* refer to the same memory location—so both statements assign the value 421 to that location.

Dynamic Allocation

Here's another variation of the program:

```
#include <alloc.h>

main()
{
   int *iptr;

   iptr = (int *) malloc(sizeof(int));
   *iptr = 421;
   printf("contents of iptr: %p\n", iptr);
   printf("value pointed to: %d\n",*iptr);
}
```

This version dropped the declaration of *ivar* altogether. Instead, it's assigning to *iptr* the value returned by some function named **malloc**, which is declared in alloc.h (hence the `#include` directive at the start). It then assigns the value 421 to **iptr,* which is the address *iptr* points to. If you run this program, you'll get a different value for *iptr* than you did before, but **iptr* will still be 421.

What does the statement `iptr = (int *) malloc(sizeof(int))` do? We'll break it down one part at a time.

- The expression `sizeof(int)` returns the number of bytes that a variable of type **int** requires; using Turbo C on the IBM PC, the value it yields is 2.
- The function **malloc(*num*)** grabs *num* consecutive bytes of the available (unused) memory in your computer. It then returns the starting address of those bytes.
- The expression `(int *)` means you will consider that starting address to be a pointer to type **int**. This is known as *type-casting*. In this case, Turbo C doesn't require it. But because many other C compilers do require it, if you leave it off, you will get the warning message `Nonportable pointer assignment`.
- Finally, this address is stored in *iptr*. This means you have *dynamically* created an integer variable, which you can refer to as **iptr*.

Given all this, the entire statement can be described as: "Allocate from the computer's memory enough space for a variable of type **int**, then assign the starting address of that memory to *iptr*, which is a pointer to type **int**."

Was all this necessary? Yes. Why? Because without it you would have no guarantee that *iptr* was pointing to an unused area of memory. *iptr* would have some value in it, and that is the address it would use, but you wouldn't know if that section of memory was being used for other reasons. The rule for using pointers is simple: **Always assign an address to a pointer before using it.** Rather, don't assign an integer value to **iptr* without first assigning an address to *iptr*.

Pointers and Functions

In the last chapter, we explained how you declare parameters for functions. Perhaps now you understand why you use pointers for formal parameters whose values you wish to change. For example, consider the following function:

```
void  swap(int *a, int *b)
{
   int  temp;
   temp = *a;   *a = *b;   *b = temp;
}
```

This function, **swap**, has declared the two formal parameters, *a* and *b*, to be pointers to **int**. This means they expect an address of an integer variable

(rather than its value) to be passed. Any changes made are made to the data at the addresses passed in.

Here's a **main** function that calls **swap**:

```
main()
{
   int  i,j;

   i = 421;
   j = 53;
   printf("before: i = %4d  j = %4d\n",i,j);
   swap(&i,&j);
   printf("after: i = %4d  j = %4d\n",i,j);
}
```

You'll notice that this program does indeed swap the values of *i* and *j*. You can think of this program as being the equivalent of:

```
main()
{
   int  i,j;
   int  *a,*b,temp;

   i = 421;
   j = 53;
   printf("before: i = %4d j = %4d\n",i,j);
   a = &i;
   b = &j;
   temp = *a; *a = *b; *b = temp;
   printf("after: i = %4d  j = %4d\n",i,j);
}
```

This program, of course, produces the same results: The call `swap(&i,&j)` assigns the values (they are addresses) of the two actual parameters (*&i* and *&j*) to the two formal parameters (*a* and *b*), then executes the statements in **swap**.

Pointer Arithmetic

What if you wanted to modify the program so that *iptr* points to three integers instead of just one?

Here's one possible solution:

```
#include <alloc.h>

main()
{
   #define NUMINTS 3
   int  *list,i;

   list = (int *) calloc(NUMINTS,sizeof(int));
   *list = 421;
   *(list+1) = 53;
   *(list+2) = 1806;

   printf("list of addresses: ");
   for (i = 0; i<NUMINTS; i++)
      printf("%4p ", (list+i));

   printf("\nlist of values  : ");
   for (i = 0; i<NUMINTS; i++)
      printf("%4d ",*(list+i));

   printf("\n");
}
```

Instead of using **malloc**, this routine uses **calloc**, which takes two parameters: how many items to allocate space for and the size of each item in bytes. So now *list* points to a chunk of memory 6 (3 × 2) bytes long, big enough to hold three variables of type **int**.

Note very carefully the three statements that follow. The first statement is familiar: `*list = 421`. It simply says, "store 421 in the **int** variable located at the address in *list*."

The next statement (`*(list+1) = 53`) is important to understand. At first glance, you might interpret this as, "store 53 in the **int** variable located 1 byte beyond the address in *list*." If so, you're probably concerned, since this would be right in the middle of the previous **int** variable (which is 2 bytes long). This, of course, would alter the value that you previously stored.

Don't worry; your C compiler is more intelligent than that. It knows that *list* is a pointer to type **int**, and so the expression `list + 1` refers to the byte address of `list + (1 * sizeof(int))`, so that the value 53 does not clobber the value 421 at all.

Likewise, `(list+2)` refers to the byte address of `list + (2*sizeof(int))`, and 1806 gets stored without affecting the previous two values.

In general, *ptr + i* denotes the memory address *ptr + (i * sizeof(int))*.

Type in and run the preceding program; the output will look something like this:

```
list of addresses:  06AA 06AC 06AE
list of values:     421  53   1806
```

Note that the addresses are 2 bytes apart, not just 1, and the three values have been kept separate.

To sum up all of this: If you use *ptr*, a pointer to *type*, then the expression `(ptr + i)` denotes the memory address `(ptr + (i*sizeof(type))`, where `sizeof(type)` returns the number of bytes that a variable of *type* requires.

Arrays

Most high-level languages, including C, allow you to define *arrays*, that is, indexed lists of a given data type. For example, you can rewrite the last program to look like this:

```
main()
{
   #define NUMINTS 3
   int   list[NUMINTS],i;

   list[0] = 421;
   list[1] = 53;
   list[2] = 1806;
   printf("list of addresses: ");
   for (i = 0; i < NUMINTS; i++)
      printf("%p ",&list[i]);
   printf("\nlist of values  : ");
   for (i = 0; i < NUMINTS; i++)
      printf("%4d ", list[i]);
   printf("\n");
}
```

The expression `int list[NUMINTS]` declares *list* to be an array of **ints**, with space set aside for exactly three **int** variables. The first variable is referred to as *list[0]*, the second as *list[1]*, and the third as *list[2]*.

The general declaration for any array is

```
type name[size];
```

where *type* is some data type, *name* is the name you give the array, and *size* is the number of elements of *type* that *name* contains. The first element in the array is *name[0]*, while the last is *name[size-1]*; the total size of the array in bytes is *size* * (`sizeof(type)`).

Arrays and Pointers

You may have already figured out that there is a close relationship between arrays and pointers. In fact, if you run the previous program, your output will look very familiar:

```
list of addresses: 163A 163C 163E
list of values:    421  53   1806
```

The starting address is different, but that is the only change. The truth is, you can use the name of an array as if it were a pointer; similarly, you can index a pointer as if it were an array.

Consider the following important identities:

```
(list + i) == &(list[i])
*(list + i) == list[i]
```

In both cases, the expression on the left is equivalent to the expression on the right; you can use one in place of the other, regardless of whether you declared *list* as a pointer or as an array.

The only difference between declaring *list* as a pointer and declaring it as an array is in allocation. If you declare *list* as an array, your program automatically sets aside the requested amount of space. If you declare *list* as a pointer, you must explicitly create space for it using **calloc** or a similar function call, or you must assign to it the address of some space that has already been allocated.

Arrays and Strings

We talked about strings in the previous chapter and referred to declaring a string in two slightly different ways: as a pointer to characters and as an array of characters. Now you can better understand that difference.

If you declare a string as an array of **char**, the space for that string is allocated. If you declare a string as a pointer to **char**, no space is allocated; you must either allocate it yourself (using **malloc** or something similar) or assign to it the address of an existing string. An example of this is given in the section "Pitfalls in C Programming" later in this chapter.

Multi-Dimensional Arrays

Yes, you can have multi-dimensional arrays. They are declared like this:

```
type  name[size1] [size2]...[sizeN];
```

Consider the following program, which initializes a couple of two-dimensional arrays, then performs matrix multiplication on them:

```
main()
{
   int a[3][4] = { { 5,  3, -21, 42},
      { 44, 15,  0, 6},
      { 97,  6, 81, 2} };

   int b[4][2] = { { 22,  7},
      { 97, -53},
      { 45,  0},
      { 72,  1} };

   int c[3][2],i,j,k;
   for (i = 0; i < 3; i++) {
      for (j = 0; j < 2; j++) {
         c[i][j] = 0;
         for (k = 0; k < 4; k++)
            c[i][j] += a[i][k] * b[k][j];
      }
   }
   for (i = 0; i < 3; i++) {
      for (j=0; j<2; j++)
         printf("c[%d][%d] = %d ",i,j,c[i][j]);
      printf("\n");
   }
}
```

Take note of two things in the preceding program: The syntax for initializing a two-dimensional array consists of nested {...} lists separated by commas, and square brackets ([ ]) that are used around each index variable.

Some languages use the syntax [i,j]; that is legal syntax in C, but is the same as saying just [j], since the comma is interpreted as the comma operator ("evaluate *i*, then evaluate *j*, then let the entire expression assume the value of *j*"). Be sure to put square brackets around every index variable.

Multi-dimensional arrays are stored in what is known as *row-column order*. This means that the last index varies the most rapidly. In other words,

given the array *arr[3][2]*, the elements in *arr* are stored in the following order:

```
arr[0][0]
arr[0][1]
arr[1][0]
arr[1][1]
arr[2][0]
arr[2][1]
```

The same principle holds true for arrays of three, four, or more dimensions.

Arrays and Functions

What happens when you pass an array to a function?

Look at this function, which returns the *index* of the lowest value in an array of **int**:

```
int  lmin(int list[],int size)
{
   int  i, minindx, min;

   minindx = 0;
   min = list[minindx];

   for (i = 1; i < size; i++)
     if (list[i] < min) {
        min = list[i];
        minindx = i;
     }
   return(minindx);
}
```

Here you see one of the great strengths of C: You don't need to know how large *list[]* is at compile time. Why? Because the compiler is content to consider *list[]* to be the starting address of the array, and it doesn't really care where it ends. A call to the function **lmin** might look like this:

```
main()
{
   #define VSIZE 22
   int  i, vector[VSIZE];

   for (i = 0; i < VSIZE; i++) {
      vector[i] = rand();
      printf("vector[%2d] = %6d\n",i,vector[i]);
   }
   i = lmin(vector,VSIZE);
   printf("minimum: vector[%2d] = %6d\n",i,vector[i]);
}
```

Question: What exactly is passed to **lmin**? Answer: The starting address of *vector*. This means that if you were to make changes to *list* within *lmin*, those changes would be made to *vector* as well. For example, you could write the following function:

```
void setrand(int list[],int size);
{
   int i;
   for (i = 0; i < size; i++) list[i] = rand();
}
```

Then you could make the call `setrand(vector,VSIZE)` in **main** to initialize *vector*.

How about multi-dimensional arrays passed to functions? Do you have the same flexibility? Suppose you wanted to modify **setrand** to work on a two-dimensional array.

You'd have to do something like this:

```
void setrand(int matrix[][CSIZE],int rsize)
{
   int i,j;
   for (i = 0; i < rsize; i++) {
      for (j = 0; j < CSIZE; j++)
      matrix[i][j] = rand();
   }
}
```

CSIZE is a global constant fixing the size of the second dimension of the array. In other words, any array you passed to **setrand** would have to have a second dimension of *CSIZE*.

There is another solution, however. Suppose you have an array *matrix[15][7]* that you want to pass to **setrand**. If you use the original declaration of `setrand(int list[], int size)`, you can call it as follows:

```
setrand(matrix,15*7);
```

The array *matrix* will then look to **setrand** like a one-dimensional array of size 105 (which is 15 × 7), and everything will work just fine.

Structures

Arrays and pointers allow you to build lists of items of the same data type. What if you want to construct something out of different data types? Declare a *structure*.

A *structure* is a conglomerate data structure, a lumping together of different data types. For example, suppose you wanted to keep track of information about a star: name, spectral class, coordinates, and so on. You might declare the following:

```
typedef struct {
   char name[25];
   char class;
   short subclass;
   float decl,RA,dist;
} star ;
```

This defines the **struct** type *star*. Each of the variables declared within the structure (*name*, *class*, and so on) is a *member* of that structure.

Having declared it—that is, having placed the previous definition at the start of your program file—you can use it as a data type to declare *structure variables* of type **star**.

```
main()
{
   star mystar;

   strcpy(mystar.name,"Epsilon Eridani");
   mystar.class    = 'K';
   mystar.subclass =  2;
   mystar.decl     =  3.5167;
   mystar.RA       = -9.633;
   mystar.dist     =  0.303;

   /* Rest of function main() */
}
```

You refer to each member of a structure variable by preceding its name with the structure variable's name followed by a period. The construct *varname.memname* is considered equivalent to the name of a variable of the

the same type as *memname,* and you can use it to perform all the same operations.

Structures and Pointers

You can declare pointers to structures, just as you can declare pointers to other data types. This ability is essential for creating linked lists and other dynamic data structures. In fact, pointers to structures are used so often in C that there is a special symbol for referring to the member of a structure pointed to by a pointer.

Consider the following rewrite of the previous program.

```
#include <alloc.h>

main()
{
   star *mystar;

   mystar = (star *) malloc(sizeof(star));
   strcpy(mystar -> name,"Epsilon Eridani");
   mystar -> class = 'K';
   mystar -> subclass = 2;
   mystar -> decl = 3.5167;
   mystar -> RA = -9.633;
   mystar -> dist = 0.303;

   /* Rest of function main() */
}
```

This rewrite declares *mystar* to be a *pointer* to type *star,* rather than to be a variable of type *star*. It allocates space for *mystar* via the call to **malloc**. Now when you refer to the members of *mystar,* you use *ptrname* -> *memname*. The symbol -> means "member of the structure pointed to by"; it is a shorthand notation for *(*ptrname).memname.*

The switch Statement

You may find yourself building long **if..else if..else if..** constructs. Look at the following function:

```
#include <ctype.h>

do_main_menu(short *done)
{
   char  cmd;

   *done = 0;
   do {
      cmd = toupper(getch());
      if (cmd == 'F') do_file_menu(done);
      else if (cmd == 'R') run_program();
      else if (cmd == 'C') do_compile();
      else if (cmd == 'M') do_make();
      else if (cmd == 'P') do_project_menu();
      else if (cmd == 'O') do_option_menu();
      else if (cmd == 'E') do_error_menu();
      else handle_others(cmd, done);
   } while (!*done);
}
```

This is so common in programming that C has a special control structure for it: the **switch** statement. Here's that same function, but rewritten using the **switch** statement:

```
#include <ctype.h>

do_main_menu(short *done)
{
   char  cmd;

   *done = 0;
   do {
      cmd = toupper(getch());
      switch (cmd) {
        case 'F': do_file_menu(done); break;
        case 'R': run_program(); break;
        case 'C': do_compile(); break;
        case 'M': do_make(); break;
        case 'P': do_project_menu(); break;
        case 'O': do_option_menu(); break;
        case 'E': do_error_menu(); break;
        default : handle_others(cmd,done);
   } while (!*done);
}
```

This function enters a loop that reads in a character, converts it to uppercase, and stores it in *cmd*. It then executes the **switch** statement based on the value of *cmd*. The loop continues until the variable *done* gets assigned zero (presumably in the functions **do_file_menu** or **handle_others**).

The **switch** statement takes the value of *cmd* and compares it against each of the **case** labels. If there is a match, execution starts at that label and continues until either you encounter a **break** statement, or you reach the end of the **switch** statement. If there is no match, and you've included the label `default` in your **switch** statement, then execution starts there; if there is no default, then the entire **switch** statement is skipped.

In the **switch** statement, *value* must be *integer compatible*. In other words, it has to be easily converted to an integer; it can be a **char**, any **enum** type, and (of course) an **int** with all its variants. You cannot use reals (such as **float** and **double**), pointers, strings, or other data structures (though you can use integer-compatible elements of a data structure).

Although *value* can be any expression (constant, variable, function call, or any combination thereof), the case labels themselves have to be constants. What's more, you can only list one value per **case** keyword.

If **do_main_menu** hadn't used the function **toupper** to convert *cmd* to uppercase, then the **switch** statement might have looked like this:

```
switch (cmd) {
   case 'f':
   case 'F': do_file_menu(done);
      break;
   case 'r' :
   case 'R' : run_program();
      break;
   ...
```

This statement executes the function **do_file_menu** if *cmd* is either a lowercase or uppercase *F*, and so on for the rest of the options.

Remember, you must use the break statement when you're finished with a given case. Otherwise, the remaining statements will be executed (until, of course, you encounter a **break** statement). If you had left off the **break** statement following the call to **do_file_menu**, typing the letter *F* would result in a call to **do_file_menu**, followed by a call to **run_program**.

There are times when you want to do that though; consider this code:

```
typdef enum { sun, mon, tues, wed, thur, fri, sat } days;

main()
{
   days today;

   ...
   switch (today) {
      case mon:
      case tues:
      case wed:
      case thur:
      case fri: puts("go to work!"); break;
      case sat: printf("clean the yard and ");
      case sun: puts("relax!");
   }
   ...
}
```

With this **switch** statement, the values *mon* through *fri* all end up executing the same **puts** statement, after which the **break** statement causes you to leave the **switch**. However, if *today* equals *sat*, then the **printf** is executed, following which the `puts("relax!")` statement is executed; if *today* equals *sun*, then only the last **puts** is executed.

Control Flow Commands

There are additional commands for use within control structures or to simulate other control structures. The **return** statement lets you exit functions early. The **break** and **continue** statements are designed to be used within loops and help you skip over statements. The **goto** statement allows you to jump around in your code. And the conditional expression (?:) lets you compress certain **if...else** statements onto just one line.

A word of advice: Think twice before using these (except, of course, for **return**). There are situations where they represent the best solution, but more often than not you can solve your problem without resorting to them. Especially avoid the use of the **goto** statement; given the **return**, **break**, and **continue** statements, there shouldn't be much need for it.

The return Statement

There are two major uses of the **return** statement. First, if a function returns a value, you *must* use it in order to pass that value back to the calling routine.

For example,

```
int   imax(int a, int b);
{
   if (a > b)
      return(a);
   else
      return(b);
}
```

Here, the routine uses the **return** statement to pass back the larger of the two values accepted.

The second major use of the **return** statement is to exit a function at some point other than its end. For example, a function might detect a condition early on that requires that it terminate. Rather than put the rest of the function inside of an **if** statement, you can just call **return** to exit. If the function is of type **void**, you can use **return** with no value passed back at all.

Consider this modification of the **lmin** function given earlier:

```
int lmin(int list[], int size)
{
   int i, minindx, min;

   if (size <= 0)
      return(-1);
      ...
}
```

In this case, if the parameter *size* is less than or equal to zero, then there is nothing in *list*; therefore, **return** is called right off the bat, to get out of the function. Note that an error value of –1 is returned. Since –1 is never a valid index into an array, the calling routine knows that it did something wrong.

The break Statement

Sometimes you want to exit a loop quickly and easily, before you reach its end.

Consider the following program:

```
#define LIMIT 100
#define MAX  10

main()
{
   int  i,j,k,score;
   int  scores[LIMIT][MAX];

   for (i = 0; i < LIMIT; i++) {
      j = 0;
      while (j < MAX-1) {
         printf("please enter score #%d: ",j);
         scanf("%d",score);
         if (score < 0)
            break;
         scores[i][++j] = score;
      }
      scores[i][0] = j;
   }
}
```

Note the statement `if (score < 0) break;`. This says that if the user enters a negative value for the score, the **while** loop is terminated. The variable *j* is used both to index *scores* and to keep track of the total number of scores in each row; that count is then stored in the first element of the row.

You may recall the **break** statement from its use in the **switch** statement in the last chapter. In that case, it caused the program to exit the **switch** statement; here, it causes the program to exit the loop and proceed with the program. The **break** statement can be used with all three loops (**for**, **while**, and **do...while**), as well as in the **switch** statement; however, it *cannot* be used in an **if...else** statement or just in the main body of a function.

The continue Statement

Sometimes, you don't want to get out of the loop completely; you just want to skip the rest of the loop and start at the top again. In those situations, you can use the **continue** statement, which does just that.

Look at this program:

```
#define LIMIT 100
#define MAX  10

main()
{
   int  i,j,k,score;
   int  scores[LIMIT][MAX];

   for (i = 0; i < LIMIT; i++) {
      j = 0;
      while (j < MAX-1) {
         printf("please enter score #%d: ",j);
         scanf("%d",score);
         if (score < 0)
            continue;
         scores[i][++j] = score;
      }
      scores[i][0] = j;
   }
}
```

When the **continue** statement is executed, the program skips over the rest of the loop and does the loop test again. As a result, this program works differently from the one before. Instead of exiting the inner loop when the user enters a score of –1, it assumes that an error has been made and goes to the top of the **while** loop again. Since *j* has not been incremented, it asks for the same score again.

The goto Statement

Yes, there is a **goto** statement in C. The format is simple: `goto` *`label`*, where *label* is some identifier, associated with a given statement. However, most intelligent uses of the **goto** statement are taken care of by the three previous statements, so consider carefully whether you really need to use it.

The Conditional Operator (?:)

Suppose you want to choose between two expressions (and the resulting values), based on some condition.

As we have seen, this is usually accomplished with an **if..else** statement, such as

```
int   imin(int a, int b)
{
   if (a < b)
      return(a);
   else
      return(b);
}
```

The **if...else** statement is used often enough to warrant a special operator. Its format is

```
expr1 ? expr2 : expr3
```

This is interpreted as follows: "If *expr1* is true, then evaluate *expr2* and let the entire expression assume its value; otherwise, evaluate *expr3* and assume its value." Using this construct, you can rewrite the function **imin** as follows:

```
int  imin(int a, int b)
{
   return((a < b) ? a : b);
}
```

Better yet, you can rewrite **imin** as an inline macro:

```
#define imin(a,b) ((a < b) ? a : b)
```

Now whenever your program sees the expression `imin(e1,e2)`, it replaces it with `((e1<e2) ? e1 : e2)` and continues compilation. This is actually a more general solution, since *a* and *b* are no longer limited to being **int**; they can be any type that allows the < relationship.

Streams and Stream I/O

What Are Streams?

Streams are the most portable means for reading or writing data using Turbo C. They are designed to allow flexible and efficient I/O that is not affected by the underlying file or device hardware.

A stream is a file or physical device (a printer or monitor, for example) that you manipulate with a pointer to a FILE object (defined in stdio.h). The FILE object contains various information about the stream, including the

current position of the stream, pointers to any associated buffers, and error or end-of-file indicators.

Your program should never create or copy FILE objects themselves; instead, it should use the pointers returned from functions like **fopen**. Be sure that you do not confuse FILE pointers with DOS file handles (which are used in low-level DOS or UNIX-compatible I/O).

You must open a stream before you can perform I/O on it. Opening the stream connects it to the named DOS file or device. The routines that open streams are **fopen**, **fdopen**, and **freopen**. When you open a stream, you indicate whether you want to read or write to the stream, or do both. You also indicate whether you will treat the data of that stream as text or binary data. This last distinction is important because of a minor incompatibility between C stream I/O and DOS text files.

Text vs. Binary Streams

Text streams are used for normal DOS text files, such as a file created with the Turbo C editor. C stream I/O assumes that text files are divided into lines separated by a single newline character (which is the ASCII linefeed). DOS text files, however, are stored on disk with two characters between each line, an ASCII carriage-return and a linefeed. In text mode, Turbo C translates carriage-return/linefeed (CR/LF) pairs into a single linefeed on input; linefeeds are translated to CR/LF pairs on output.

Binary streams are much simpler than text streams. No such translations are performed. Any character is read or written without change.

A file can be accessed in either text or binary mode without any problems as long as you are aware of and understand the translations taking place in text streams. Turbo C doesn't "remember" how a file was created or last accessed.

If no translation mode is specified when a stream is opened, it is opened in the default translation mode given by the global variable *_fmode*. By default, *_fmode* is set to text mode.

Buffering Streams

Streams that are associated with files are typically buffered. This allows I/O at the individual character level, such as with **getc** and **putc**, to be very

fast. You can supply your own buffer, change the size of the buffer used, or force the stream to use no buffer at all by calling **setvbuf** or **setbuf**.

Buffers are automatically flushed when the buffer is full, the stream is closed, or the program terminates normally. You can use **fflush** and **flushall** to flush the buffers manually.

Normally, you use streams to read or write data sequentially. I/O takes place at the current file position. Whenever you read or write data, the program moves the file position to immediately after the just-accessed data. A stream that is connected to a disk file can also be accessed randomly. You can use **fseek** to position a file, then issue several read or write operations to access the data after that point.

When you are both reading and writing data to a stream, you should not freely mix reading and writing operations. You must flush the stream's buffer between reading and writing. A call to **fflush**, **flushall**, or **fseek** clears the buffer and allows you to switch operations. For maximum portability, you should flush even when no buffer is present, since other systems may have additional restrictions on mixing input and output operations even without a buffer.

Predefined Streams

In addition to streams created by calling **fopen**, five predefined streams are available whenever your program begins execution.

Name	I/O	Mode	Stream
stdin	Input	Text	Standard Input
stdout	Output	Text	Standard Output
stderr	Output	Text	Standard Error
stdaux	Both	Binary	Auxiliary I/O
stdprn	Output	Binary	Printer Output

The *stdaux* and *stdprn* streams are specific to DOS and are not portable to other systems.

The *stdin* and *stdout* can be redirected by DOS, while the others are connected to specific devices: *stderr* to the console (CON:), *stdprn* to the printer (PRN:), and *stdaux* to the auxiliary port.

The auxiliary port depends on your machine's configuration; it is typically COM1:. Consult your DOS documentation for information about

redirecting input or output on a DOS command line. Unless they are redirected, *stdin* and *stdout* are connected to the console (CON: device). Furthermore, if not redirected, *stdin* is line buffered, while *stdout* is unbuffered. The other predefined streams are unbuffered.

To process a predefined stream in a mode other than its default (for example, to process *stdprn* in text mode), use **setmode**. The predefined stream names are constants; you cannot assign values to them. If you want to reassociate one of them to a file or device, use **freopen**.

Style in C Programming: Modern vs. Classic

There is a trend today in C programming to introduce techniques that make C easier to use. This counteracts classic traditions or methods of C programming. Most have been made possible by language extensions defined by the ANSI C Standards Committee. This section should give you a feeling for how things have been done in the past and how the new standards can help you write better C programs.

Turbo C, of course, supports both the classic programming style and the modern style.

Using Function Prototypes and Full Function Definitions

In the classic style of C programming, you declare functions merely by specifying the name and type returned.

For example, you would define the function **swap** as

```
int  swap();
```

No parameter information is given, either as to number or type. The classic-style definition of the function looks like this:

```
int  swap(a,b)
int  *a,*b;
{
   /* Body of function */
}
```

This style results in very little error-checking, which in turn can result in some very subtle and hard-to-trace bugs. Avoid it.

The modern style involves the use of function prototypes for function declarations and parameter lists for function definitions.

Redeclare **swap** using a function prototype:

```
int  swap(int *a, int *b);
```

Now when your program compiles, it has all the information it needs to do complete error-checking on any call to **swap**. And you can use a similar format when you define the function:

```
int  swap(int *a, int *b)
{
     /* Body of function */
}
```

The modern style increases the error-checking performed even if you don't use function prototypes; if you do use prototypes, this will cause the compiler to ensure that the declarations and definitions agree.

Using enum Definitions

In classic C, lists of values are defined using the `#define` directive, like this:

```
#define sun   0
#define mon   1
#define tues  2
#define wed   3
#define thur  4
#define fri   5
#define sat   6
```

In the modern style, however, you can declare enumerated data types using the keyword **enum**, as shown here:

```
typedef enum {sun, mon, tues, wed, thur, fri, sat} days;
```

This has the same effect as the classic method, right down to setting *sun* = 0 and *sat* = 6; however, the modern method does more information hiding and abstraction than the long list of `#define` directives. And you can declare variables to be of type *days*.

Using typedef

In classic-style C, user-defined data types were seldom named, with the exception of structures and unions—and even with them you had to precede any declaration with the keyword **struct** or **union**.

In modern-style C, another level of information hiding is available through the `typedef` directive. This allows you to associate a given data type (including **structs** and **enums**) with a name, then declare variables of that type.

Here are some sample type definitions with variable declarations:

```
typedef int  *intptr;
typedef char  namestr[30];
typedef enum  { male, female, unknown } sex;
typedef struct {
   namestr  last,first;
   char     ssn[9];
   sex      gender;
   short    age;
   float    gpa;
} student;
typedef student class[100];

class     hist104,ps102;
student   valedictorian;
intptr    iptr;
```

Using **typedef**s makes the program more readable; it also allows you to change a single location—the place where a type is actually defined—and have that change propagated through the entire program.

Declaring void functions

In the original definition of C, every function returned a value of some type; if no type was declared, the function was assumed to be of type **int**. In a similar fashion, functions that returned "generic" (untyped) pointers were usually declared to return a pointer to **char**, just because they had to return *something*.

Now there is a standard type **void**, which can be thought of as a kind of null type. Any function that does not explicitly return a value should be declared as being of type **void**. Note that many of the runtime memory allocation routines (such as **malloc**) are declared to be of type `void *`. This means they return an untyped pointer, which you can then (in Turbo C)

assign to a pointer of any type without type-casting (though you should type cast anyway, to preserve portability).

Make Use of Extensions

There are a number of minor extensions to the C language that aid program readability, replace some anachronisms, and allow you to move forward. Here's a brief listing.

String Literals

In classic C, you had to use continuation characters or some kind of concatenation in order to have large string literals in your program.

In modern-style C, you can easily spread a large literal across several lines, like this:

```
main()
{
   char  *msg;

   msg = "Four score and seven years ago, our fathers"
         " brought forth upon\nthis continent a new"
         " nation, dedicated to the ideal that all"
         " men\nare created equal";

   printf("%s",msg);
}
```

Hexadecimal Character Constants

In classic C, escape sequences specifying particular ASCII codes were all done in octal (base 8). This was because C was originally developed on machines where binary numbers were usually represented in octal form.

Today, most computers use hexadecimal (base 16) to represent binary numbers. Because of this, modern C allows you to declare character constants in hex notation. The general format is `'\xDD'`, where *DD* represents one or two hexademical digits (0..9, *A..F*). These escape sequences can be assigned directly to **char** variables, or they can be embedded in strings, for example, `ch = '\x20'`.

signed Types

Classic C assumed that all integer-based types were signed, and so included the type modifier **unsigned** so that you could specify otherwise. By default, variables of type **char** were considered **signed**, which meant that the underlying range of values was –128 to 127.

On microcomputers today, however, the **char** type is often thought of as being **unsigned**, and Turbo C has a compiler option to allow you to make that the default. In such a case, however, you may still want to be able to declare a **signed char**. In modern C, you can do so, since **signed** is recognized as a valid modifier.

Pitfalls in C Programming

There are a number of common errors that programmers make when they first start coding in C. Here's a list of some of them, along with suggestions about how to avoid them.

Path Names with C Strings

Everyone knows that the backslash (\) in MS-DOS indicates a directory name. However, in C, the backslash is the escape character in a string. This conflict causes a bit of a problem if you give a path name with a C string.

For example, if you had the statement

```
file = fopen("c:\new\tools.dat", "r");
```

you'd expect to open the file TOOLS.DAT in the NEW directory on drive C. You won't. Instead, the `\n` gets you the escape sequence for the newline character (LF), and the `\t` gets you the tab character.

The result is your file name will have embedded in it the newline and tab characters. DOS would reject the string as an improper file name, since file names may not have newline or tab in them. The proper string is

```
"c:\\new\\tools.dat"
```

Using and Misusing Pointers

Pointers may well be the single most confusing issue to novice C programmers. When do you use pointers, and when don't you? When do you use the indirection operator (*)? When do you use the address-of operator (&)? And how can you avoid really messing up the operating system when you run your program?

Using an Uninitialized Pointer

One serious mistake is to assign a value to the address contained by a pointer without having assigned an address to that pointer.

For example,

```
main()
{
   int *iptr;

   *iptr = 421;
   printf("*iptr = %d\n",*iptr);
}
```

What makes this pitfall so dangerous is that you can often get away with it. In the previous example, the pointer *iptr* has some random address in it; that's where the value 421 is stored. This program is small enough that there is very little chance of anything being clobbered. In a larger program, though, there is an increasing chance of that happening, since you may well have other information stored at the address that *iptr* happens to contain. And if you're using the tiny memory model, where the code and data segments occupy the same space, you run the risk of corrupting the machine code itself. Remember to use **malloc** to set aside memory at the address the pointer points to, and to place that address in the pointer.

Strings

You may recall that you can declare strings as pointers to **char** or as arrays of **char**. You may also recall that these are the same, except for one very important difference: If you use a pointer to **char**, no space for the string is allocated; if you use an array, space *is* allocated, and the array variable holds the address of that space.

Failure to understand this difference can lead to two types of errors. Consider the following program:

```
main()
{
   char  *name;
   char  msg[10];

   printf("What is your name? ");
   scanf("%s",name);
   msg = "Hello, ";
   printf("%s%s",msg,name);
}
```

At first glance, this might appear to be perfectly fine; a little clumsy, but still allowable.

But this has introduced two separate errors.

The first error has to do with the statement

```
scanf("%s",name)
```

The statement itself is legal and correct. Since *name* is a pointer to **char**, you don't need to use the address-of (&) operator in front of it.

However, the program has *not* allocated any memory for *name*; the name you type in will be stored at whatever random address that *name* happens to have. You will get a warning on this (`Possible use of 'name' before definition`), but no error.

The second problem *will* cause an error. The problem lies in the statement `msg = "Hello, "`. The compiler thinks you are trying to change *msg* to the address of the constant string `"Hello, "`. You can't do that, because array names are constants that cannot be modified (just like 7 is a constant, and you can't say "7 = i"). The compiler will give you an error message `Lvalue required`.

What are the solutions to these errors? The simplest approach is to switch the ways in which *name* and *msg* have been declared:

```
main()
{
   char name[10];
   char *msg;

   printf("What is your name? ");
   scanf("%s",name);
   msg = "Hello, ";
   printf("%s%s",msg,name);
}
```

This works perfectly well. The variable *name* has space set aside to hold your name as you type it in, while *msg* lets you assign to it the address of the constant string `"Hello, "`.

If, however, you are bound and determined to keep the declarations the way they were, then you'll need to make the following changes to the code:

```
#include <alloc.h>

main()
{
   char *name;
   char msg[10];

   name = (char *) malloc(10);
   printf("What is your name? ");
   scanf("%s",name);
   strcpy(msg,"Hello, ");
   printf("%s%s",msg,name);
}
```

The call to **malloc** sets aside 10 bytes of memory and assigns the address of that memory to *name,* taking care of our first problem. The function **strcpy** does a character-by-character copy from the constant string `"Hello,"` to the array *msg*.

Confusing Assignment (=) with Equality (==)

In the languages Pascal and BASIC, a comparison for equality is made with the expression `if (a = b)`. In C, that is a valid construct, but it has quite a different meaning.

Look at this code fragment:

```
if (a = b)
   puts("Equal");
else
   puts("Not equal");
```

If you're a Pascal or BASIC programmer, then you might expect this to print `Equal` if *a* and *b* have the same value, and `Not equal` otherwise. That's not what happens. In C, the expression `a = b` means "assign the value of *b* to *a*," and the entire expression takes on the value of *b*. So, the previous fragment will assign the value of *b* to *a*, then print `Equal` if *b* has a nonzero value, otherwise it will print `Not equal`.

What you *really* want is the following:

```
if (a == b)
   puts("Equal);
else
   puts("Not equal");
```

Forgetting the break in switch Statements

You may remember that the **break** statement is used in a **switch** statement to end a particular case. Please continue to remember that. If you forget to put a **break** statement in for a given case, the case(s) after it is executed as well.

Array Indexing

Don't forget that arrays start at [0], not at [1]. A common error is to write code like this:

```
main()
{
   int list[100],i;

   for (i = 1; i <= 100; i++)
      list[i] = i*i;
}
```

This program leaves the first location in *list*—namely *list[0]*—uninitialized, and it stores a value in a nonexistent location of *list*—*list[100]*—possibly overwriting other data in the process.

The correct code should be written like this:

```
main()
{
   int list[100],i;

   for (i = 0; i < 100; i++)
      list[i] = i*i;
}
```

Failure to Pass-by-Address

Look at the following program and figure out what's wrong with it:

```
main()
{
   int a,b,sum;

   printf("Enter two values: ");
   scanf("%d %d",a,b);
   sum = a + b;
   printf("The sum is %d\n",sum);
}
```

Give up? The error is in the statement `scanf("%d %d",a,b)`. Remember that **scanf** requires you to pass *addresses* instead of values? The same is true of any function whose formal parameters are pointers. The previous program will compile and run, since **scanf** will take whatever random values are in *a* and *b* and use them as addresses in which to store the values you enter.

The correct statement should read `scanf("%d %d",&a,&b);` that way, the addresses of *a* and *b* are passed to **scanf**, and the values you enter are correctly stored in those variables. This same pitfall can happen with your own functions. Remember the function **swap** defined back in the section on pointers?

What would happen if you called it like this:

```
main()
{
   int i,j;

   i = 421;
   j = 53;
   printf("before: i = %4d j = %4d\n",i,j);
   swap(i,j);
   printf("after: i = %4d j = %4d\n",i,j);
}
```

The variables *i* and *j* would have the same values before and after the call to **swap**; however, the values at data addresses 421 and 53 would have their values swapped, which could cause some subtle and hard-to-trace problems.

How do you avoid this? Use function prototypes and full function definitions.

Actually, you would have gotten a compiler error in the previous version of **main** if **swap** were defined as it was earlier in this chapter.

If, however, you defined it in the following manner, the program would compile just fine:

```
void swap(a,b)
int *a,*b;
{
   ...
}
```

Moving the definitions of *a* and *b* out of the parentheses disables the error-checking that would go on otherwise, which is the best reason for not using the classic style of function definition.

Sailing Away

As we said at the start of the previous chapter, we can't give you a complete tutorial on C in just two chapters. But we have gotten you under way. What you should do now—if you haven't been doing it all along—is key in these example programs, compile them, run them, and (most important) modify them to see what happens when you change things around. Best of luck, and bon voyage.

C H A P T E R

8

Turbo C's Video Functions

Turbo C comes with a complete library of graphics functions, so that you can produce onscreen charts and diagrams in color or black and white.

In This Chapter...

First, we will briefly discuss video modes and windows. After that, we will explain how to program in text mode and in graphics mode.

Turbo C's new video functions are based on corresponding routines in Turbo Pascal. If you are not already familiar with controlling your PC's screen modes or creating and managing windows and viewports, take a few minutes to read the following words on those topics.

Some Words about Video Modes

Your PC has some kind of video adapter. This can be a Monochrome Display Adapter (MDA) for your basic text-only display, or it can be capable of displaying graphics, such as a Color Graphics Adapter (CGA), an Enhanced Graphics Adapter (EGA), or a Hercules Monochrome Graphics Adapter. Each adapter can operate in a variety of modes; the mode specifies whether the screen displays 80- or 40 columns (text mode only), the display resolution (graphics mode only), and the display type (color or black and white).

The screen's operating mode is defined when your program calls one of the mode-defining functions (**textmode**, **initgraph**, or **setgraphmode**).

- In *text mode*, your PC's screen is divided into cells (80- or 40 columns wide by 25 lines high). Each cell consists of an attribute and a character. The character is the displayed ASCII character, while the attribute specifies *how* the character is displayed (its color, intensity, etc.). Turbo C provides a full range of routines for manipulating the text screen, for writing text directly to the screen, and for controlling the cell attributes.
- In *graphics mode*, your PC's screen is divided into pixels; each pixel displays a single dot on the screen. The number of pixels (the resolution) depends on the type of video adapter connected to your system and the mode that adapter is in. You can use functions from Turbo C's new graphics library to create graphic displays on the screen: You can draw lines and shapes, fill enclosed areas with patterns, and control the color of each pixel.

In text modes, the upper left corner of the screen is position (1,1), with *x*-coordinates increasing from left-to-right, and *y*-coordinates increasing from screen-top to screen-bottom. In graphics modes, the upper left corner is position (0,0), with the *x*- and *y*-coordinate values increasing in the same manner.

Some Words about Windows and Viewports

Turbo C provides functions for creating and managing windows on your screen in text mode (and viewports in graphics mode). If you are not familiar with windows and viewports, you should read this brief overview. Turbo C's new window- and viewport-management functions are explained in "Programming in Text Mode" and "Programming in Graphics Mode" later in this chapter.

What Is a Window?

A window is a rectangular area defined on your PC's video screen when it's in a text mode. When your program writes to the screen, its output is restricted to the active window. The rest of the screen (outside the window) remains untouched.

The default window is a full-screen text window. Your program can change this default full-screen text window to a text window smaller than the full

screen (with a call to the **window** function). This function specifies the window's position in terms of screen coordinates.

What Is a Viewport?

In graphics mode, you can also define a rectangular area on your PC's video screen; this is a viewport. When your graphics program outputs drawings and so on, the viewport acts as the virtual screen. The rest of the screen (outside the viewport) remains untouched. You define a viewport in terms of screen coordinates with a call to the **setviewport** function.

Coordinates

Except for these window- and viewport-defining functions, all coordinates for text-mode and graphics-mode functions are given in window- or viewport-relative terms, not in absolute screen coordinates. The upper left corner of the text-mode window is the coordinate origin, referred to as (1,1); in graphics modes, the viewport coordinate origin is position (0,0).

Programming in Text Modes

In this section, we give a brief summary of the functions you use in text mode: For more detailed information about these functions, refer to Chapter 2 of the *Turbo C Reference Guide.*

Turbo C, the direct console I/O package (**cprintf**, **cputs**, and so on) has been enhanced to provide higher-performance text output and extended to provide window management, cursor positioning, and attribute control functions. These functions are all part of the standard Turbo C libraries; they are prototyped in the header file conio.h.

The Console I/O Functions

Turbo C's text-mode functions work in any of the five possible video text modes. The modes available on your system depend on the type of video adapter and monitor you have. You specify the current text mode with a call to **textmode**. We explain how to use this function later in this chapter and under the **textmode** entry in Chapter 2 of the *Turbo C Reference Guide.*

These text-mode functions are divided into four separate groups:

- text output and manipulation
- window and mode control
- attribute control
- state query

We cover these four text-mode function groups in the following sections.

Text Output and Manipulation

Here's a quick summary of the text output and manipulation functions:

===

Writing and reading text:

cprintf	sends formatted output to the screen
cputs	sends a string to the screen
putch	sends a single character to the screen
getche	reads a character and echoes it to the screen

Manipulating text (and the cursor) onscreen:

clrscr	clears the text window
clreol	clears from the cursor to the end of the line
delline	deletes the line where the cursor rests
gotoxy	positions the cursor
insline	inserts a blank line below the line where the cursor rests
movetext	copies text from one area onscreen to another

Moving blocks of text into and out of memory:

gettext	copies text from an area onscreen to memory
puttext	copies text from memory to an area onscreen

===

Your screen-output programs will come up in a full-screen text window by default, so you can immediately write, read, and manipulate text without any preliminary mode-setting. You write text to the screen with the direct console output functions **cprintf**, **cputs**, and **putch**, and echo input with the function **getche**. Text wraps within the window as follows: If text extends beyond the window's right border, the text extending beyond the right border is moved down to the beginning of the next line.

Once your text is on the screen, you can erase the active window with **clrscr**, erase part of a line with **clreol**, delete a whole line with **delline**, and insert a blank line with **insline**. The latter three functions operate relative to the cursor position; you move the cursor to a specified location with **gotoxy**. You can also copy a whole block of text from one rectangular location in the window to another with **movetext**.

You can capture a rectangle of onscreen text to memory with **gettext**, and put that text back on the screen (anywhere you want) with **puttext**.

Window and Mode Control

There are two window- and mode-control functions:

textmode	sets the screen to a text mode
window	defines a text-mode window

You can set your screen to any of several video text modes with **textmode** (limited only by your system's type of monitor and adapter). This initializes the screen as a full-screen text window, in the particular mode specified, and clears any residual images or text.

When your screen is in a text mode, you can output to the full screen, or you can set aside a *portion* of the screen—a window—to which your program's output is confined. To create a text window, you call **window**, specifying what area on the screen it will occupy.

Attribute Control

Here's a quick summary of the text-mode attribute control functions:

==

Setting foreground and background:

textcolor	sets the foreground color (attribute)
textbackground	sets the background color (attribute)
textattr	sets the foreground and background colors (attributes) at the same time

Converting intensity:

highvideo	sets text to high intensity
lowvideo	sets text to low intensity
normvideo	sets text to original intensity

==

The attribute-control functions set the current attribute, which is represented by an 8-bit value: the four lowest bits represent the foreground color, the next three bits give the background color, and the high bit is the "blink enable" bit.

Subsequent text is displayed in the current attribute. With the attribute-control functions, you can set the background and foreground (character) colors separately (with **textbackground** and **textcolor**) or combine the color specifications in a single call to **textattr**. You can also specify that the character—the foreground—will blink. Most color monitors in color modes will display the true colors. Non-color monitors may convert some or all of the attributes to various monochromatic shades or other visual effects, such as bold, underscore, reverse video, and so on.

You can direct your system to map the high-intensity foreground colors to low-intensity colors with **lowvideo** (which turns *off* the high-intensity bit for the characters). Or you can map the low-intensity colors to high intensity with **highvideo** (which turns *on* the character high-intensity bit). When you're through playing around with the character intensities, you can restore the settings to their original values with **normvideo**.

State Query

Here's a quick summary of the state-query functions:

===

gettextinfo	fills in a **text_info** structure with information about the current text window
wherex	gives the x-coordinate of the cell containing the cursor
wherey	gives the y-coordinate of the cell containing the cursor

===

Turbo C's console I/O functions include some designed for *state query*. With these functions, you can retrieve information about your text-mode window and the current cursor position within the window.

The **gettextinfo** function fills a **text_info** structure (defined in conio.h) with several details about the text window, including:

- the current video mode
- the window's position in absolute screen coordinates
- the window's dimensions
- the current foreground and background colors
- the cursor's current position

Sometimes you might need only a few of these details. Rather than retrieving all the text window information, you can find out just the cursor's (window-relative) position with **wherex** and **wherey**.

Text Windows

The default text window is full screen; you can change this to a less-than-full-screen text window with a call to the **window** function. Text windows can contain up to 25 lines (the maximum number of lines onscreen in any text mode) and up to 40- or 80 columns (depending on your text mode).

The coordinate origin of a Turbo C text window is the *upper left corner* of the *window*. The coordinates of the window's upper left corner are (1,1); the coordinates of the bottom right corner of a full-screen 80-column text window are (80,25).

An Example

Suppose your 100% PC-compatible system is in 80-column text mode, and you want to create a window. The upper left corner of the window will be at screen coordinates (10, 8), and the lower right corner of the window will be at screen coordinates (50, 21). To do this, you call the **window** function, like this:

```
window(10, 8, 50, 21);
```

Now that you've created the text-mode window, you want to move the cursor to the *window* position (5, 8) and write some text in it, so you decide to use **gotoxy** and **cputs**.

```
gotoxy(5, 8);
cputs("Happy Birthday, Frank Borland");
```

Figure 8.1 illustrates these ideas.

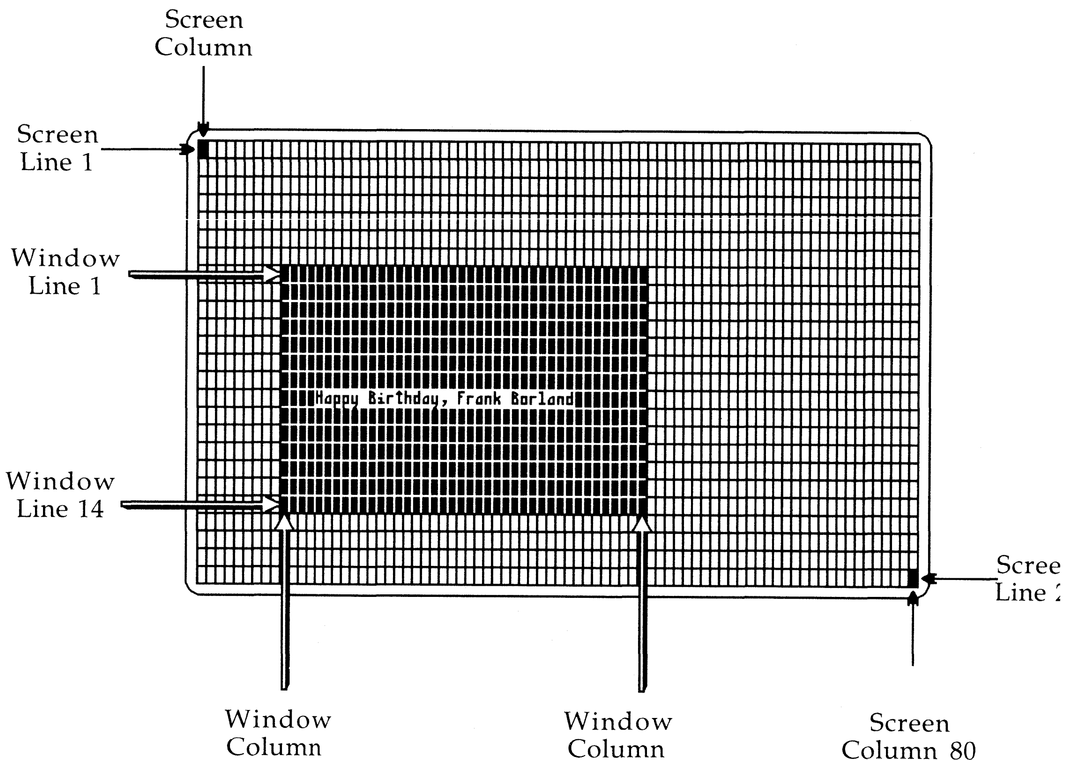


Figure 8.1: A Window in 80x25 Text Mode

The text_modes Type

You can put your monitor into one of five PC text modes with a call to the **textmode** function. The enumeration type *text_modes*, defined in conio.h, enables you to use symbolic names for the *mode* argument to the **textmode** function, instead of "raw" mode numbers. However, if you use the symbolic constants, you must `#include <conio.h>` in your source code.

The numeric and symbolic values defined by *text_modes* are as follows:

Symbolic Constant	*Numeric Value*	*Video Text Mode*
LASTMODE	–1	Previous text mode enabled
BW40	0	Black & White, 40 columns
C40	1	16-Color, 40 columns
BW80	2	Black & White, 80 columns
C80	3	16-Color, 80 columns
MONO	7	Monochrome, 80 columns

For example, the following calls to **textmode** will put your color monitor in the indicated operating mode:

Call	*Operating Mode*
`textmode(0)`	Black&White, 40 column
`textmode(BW80)`	Black&White, 80 column
`textmode(C40)`	16-Color, 40 column
`textmode(3)`	16-Color, 80 column

Text Colors

For a detailed description of how cell attributes are laid out, refer to the **textattr** entry in Chapter 2 of the *Turbo C Reference Guide.*

When a character occupies a cell, the color of the character is the *foreground;* the color of the cell's remaining area is the *background.* Color monitors with color video adapters can display up to 16 different colors; monochrome monitors substitute different visual attributes (highlighted, underscored, reverse video, and so on) for the colors.

The include file conio.h defines symbolic names for the different colors. If you use the symbolic constants, you must `#include <conio.h>` in your source code.

The following table lists these symbolic constants and their corresponding numeric values. Note that only the first eight colors are available for the background, while all sixteen colors are available for the foreground (the characters themselves).

Symbolic Constant	Numeric Value	Foreground or Background?
BLACK	0	both
BLUE	1	both
GREEN	2	both
CYAN	3	both
RED	4	both
MAGENTA	5	both
BROWN	6	both
LIGHTGRAY	7	both
DARKGRAY	8	foreground only
LIGHTBLUE	9	foreground only
LIGHTGREEN	10	foreground only
LIGHTCYAN	11	foreground only
LIGHTRED	12	foreground only
LIGHTMAGENTA	13	foreground only
YELLOW	14	foreground only
WHITE	15	foreground only
BLINK	128	foreground only

You can add the symbolic constant BLINK (numeric value 128) to a foreground argument if you want the character to blink.

High-Performance Output: the directvideo Variable

Turbo C's console I/O package includes a variable called *directvideo*. This variable controls whether your program's console output goes directly to the video RAM (*directvideo* = 1) or goes via BIOS calls (*directvideo* = 0).

The default value is *directvideo* = 1 (console output goes directly to the video RAM). In general, going directly to video RAM gives very high performance (spelled f-a-s-t-e-r o-u-t-p-u-t), but doing so requires your computer to be 100% IBM PC-compatible: Your video hardware must be

identical to IBM display adapters. Setting *directvideo* = 0 will work on any machine that is IBM BIOS-compatible, but the console output will be slower.

Programming in Graphics Mode

In this section, we give a brief summary of the functions you use in graphics mode: For more detailed information about these functions, refer to Chapter 2 of the *Turbo C Reference Guide.*

Turbo C provides a separate library of over 70 graphics functions, ranging from high-level calls (like **setviewport**, **bar3d**, and **drawpoly**) to bit-oriented functions (like **getimage** and **putimage**). The graphics library supports numerous fill and line styles, and provides several text fonts that you can size, justify, and orient horizontally or vertically.

These functions are in the library file GRAPHICS.LIB, and they are prototyped in the header file GRAPHICS.H. In addition to these two files, the graphics package includes graphics device drivers (*.BGI files) and stroked character fonts (*.CHR files); we discuss these additional files in following sections.

In order to use the graphics functions:

- If you're using TC.EXE, toggle Options/Linker/Graphics library to On. When you make your program, the linker will automatically link in the Turbo C graphics library.
- If you're using TCC.EXE, you have to list GRAPHICS.LIB on the command line. For example, if your program, MYPROG.C, uses graphics, the TCC command line would be:

```
tcc myprog graphics.lib
```

Important Note: There is only one graphics library, not separate versions for each memory model (in contrast to the standard libraries CS.LIB, CC.LIB, CM.LIB, etc., which are memory-model specific). Each function in GRAPHICS.LIB is a **far** function, and those graphics functions that take pointers take **far** pointers. For these functions to work correctly, it is important that you `#include <graphics.h>` in every module that uses graphics.

The Graphics Library Functions

Turbo C's graphics functions comprise seven categories:

- graphics system control
- drawing and filling
- manipulating screens and viewports
- text output
- color control
- error handling
- state query

Graphics System Control

Here's a quick summary of the graphics system control functions:

closegraph	shuts down the graphics system
detectgraph	checks the hardware and determines which graphics driver to use; recommends a mode
graphdefaults	resets all graphics system variables to their default settings
_graphfreemem	deallocates graphics memory; hook for defining your own routine
_graphgetmem	allocates graphics memory; hook for defining your own routine
getgraphmode	returns the current graphics mode
getmoderange	returns lowest and highest valid modes for specified driver
initgraph	initializes the graphics system and puts the hardware into graphics mode
installuserdriver	installs a vendor-added device driver to the BGI device driver table
installuserfont	loads a stroked font file not known to the graphics routines
registerbgidriver	registers a linked-in or user-loaded driver file for inclusion at link time
restorecrtmode	restores the original (pre-**initgraph**) screen mode
setgraphbufsize	specifies size of the internal graphics buffer
setgraphmode	selects the specified graphics mode, clears the screen, and restores all defaults

Turbo C's graphics package provides graphics drivers for the following graphics adapters (and true compatibles):

- Color Graphics Adapter (CGA)
- Multi Color Graphics Array (MCGA)
- Enhanced Graphics Adapter (EGA)
- Video Graphics Array (VGA)
- Hercules Graphics Adapter
- AT&T 400-line Graphics Adapter
- 3270 PC Graphics Adapter
- IBM 8514 Graphics Adapter

To start the graphics system, you first call the **initgraph** function. **initgraph** loads the graphics driver and puts the system into graphics mode. You can tell **initgraph** to use a particular graphics driver and mode, or to autodetect the attached video adapter at run time and pick the corresponding driver. If you tell **initgraph** to autodetect, it calls **detectgraph** to select a graphics driver and mode. If you tell **initgraph** to use a particular graphics driver and mode, you must be sure that the hardware is present. If you force **initgraph** to use hardware that is not present, the results will be unpredictable.

Once a graphics driver has been loaded, you can find out the name of the driver by using the **getdrivename** function and how many modes a driver supports with **getmaxmode**. **getgraphmode** will tell you which graphics mode you are currently in. Once you have a mode number, you can find out the name of the mode with **getmodename**. You can change graphics modes with **setgraphmode** and return the video mode to its original state (before graphics was initialized) with **restorecrtmode**. **restorecrtmode** returns the screen to text mode, but it does not close the graphics system (the fonts and drivers are still in memory).

graphdefaults resets the graphics state's settings (viewport size, draw color, fill color and pattern, etc.) to their default values.

installuserdriver and **installuserfont** allow you to add new device drivers and fonts to your BGI.

Finally, when you're through using graphics, call **closegraph** to shut down the graphics system. **closegraph** unloads the driver from memory and restores the original video mode (via **restorecrtmode**).

A More Detailed Discussion

The previous discussion provided an overview of how **initgraph** operates. In the following paragraphs, we describe the behavior of **initgraph**, **_graphgetmem**, and **_graphfreemem** in some detail.

Normally, the **initgraph** routine loads a graphics driver by allocating memory for the driver, then loading the appropriate .BGI file from disk. As an alternative to this dynamic loading scheme, you can link a graphics driver file (or several of them) directly into your executable program file. You do this by first converting the .BGI file to an .OBJ file (using the BGIOBJ utility), then placing calls to **registerbgidriver** in your source code (before the call to **initgraph**) to *register* the graphics driver(s). When you build your program, you need to link the .OBJ files for the registered drivers.

After determining which graphics driver to use (*via* **detectgraph**), **initgraph** checks to see if the desired driver has been registered. If so, **initgraph** uses the registered driver directly from memory. Otherwise, **initgraph** allocates memory for the driver and loads the .BGI file from disk.

Note: Using **registerbgidriver** is an advanced programming technique, not recommended for novice programmers. This function is described in more detail in Appendix D in the *Turbo C Reference Guide*.

During run time, the graphics system might need to allocate memory for drivers, fonts, and internal buffers. If this is necessary, it calls **_graphgetmem** to allocate memory, and calls **_graphfreemem** to free it. By default, these routines simply call **malloc** and **free**, respectively.

You can override this default behavior by defining your own **_graphgetmem** and **_graphfreemem** functions. By doing this, you can control graphics memory allocation yourself. You must, however, use the same names for your own versions of these memory-allocation routines: They will override the default functions with the same names that are in the standard C libraries.

Note: If you have provided your own **_graphgetmem** or **_graphfreemem**, you may get a "duplicate symbols" warning message because these functions are also in the graphics library. Just ignore the warning.

Drawing and Filling

Here's a quick summary of the drawing and filling functions:

Drawing:

arc	draws a circular arc
circle	draws a circle
drawpoly	draws the outline of a polygon
ellipse	draws an elliptical arc
getarccoords	returns the coordinates of the last call to **arc** or **ellipse**
getaspectratio	returns the aspect ratio of the current graphics mode
getlinesettings	returns the current line style, line pattern, and line thickness
line	draws a line from (*x0*, *y0*) to (*x1*, *y1*)
linerel	draws a line to a point some relative distance from the current position (CP)
lineto	draws a line from the current position (CP) to (*x*,*y*)
moveto	moves the current position (CP) to (*x*,*y*)
moverel	moves the current position (CP) a relative distance
rectangle	draws a rectangle
setaspectratio	changes the default aspect ratio-correction factor
setlinestyle	sets the current line width and style

Filling:

bar	draws and fills a bar
bar3d	draws and fills a 3-D bar
fillellipse	draws and fills an ellipse
fillpoly	draws and fills a polygon
floodfill	flood-fills a bounded region
getfillpattern	returns the user-defined fill pattern
getfillsettings	returns information about the current fill pattern and color
pieslice	draws and fills a pie slice
sector	draws and fills an elliptical pie slice
setfillpattern	selects a user-defined fill pattern
setfillstyle	sets the fill pattern and fill color

With Turbo C's drawing and painting functions, you can draw colored lines, arcs, circles, ellipses, rectangles, pieslices, 2- and 3-dimensional bars, polygons, and regular or irregular shapes based on combinations of these. You can fill any bounded shape (or any region surrounding such a shape)

with one of 11 predefined patterns, or your own user-defined pattern. You can also control the thickness and style of the drawing line, and the location of the current position (CP).

You draw lines and unfilled shapes with the functions **arc**, **circle**, **drawpoly**, **ellipse**, **line**, **linerel**, **lineto**, and **rectangle**. You can fill these shapes with **floodfill**, or combine drawing/filling into one step with **bar**, **bar3d**, **fillellipse**, **fillpoly**, **pieslice**, and **sector**. You use **setlinestyle** to specify whether the drawing line (and border line for filled shapes) is thick or thin, and whether its style is solid, dotted, etc., or some other line pattern you've defined. You can select a predefined fill pattern with **setfillstyle**, and define your own fill pattern with **setfillpattern**. You move the CP to a specified location with **moveto**, and move it a specified displacement with **moverel**.

To find out the current line style and thickness, you call **getlinesettings**. For information about the current fill pattern and fill color, you call **getfillsettings**; you can get the user-defined fill pattern with **getfillpattern**.

You can get the aspect ratio (the scaling factor used by the graphics system to make sure circles come out round) with **getaspectratio**, and get coordinates of the last drawn arc or ellipse by calling **getarccoords**. If your circles are not perfectly round, use **setaspectratio** to correct them.

Manipulating the Screen and Viewport

Here's a quick summary of the image-manipulation functions:

==

Screen Manipulation

cleardevice	clears the screen (active page)
setactivepage	sets the active page for graphics output
setvisualpage	sets the visual graphics page number

Viewport Manipulation

clearviewport	clears the current viewport
getviewsettings	returns information about the current viewport
setviewport	sets the current output viewport for graphics output

Image Manipulation

getimage	saves a bit image of the specified region to memory
imagesize	returns the number of bytes required to store a rectangular region of the screen
putimage	puts a previously saved bit image onto the screen

Pixel Manipulation

getpixel	gets the pixel color at (*x*,*y*)
putpixel	plots a pixel at (*x*,*y*)

==

Besides drawing and painting, the graphics library offers several functions for manipulating the screen, viewports, images, and pixels. You can clear the whole screen in one fell swoop with a call to **cleardevice**; this routine erases the entire screen and homes the CP in the viewport, but leaves all other graphics system settings intact (the line, fill, and text styles, the palette, the viewport settings, and so on).

Depending on your graphics adapter, your system has between one and eight screen-page buffers, which are areas in memory where individual whole-screen images are stored dot-by-dot. You can specify which screen page is the active one (where graphics functions place their output) and which is the visual page (the one displayed onscreen) with **setactivepage** and **setvisualpage**, respectively.

Once your screen's in a graphics mode, you can define a viewport (a rectangular "virtual screen") on your screen with a call to **setviewport**. You

define the viewport's position in terms of absolute screen coordinates and specify whether clipping is on (active) or off. You clear the viewport with **clearviewport**. To find out the current viewport's absolute screen coordinates and clipping status, call **getviewsettings**.

You can capture a portion of the onscreen image with **getimage**, call **imagesize** to calculate the number of bytes required to store that captured image in memory, then put the stored image back on the screen (anywhere you want) with **putimage**.

The coordinates for all output functions (drawing, filling, text, and so on) are viewport-relative.

You can also manipulate the color of individual pixels with the functions **getpixel** (which returns the color of a given pixel) and **putpixel** (which plots a specified pixel in a given color).

Text Output in Graphics Mode

Here's a quick summary of the graphics-mode text output functions:

gettextsettings	returns the current text font, direction, size, and justification
outtext	sends a string to the screen at the current position (CP)
outtextxy	sends a string to the screen at the specified position
registerbgifont	registers a linked-in or user-loaded font
settextjustify	sets text justification values used by **outtext** and **outtextxy**
settextstyle	sets the current text font, style, and character magnification factor
setusercharsize	sets width and height ratios for stroked fonts
textheight	returns the height of a string in pixels
textwidth	returns the width of a string in pixels

The graphics library includes an 8×8 bit-mapped font and several stroked fonts for text output while in graphics mode.

- In a *bit-mapped* font, each character is defined by a matrix of pixels.
- In a *stroked* font, each character is defined by a series of vectors that tell the graphics system how to draw that character.

The advantage of using a stroked font is apparent when you start to draw large characters. Since a stroked font is defined by vectors, it will still retain good resolution and quality when the font is enlarged. On the other hand, when you enlarge a bit-mapped font, the matrix is multiplied by a scaling factor; as the scaling factor becomes larger, the characters' resolution becomes coarser. For small characters, the bit-mapped font should be sufficient, but for larger text you should select a stroked font.

You output graphics text by calling either **outtext** or **outtextxy**, and control the justification of the output text (with respect to the CP) with **settextjustify**. You choose the character font, direction (horizontal or vertical), and size (scale) with **settextstyle**. You can find out the current text settings by calling **gettextsettings**, which returns the current text font, justification, magnification, and direction in a **textsettings** structure. **setusercharsize** allows you to modify the character width and height of stroked fonts.

If clipping is *on*, all text strings output by **outtext** and **outtextxy** will be clipped at the viewport borders. If clipping is *off*, these functions will throw away bit-mapped font output if any part of the text string would go off the screen edge; stroked font output is truncated at the screen edges.

To determine the onscreen size of a given text string, call **textheight** (which measures the string's height in pixels) and **textwidth** (which measures its width in pixels).

The default 8×8 bit-mapped font is built into the graphics package, so it is always available at run time. The stroked fonts are each kept in a separate .CHR file; they can be loaded at run time or converted to .OBJ files (with the BGIOBJ utility) and linked into your .EXE file.

Normally, the **settextstyle** routine loads a font file by allocating memory for the font, then loading the appropriate .CHR file from disk. As an alternative to this dynamic loading scheme, you can link a character font file (or several of them) directly into your executable program file. You do this by first converting the .CHR file to an .OBJ file (using the BGIOBJ utility), then placing calls to **registerbgifont** in your source code (before the call to **settextstyle**) to *register* the character font(s). When you build your program, you need to link in the .OBJ files for the stroked fonts you register.

Note: Using **registerbgifont** is an advanced programming technique, not recommended for novice programmers. This function is described in more detail in Appendix D in the *Turbo C Reference Guide*.

Color Control

Here's a quick summary of the color control functions:

==

Get color information

getbkcolor	returns the current background color
getcolor	returns the current drawing color
getdefaultpalette	returns the palette definition structure
getmaxcolor	returns the maximum color value available in the current graphics mode
getpalette	returns the current palette and its size
getpalettesize	returns the size of the palette lookup table

Set one or more colors

setallpalette	changes all palette colors as specified
setbkcolor	sets the current background color
setcolor	sets the current drawing color
setpalette	changes one palette color as specified by its arguments

==

Before summarizing how these color control functions work, we first present a basic description of how colors are actually produced on your graphics screen.

Pixels and Palettes

The graphics screen consists of an array of pixels; each pixel produces a single (colored) dot on the screen. The pixel's value does not specify the precise color directly; it is an index into a color table called a *palette*. The palette entry corresponding to a given pixel value contains the exact color information for that pixel.

This indirection scheme has a number of implications. Though the hardware might be capable of displaying many colors, only a subset of those colors can be displayed at any given time. The number of colors that can be displayed at any one time is equal to the number of entries in the palette (the palette's *size*). For example, on an EGA, the hardware can display 64 different colors, but only 16 of them at a time; the EGA palette's *size* = 16.

The *size* of the palette determines the range of values a pixel can assume, from 0 to (*size* – 1). The **getmaxcolor** function returns the highest valid pixel value (*size* – 1) for the current graphics driver and mode.

When we discuss the Turbo C graphics functions, we often use the term *color*, such as the current drawing color, fill color and pixel color. In fact, this color is a pixel's value: it's an index into the palette. Only the palette determines the true color on the screen. By manipulating the palette, you can change the actual color displayed on the screen even though the pixel values (drawing color, fill color, and so on) have not changed.

Background and Drawing Color

The *background color* always corresponds to pixel value 0. When an area is cleared to the background color, that area's pixels are simply set to 0.

The *drawing color* is the value to which pixels are set when lines are drawn. You choose a drawing color with `setcolor(n)`, where *n* is a valid pixel value for the current palette.

Color Control on a CGA

Due to graphics hardware differences, how you actually control color differs quite a bit between the CGA and the EGA, so we'll present them separately. Color control on the AT&T driver, and the lower resolutions of the MCGA driver is similar to CGA color control.

On the CGA, you can choose to display your graphics in low resolution (320x200), which allows you to use four colors, or high resolution (640x200), in which you can use two colors.

CGA Low Resolution

In the low resolution modes, you can choose from four predefined four-color palettes. In any of these palettes, you can only set the first palette entry; entries 1, 2, and 3 are fixed. The first palette entry (color 0) is the background color. This background color can be any one of the 16 available colors (see table of CGA background colors below).

You choose which palette you want by the mode you select (CGAC0, CGAC1, CGAC2, CGAC3); these modes use color palette 0 through color palette 3, as detailed in the following table. The CGA drawing colors, and the equivalent constants, are defined in graphics.h.

Palette Number	*Constant assigned to color number (pixel value)* 1	2	3
0	CGA_LIGHTGREEN	CGA_LIGHTRED	CGA_YELLOW
1	CGA_LIGHTCYAN	CGA_LIGHTMAGENTA	CGA_WHITE
2	CGA_GREEN	CGA_RED	CGA_BROWN
3	CGA_CYAN	CGA_MAGENTA	CGA_LIGHTGRAY

To assign one of these colors as the CGA drawing color, call **setcolor** with either the color number or the corresponding constant name as an argument; for example, if you are using palette 3 and you want to use cyan as the drawing color:

`setcolor(1);` or `setcolor(CGA_CYAN);`

The available CGA background colors, defined in graphics.h, are listed in the following table.

Numeric Value	*Symbolic Name*
0	BLACK
1	BLUE
2	GREEN
3	CYAN
4	RED
5	MAGENTA
6	BROWN
7	LIGHTGRAY
8	DARKGRAY
9	LIGHTBLUE
10	LIGHTGREEN
11	LIGHTCYAN
12	LIGHTRED
13	LIGHTMAGENTA
14	YELLOW
15	WHITE

To assign one of these colors to the CGA background color, use `setbkcolor(`*`color`*`)`, where *color* is one of the entries in the preceding table. Note that for CGA, this color is not a pixel value (palette index); it directly specifies the *actual* color to be put in the first palette entry.

CGA High Resolution

In high resolution mode (640x200), the CGA displays two colors: a black background and a colored foreground. Pixels can take on values of either 0 or 1. Because of a quirk in the CGA itself, the foreground color is actually what the hardware thinks of as its background color; you set it with the **setbkcolor** routine. (Strange but true.)

The colors available for the colored foreground are those listed in the preceding table. The CGA uses this color to display all pixels whose value equals 1.

The modes that behave in this way are CGAHI, MCGAMED, MCGAHI, ATT400MED, and ATT400HI.

CGA Palette Routines

Because the CGA palette is predetermined, you should not use the **setallpalette** routine on a CGA. Also, you should not use `setpalette(`*`index, actual_color`*`)`, except for *index* = 0. (This is an alternate way to set the CGA background color to *actual_color*.)

Color Control on the EGA and VGA

On the EGA, the palette contains 16 entries from a total of 64 possible colors, and each entry is user-settable.

You can retrieve the current palette with **getpalette**, which fills in a structure with the palette's size (16) and an array of the actual palette entries (the "hardware color numbers" stored in the palette). You can change the palette entries individually with **setpalette**, or all at once with **setallpalette**.

The default EGA palette corresponds to the 16 CGA colors, as given in the previous color table: black is in entry 0, blue in entry 1, ..., white in entry 15. There are constants defined in GRAPHICS.H that contain the corresponding hardware color values: these are EGA_BLACK, EGA_WHITE, and so on. You can also get these values with **getpalette**.

The **setbkcolor**(*color*) routine behaves differently on an EGA than on a CGA. On an EGA, **setbkcolor** copies the actual color value that's stored in entry #*color* into entry #0.

As far as colors are concerned, the VGA driver behaves like the EGA driver; it just has higher resolution (and smaller pixels).

Error Handling in Graphics Mode

Here's a quick summary of the graphics-mode error-handling functions:

==

grapherrormsg	returns an error message string for the specified error code
graphresult	returns an error code for the last graphics operation that encountered a problem

==

If an error occurs when a graphics library function is called (such as a font requested with **settextstyle** not being found), an internal error code is set. You retrieve the error code for the last graphics operation that reported an error by calling **graphresult**. The following error return codes are defined:

error code	***graphics_errors*** **constant**	**corresponding error message string**
0	grOk	No error
–1	grNoInitGraph	(BGI) graphics not installed (use **initgraph**)
–2	grNotDetected	Graphics hardware not detected
–3	grFileNotFound	Device driver file not found
–4	grInvalidDriver	Invalid device driver file
–5	grNoLoadMem	Not enough memory to load driver
–6	grNoScanMem	Out of memory in scan fill
–7	grNoFloodMem	Out of memory in flood fill
–8	grFontNotFound	Font file not found
–9	grNoFontMem	Not enough memory to load font
–10	grInvalidMode	Invalid graphics mode for selected driver
–11	grError	Graphics error
–12	grIOerror	Graphics I/O error
–13	grInvalidFont	Invalid font file
–14	grInvalidFontNum	Invalid font number
–15	grInvalidDeviceNum	Invalid device number
–18	grInvalidVersion	Invalid version of file

A call to `grapherrormsg(graphresult())` will return the error strings listed in the previous table.

The error return code accumulates, changing only when a graphics function reports an error. The error return code is reset to 0 only when **initgraph** executes successfully, or when you call **graphresult**. Therefore, if you want to know which graphics function returned which error, you should store the value of **graphresult** into a temporary variable and then test it.

State Query

Here's a quick summary of the graphics mode state query functions:

==

getarccoords	returns information about the coordinates of the last call to **arc** or **ellipse**
getaspectratio	returns the aspect ratio of the graphics screen
getbkcolor	returns the current background color
getcolor	returns the current drawing color
getdrivername	returns name of current graphics driver
getfillpattern	returns the user-defined fill pattern
getfillsettings	returns information about the current fill pattern and color
getgraphmode	returns the current graphics mode
getlinesettings	returns the current line style, line pattern, and line thickness
getmaxcolor	returns the current highest valid pixel value
getmaxmode	returns maximum mode number for current driver
getmaxx	returns the current *x* resolution
getmaxy	returns the current *y* resolution
getmodename	returns name of a given driver mode
getmoderange	returns the mode range for a given driver
getpalette	returns the current palette and its size
getpixel	returns the color of the pixel at *x,y*
gettextsettings	returns the current text font, direction, size, and justification
getviewsettings	returns information about the current viewport
getx	returns the *x* coordinate of the current position (CP)
gety	returns the *y* coordinate of the current position (CP)

==

In each of Turbo C's graphics functions categories there is at least one state query function. These functions are mentioned under their respective categories and also covered here. Each of the Turbo C graphics state query functions is named **get**<*something*> (except in the error-handling category).

Some of them take no argument and return a single value representing the requested information; others take a pointer to a structure defined in graphics.h, fill that structure with the appropriate information, and return no value.

The state query functions for the graphics system control category are **getgraphmode**, **getmaxmode**, and **getmoderange**: The first returns an integer representing the current graphics driver and mode, the second returns the mode range for a given driver, and the third returns the range of modes supported by a given graphics driver. **getmaxx** and **getmaxy** return the maximum *x* and *y* screen coordinates for the current graphics mode.

The drawing and filling state query functions are **getarccoords**, **getaspectratio**, **getfillpattern**, **getfillsettings**, and **getlinesettings**. **getarccoords** fills a structure with coordinates from the last call to **arc** or **ellipse**; **getaspectratio** tells the current mode's aspect ratio, which the graphics system uses to make circles come out round. **getfillpattern** returns the current user-defined fill pattern. **getfillsettings** fills a structure with the current fill pattern and fill color. **getlinesettings** fills a structure with the current line style (solid, dashed, and so on), line width (normal or thick), and line pattern.

In the screen- and viewport-manipulation category, the state query functions are **getviewsettings**, **getx**, **gety**, and **getpixel**. When you have defined a viewport, you can find out its absolute screen coordinates and whether clipping is active by calling **getviewsettings**, which fills a structure with the information. **getx** and **gety** return the (viewport-relative) *x*- and *y*-coordinates of the CP. **getpixel** returns the color of a specified pixel.

The graphics mode text-output function category contains one all-inclusive state query function: **gettextsettings**. This function fills a structure with information about the current character font, the direction in which text will be displayed (horizontal or bottom-to-top vertical), the character magnification factor, and the text-string justification (both horizontal and vertical).

Turbo C's color-control function category includes three state query functions. **getbkcolor** returns the current background color, and **getcolor** returns the current drawing color. **getpalette** fills a structure with the size of the current drawing palette and the palette's contents. **getmaxcolor** returns the highest valid pixel value for the current graphics driver and mode (palette *size* – 1).

Finally, **getmodename** and **getdrivername** return the name of a given driver mode and the name of the current graphics driver, respectively.

C H A P T E R

9

Notes for Turbo Pascal Programmers

Now, before you go any farther, go back to Chapters 7 and 3 and at least skim through them. Learn how C implements the basic elements of programming. We will cover some of the same ground in this chapter, but there are many details in those two chapters that you won't find here.

In a nutshell, Pascal is a fairly disciplined and structured language, whereas C is rather free-wheeling and flexible. But C is also *caveat programmer*; the same freedom that gives power to an experienced user can get a beginner in a lot of trouble. Pascal takes care of you better than C does, and thus is more suited as a language to learn the fundamentals of programming.

Turbo C and Turbo Pascal are moving toward the center of this C-Pascal language spectrum: Turbo C adds some structure to C, and Turbo Pascal adds some flexibility to Pascal.

In This Chapter...

This chapter is not meant to be a comprehensive discussion of C and its many fine features; its goal is to help you, as a Turbo Pascal programmer, learn enough about Turbo C to start writing programs quickly. Expertise and insight will come only with time, practice, and the hundreds of lines of code that you will write.

In this chapter, we'll show you the similarities and differences between Pascal and Turbo C programming. We start off with the basics: program

structure and the elements of programming. After that, we use a major example to illustrate our discussion of data structures. The end of this chapter is devoted to a discussion of programming issues that you need to be aware of, and an overview of common pitfalls that trap Pascal programmers learning C.

Throughout this chapter, we use examples of program code to illustrate the points we're making. Each example consists of a Turbo Pascal program or fragment on the left, and its equivalent in Turbo C on the right.

Program Structure

As you know, program structure in Turbo Pascal takes the following form:

```
program ProgName;
< declarations:
const
type  freely mixed
var
procedures and functions >
begin { Main body of prog ProgName }
   < statements >
end. { End of prog ProgName }
```

The main body of the program is executed; if it calls additional procedures and functions, they are executed as well. All identifiers—constants, types, variables, procedures, and functions—must be declared before they are used. Procedures and functions are organized in a nearly identical manner.

Program structure in C is a little more flexible:

```
< preprocessor commands >
< type definitions >
< function prototypes > freely mixed
< variables >
< functions >
```

Functions, in turn, have the following structure:

```
<type> FuncName(<parm declarations>)
{
   <local declarations>
   <statements>
}
```

Of all the functions you declare, one must be named **main**; that is the *main body* of your program. In other words, when your Turbo C program starts

execution, **main** is called, and it can in turn call other functions. A C program consists entirely of functions. However, some functions are of type **void**, meaning that they return no values; so they are like Pascal procedures. Also (unlike Pascal) you are free to ignore any values that a function returns.

An Example

Following are two programs, one written in Turbo Pascal, the other written in Turbo C, which illustrate some of the similarities and differences between the two in program structure:

Turbo Pascal	Turbo C

Turbo Pascal

```
program MyProg;
var
  I,J,K   : integer;

function Max(A,B : integer) : integer;
begin
  if A > B
    then Max := A

    else Max := B
end;
{ End of function Max }

procedure Swap(var A,B : integer);
var
  Temp    : integer;
begin
  Temp := A; A := B; B := Temp
end;
{ End of procedure Swap }


begin  { Main body of MyProg }

  I := 10; J := 15;
  K := Max(I,J);
  Swap(I,K);
  Write('I = ',I:2,' J = ',J:2);
  Writeln(' K = ',K:2)
end.
{ End of program MyProg }
```

Turbo C

```
int      i,j,k;

int  max(int a, int b)
{
   if (a > b)
      return(a);
   else
      return(b);
}
/* End of max() */

void swap(int *a, int *b)

{
   int    temp;
   temp = *a; *a = *b; *b = temp;
}
/* End of swap() */

main()
{

   i = 10; j = 15;
   k = max(i,j);
   swap(&i,&k);
   printf("i = %2d j = %2d",i,j);
   printf(" k = %2d\n",k);
}
/* End of main */
```

If we had chosen to, we could have declared *i*, *j*, and *k* inside of **main**, instead of as global variables. In many cases, that's better programming practice, since it eliminates the chance (and temptation) of directly modifying global variables within functions, while still creating variables that exist throughout the course of the program.

Right now the C program on the right probably looks bizarre to you. But by the time you finish this chapter, you'll be right at home with it; in fact, you'll probably be writing things that look even more bizarre.

A Comparison of the Elements of Programming

Back in Chapter 7, we talked about the seven basic elements of programming—output, data types, operations, input, conditional execution, iterative execution, and subroutines. Let's look at those again, seeing how Pascal and C both resemble and differ from each other.

Output

The main output commands in Turbo Pascal are *Write* and *Writeln*. Turbo C, on the other hand, has a variety of commands, based on what exactly you want to do. The most commonly used, and the one that requires the most overhead, is **printf**, which takes the format:

```
printf(<format string>,<item>,<item>,...);
```

where *<format string>* is a *string literal* or a *string variable* (remember, C uses double quotes) and the *<item>*s are optional variables, expressions, etc., that match up with format commands in the format string; see Chapter 7 for more details. To get a newline (=*Writeln*) in C, insert the escape sequence `\n` (newline) at the end of the format string.

Here are some example routines in Turbo Pascal with equivalent (or near equivalent) C routines:

Turbo Pascal	Turbo C
`var`	
`A,B,C : integer;`	`int   a,b,c;`
`Amt   : real;`	`float amt;`
`Name  : string[20];`	`char  name[21]; (or *name)`
`Ans   : char;`	`char  ans;`
`Writeln('Hello, world.');`	`printf("Hello, world.\n");`
`Write('What''s your name? ');`	`printf("What's your name? ");`
`Writeln('"Hello," said John');`	`printf("\"Hello,\" said John\n");`
`Writeln(A,' + ',B,' = ',C);`	`printf("%d + %d = %d\n",a,b,c);`
`Writeln('You owe us $',Amt:6:2);`	`printf("You owe us $%6.2f\n",amt);`
`Writeln('Your name is ',Name,'?');`	`printf("Your name is %s?\n",name)`
`Writeln('The answer is ',Ans);`	`printf("The answer is %c\n",ans);`
`Write(' A = ',A:4);`	`printf(" a = %4d",a);`
`Writeln(' A*A = ',(A*A):6);`	`printf(" a*a = %6d\n",a*a);`

Two other C output routines you'll probably want to be aware of are **puts** and **putchar**. **puts** takes a single string as its argument and writes it out, automatically adding a new line. **putchar** is even simpler: It writes a single character. So, for example, the following commands are equivalent:

```
Writeln(Name);                      puts(Name);
Writeln('Hi, there!'); Writeln;     puts("Hi, there!\n");

Write(Ch);                          putchar(ch);
```

Data Types

Most Turbo Pascal data types have equivalents in Turbo C. C actually has a greater variety of data types, with different sizes of integers and floating-point values, as well as the modifiers **signed** and **unsigned**.

Here's a table giving rough equivalents between Pascal and C data types.

Turbo Pascal			Turbo C		
char	(1 byte)	chr(0 - 255)	char	(1 byte)	-128 - 127
byte	(1 byte)	0 - 255	unsigned char	(1 byte)	0 - 255
integer	(2 bytes)	-32768 - 32767	short	(2 bytes)	-32768 - 32767
			int	(2 bytes)	-32768 - 32767
			unsigned int	(2 bytes)	0 - 65535
			long	(4 bytes)	-2^{31} - $(2^{31}-1)$
			unsigned long	(4 bytes)	0 - $(2^{32}-1)$
real	(6 bytes)	1E-38 - 1E+38	float	(4 bytes)	± 3.4 E ±38
			double	(8 bytes)	± 1.7 E ±308
			long double	(10 bytes)	± 3.4E-4932 - 1.1E+4932
boolean	(1 byte)	False, True	0 = false, nonzero = true		

Note that there is no Boolean data type in C; expressions that require a Boolean value interpret a value of zero as being *false* and any other value as being *true*.

In addition to the data types listed, Turbo C supports *enumerated* data types; however, unlike Pascal, these are effectively just pre-assigned integer constants and are completely compatible with all integral types.

Turbo Pascal	Turbo C
type Days = (Sun,Mon,Tues,Wed, Thurs,Fri,Sat); **var** Today : Days;	enum days = { Sun,Mon,Tues,Wed, Thurs,Fri,Sat }; enum days today;

Operations

Turbo C has all the operators of Turbo Pascal, and then some.

One of the more basic differences between the two languages is how *assignment* is handled. In Pascal, assignment (:=) is a statement. In C, assignment (=) is an operator that may be used in an expression.

Table 9.1 shows a side-by-side comparison of operators in Turbo Pascal and Turbo C. They are listed in order of precedence, with operations grouped together having the same precedence.

Table 9.1: Pascal and C Operators

<table>
<tr><th>Operator</th><th>Pascal</th><th>C</th></tr>
<tr><td>unary minus</td><td>A := –B;</td><td>a = –b;</td></tr>
<tr><td>unary plus</td><td>A := +B;</td><td>a = +b;</td></tr>
<tr><td>logical not</td><td>not Flag</td><td>!flag</td></tr>
<tr><td>bitwise complement</td><td>A := not B;</td><td>a = ~b;</td></tr>
<tr><td>address</td><td>A := Addr(B);</td><td>a = &b;</td></tr>
<tr><td>pointer reference</td><td>A := IntPtr^;</td><td>a = *intptr;</td></tr>
<tr><td>size of</td><td>A := SizeOf(B);</td><td>a = sizeof(b);</td></tr>
<tr><td>increment</td><td>A := Succ(A);</td><td>a++ and ++a</td></tr>
<tr><td>decrement</td><td>A := Pred(A);</td><td>a – – and – – a</td></tr>
<tr><td>multiplication</td><td>A := B * C;</td><td>a = b * c;</td></tr>
<tr><td>integer division</td><td>A := B div C;</td><td>a = b / c;</td></tr>
<tr><td>floating division</td><td>X := B / C;</td><td>x = b / c;</td></tr>
<tr><td>modulus</td><td>A := B mod C;</td><td>a = b % c;</td></tr>
<tr><td>addition</td><td>A := B + C;</td><td>a = b + c;</td></tr>
<tr><td>subtraction</td><td>A := B – C;</td><td>a = b – c;</td></tr>
<tr><td>shift right</td><td>A := B shr C;</td><td>a = b » c;</td></tr>
<tr><td>shift left</td><td>A := B shl C;</td><td>a = b « c;</td></tr>
<tr><td>greater than</td><td>A > B</td><td>a > b</td></tr>
<tr><td>greater or equal</td><td>A >= B</td><td>a >= b</td></tr>
<tr><td>less than</td><td>A < B</td><td>a < b</td></tr>
<tr><td>less or equal</td><td>A <= B</td><td>a <= b</td></tr>
<tr><td>equal</td><td>A = B</td><td>a == b</td></tr>
<tr><td>not equal</td><td>A <> B</td><td>a != b</td></tr>
<tr><td>bitwise AND</td><td>A := B and C;</td><td>a = b &</td></tr>
<tr><td>bitwise OR</td><td>A := B or C;</td><td>a = b | c;</td></tr>
<tr><td>bitwise XOR</td><td>A := B xor C;</td><td>a = b ^ c;</td></tr>
<tr><td>logical AND</td><td>Flag1 and Flag2</td><td>flag1 && flag2</td></tr>
<tr><td>logical OR</td><td>Flag1 or Flag2</td><td>flag1 || flag2</td></tr>
<tr><td>assignment</td><td>A := B;
A := A <op> B;</td><td>a = b;
a <op> = b;</td></tr>
</table>

There are some important differences in C operators and operator precedence.

First, the *increment* (++) and *decrement* (--) operators can be placed before or after the variable name. If the operator is placed before the variable, then the variable is incremented (or decremented) before the rest of the expression is evaluated; if after, the expression is evaluated first, then the variable is incremented (or decremented).

Second, the logical operators in C (&&, ||) are *short-circuit* operators. This means that if the first item determines the truth of the expression, then the second is never evaluated. So, unlike Pascal, C lets you safely write this:

```
while (i <= limit && list[i] != 0) ... ;
```

where *limit* is the largest allowable index into the *list* array.

If the first item (`i <= limit`) is *false*, then C knows that the entire expression must be *false*, and it doesn't evaluate the second item (`list[i] != 0`), which would be an index range error.

Third, C allows you to take the general expression

```
A = A <op> B
```

where *<op>* is any binary operator (except for && and ||) and replace it with

```
A <op> = B
```

So, for example, instead of `A = A * B`, you could write `A *= B`, and so on.

Input

Again, in Turbo Pascal, you have one basic input command, *Read()*, with a few variations (*Readln()*, *Read(f,)*, etc.). In Turbo C, the main function used for keyboard input is **scanf**, which takes the format:

```
scanf(<format string>,<addr1>,<addr2>,...);
```

where *<format string>* is a string with format indicators (as in **printf**), and each *<addr>* is an address into which **scanf** stores the incoming data. This means that you will often need to use the address-of operator (&). There are other commonly used commands as well: **gets**, which reads in an entire string until you press *Enter*; and **getch**, which reads a character straight from the keyboard with no echo.

Here are some Pascal input commands with corresponding C commands:

Turbo Pascal	Turbo C
`Readln(A,B);`	`scanf("%d %d",&a,&b);`
`Readln(Name);`	`scanf("%s",name);` `/* or gets(name); */`
`Readln(X,A);`	`scanf("%f %d",&x,&a);`
`Readln(Ch);`	`scanf("%c",ch);`
`Read(Kbd,Ch);`	`ch = getch();`

Be aware of one important distinction between these two ways of reading in a string (**scanf** and **gets**). **scanf** reads in all characters until whitespace (blanks, tabs, newline) is encountered. By contrast, **gets** will read *everything* in (including blanks and tabs) until you press *Enter*.

Block Statement

Both Pascal and C have the concept of a *block statement* (a collection of statements that can be put in anywhere a single statement can). In Pascal, the block statement takes the form

```
begin <statement>; <statement>; ... <statement> end;
```

In C, it takes this form:

```
{ <statement>; <statement>; ... <statement>; }
```

While the form is very similar, there are two important differences:

- In Pascal you don't have to put a semicolon after the last *<statement>* in a block; in C, you do.
- In C you never put a semicolon after the closing brace (}) of a block; in Pascal, you might have to.

Conditional Execution

Both Pascal and C support two *conditional execution constructs*: the **if/then/else** statement and the **case** statement.

The **if/then/else** is very similar for both of them:

```
if <bool expr>                  if (<expr>)
  then <statement>                 <statement>;
  else <statement>              else
                                   <statement>;
```

In both Pascal and C, the **else** clause is optional, and *<statement>* can be replaced with a block statement (as already described). There are a few important differences, though.

- In C, the *<expr>* doesn't have to be Boolean; it just has to somehow resolve to a zero or nonzero value, where zero is considered *false*, and nonzero is considered *true*.
- In C, the *<expr>* must be in parentheses.
- In C, there is no **then**.
- In C, semicolons are always required after the statements—unless, of course, you have a block statement there instead.

Here are a few examples in Pascal and C:

Turbo Pascal	Turbo C

`if B = 0`
`  then Writeln('C is undefined')`
`else begin`
`  C := A div B;`
`  Writeln('C = ',C)`
`end;`
	`if (B == 0)`
	`   puts("c is undefined");`
	`else {`
	`   c = a / b;`
	`   printf("c = %d\n",c);`
	`}`

`C := A * B;`
`if C < > 0`
`  then C := C + B;`
`  else C := A`
	`if ((c = a * b) != 0)`
	`   c += b;`
	`else`
	`   c = a;`

The **case** statement is also implemented in both Pascal and C (in which it's known as the **switch** statement), but with some important differences.

Here's the general format for Pascal and C:

Turbo Pascal

```
case <expr> of
  <list>    : <statement>;
  <list>    : <statement>;
  ...
  <list>    : <statement>;
  else <statements>
end;
```

Turbo C

```
switch (<expr>) {
   case <item>  : <statements>
   case <item>  : <statements>
   ...
   case <item>  : <statements>
   default      : <statements>
}
```

Besides the cosmetic changes, there are critical distinctions as well.

First, in Pascal, *<list>* can be a list of values; in Turbo Pascal, it can include ranges (*A*...*Z*) as well. In C, *<item>* is exactly one value. In both languages, you're limited to ordinal and constant values: integers, characters, and enumerated data types.

Second (and this is *very* important), in Pascal, *<statement>* is either a single statement or a block statement; once it is executed, the rest of the **case** statement is skipped over. In C, *<statements>* consists of zero or more statements, each ending with a semicolon. However, once they are executed, control does *not* pass to the end of the **switch** statement; instead, it continues down the list of *<statements>* until and unless it hits a **break** statement. Then, and only then, is the rest of the **switch** statement skipped. It may help to think of each `case <item> :` as a label, with the `switch(<expr>)` statement determining which one to jump to.

Here are a few examples:

Turbo Pascal

```
case Ch of
  'C' : DoCompile;
  'R' : begin
    if not Compiled
    then DoCompile
    RunProgram;
  end;
  'S' : SaveFile;
  'E' : EditFile;
  'Q' : begin
    if not Saved
    then SaveFile
  end;
end;
```

Turbo C

```
switch (ch) {
   case 'C' : DoCompile(); break;
   case 'R' :
      if (!compiled)
         DoCompile();
      RunProgram();
      break;
   case 'S' : SaveFile(); break;
   case 'E' : EditFile(); break;
   case 'Q' :
      if (!saved)
         SaveFile();
      break;
   }
```

Turbo Pascal

```
case Today of
  Mon..Fri : Writeln('go work!');
  Sat,Sun  : begin
    if Today = Sat then begin
      Write('clean the ');
      Write('yard and ')
      end;


    Writeln('relax!')
    end
end;
```

Turbo C

```
switch (today) {
   case Mon :
   case Tue :
   case Wed :
   case Thur:
   case Fri : puts("go work!");
      break;
   case Sat : printf("%s","clean the "
      "yard and ");
   case Sun : puts("relax!");

}
```

Note the second set of examples. The `case <item>` parts of the **switch** statement label each case you want to handle; in the case of *Mon* through *Thur*, the *<statements>* sections are empty, and control falls downward until it finds the statements labeled by `case Fri :`. The **break** statement then causes control to skip to the end of the **switch** statement. However, the program takes advantage of the same feature with the weekend; the label `case Sat :` causes the **printf** statement to execute, after which control falls to the following **puts** statement.

Iteration

C, like Pascal, has three types of loops : **while, do...while**, and **for**, which correspond closely to Pascal's three loops (**while**, **repeat..until**, and **for**). We'll present them in that order.

The while Loop

Of the three loops, the **while** loop is most similar in both languages. Here are the formats:

```
while <bool expr> do
  <statement>;
```

```
while (<expr>)
   <statement>;
```

In both languages, you use a block statement to put more than one statement in the loop. The only real difference, again, is C's greater flexibility in what it accepts for *<expr>*. For example, compare the following two loops:

```
Read(Kbd,Ch);
while Ch <> 'q' do begin
  Write(Ch); Read(Kbd,Ch)
end;
```

```
while ((ch = getch()) != 'q')
   putchar(ch);
```

The do..while Loop

The **do...while** loop is similar to Pascal's **repeat...until** loop; here are the formats:

```
repeat
  <statements>
until <bool expr>;
```

```
do
   <statement>;
while (<expr>);
```

But, there are two important differences between these two loops:

- The **do..while** loop executes *while <expr>* is *true*, whereas the **repeat..until** executes *until <bool expr>* is *true.*
- The **repeat..until** statement doesn't require a block statement for multiple statements, while the **do..while** does.

Here's an example of each:

```
repeat                                   do {
  Write('Enter a value: ');                 printf("Enter a value: ");
  Readln(A)                                 scanf("%d",&a);
until (Low <= A) and (A <= High);        } while (a < low || a > high);
```

Note another important difference between C and Pascal: In C, relational operators (>, <, etc.) have a higher precedence than logical operators (&&, | |). This means that you don't have to surround each relational expression with parentheses, as you often have to in Pascal.

The for Loop

The **for** loop shows the greatest differences between Pascal and C. In Pascal, the **for** loop is rather fixed and inflexible; in C, it is almost too flexible, allowing some constructs that tend to lose all resemblance to a **for** loop.

Here are the formats of both:

```
for <indx> := <start> to <finish> do      for (<expr1>; <expr2>; <expr3>)
  <statement>;                               <statement>;
```

In C (as it really is in Pascal), the **for** statement is simply a special case of the **while** statement. The given format is equivalent to

```
<expr1>;
while (<expr2>) {
   <statement>;
   <expr3>;
}
```

<expr1> is used for initialization, *<expr2>* for testing the end of the loop, and *<expr3>* to increment or otherwise modify the loop variable(s).

Here are a few examples, some of which use the **while** loop in Pascal:

Turbo Pascal	Turbo C

`for I := 1 to 10 do begin`
`  Write('I = ',I:2);`
`  Write('  I*I = ',(I*I):4);`
`  Writeln('  I**3 = ',(I*I*I):6)`
`end;`
	`for (i = 1; i <= 10; i++) {`
	`   printf("i = %2d  ",i);`
	`   printf("i*i = %4d  ",i*i);`
	`   printf("i**3 = %6d\n",i*i*i);`
	`}`

`I := 17; K := I;`
`while (I > -450) do begin`
`  K := K + I;`
`  Writeln('K = ',K,' I = ',I);`
`  I := I - 15`
`end;`
	`for (i = 17, k = i; i >- 450; i -= 15)`
	`{`
	`   k += i,`
	`   printf("k = %d i = %d\n",k,i);`
	`}`

`X := D/2.0;`
`while (Abs(X*X-D) > 0.01) do`
`  X := (X + D/X)/2.0;`
	`for (x = d/2; fabs(x*x-d) > 0.01;`
	`     x = (x+d/x)/2)`
	`     ; /* Empty statement */`

Notice that these loops are doing more and more inside the **for** section, until the last one actually has no statement to execute; all the work is done within the header of the loop itself.

Subroutines

Both Pascal and C have subroutines; Pascal has procedures and functions, while C has just functions. However, you can declare functions to be of type **void**, which means that they return no value at all; if you want to, you can also ignore the value that a function does return.

Here are the formats for functions in both languages:

Turbo Pascal	Turbo C

`function FName(<parm decls>) : <type>;`
`  <local declarations>`
`begin`
`  <statements>`
`end;`
	`<type> FName(<parm decls>)`
	`{`
	`   <local declarations>`
	`   <statements>`
	`}`

In Pascal, *<parm decls>* takes the form *<pnames>* : *<type>*; for each set of parameters, while in C it's *<type>* *<pnames>*.

There are other important differences as well, but they're best shown by example. Here are a few:

Turbo Pascal	Turbo C

`function Max(A,B : integer) : integer;`
`begin`
`  if A > B`
`    then Max := A`
`    else Max := B`
 	
`end;`
	`int max(int a, int b)`
	`{`
	`   if (a > b)`
	`      return(a);`
	`   else`
	`      return(b);`
	`}`

Note that in C the **return** statement is used to return a value through the function, while Pascal has you assign a value to the function name.

Turbo Pascal	Turbo C
`procedure Swap(var X,Y : real);`	`void swap(float *x, float *y)`
`var`	
`  Temp    : real;`	
`begin`	`{`
	`   float  temp;`
`  Temp := X;`	`   temp = *x;`
`  X := Y;`	`   *x = *y;`
`  Y := Temp`	`   *y = temp;`
`end;`	`}`

In Pascal, you have two types of parameters: *var* (pass-by-address) and *value* (pass-by-value). In C, you only have pass by value. If you want a pass-by-address parameter, then that's what you have to do: Pass the address, and define the formal parameter as a pointer. That's what was done for **swap**.

Here's sample code calling these routines:

Turbo Pascal	Turbo C
`Q := 7.5;`	`q = 7.5;`
`R := 9.2;`	`r = 9.2;`
`Writeln('Q = ',Q:5:1,' R = ',R:5:1);`	`printf("q = %5.1f r = %5.1f\n",q,r);`
`Swap(Q,R);`	`swap(&q,&r);`
`Writeln('Q = ',Q:5:1,' R = ',R:5:1);`	`printf("q = %5.1f r = %5.1f\n",q,r);`

Note the use of the address-of operator (&) in the C code when passing *q* and *r* to **swap**.

Function Prototypes

There is an important difference between Pascal and C concerning functions: Pascal always does error-checking to make sure that the number and types of parameters declared in the function match those used when the function is called.

In other words, suppose you define the Pascal function

```
function Max(I,J : integer) : integer;
```

and then try to call it with real values (`A := Max(B,3.52);`).

What will happen? You'll get a compilation error telling you that there is a type mismatch, since 3.52 is not an acceptable substitute for an integer.

Not so in C. By default, C does *no* error checking on function calls: It does not check the number of parameters, parameter types, or even the type returned by the function. This allows you a certain amount of flexibility, since you can call a function before it is ever defined. But it can also get you into deep trouble; see "Pitfall #2" later in this chapter. So, how do you avoid this?

Turbo C supports *function prototypes*. You can think of these as being somewhat analogous to **forward** declarations in Pascal. You typically place function prototypes near the start of your file, before you make any calls to those functions. The key point to remember is that the function prototype—which is a kind of declaration—must precede the actual call to the function.

A function prototype takes the format:

```
<type> FName(<type> <pname>, <type> <pname>, etc. );
```

This is very similar to how Pascal declares functions, but with a few differences. Commas (not semicolons) are used to separate the definition of each parameter; also, you can't list multiple *<pname>*s for a single *<type>*.

Here are some sample prototypes, based on the routines already given, as well as on the routines in "A Major Example" (following):

```
int   max(int a, int b);
void  swap(float *x, float *y);
void  swapitem(listitem *i, listitem *j);
void  sortlist(list l, int c);
void  dumplist(list l, int c);
```

Unlike the Pascal **forward** statement, the C function prototype does not force you to do anything different when you actually define your function; in other words, you define your function just as you would otherwise (or you can define it using modern C style). In fact, if your function definition doesn't match the prototype, Turbo C will give you a compilation error.

Turbo C supports both the classic and modern styles, although—since C is migrating toward using the modern style—we recommend that you use function prototypes and prototype-style function definitions.

Using function prototypes can prevent a lot of problems, especially when you start compiling libraries of C routines. You should create a separate file and put in it function headers for all the routines in a given library. When you want to use any routines in that library, you include the header file into your program (with the directive `#include`). That way, error checking can take place at compile time, possibly saving you a fair amount of grief.

A Major Example

Now here's a long example, a complete program using most of what you've learned up until now, and then some. It defines an array *myList*, whose length is defined by the constant *LMax* and whose base type is defined as *ListItem* (which here is just *Integer*). It initializes that array to a set of numbers in descending order, displays them using the *DumpList* routine, sorts them in ascending order with *SortList*, then displays them again.

Note that the C version of this program is not necessarily the best C version. It has been written to correspond as much as possible to the Pascal version; the few places where it doesn't correspond are designed to demonstrate certain differences between C and Pascal.

Turbo Pascal	Turbo C

```
program DoSort;
const
  LMax = 100;
type
  Item = integer;
  List = array[1..LMax] of Item;
var
  myList  : List;
  Count,I : integer;
  Ch      : char;

procedure SortList(var L : List;
  C : integer);
var
  Top,Min,K : integer;

procedure SwapItem(var I,J : Item);
var
  Temp : Item;
begin

  Temp := I; I := J; J := Temp
end; { of proc SwapItem }


begin { Main body of SortList }

  for Top := 1 to C-1 do begin
    Min := Top;
    for K := Top + 1 to C do
      if L[K] < L[Min]
        then Min := K;
    SwapItem(L[Top],L[Min])
  end
end; { of proc SortList }

procedure DumpList(L : List;

var
  I : integer;
begin

  for I := 1 to C do
    Writeln('L[',I:3,'] = ',L[I]:4)
end; { End of proc DumpList }
```

```
#define LMAX  100

typedef int   item;
typedef item  list[LMAX];

list  myList;
int   count, i;

void swapitem(item *i,item *j)


{
   item  temp;
   temp = *i; *i = *j; *j = temp
} /* swapitem */

void sortlist(list l, int c)
{
   int  top,min,k;
   for (top = 0; top < c-1; top++){
      min = top;
      for (k = top + 1; k <= c; k++)
         if (l[k] < l[min])
            min = k;
      swapitem(&l[top],&l[min]);
   }
} /*end of sortlist */

void dumplist(list l,int c)
              C : integer);


{
   int    i;
   for (i = 0; i <= c; i++)
      printf("l[%3d] = %4d\n",i,l[i]);
} /* dumplist() */
```

```
begin { Main body of DoSort }
  for I := 1 to LMax do
    myList[I] := Random(1000);
  Count := LMax;
  DumpList(myList,Count);
  Read(Kbd,Ch);
  SortList(myList,Count);
  DumpList(myList,Count);
  Read(Kbd,Ch)
end. { of DoSort }
```

```
main()
{
   for (i = 0; i < LMAX; i++)
      myList[i] = rand() % 1000;
   count = LMAX;
   dumplist(myList,count);
   getch();
   sortlist(myList,count);
   dumplist(myList,count);
   getch();
} /* main */
```

There are some important things to note here:

- In the Pascal version, we nested the procedure *SwapItem* inside of the procedure *SortList*; in C, you can't nest functions, so we had to move **swapitem** outside of **sortlist**.
- In C, arrays always start at location *0* and go up through *size-1*. For example, the first location in *myList* is *myList[0]*, while the last is *myList[LMAX-1]*. That's why the various **for** loops are set up the way they are.
- We didn't have to use the address-of and pointer operators when we passed *myList[]* to **sortlist**. Why? Because C always passes the address of arrays used as parameters, rather than the array itself, since it can't pass all the values in the array without causing serious problems. Likewise, when we declare the formal parameter `list l;` in **dumplist** and **sortlist**, C knows to create that array at the address passed to it, so we don't have to mess with pointer operators.
- We didn't need function prototypes in this example because each function is defined before it is used. If we wanted to, we could place them in the program anywhere *after* defining the data types *item* and *list*, and they would have looked like this:

```
void swapitem(item *i, item *j);
void sortlist(list l, int c);
void dumplist(list l, int c);
```

Again, note that using the function prototype does not change how you define the function.

A Survey of Data Structures

In this section we'll give you an overview of how data structures in C do (and don't) resemble Turbo Pascal data structures. The elements we'll talk about are pointers, arrays, strings, structures, and unions.

Pointers

It's possible to program for a long time in Pascal and never use pointers; not so in C. Why? Because, as mentioned before, C only uses pass-by-value parameters for its functions. If you want to modify a formal parameter and have that change the actual parameter, you have to pass the address yourself, then declare the formal parameter to be a pointer to the actual data type. Furthermore, strings are implemented in C as pointers to **char**, so any string manipulation will need pointers as well.

Here's a quick comparison of pointer declarations and use in Pascal and C, with a few examples of each:

	Turbo Pascal	Turbo C
Declaration:	<pname> : ^<type>;	<type> *<pname>;
	IntPtr : ^Integer;	int *intptr;
	Buff1 : ^Intarray;	int buff1[];
	Buff2 : **array**[0..N] **of** IntPtr;	int *buff2[];
	PHead : ^Node;	node *phead;
	Head : Node;	node head;
Use:	<pname>^ := <value>;	*<pname> = <value>;
	IntPtr^ := 22;	*intptr = 22;
	Buffer^[152] := 0;	buff1[152] = 0;
	PHead^.Next := **nil**;	(*phead).next = NULL;
		/* or phead->next = NULL; */

Note the use of parentheses for the last example (*phead*) in C, as well as the special symbol (->) in the second version for *phead*. Here are some more examples:

```
1.  Buff1^[152]   := 0;         buff1[152]     = 0;
2.  Buff2[152]^   := 0;         *buff2[152]    = 0;
3.  Head.Data^    := 0;         *head.data     = 0;
4.  Head.Next     := nil;       head.next      = NULL;
5.  PHead^.Next   := nil;       (*phead).next = NULL;     /* or phead -> next =
                                                                    NULL; */
```

The first example presumes that *buff1* points to an array of integers.

The second example indicates that *buff2* is an array of pointers to integers, so it is indexed before it is referenced.

The third assumes that *head* is a **record** (a **struct** in C) with a field *next*, which is a pointer to an integer.

The fourth assumes that *head* also has a field *next*, which is a pointer to something (it's unclear what).

The last example shows that *phead* is a *pointer* to a **record** (also a **struct**), and that record has a field *next*, which is a pointer.

The symbol -> is used as shorthand notation; that is, the expression

```
pname -> fname = value;
```

says that *pname* is a pointer to some type of record, *fname* is the name of some field in that record, and that *value* is going to be assigned to the *fname* field in the record to which *pname* points.

Arrays

Arrays are fairly simple creatures in C, compared to Pascal. Arrays in C can have integer, character, or enumerated-type indices, while Pascal allows you to use *any* ordinal type. All array index ranges in C start at 0 and go to *n*-1 (where *n* is the size of the array). This is very unlike Pascal, which lets you start and end the index ranges wherever you choose.

In C, array indexing is like pointer arithmetic, and the same identity holds true: The Pascal *a*[*i*] is equivalent to both *a*[*i*] and *(*a* + *i*) in C.

The general format for arrays in the two languages follows:

```
<name> : array[<low>..<high>] of <type>;        <type>  <name>[<size>];
```

where <*size*> is equal to (1 + <*high*> – <*low*>).

Multi-dimensional arrays in C are declared much like in Pascal: Either <*type*> is itself an array of some sort, or you add additional sizes on the end, like this:

```
<type> <name>[<size1>][<size2>][<size3>];
```

Note that, unlike Pascal, you cannot write `arr[x][y]` as `arr[x,y]` (see "Pitfall #5" following).

In C, a block of memory large enough for <*size*> instances of <*type*> is set aside, and <*name*> is a constant pointer to the beginning of that block.

This aids passing arrays to functions; more importantly, it means that (unlike Pascal) the function does not have to know how big the array is at compile time.

The result: You can pass arrays of different sizes (but the same type) to a given function.

Consider, for example, the following function, which receives an array of type **int** and returns the lowest value in the array:

```
int  amin(int a[], int n); /* Function declaration */
{
   int    min,i;

   min = a[0];

   for (i = 1; i < n; i++)
      if (a[i] < min)
         min = a[i];
   return(min);
}
```

You can pass an integer array of any size to this function; it will find the lowest value of the first *n* elements. Losing this flexibility is one of the biggest complaints C programmers have when using Pascal.

Strings

Standard Pascal doesn't define strings as a separate data type; Turbo Pascal does, and supplies a number of procedures and functions for working with them.

C (including Turbo C) does not define a separate string data type; instead, a string is defined as either an array of **char** or a pointer to **char**, which (as you've seen) are almost the same thing.

Here are some comparative declarations:

Turbo Pascal	Turbo C
<name> : **string**[<size>];	char <name>[<size>];
type	
BigStr = **string**[255];	typedef char bigstr[256];
StrPtr = ^BigStr;	typedef char *strptr;
var	
Line : **string**[80];	char line[81];
Buffer : BigStr;	bigstr buffer;
Word : **string**[35];	char word[36];
Ptr : StrPtr;	strptr ptr;

The key differences between strings in Turbo Pascal and strings in Turbo C are closely tied to the differences between arrays in the two languages.

In Turbo Pascal, the declaration

```
S : string[N]
```

is equivalent to

```
S : array of [0..N] of char
```

The string has a maximum length of *N* characters; the current length is stored in *S[0]*, while the actual string itself starts in location *S[1]*. You can directly assign string literals and constants to a string variable; Pascal will do the byte-by-byte transfer and correctly adjust the length.

In Turbo C, you can declare a string as

```
char strarr[N]
```

or as

```
char *strptr
```

The first declaration sets aside *N* memory for holding a string, then represents the address of those bytes with *strarr*. The second declaration only sets aside bytes for the pointer *strptr*, which points to **char** types.

In C, a string's length is not stored separately; instead, a *string terminator* is used to mark the end of the string. This terminator is the *null character* (ASCII 0), which requires an extra byte at the end of the string; the string itself starts in *strarr[0]*.

Because of this, the string *strarr* can only hold *N-1* "real" characters, since 1 byte will have to be reserved for the null-terminator. That's why the C declarations in the comparison table all have lengths one greater than their corresponding Pascal declarations.

Furthermore, since *strarr* is not the actual collection of bytes, you cannot directly assign string literals. Instead, you must use the routine **strcpy** (or one of its derivatives) to do a byte-by-byte transfer from one string to another: `strcpy(strarr,"Hello, world!");`. However, you can directly read into *strarr* using **scanf** or **gets**.

The other method of string declaration, `char *strptr`, requires you to use more care. In this case, *strptr* is just a pointer to **char**; no space for any string has been allocated, just the few bytes for the pointer itself.

You can assign string literals directly to *strptr*; since those literals are created as part of the object code itself, you merely assign their addresses to *strptr*. If you assign *strarr* to *strptr*, then both *strarr* and *strptr* now point to the same string; the same thing occurs if you assign another string pointer to *strptr*.

So, how do you get *strptr* to point to its own private string instead of somewhere else? By allocating space to it:

```
strptr = (char *) malloc(N);
```

This will set aside *N* bytes of available memory, using the **malloc** routine and assign to *strptr* the address of that string. You can then use *strcpy* to copy strings (literals and variables) into those allocated bytes.

The Pascal equivalents for this (*StrPtr*, *Ptr*) are only very rough equivalents. Instead of being *^Char*, *StrPtr* is defined as *^BigStr*. This is so that Turbo Pascal will recognize *Ptr* as being a string; it also helps to avoid any range-checking problems. Note in the following example that only the amount of space requested is actually allocated to *Ptr*.

Here is a list of roughly comparable statements; refer to the *Turbo C Reference Guide* for a complete list of Turbo C's string (**str...**) functions. These statements presume the type declarations given in the previous comparison:

Turbo Pascal	Turbo C

var
  Line,Name   : BigStr;
  First,Temp : **string**[80];
  Ptr   : StrPtr;
  I,Len,Err   : integer;
	main ()
	{
	   bigstr line,name;
	   char first[81],temp[81];
	   char *ptr;
	   int i,len;
	   extern char *strchr(char *s,char ch);

begin
  Write('Enter name: ');
  Readln(Name);
  I := Pos(' ',Name);
  if I = 0 **then**
    First := Name
  else
    First := Copy(Name,1,I-1);
  Len := Length(First);
  Writeln('Len = ',Len);
  Temp := Concat('Hi, ',Name);
 	
  Writeln(Temp);
  if Name <> First
    then Name := First;
  I := 823; Str(I,Temp);
  Val(Temp,I,Err);
  GetMem(Ptr,81);
  Ptr^ := 'This is a test.';
  Writeln('Ptr = ',Ptr^);
  FreeMem(Ptr,81);
end.
	
	   printf("Enter name: ");
	   gets(name);
	   ptr = strchr(name,'.');
	   if (ptr == NULL)
	      strcpy(first,name);
	   else
	      strncpy(first,name, ptr-name-1);
	   len = strlen(first);
	   printf("len = %d\n",len);
	   strcpy(temp,"Hi, ");
	   strcat(temp,name);
	   puts(temp);
	   if (strcmp(name,first))
	     strcpy(name,first);
	   i = 823; sprintf(temp,"%d",i);
	   i = atoi(temp);
	   ptr = (char *) malloc(81);
	   strcpy(ptr,"This is a test.");
	   printf("ptr = %s\n",ptr);
	   free(ptr);
	}

The use of *Ptr* in the Pascal source code is something of a kludge; it's included here only to give you a feeling for what the equivalent C code does.

One last point: The function prototypes for the C routines called in this example are listed in header (.h) files; so, for proper error-checking, you should place the following `#include` statements at the start of the Turbo C program.

```
#include  <stdio.h>
#include  <string.h>
#include  <stdlib.h>
#include  <alloc.h>
```

Structures

Both Pascal and C allow you to define aggregate, heterogeneous data structures. In Pascal, they're called *records;* in C, *structures*. Here's the format for both:

Turbo Pascal	Turbo C
type	typedef struct {
<rname> = **record**	<type> <fnames>;
<fnames> : <type>;	<type> <fnames>;
<fnames> : <type>;	...
...	<type> <fnames>;
<fnames> : <type>	} <rname>;
end;	
var	
<vnames> : <rname>;	<rname> <vnames>;

There's also a more concise format in C for directly declaring structure variables, much as there is in Pascal:

Turbo Pascal	Turbo C

`var`
`  <vnames> : record`
`  <fnames> : <type>;`
`  ...`
`  <fnames> : <type>`
`end;`
	`struct <rname> {`
	`   <type>  <fnames>;`
	`   ...`
	`   <type>  <fnames>;`
	`}  <vnames>;`

In this case, *<rname>* of the structure is optional; you should put it there if you plan to declare other variables to be of type *<rname>*. Beyond that, records in Pascal and structures in C are pretty much the same. Here's an example:

Turbo Pascal

```
type
  Student = record
  Last,First : string[20];
  SSN    : string[11];
  Age    : integer;
  Tests  : array[1..5] of integer;
  GPA    : real
end;

var
  Current : Student;
begin
  Current.Last = 'Smith';
  Current.Age = 21;
  Current.Tests[1] = 97;
  Current.GPA = 3.94;
end.
```

Turbo C

```
struct student {
   char   last[20],first[20];
   char   ssn[11];
   int    age;
   int    tests[5];
   float  gpa;
} current ;

main()
{

   strcpy(current.last = "Smith");
   current.age = 21;
   current.tests[0] = 97;
   current.gpa = 3.94;
}
```

The only major difference between Pascal and C here is that Pascal has the **with** statement and C doesn't. We could rewrite the Pascal previous code to say **with** `Current` **do** and then refer to the fields without the `Current` in front of them. In C, you always have to have the `current.` in front. C also has the *member access* operator (->), which is used when the identifier on the left of

the operator is a pointer to a structure rather a structure itself. For example, if *pstudent* is a pointer to a **struct**, then

```
pstudent -> last = "Jones";
```

assigns the string `Jones` to the last name.

Unions

Again, Pascal and C support similar concepts. In Pascal, it is called a *free union variant record*; in C, it's just called a *union*. Here are the definitions of each, along with an example:

Turbo Pascal	Turbo C

```
type
  <uname> = record
  <fieldlist>
case <type> of
  <vlist> : (<fieldlist>);
  <vlist> : (<fieldlist>);
  ...
  <vlist> : (<fieldlist>)
end;
```

```
union <uname> {
   <type>  <fnames>;
   <type>  <fnames>;
   ...
   <type>  <fnames>;
} ;
```

In the Pascal version, *<fieldlist>* is the usual record sequence of *<fnames>* : *<type>*;, repeated as needed.

There are two major differences between Pascal and C on this one:

- First, Pascal makes you put the union at the end of a regular record, whereas C does not. However, you can declare the union first, then declare a field in a structure to be of that **union** type.
- Second, Pascal allows you to have multiple types for each variant in the union. C does let you have multiple fields (hence *<fnames>*), but all must be of the same type.

Here's a sample to study, written to make the Pascal and C versions as close to each other as possible (although, admittedly, they are not fully equivalent):

Turbo Pascal	Turbo C

Turbo Pascal

```
type
  trick_word = record
case integer of
  0 : (w : integer);
  1 : (lob,hib: byte);
end;
var xp:trick_word;
```

Turbo C

```
typedef union {
   int w;
   struct {
      char lob;
      char hib;
   }b;
} trick_word;
trick_word xc;
```

Note that neither the C nor the Pascal definition of *trick_word* is portable. They both depend on the byte-order of the 8086.

In C unions, as with structures, you can insert a <*vnames*> field between the closing brace and the semicolon to directly declare variables of that type. In that case, you can leave off <*uname*> if you're not going to declare any more such variables. Field references in Pascal are *xp.w*, *xp.hib*, and *xp.lob*; in C, they are *xc.w*, *xc.b.hib*, and *xc.b.lob*.

Programming Issues

As a Pascal programmer, you shouldn't have a difficult time getting up to speed with Turbo C. But there are a few areas of programming that are implemented somewhat differently in the two languages. We'll discuss each of these programming issues in this section.

Case Sensitivity

Pascal is not case sensitive; C is. This means that the identifiers *indx*, *Indx*, and *INDX* all refer to the same variable in Pascal but would refer to three different variables in C.

Note: Since function calls are not resolved until the C program is linked, differences due to case may not show up until then. For your own good, be careful with case in C.

Type-Casting

Pascal, as a rule, allows only limited type-casting (converting data from one type to another). The function *Ord()* will cast from any ordinal type to integer; *Chr()* will cast from integer (or a related type) to char. Turbo Pascal allows some additional type-casting (called *retyping*) between all ordinal types (integer, char, boolean, and enumerated data type). C is much freer, allowing you to attempt to cast from any type to any type, with results that are not always favorable.

Here's the standard format for each, with a few examples:

Turbo Pascal	Turbo C
`<var> := <type>(<expr>);` `var Ch : char;`	`<var> = (<type>)<expr>;` `char ch;`

`I  := integer(Ch);`
`Ch := char(Today);`
`Today := Days(3);`
	`i = (int) ch;`
	`ch = (char) today;`
	`today = (days) 3;`

In addition, Turbo C will do a lot of automatic type-casting, mostly between types that are *integer compatible* (types whose underlying representation is an integer value). Because of that, all three previous statements could have left out the explicit cast: You could have written

```
i = ch;
ch = today;
today = 3;
```

Constants, Variable Storage, Initialization

Turbo Pascal does not initialize variables that you declare. Neither does it preserve the value of variables declared within procedures (and functions) between calls to those subroutines. The major exception to this is that typed constants are initialized, and they will hold their values *between* calls to a subroutine in which they are defined (including any value you might assign to them during execution). In C, all global variables are initialized to 0 by default unless you explicitly initialize them to a different value.

Note: You should keep in mind that uninitialized variables are not necessarily created in the order that they are declared.

Turbo C gives you two types of constants, allows you to pre-initialize any variables, and lets you declare variables within a function as being *static*.

Constant Types

The two types of constants take the format:

```
#define <cname> <value>
const <type> <cname> = <value>;
```

The first type (`#define...`) more closely matches Pascal's **const** definition, in that *<value>* is directly substituted wherever *<cname>* is found.

The second type (`const...`) is more like Turbo Pascal's typed constant, except that you really can't change *<cname>*; any attempt to modify or assign a new value to it will result in a compilation error.

Bear in mind that C allows constant expressions, for example, `char s[SIZE + 1]`. A **const** variable won't do here. A manifest constant will, but it's substituted as if by a word processor with sometimes surprising results:

```
#define MAX_SIZE   80 + 1
char   s[MAX_SIZE * 2];                                /* 82, not 162! */
```

A more careful definition would be

```
#define MAX_SIZE   (80 + 1)
```

In other words, to be safe, it's always wise to parenthesize expressions in a `#define`.

Variable Initialization

Turbo C lets you initialize any variable in a manner that does match Turbo Pascal's typed constant. The format is as follows:

```
<type> <vname> = <value>;
```

Items requiring more than one value (arrays, structures) should have the values enclosed in braces and separated by commas { *"like_this"*, *"and_this"*, *"and_this_too"* }.

```
int x = 1, y = 2;
char name[] = "Frank";
char answer = 'Y';
char key = 3;
char list[2][10] = {"First", "Second"};
```

Variable Storage

C defines several storage classes for variables; the two most important are *external* and *automatic* (local). Global variables (those declared outside of any function, including **main**) are by default external. This means that they are initialized to 0 at the start of program execution—unless, of course, you initialize them yourself.

Variables declared within functions (including within **main**) are, by default, automatic. They are not initialized to anything, unless you do it, and they lose their values between calls to that function. However, you can declare such variables to be *static*; that way, they will be initialized to 0 (once, at the start of program execution), and they will retain their values between calls to the function.

In the following example

```
init test(void)
{
   int i;
   static int count;

   ...
}
```

the variable *i* resides on the stack and must be initialized by function **test** each time the function is called. The static variable *count*, on the other hand, resides in the global data area and is initialized to zero when the program is first executed. *Count* retains its previous value each time function **test** is invoked.

Dynamic Memory Allocation

In Turbo Pascal, there are several different methods for managing the heap. Given these Turbo Pascal declarations

```
type
   ItemType = integer;
   ItemPtr = ^ItemType;
var
   p : ItemPtr;
```

here are three different methods of allocating and deallocating dynamic memory:

```
/* New and Dispose */
New(p);                    { Automatically allocates required amount of storage }
...
Dispose(p);              { Automatically deallocates amount of storage allocated }

/* New, Mark, and Release */
New(p);                    { Automatically allocates required amount of storage }
...
Mark(p);
Release(p);              { Deallocates all dynamic memory from p^ to end of heap }

/* Freemem and Getmem */
GetMem(p, SizeOf(ItemType));       { Must specify amount of storage to allocate }
...
FreeMem(p, SizeOf(ItemType));    { Must specify amount of storage to deallocate }
```

In Turbo C, allocating and deallocating dynamic memory is done using routines that are quite similar to Turbo Pascal's *GetMem* and *Dispose*:

```
<type> *<ptr>;

<ptr> = (<type>*) calloc(<num>,<size>);
/* or <ptr> = (<type>*) malloc(<total size>); */
/* or <ptr> = (<type>*) realloc(<op>,<nusz>); */
free(<ptr>);

typedef int ItemType;
ItemType *p;

p = (ItemType*) malloc(sizeof(ItemType));
...
free(p);
```

All three of the C routines (**calloc**, **malloc**, and **realloc**) return a generic pointer, which can be cast to the appropriate type. All three also return NULL if there is not enough memory available on the heap.

- The function **calloc** expects you to pass it the number of items to create and the size (in bytes) of one item; it creates the items, sets them all to 0, and returns a pointer to the entire block. This is very handy for dynamic creation of arrays.
- **malloc** is told how many bytes to allocate.

■ **free** just frees up the memory pointed to by *<ptr>*.

Command-Line Arguments

When you create a .COM file using Turbo Pascal, your program can read in any arguments that you might type on the line, using the *ParamCount* and *ParamStr* functions. For example, if you were to create a program called DUMPIT.COM and execute it as follows:

```
A>dumpit myfile.txt other.txt 72
```

ParamCount would return a value of 3, and *ParamStr* would return the following values:

```
ParamStr(1)   myfile.txt
ParamStr(2)   other.txt
ParamStr(3)   72
```

Likewise, Turbo C (following standard C conventions) allows you to declare the identifiers *argc*, *argv*, and *env* as parameters to **main** as follows:

```
main(int argc, char *argv[], char *env[]);
{
   ...body of main...
}
```

where *argc* is the number of arguments, and *argv[]* is an array of strings holding the parameters. With the same example, *argc* would yield 4, and *argv[]* would point to the following:

```
argv[0]   A:\DUMPIT.EXE
argv[1]   myfile.txt
argv[2]   other.txt
argv[3]   72
argv[4]   (null)
```

In C, under 3.x versions of MS-DOS, *argv*[0] is defined (whereas *ParamStr(0)* is not) and contains the name of the program being executed. For MS-DOS 2.x, *argv*[0] points to the null string (""). Also note that *argv*[4] actually contains NULL.

The third argument, *env[]*, is an array of strings, each holding a string of the form

```
envvar = value
```

where *envvar* is the name of an environment variable, and *value* is the string value to which *envvar* is set.

File I/O

In standard Pascal, you have two types of files: *text* (declared as text) and *data* (declared as file of <*type*>). The sequence for opening, modifying, and closing the file is almost identical for both types. Turbo Pascal also provides a third file type (untyped files) that is quite similar to the binary file operations used in Turbo C.

C files are usually treated as streams of bytes; text versus data distinctions are largely up to you, though the *t* (*text*) and *b* (*binary*) modifiers on **fopen** can be significant.

Here are some rough equivalencies between the two languages:

Table 9.2: File I/O Similarities

Turbo Pascal	Turbo C
var	
I : integer;	int i;
X : real;	float x;
Ch : char;	char ch;
Line : **string**[80];	char line[80];
myRec : RecType;	struct rectype myrec;
buffer : **array**[1..1024] **of** char	char buffer[1024]
F1 : text;	FILE *f1;
F2 : **file of** RecType;	FILE *f2;
F3 : **file**;	FILE *f3;
Assign(<fvar>,<fname>); Reset(<fvar>); Reset(<untyped fvar>, <blocksize>);	<fvar> = fopen(<fname>,"r"); /* or <fvar> = fopen(<fname>,"r+"); */ /* or f1 = fopen(<fname>,"r+t"); */ /* or f2 = fopen(<fname>,"r+b"); */
Assign(<fvar>,<fname>); Rewrite(<fvar>); Rewrite(<untyped fvar>, <blocksize>);	<fvar> = fopen(<fname>,"w"); /* or <fvar> = fopen(<fname>,"w+"); */ /* or f1 = fopen(<fname>,"w+t"); */ /* or f2 = fopen(<fname>,"w+b"); */
Assign(<fvar>,<fname>); Append(<text fvar>);	<fvar> = fopen(<fname>,"a+"); /* or <fvar> = fopen(<fname>,"a+t"); */ /* or <fvar> = fopen(<fname>,"a+b"); */
Read(F1,Ch);	ch = getc(f1);
Readln(F1,Line);	fgets(line, 80, f1);
Readln(F1,I,X);	fgets(line, 80, f1);sscanf(line,"%d %f",&i,&x);
Read(F2,MyRec);	fread(&myrec,sizeof(myrec),1,f2);
BlockRead(F3,buffer,SizeOf(buffer));	fread(buffer,1,sizeof(buffer),f3);
Write(F1,Ch);	fputc(ch,f1); /* or fprintf(f1,"%c",ch); */
Write(F1,Line);	fputs(line,f1); /* or fprintf(f1,"%s",line); */

Table 9.2: File I/O Similarities (continued)

Turbo Pascal	Turbo C
`Write(F1,I,X);`	`fprintf(f1,"%d %f",i,x);`
`Writeln(F1,I,X);`	`fprintf(f1,"%d %f\n",i,x);`
`Write(F2,MyRec);`	`fwrite(&myrec,sizeof(myrec),1,f2);`
`Seek(F2,<rec#>);`	`fseek(f2,<rec#>*sizeof(rectype),0);`
`Flush(<fvar>);`	`fflush(<fvar>);`
`Close(<fvar>);`	`fclose(<fvar>);`
`BlockWrite(F3,buffer,SizeOf(buffer));`	`fwrite(buffer,1,sizeof(buffer),f3);`

You should refer to the *Turbo C Reference Guide* for more details on how each of these Turbo C I/O routines work.

Here's a short example of a program that dumps a text file (whose name is given on the command line) to the screen:

Turbo Pascal

```
program DumpIt;
var
  F  : Text;
  Ch : char;
begin
  Assign(F,ParamStr(1));
  {$I-} Reset(F); {$I+}
  if IOResult <> 0 then begin
    Writeln('Cannot open ',ParamStr(1));
    Halt(1);
  end;
  while not EOF(F) do begin
    Read(F,Ch);
    Write(Ch)
  end;
  Close(F)
end.
```

Turbo C

```
#include <stdio.h>
main(int argc, char *argv[])
{
   FILE   *f;
   int    ch;

   f = fopen(argv[1],"r");
   if (f == NULL) {
      printf("Cannot open
      %s\n",argv[1]);
      return(1);
   }

   while ((ch = getc(f)) != EOF)
      putchar(ch);

   fclose(f);}
}
```

Common Pitfalls for Pascal Programmers Using C

There are enough similarities between Pascal and C that Pascal programmers working in C fall prey to certain predictable mistakes. Here are some of the pitfalls to avoid. There is a rough order to this list, based on a combination of how likely they are to occur, how difficult they would be for a Pascal programmer to see, and whether or not the compiler might catch them. (Chapter 3 also discusses some common pitfalls.)

PITFALL #1: Assignment vs. Comparison

In Pascal, `A = B` is the Boolean expression *A equals B* and returns *true* or *false*. In C, `A = B` is the assignment *A gets the value of B*; however (and this is critical to understand), this expression also returns a value, namely the value of *B* (which has just been assigned to *A*). The single most pernicious bug for Pascal programmers is the statement:

```
if (A = B) <statement>;
```

This is perfectly legal in C, and is evaluated as follows:

- the value of *B* is assigned to *A*
- the expression `A = B` takes on the value of *B*
- if its value is nonzero (which, in C, is *true*), *<statement>* is executed.

What you really want to write is

```
if (A == B) <statement>;
```

which does what you think it should: If *A* and *B* are equal, then *<statement>* is executed.

Remember: In C, the *is equal to* comparator is double equal signs (==), *not* a single equal sign (=). The single equal sign in C is the *assignment* operator.

PITFALL #2: Forgetting to Pass Addresses (Especially When Using scanf)

As we've explained, C only lets you pass parameters to a function by value; if you want to pass a parameter by address, you need to explicitly pass the

address yourself. Suppose you've written the function **swap** as shown earlier in the chapter. You might make the mistake of calling it like this: `swap(q,r);`, when *q* and *r* are of type **float**. In that case, **swap** will take the values of *q* and *r*, interpret them as addresses, then cheerfully swap the values at those addresses.

How do you avoid this pitfall? The best way is to use function prototypes; that way, Turbo C can do the appropriate error checking when you compile. For **swap**, you would put the following prototype somewhere near the start of your source file:

```
void swap(float *x, float *y);
```

Now, if you compile your program with the statement `swap(q,r);` you'll get an error telling you that you have a type mismatch in parameter *x* in a call to **swap**.

PITFALL #3: Omitting Parentheses on Function Calls

In Pascal, a procedure that takes no parameters is called merely by using the procedure name :

```
AnyProcedure;
i := AnyFunction;
```

In C, a call to a function—even one that has no parameters—must always include a left (open) and right (close) parenthesis. It's easy to do this:

```
AnyFunction;                              /* Code has no effect */
i = AnyFunction;             /* Stores the address of AnyFunction in i */
```

when you really want this:

```
AnyFunction();                               /* Calls AnyFunction */
i = AnyFunction();           /* Calls AnyFunction, stores result in i */
```

PITFALL #4: Warning Messages

In addition to generating error messages, Turbo C also reports nonfatal warnings. Using the incorrect function calls from the previous example, these are the warnings that Turbo C would report:

```
Warning test.c 5: Code has no effect in function main
Warning test.c 6: Nonportable pointer assignment in function main
```

Both statements are actually legal and, since no errors occurred, an .OBJ file would be created. Beware! These types of warnings would always be fatal errors in Turbo Pascal. Don't get in the habit of taking Turbo C warning messages lightly.

PITFALL #5: Indexing Multi-dimensional Arrays

Suppose you have a two-dimensional array named *matrix*, and you want to reference location (*i,j*). As a Pascal programmer, you might be inclined to write something like this:

```
x = matrix[i,j];
```

That will compile all right in C; however, it won't do what you were expecting.

In C, it is legal to have a series of expressions separated by commas; in such a case, the entire expression takes on the value of the last expression, so the preceding statement is equivalent to

```
x = matrix[j];
```

It's definitely not what you wanted, but that is still a legal statement in C. All you'll get is a warning, since C thinks you are trying to assign the address of *matrix*[*j*]—that is, the *j*th row of *matrix*[ ]—to *x*.

In C, you must explicitly surround each array index with brackets. What you really wanted to write was

```
x = matrix[i][j];
```

Remember: For multi-dimensional arrays, you must put each index in its own set of brackets.

PITFALL #6: Forgetting the Difference Between Character Arrays and Character Pointers

Suppose that you have the following statements:

```
char   *str1,str2[30];
str1 = "This is a test";
str2 = "This is another test";
```

The first assignment is acceptable; the second isn't. Why? *str1* is a pointer to a string; when the compiler sees that assignment statement, it creates the

string `This is a test` somewhere in your object file and assigns its address to *str1*.

By contrast, *str2* is a *constant* pointer to a block of 30 bytes somewhere; you can't change the address it contains. What you ought to write instead is

```
strcpy(str2,"This is another test");
```

A byte-by-byte copy is done from the constant string `This is another test` to the address pointed to by *str2*.

PITFALL #7: Forgetting that C is Case Sensitive

In Pascal, the identifiers *indx*, *Indx*, and *INDX* are all the same; uppercase and lowercase letters are treated identically. In C, they are not.

So if you declare

```
int Indx;
```

then later try to write

```
for (indx=1; indx<10; indx++) <statement>;
```

the compiler will give you an error, saying that it doesn't recognize *indx*.

PITFALL #8: Leaving Semicolons Off the Last Statement in a Block

If you're a Pascal purist who only puts semicolons where they are required (as opposed to where they are allowed), you'll have problems with this for a while. Luckily, the compiler will catch it and flag it pretty clearly. Just remember that every C statement, with two major exceptions, must have a semicolon after it.

One major exception is the function definition

```
<type> FuncName(<parm names>)
```

which should not have a semicolon after it.

This is not to be confused with the function prototype

```
<type> FuncName(<type> <pname>,<type> <pname>,...);
```

which is used to declare the function but not to actually define it, somewhat like a **forward** declaration in Pascal.

The other major exception is the set of preprocessor commands `(#<cmd>)`, such as

```
#include <stdio.h>
#define  LMAX   100
```

If you forget and enter `#define LMAX 100;`, the preprocessor will substitute *100;* every place it finds *LMAX*, semicolon and all.

Remember in C, you simply have to be more careful; it's not the forgiving language Pascal is.

C H A P T E R

10

Interfacing Turbo C with Turbo Prolog

With the introduction of Turbo C, you can now merge two powerful languages currently available for a PC. By linking Turbo C modules with Turbo Prolog modules, you can incorporate artificial intelligence (AI) into your Turbo C applications. If you are an experienced C programmer, you are already aware of Turbo C's several advantages over other C implementations. If you are just learning C, now is a good time to see how Turbo C and Turbo Prolog enhance one another.

Turbo C is a procedural language, and Turbo Prolog is a language based upon logic programming. Linking your Turbo C application with Turbo Prolog can provide the following AI advantages:

- rule-based control structure
- easy integration of natural language

Linking with Turbo Prolog also provides added AI power to your Turbo C application, so that you can solve advanced problems by simply describing the problem and letting Turbo Prolog's inference engine do the work. In many Turbo C applications, linking in Turbo Prolog programs will significantly reduce software development time and increase code clarity and program flexibility.

In This Chapter...

In this chapter, we explain the steps to compile and link Turbo C and Turbo Prolog programs, and provide four examples that demonstrate the process. The first example is a simple program that demonstrates compilation and linking. The second goes a little further and shows how to link in added C libraries. The third demonstrates allocating memory. The last example describes a practical graphics program that shows some of the power you gain by combining the two languages.

Linking Turbo C and Turbo Prolog: An Overview

Compiling and linking your Turbo C modules with Turbo Prolog modules and programs is straightforward. You only need to keep in mind the following points:

Compiling your program modules:

- Your C functions must have the _0 suffix to be called by Turbo Prolog (see the first C example program, CSUM.C, in this chapter), unless you use the `as "<filename>"` extension in Turbo Prolog.
- Your Turbo Prolog main module (the one containing a goal) replaces your C main module.
- The Turbo Prolog main module must have your C functions declared as global predicates. (See the first Prolog example program, PROSUM.PRO, in this chapter.)
- All program modules must compile to the large memory model (which is the *only* size memory model Turbo Prolog compiles in).
- If your program calls the Turbo Prolog library for version 1.1 through 2.0, you must compile your modules with register allocation turned *off* (`-r-`).
- Generate underbars should be set to *off* (`-u-`).

Linking your program modules:

- INIT.OBJ must be the first object file linked. (This is Turbo Prolog's initialization module and is found on the Turbo Prolog library disk.)
- If you need Turbo C library routines, use CL.LIB, and if using real arithmetic, EMU.LIB and MATHL.LIB.

The Link command line must have the form

```
tlink init <T_Prolog_Main> Other_files <T_Prolog_Main.sym>,
[exename], [your_libs] prolog [emulib mathl] cl
```

(This should be on a single command line.)

In addition to the preceding points, you should keep in mind the following:

- Turbo Prolog predicates may call functions written in Turbo C that are similar to built-in Turbo Prolog *predicates*.
- All calls to Turbo C library functions must be prefixed by an underbar (_). **Note**: All Turbo C library functions are prefixed by underbars. Because underbar generation is turned off, calls to library functions must have the underbars explicitly added. User-defined functions do not need the underbars.
- **malloc**, **calloc**, **free**, and other Turbo C memory allocation functions are replaced in Turbo Prolog by **alloc_gstack**, **_malloc**, and **_free**. **alloc_gstack**, **_malloc**, and **_free** are available in Turbo Prolog for memory allocation within your Turbo C functions.

  alloc_gstack allocates memory on the global stack and is called as

  ```
  void *alloc_gstack(int size)
  ```

  _malloc allocates memory on the Prolog heap and is called as

  ```
  void *_malloc(int size)
  ```

  _free releases memory allocated on the Prolog heap and is called by

  ```
  _free (void *ptr,int size)
  ```

  When **alloc_gstack** is used, the memory will automatically be freed when a fail happens, causing Turbo Prolog to backtrack across the memory allocation.
- **printf**, **putc**, and related screen output functions are not functional when you link Turbo C and Turbo Prolog. However, **wrch** can write a character to a Prolog window, and **zwf** has the same functionality as **writef** in Turbo Prolog. **zwf** is similar to a limited **printf**:

  ```
  zwf(FormatString,Arg1,Arg2,...)
  ```

  FormatString is a **printf**-type format string. Refer to the *Turbo Prolog Reference Manual* to see which conversion specifications are supported.

  zwf and **wrch** are in PROLOG.LIB.

  C functions called by Turbo Prolog should not have return values and should be defined as **void**. The flow patterns for the arguments are specified by the Turbo Prolog global predicate declaration. For example:

  ```
  factorial(integer,real) - (i,o) language c
  ```

lets Turbo Prolog know that **factorial** is a function that has two arguments—the first an integer, the second a real (floating point). The `(i,o)` means that the first argument (the integer) is passed in, and the second argument is a pointer to a floating point that will be assigned within **factorial**. The *c* lets Turbo Prolog know that the function uses C calling conventions. (See DUBLIST.C and PLIST.PRO in the third example program in this chapter (page 303).)

Notice that values are returned by reference. For more information on flow patterns, see the discussion of alternate flow patterns in example 3.

Example 1: Adding Two Integers

The following example combines a Turbo C function (one that adds two integer numbers) with a Turbo Prolog module that writes the C function result in the current window.

Turbo C Source File: CSUM.C

```
/*
The output routine zwf works nearly like the C output
routine printf. It prints the output in the current window.
*/

extern void zwf(char *format, ...);

void sum_0(int parm1,int parm2, int *res_p)
{
   zwf("This is the sum function: parm1=%d, parm2=%d" ,parm1,parm2);
   *res_p = parm1 + parm2;
} /* End of sum_0 */
```

Compiling CSUM.C to CSUM.OBJ

After you have edited and saved CSUM.C, you need to choose the compile-time options. Turbo C provides you with two methods for doing this:

1. Choose the following compile-time options from the TC (Turbo C integrated development environment) menus:

O/C/Model/Large (-ml)
O/C/Optimization/Jump Optimization ... On (-O)
O/C/Code Generation/Generate Underbars ...Off (-u-)
O/C/Optimization/Use Register Variables ... Off (-r-)

Once you have selected these options, choose **Options/Store Options** from the TC main menu; when the setup parameters are saved, choose **Compile/Compile to OBJ**. Turbo C will compile CSUM.C with the selected options, producing the object module CSUM.OBJ.

2. If you prefer to compile CSUM.C with a standard DOS command line instead of using TC's menus, enter the following at the DOS prompt:

```
tcc -ml -O -c -u- -r- csum
```

Note: Turbo Prolog only compiles to the large memory model; so if you are going to link Turbo C with Turbo Prolog, you must use the `-ml` (large memory model) compile option.

Turbo Prolog Source File: PROSUM.PRO

```
global predicates
   sum(integer,integer,integer) - (i,i,o) language c
                               /* The flow pattern of sum is defined as (i,i,o)
                                     specifying that the third argument is the
                                returned value and the first two are inputs. */

goal sum(7,6,X),write("Sum=",X).
```

Compiling PROSUM.PRO to PROSUM.OBJ

After you have edited and saved PROSUM.PRO, you need to compile it to an object (.OBJ) file, so it will link with the Turbo C object module. To do this, choose **Compile** from the Turbo Prolog main menu, then choose **OBJ File**. When Turbo Prolog finishes compiling the source file to an object file, you can link and run this example.

Linking CSUM.OBJ and PROSUM.OBJ

To link Turbo Prolog modules with Turbo C modules, you can use either TC (Turbo C's integrated development environment) or TLINK (the stand-alone linker included with your Turbo C package). Beyond the `tlink`

command, the linker command-line arguments consist of Turbo Prolog main modules, assorted other modules, output files, and libraries; except where noted, these must appear in the following order:

Turbo Prolog Initialization:

- INIT.OBJ (Turbo Prolog initialization module)

Turbo Prolog Main Module:

- a main Turbo Prolog module that contains a goal

Assorted Modules:
(These modules do not need to appear in any particular order.)

- assembler .OBJ modules
- Turbo C .OBJ modules
- Turbo Prolog .OBJ modules

Symbol Table Module:

- Turbo Prolog main symbol table name (this is required and must appear last in the list of modules).

Output File Names:

- the name of the executable file to be generated

Libraries:

- List all libraries containing routines needed by the assorted modules. Order is important: first, user-defined libraries; next, PROLOG.LIB; then if needed, EMU.LIB and MATHL.LIB; and last, CL.LIB.

In this example, we use Turbo Link (note the `tlink` command) and give it the following arguments:

- the Turbo Prolog programs INIT.OBJ and PROSUM.OBJ
- the Turbo C object module CSUM.OBJ
- the symbol table PROSUM.SYM and the executable file TEST.EXE
- the libraries PROLOG.LIB and CL.LIB (use EMU.LIB and MATHL.LIB to do floating point)

Note: PROSUM.SYM is a file that contains the symbol table of the name and type of variables in the program PROSUM.OBJ.

This is the link command line for our first example:

```
tlink init prosum csum prosum.sym,test.exe,,prolog+cl
```

Example 2: Using the Math Library

The second example is similar to the first; it shows how to write two Turbo C functions and how to combine these functions with a Turbo Prolog program. We present each of the Turbo C functions in its own separate source file; CSUM1.C adds two real numbers together and returns the sum, and FACTRL.C calculates the factorial of an integer. The Turbo Prolog program, FACTSUM.PRO, writes the program results in two Prolog windows. This example uses the Turbo C large memory-model math library, MATHL.LIB.

Turbo C Source File: CSUM1.C

```
extern void zwf(char *format, ...);
void sum_0(double parm1, double parm2, double *res_p)
{
   *res_p=parm1+parm2;
   zwf("This is the sum function: parm1=%f, parm2=%f, result=%f",
        parm1,parm2,*res_p);
}
```

Turbo C Source File: FACTRL.C

```
void factorial_0(int top, double *result)      /* Product of factorial series */
{
   double x;
   int i;
   if(top<1)
   {
     *result = 0.0;
     return;
   }
   *result = 1.0;
   x = 2.0;
   while (top-- > 1)
   {
      *result = *result * x;
      x++;
   }
} /* End of factorial_0 */
```

Compiling CSUM1.C and FACTRL.C to .OBJ

As in the first example, you must compile the two Turbo C modules to object (.OBJ) files before linking them with the other modules and with the Turbo Prolog main program. You can choose and save compile-time options with the TC **O**ptions/**S**tore options command, then choose the **C**ompile/**C**ompile to OBJ command for each of the .C source files. Or you can opt to compile both .C source files from a standard C command line, using the `tcc` command. In either case, you must choose at least the following compile-time options:

O/**C**/**M**odel/**L**arge (`-ml`)
O/**C**/**O**ptimization/**J**ump Optimization ... On (`-O`)
O/**C**/**C**ode Generation/**G**enerate Underbars ... Off (`-u-`)
O/**C**/**O**ptimization/**U**se Register Variables ... Off (`-r-`)

Turbo Prolog Source File: FACTSUM.PRO

FACTSUM.PRO is the main Turbo Prolog program, which makes two windows: One displays the output from your Turbo C modules, and the other displays the Turbo Prolog program output. This is the order in which the modules and program interact:

1. FACTSUM.PRO prompts the user to input an integer *Int*, which the Turbo Prolog program then passes to FACTRL.C.
2. The Turbo C function **factorial** in FACTRL.C returns *Result*, the factorial of *Int*, to FACTSUM.PRO.
3. FACTSUM.PRO writes *Result* in a window and again prompts the user for a number (this time, a real).
4. FACTSUM.PRO passes this second input number, *Real*, and the previously calculated factorial, *Result*, to the module CSUM1.C.
5. The Turbo C function **sum** in CSUM1.C adds *Real* and *Result*, then returns the answer, *Sum*, to FACTSUM.PRO.
6. FACTSUM.PRO writes *Sum* in a window, and the program is finished.

Here is the Turbo Prolog program FACTSUM.PRO:

```
/*
   Declaration of the Turbo C module must be located after the Turbo Prolog
   domains and database declarations (if any are present). All global modules
   are called from Turbo Prolog as global predicates, and must be followed by
   the flow pattern and language specification.
```

```
*/
global predicates
   sum(real,real,real) - (i,i,o) language c
   factorial(integer,real) - (i,o) language c
/*
   This is a very simple example that has only external clauses
   (Turbo C modules), so only a goal section is needed. However,
   in any real application, a clauses section would also be needed.
*/
goal
   makewindow(1,49,31,
            " A Turbo Prolog window to the Turbo C program ",0,0,15,80),
   makewindow(2,47,3,
            " A Turbo Prolog window to the Turbo Prolog program ",15,0,10,80),

   /* Prompt user for first input */
   write("Enter an integer; Turbo C will calculate the factorial: "),
   readint(Int),nl,
   shiftwindow(1),                 /* Change output window to Turbo C window */

   /* Call Turbo C factrl module and calculate the factorial */
   factorial(Int,Result),
   shiftwindow(2),            /* Change output window to Turbo Prolog window */

   /* Prompt user for second input */
   write("Enter a real number to add to the factorial "),
   readreal(Real),nl,
   shiftwindow(1),                 /* Change output window to Turbo C window */

   /* Call Turbo C csum1 module and calculate the sum */
   sum(Result,Real,Sum),
   shiftwindow(2),            /* Change output window to Turbo Prolog window */

   /* Write result of first calculation in window */
   write("The factorial of ",Int," is ",Result),nl,

   /* Write result of second calculation in window */
   write("The result: ",Result," + ",Real," = ",Sum),nl.
```

Compiling FACTSUM.PRO to FACTSUM.OBJ

As in the first example, you must compile the Turbo Prolog program source file to an object (.OBJ) file before linking it with the modules. Choose Compile from the Turbo Prolog main menu, as before, then choose OBJ File.

Linking CSUM1.OBJ, FACTRL.OBJ, and FACTSUM.OBJ

In the link command used in this example,

- The Turbo Prolog object modules are INIT.OBJ and FACTSUM.OBJ.
- The Turbo C object modules are CSUM1.OBJ and FACTRL.OBJ.
- The output file names are FACTSUM.SYM (symbol table) and SUM.EXE (executable file).
- The libraries needed are PROLOG.LIB, EMU.LIB, MATHL.LIB, and CL.LIB.

This is the command linking the modules:

```
tlink init factsum factrl csum1 factsum.sym,sum,,prolog+emu+mathl+cl
```

Example 3: Flow Patterns and Memory Allocation

The following program presents the code for creating a Turbo Prolog *functor* and *list* in Turbo C and returning these new structures to Turbo Prolog. This example also demonstrates how memory can be allocated in Turbo Prolog's global stack. Lists are recursive structures of three elements, and functors are C structures with two members (these are described more fully after this example).

- A Turbo C module DUBLIST.C contains three functions. The first two can take an integer list and return a structure with the first integer in it, or can take a structure with an integer and return a list with that integer. The third function takes an integer *n* and generates a list of two integers; the first being *n* and the second 2*n*.
- It is important to notice that there can be alternate flow patterns for each Turbo Prolog global predicate, and that each flow pattern requires an alternate Turbo C function. For the following example, **clist_0** must correspond to the first flow pattern (i,o), and **clist_1** to the second flow pattern (o,i).

```
global predicates
   clist(ilist,ifunc) - (i,o) (o,i) language c
```

- The `(i,o)` specifies that *ilist* is to be passed into your Turbo C function *clist_0*, and *ifunc* is a pointer to a structure that will be defined within the

Turbo C function *clist_0*. The (o,i) specifies that *ifunc* is passed into *clist_1*, and *ilist* is a pointer to a list structure that will be defined within *clist_1*.

- If an additional flow pattern was specified in your Turbo Prolog global **domains**, a *clist_2* would be needed to handle the additional flow pattern.

Turbo C Source File: DUBLIST.C

```
void fail_cc(void);
void *alloc_gstack(int size);
struct ilist {
   char functor;                              /* Type of the list element */
                                                    /* 1 = a list element */
                                                      /* 2 = end of list */
   int val;                                         /* The actual element */
   struct ilist *next;                        /* Pointer to the next node */
};

struct ifunc {
   char type;                                         /* Type of functor */
   int value;                                    /* Value of the functor */
};

void clist_0(struct ilist *in, struct ifunc **out)
{
   if (in->functor != 1)
      fail_cc();                                    /* Fail if empty list */
   *out = alloc_gstack(sizeof(struct ifunc));

   (*out)->value = in->val;                    /* This sets out to f(X) */
   (*out)->type = 1;                            /* Set the functor type */
}

void clist_1(struct ilist **out, struct ifunc *in)
{
   int temp = 0;
   struct ilist *endlist = alloc_gstack(sizeof(struct ilist));
   endlist->functor = 2;
   temp = in->value;
   temp += temp;
   *out = alloc_gstack(sizeof(struct ilist));
   (*out)->val = temp;                  /* This returns [2*X] as a list */
   (*out)->functor = 1;            /* Set the list type. If this is not */
                                   /* done, no meaningful value will be */
                                                            /* returned */
   (*out)->next = endlist;
```

```
}

void dublist_0(int n, struct ilist **out)
{
/*
   This function creates the list [n,n+n]
*/
   struct ilist *temp;
   struct ilist *endlist = alloc_gstack(sizeof(struct ilist));

   endlist->functor = 2;
   temp = alloc_gstack(sizeof(struct ilist));
   temp->val = n;                                   /* This sets the first
   temp->functor = 1;                          /* element of the list to n */
   *out = temp;

   /* Now we have to allocate a second list element */
   temp = alloc_gstack(sizeof(struct ilist));

   temp->val = n + n;                     /* This assigns the value n + n to */
   temp->functor = 1;                                   /* The second element */
   temp->next = endlist;                /* Set the node after the second to an */
                                                          /* end of list node */
   (*out)->next = temp;                              /* after the first element */
}
```

Calling Turbo Prolog from Turbo C

Not only can Turbo Prolog call predicates written in Turbo C, Turbo C can also call Turbo Prolog predicates. If a global predicate is declared to exist in Turbo Prolog as `language c`, and there are Turbo Prolog clauses for that predicate, Turbo Prolog will generate a routine that can be called by Turbo C.

The following Turbo Prolog program declares two C language global predicates; ***message*** and ***hello_c***. The ***message*** predicate can be called from a C module by using the function name **message_0** in the C source code.

```
global predicates
   message(string) - (i) language c
   hello_c - language c

clauses
   message(S) :-
      makewindow(13,7,7,"",10,10,3,50),
      write(S), readchar(_),
      removewindow.

goal
```

```
   message("Hello from Turbo Prolog"),
   hello_c.
```

The **goal** section of this example calls the Turbo C function **hello_c**, which, in turn, calls the Turbo Prolog predicate ***message_0*** to display a message.

```
void message_0(char * str);

void hello_c_0(void)
{
   message_0("Hello from Turbo C");
}
```

You can use this feature to give easy access to Turbo Prolog's powerful library from other languages.

You can easily define your own library routines in a Turbo Prolog module like this:

```
project "dummy"                          /* Use your own project name here */

global predicates
   myfail language c as "fail"
   mymakewindow(integer,integer,integer,string,integer,integer,integer,integer)
      - (i,i,i,i,i,i,i,i) language c as "makewindow"
   myshiftwindow(integer) - (i) language c as "shiftwindow"
   myremovewindow language c as "removewindow"
   write_integer(integer) - (i) language c as "write_integer"
   write_real(real) - (i) language c as "write_real"
   write_string(string) - (i) language c as "write_string"
   myreadchar(char) - (o) language c as "readchar"
   myreadline(string) - (o) language c as "readline"

   extprog language c as "extprog"

clauses
   myfail :- fail.

   mymakewindow(Wno, Wattr, Fattr, Text, Srow, Scol, Rows, Cols) :-
      makewindow(Wno, Wattr, Fattr, Text, Srow, Scol, Rows, Cols).

   myshiftwindow(WNO) :- shiftwindow(WNO).

   myremovewindow :- removewindow.

   write_integer(I) :- write(I).

   write_real(R) :- write(R).

   write_string(S) :- write(S).

   myreadchar(CH) :- readchar(CH).

   myreadline(S) :- readln(S).
```

The following C procedure, **extprog**, demonstrates using these new library routines. **extprog** creates a Turbo Prolog window, then does some reading and writing in it.

```
void makewindow(int who, int wattr, int fattr, char *title, int
     row, int srow, int scol);
void write_string(char *text);
void readchar(char *ch);
void readline(char *in_str[]);
void removewindow(void)

void extprog(void)
{
   char dummychar;
   char *Name;

   makewindow(1,7,7,"Hello there",5,5,15,60);
   write_string("\n\nIsn't it easy");
   readchar(&dummychar);
   write_string("\nEnter your name: ");
   readline(&Name);
   write_string("\nYour name is: ");
   write_string(Name);
   readchar(&dummychar);
   removewindow();
}
```

The only restriction in calling Turbo Prolog from Turbo C is that the Turbo Prolog program must be the main program because Turbo Prolog needs to set up the heap and stacks.

Lists and Functors

Turbo Prolog lists and functors are structures in Turbo C (see DUBLIST.C).

Lists are recursive structures that have three elements. The first element is the type; the value may be 1 if it is a list element, and 2 if it is an end of the list. The second element is the actual value; it must be the same type as the element in Turbo Prolog.

For example, a list of reals would be

```
struct alist {
   char funct;
   double elem;                        /* The list elements are real */
   struct alist *next;
}
```

The third element is a pointer to the next node.

Turbo Prolog functors are C structures that have two elements. The first element corresponds to the Turbo Prolog domain declaration. (Refer to your *Turbo Prolog Reference Guide* for more information on domain declarations in Turbo Prolog.) For example,

```
domains
   func = i(integer); s(string)
predicates
   call(func)
goal
   call(X)
```

The Turbo Prolog functor *func* has two types: The first is an integer, and the second is a string. So in this example the value of the type member of the Turbo C structure may be 1 or 2; 1 corresponding to the first type, and 2 to the second type.

The second element of the Turbo C structure is the actual value of this element of the functor and is defined as the *union* of the possible types of the argument.

```
union val {
   int    ival;
   char  *svar;
};
struct func {
   char type;                          /* Type may be 1 or 2 corresponding to
                                         the Turbo Prolog domain declarations */
   union val value;                   /* The value of the functor element     */
}
```

Note: The functions **alloc_gstack, _malloc,** and **_free** must be used for memory management (these are found in CPINIT.OBJ). These functions are needed to

1. allocate memory for Turbo C structures stored in the Turbo Prolog heap or stack
2. release memory in Turbo Prolog's heap

When **alloc_gstack** is used, the memory will automatically be released when a fail occurs, causing Turbo Prolog to backtrack across the memory allocation.

Here is the Turbo C syntax for each:

```
void *alloc_gstack(size)                  /* Allocates storage in stack */

void *_malloc(size)                        /* Allocates storage in heap */

_free(void *ptr,size)                            /* Releases heap space */
```

Here is the Turbo Prolog main module, PLIST.PRO, which calls the functions in DUBLIST.C and prints the results.

```
domains
   ilist = integer*
   ifunc = f(integer)

global predicates
   clist(ilist,ifunc) - (i,o) (o,i) language c
   dublist(integer,ilist) - (i,o) language c
goal
   clearwindow,
   clist([3],X),                                      /* Binds X to f(3) */
   write("X = ",X),nl,
   clist(Y,X),                                         /* Binds Y to [6] */
   write("Y = ",Y),nl,
   dublist(6,Z),                                    /* Binds Z to [6,12] */
   write(Z),nl.
```

Compiling DUBLIST.C

As in the first two examples, you must compile the Turbo C module DUBLIST.C to an object (.OBJ) file before linking it with the Turbo Prolog main module PLIST.PRO.

This is the link command:

```
tlink init plist dublist plist.sym, dublist, ,prolog+emu+mathl+cl
```

Example 4: Drawing a 3-D Bar Chart

In this example, we show you how to compile and link the C and Prolog modules to create a unified, mixed-language program that combines AI flexibility with C graphics-handling capability. Specifically, the code provided includes the following:

- a Turbo C module CBAR.C that draws bar charts using input from another file

- a Turbo Prolog main module PBAR.PRO that requests input from the user

Turbo C Source File: CBAR.C

The source code for this program is the file CBAR.C on your distribution disks.

Compiling CBAR.C

As in the first three examples, you must compile the Turbo C module CBAR.C to an object (.OBJ) file before linking it with the Turbo Prolog main module PBAR.PRO.

Turbo Prolog Program: PBAR.PRO

The source code for this program is the file PBAR.PRO on your disk. PBAR is a Turbo Prolog program that prompts the user to make, save, load, or draw a bar chart.

If the user wishes to make a bar chart, this program will accept input specifications for the chart, position each bar in a window and call your C module to draw the bar. After each bar is drawn, it is *asserted* (a Prolog term for inserted) into the database.

The user may opt to save the bar chart; PBAR will save a description of the current bar chart into a file for later use.

If the user selects the load option, PBAR will delete the current bar chart description and load a user-specified bar chart description from a file.

Given the final option, *draw*, PBAR will use the description in the database in a recursive call to the Turbo C BAR module, which will then draw a bar chart to the specifications currently in the database.

Compiling PBAR.PRO to PBAR.OBJ

As in example 1, you must compile the Turbo Prolog main module source file to an object (.OBJ) file before linking it with the Turbo C modules.

Linking PBAR.OBJ with the Module CBAR.OBJ

In the following link command, PBAR.OBJ is linked with the previously compiled Turbo C module, CBAR.OBJ. The components of this link command are

- the Turbo Prolog object modules INIT.OBJ and PBAR.OBJ
- the Turbo C object module CBAR.OBJ
- the symbol table PBAR.SYM and the output file BARCHART.EXE (executable file)
- the libraries PROLOG.LIB and CL.LIB

This is the link command:

```
tlink init pbar cbar pbar.sym,barchart,,prolog+cl
```

That's All There Is to It

With these four examples, we have shown you how to link Turbo Prolog modules with your Turbo C programs. If you are an experienced Turbo Prolog programmer but would like to know more about programming in C, we recommend reading Chapters 7 and 3 in this manual. If you are an experienced C programmer and would like to find out more about Turbo Prolog, we recommend consulting a Turbo Prolog tutorial, such as *Using Turbo Prolog* by P. Robinson (McGraw-Hill).

C H A P T E R

11

Turbo C Language Reference

The traditional reference for C is *The C Programming Language* (First Edition), by Brian W. Kernighan and Dennis M. Ritchie (which we will refer to as "K&R" from now on). Their book doesn't define a complete standard for C; that task has been left to the American National Standards Institute (ANSI). Instead, K&R presents a minimum standard so that a program using only those aspects of C found in K&R can be compiled by any C implementation that supports the K&R definition.

Turbo C not only supports the K&R definition, it also implements most of the ANSI extensions. In doing so, Turbo C seeks to improve and extend the C language by adding new features and increasing the power and flexibility of old ones. We don't have space to reprint K&R or the ANSI standard here; instead, we'll tell you about the additions to the K&R definition that Turbo C provides, noting which come from the ANSI standard and which are our own improvements.

In This Chapter...

To make cross-referencing easier for you, this chapter follows (more or less) the outline of Appendix A in K&R, which is titled *C Reference Manual.* Not all sections of that appendix are referenced here; for any section we passed over, you may assume that there are no significant differences between Turbo C and the K&R definition. Also, to more easily accommodate some of the ANSI and Turbo C extensions, we have presented some information

in the same order as given in the ANSI C standard rather than adhere to the K&R organization.

Comments (K&R 2.1)

The K&R definition of C does not allow comments to be nested. For example, the construct

```
/* Attempt to comment out myfunc() */

/*
   myfunc()
   {
      printf("This is my function\n");                    /* The only line */
   }
*/
```

would be interpreted as a single comment ending right after the phrase `The only line`; the dangling brace and end-of-comment would then trigger a syntax error. By default, Turbo C does not allow comment nesting; however, you can correctly compile a program (such as that shown) with nested comments by using the `-C` compiler option (Nested comments...ON in the **O/C/**Source menu). A more portable approach, though, is to bracket the code to be commented out with `#if 0` and `#endif`.

Comments are replaced with a single-space character after macro expansion. In other implementations, comments are removed completely and are sometimes used for token pasting. See "Token Replacement" in this chapter.

Identifier (K&R 2.2)

An identifier is just the name you give to a variable, function, data type, or other user-defined object. In C, an identifier can contain letters (*A*...*Z*, *a*...*z*) and digits (0...9) as well as the underscore character (_). However, an identifier can only start with a letter or an underscore.

Case is significant; in other words, the identifiers *indx* and *Indx* are different. In Turbo C, the first 32 characters of an identifier are significant within a program; however, you can modify this with the `-i#` compiler option, where # is the number of significant characters. (This is the menu option **O/C/S/**Identifier Length.)

Likewise, the first 32 characters are significant for global identifiers imported from other modules. However, you can decide whether case is significant for those identifiers by using the **Case-Sensitive Link...On** option from the **Options/Linker** menu or the /c option on a TLINK command line. Note, however, that identifiers of type **pascal** are never case sensitive at link time.

Keywords (K&R 2.3)

Table 11.1 shows the keywords reserved by Turbo C; these cannot be used as identifier names. Those preceded by "AN" are ANSI extensions to K&R; those preceded by "TC" are Turbo C extensions. The keywords **entry** and **fortran**, mentioned in K&R, are neither used nor reserved by Turbo C.

Table 11.1: Keywords Reserved by Turbo C

TC asm	AN enum	TC pascal	AN volatile	
auto	extern	register	while	
break	TC far	return		
case	float	short	TC _cs	TC _CH
TC cdecl	for	AN signed	TC _ds	TC _CL
char	goto	sizeof	TC _es	TC _CX
AN const	TC huge	static	TC _ss	TC _DH
continue	if	struct	TC _AH	TC _DL
default	int	switch	TC _AL	TC _DX
do	TC interrupt	typedef	TC _AX	TC _BP
double	long	union	TC _BH	TC _DI
else	TC near	unsigned	TC _BL	TC _SI
AN void	TC _BX	TC _SP		

Constants (K&R 2.4)

Turbo C supports all the constant types defined by K&R, with a few enhancements.

Integer Constants (K&R 2.4.1)

Constants from 0 to 4294967295 (base 10) are allowed. (Negative constants are simply unsigned constants with the unary minus operator.) Both octal (base 8) and hexadecimal (base 16) representations are accepted.

The suffix *L* (or *l*) attached to any constant forces it to be represented as a **long**. Similarly, the suffix *U* (or *u*) forces it to be **unsigned**, and it will be **unsigned long** if the value of the number itself is greater than 65535,

regardless of which base is used. **Note:** You may use both *L* and *U* suffixes on the same constant.

Table 11.2 summarizes the representations of constants in all three bases.

Table 11.2: Turbo C Integer Constants Without L or U

Range	Type
Decimal Constants	
`0 - 32767`	`int`
`32768 - 2147483647`	`long`
`2147483648 - 4294967295`	`unsigned long`
`> 4294967295`	`Will overflow without warning; the resulting constant will be the low-order bits of the actual value`
Octal Constants	
`00 - 077777`	`int`
`0100000 - 0177777`	`unsigned int`
`02000000 - 017777777777`	`long`
`020000000000 - 037777777777`	`unsigned long`
`> 037777777777`	`Will overflow (as previously described)`
Hexadecimal Constants	
`0x0000 - 0x7FFF`	`int`
`0x8000 - 0xFFFF`	`unsigned int`
`0x10000 - 0x7FFFFFFF`	`long`
`0x80000000 - 0xFFFFFFFF`	`unsigned long`
`> 0xFFFFFFFF`	`Will overflow (as previously described)`

Character Constants (K&R 2.4.3)

Turbo C supports two-character constants, for example, 'An', '\n\t', and '\007\007'. These constants are represented as 16-bit **int** values with the first character in the low-order byte and the second character in the high-order byte. Note that these constants are not portable to other C compilers.

One-character constants, such as 'A', '\t', and '\007', are also represented as 16-bit **int** values. In this case, the low-order byte is *sign extended* into the high byte; that is, if the value is greater than 127 (base 10), the upper byte is set to –1 (=0xFF). This can be disabled by declaring that the default **char** type is unsigned (use the `-K` compiler option or choose Default Char Type...Unsigned in the Options/Compiler/Source menu), which forces the high byte to be zero regardless of the value of the low byte.

Turbo C supports the ANSI extension of allowing hexadecimal representation of character codes, such as '\x1F', '\x82', and so on. Either *x* or *X* is allowed, and you may have one to three digits.

Turbo C also supports the other ANSI extensions to the list of allowed *escape sequences*. Escape sequences are values inserted into character and string constants, preceded by a backslash (\). Table 11.3 lists all allowed sequences; those marked with an asterisk (*) are extensions to K&R.

Table 11.3: Turbo C Escape Sequences

Sequence	Value	Char	What It Does
* \a	0x07	BEL	Audible bell
\b	0x08	BS	Backspace
\f	0x0C	FF	Formfeed
\n	0x0A	LF	Newline (linefeed)
\r	0x0D	CR	Carriage return
\t	0x09	HT	Tab (horizontal)
* \v	0x0B	VT	Vertical tab
\\	0x5c	\	Backslash
\'	0x27	'	Single quote (apostrophe)
* \"	0x22	"	Double quote
* \?	0x3F	?	Question mark
\DDD		any	DDD = 1 to 3 digit octal value
* \xHHH	0xHHH	any	HHH = 1 to 3 digit hex value

*ANSI extensions to K&R

Note: Since Turbo C allows two-character constants, ambiguities may arise if an octal escape sequence of less than three digits is followed by a digit. In such cases, Turbo C will presume that the following character is part of the escape sequence, unless the character is not allowed for that type of number. For example, because the digits 8 and 9 in an octal value are not allowed, the constant \258 would be interpreted as a two-character constant made up of the characters \25 and 8.

Floating Constants (K&R 2.4.4)

All floating constants are by definition of type **double** as specified in K&R. However, you can coerce a floating constant to be of type **float** by adding an *F* suffix to the constant.

Strings (K&R 2.5)

According to K&R, a string constant consists of exactly one string unit, containing double quotes, text, double quotes ("like this"). You must use the backslash (\ ) as a continuation character in order to extend a string constant across line boundaries.

Turbo C allows you to use multiple string units in a string constant; it will then do the concatenation for you. For example, you could do the following:

```
main()
{
   char    *p;

   p = "This is an example of how Turbo C"
       " will automatically\ndo the concatenation for"
       " you on very long strings,\nresulting in nicer"
       " looking programs.";
   printf(p);
}
```

The output of the program is:

```
This is an example of how Turbo C will automatically
do the concatenation for you on very long strings,
resulting in nicer looking programs.
```

Hardware Specifics (K&R 2.6)

K&R recognizes that the size and numeric range of the basic data types (and their various permutations) are very implementation specific and usually derive from the architecture of the host computer. This is true for Turbo C, just as it is for all other C compilers. Table 11.4 lists the sizes and resulting ranges of the different data types for Turbo C.

Table 11.4: Turbo C Data Types, Sizes, and Ranges

Type	Size (bits)	Range
unsigned char	8	0 – 255
char	8	–128 – 127
enum	16	–32768 – 32767
unsigned short	16	0 – 65535
short	16	–32768 – 32767
unsigned int	16	0 – 65535
int	16	–32768 – 32767
unsigned long	32	0 – 4294967295
long	32	–2147483648 – 2147483647
float	32	3.4E-38 – 3.4E+38
double	64	1.7E-308 – 1.7E+308
long double	80	3.4E-4932 – 1.1E+4932
pointer	16	(near, _cs, _ds, _es, _ss pointers)
pointer	32	(far, huge pointers)

Conversions (K&R 6)

Turbo C supports the standard mechanisms for automatically converting from one data type to another. The following sections indicate additions to K&R or implementation-specific information.

char, int, and enum (K&R 6.1)

Assigning a character constant to an integer object results in a full 16-bit assignment, since both one- and two-character constants are represented as 16-bit values (see K&R 2.4.3). Assigning a character object (such as a variable) to an integral object will result in automatic sign extension, unless you've made the default **char** type unsigned (with the `-K` compiler option). Objects of type **signed char** always use sign extension; objects of type **unsigned char** always set the high byte to zero when converted to **int**.

Values of type **enum** convert straight to **int** with no modifications; similarly, **int** values can be converted straight to an enumerated type. **enum** values and characters convert exactly, as do **int** values and characters.

Pointers (K&R 6.4)

In Turbo C, different pointers in your program may be of different sizes, depending upon the memory model or pointer type modifers you use. For example, when you compile your program in a particular memory model, the addressing modifiers (**near**, **far**, **huge**, **_cs**, **_ds**, **_es**, **_ss**) in your source code can override the pointer size given by that memory model.

A pointer must be declared as pointing to some particular type, even if that type is **void** (which really means a pointer to anything). However, having been declared, that pointer can point to an object of any other type. Turbo C allows you to reassign pointers like this, but the compiler will warn you when pointer reassignment happens—unless the pointer was originally declared to be of type pointer to **void**. However, pointers to data types cannot be converted to pointers to functions, and vice versa.

Arithmetic Conversions (K&R 6.6)

K&R refers to the *usual arithmetic conversions*, which specify what happens when any values are used in an arithmetic expression (operand, operator, operand). Here are the steps used by Turbo C to convert the operands in an arithmetic expression:

1. Any noninteger or nondouble types are converted as shown in Table 11.5. After this, any two values associated with an operator are either **int** (including the **long** and **unsigned** modifiers) or **double**.
2. If either operand is of type **long double**, the other operand is converted to **long double**.
3. If either operand is of type **double**, the other operand is converted to **double**.
4. Otherwise, if either operand is of type **unsigned long**, the other operand is converted to **unsigned long**.
5. Otherwise, if either operand is of type **long**, then the other operand is converted to **long**.
6. Otherwise, if either operand is of type **unsigned**, then the other operand is converted to **unsigned**.
7. Otherwise, both operands are of type **int**.

The result of the expression is the same type as that of the two operands.

Table 11.5: Methods Used in Usual Arithmetic Conversions

Type	Converts to	Method
char	int	Sign-extended
unsigned char	int	Zero-filled high byte (always)
signed char	int	Sign-extended (always)
short	int	If unsigned, then unsigned int
enum	int	Same value
float	double	Pads mantissa with 0's

Operators (K&R Section 7.2)

Turbo C supports the unary plus operator, while K&R does not. This operator has no effect, but provides symmetry with the negation operator.

Normally, Turbo C will regroup integral expressions, rearranging commutative operators (such as * and binary +) in an effort to create an efficiently compiled expression. However, Turbo C will not reorganize expressions containing floating-point quantities. Consequently, you must use parentheses to force the order of evaluation in floating point expressions.

For example, if *a*, *b*, *c*, and *f* are all of type **float**, then the expression

```
f = a + (b + c);
```

forces the expression *(b + c)* to be evaluated before adding the result to *a*.

Type Specifiers and Modifiers (K&R 8.2)

Turbo C supports the following basic types not found in K&R:

- unsigned char
- unsigned short
- unsigned long
- long double
- enumeration
- void

The types **int** and **short** are equivalent in Turbo C, both being 16 bits. See "Hardware Specifics" for more details on how different types are implemented.

The enum Type

Turbo C implements enumerated types as found in the ANSI standard. An enumerated data type is used to describe a discrete set of integer values. For example, you could declare the following:

```
enum days { sun, mon, tues, wed, thur, fri, sat };
```

The names listed in *days* are integer constants with the first (*sun*) being automatically set to zero, and each succeeding name being one more than the preceding one (*mon* = 1, *tues* = 2, and so on). However, you can set a name to a specific value; following names without specified values will then increase by one, as before. For example,

```
enum coins { penny = 1, nickle = 5, dime = 10, quarter = 25};
```

A variable of an enumerated type can be assigned any value of type **int**—no type checking beyond that is enforced.

The void Type

In K&R, every function returns a value; if no type is declared, then the function is of type **int**. Turbo C supports the type **void** as defined in the ANSI standard. This is used to explicitly document a function that does not return a value. Likewise, an empty parameter list can be documented with the reserved word **void**. For example,

```
void putmsg(void)
{
   printf("Hello, world\n");
}

main()
{
   putmsg();
}
```

As a special construct, you may cast an expression to **void** in order to explicitly indicate that you're ignoring the value returned by a function. For example, if you want to pause until the user presses a key but ignore what is typed, you might write this:

```
(void) getch();
```

Finally, you can declare a pointer to **void**. This doesn't create a pointer to nothing; it creates a pointer to any kind of data object, the type of which is not necessarily known. You can assign any pointer to a **void** pointer (and

vice versa) without a cast. However, you cannot use the indirection operator (*) with a **void** pointer, since the underlying type is undefined.

The signed Modifier

In addition to the three types of adjectives defined by K&R—**long**, **short**, and **unsigned**—Turbo C supports three more: **signed**, **const**, and **volatile** (all of which are defined in the ANSI standard).

The **signed** modifier is the opposite of **unsigned** and explicitly says that the value is stored as a signed (two's complement) value. This is done primarily for documentation and completeness. However, if you compile with the default **char** type **unsigned** (instead of **signed**), you must use the **signed** modifier in order to define a variable or function of type **signed char**. The modifier **signed** used by itself signifies **signed int**, just as **unsigned** by itself means **unsigned int**.

The const Modifier

The **const** modifier, as defined in the ANSI standard, prevents any assignments to the object or any other side effects, such as increment or decrement. A **const** pointer cannot be modified, though the object to which it points can be. **Note:** The modifier **const** used by itself is equivalent to **const int**. Consider the following examples:

```
const float pi      = 3.1415926;
const       maxint  = 32767;
char *const str     = "Hello, world";                /* A constant pointer */
char const  *str2   = "Hello, world";     /* A pointer to a constant string */
```

Given these, the following statements are illegal:

```
pi  = 3.0;                                       /* Assigns a value to a const */
i   = maxint++;                                          /* Increments a const */
str = "Hi, there!";                             /* Points str to something else */
```

Note, however, that the function call `strcpy(str,"Hi, there!")` is legal, since it does a character-by-character copy from the string literal `"Hi, there!"` into the memory locations pointed to by *str*.

The volatile Modifier

The **volatile** modifier, also defined by the ANSI standard, is almost the opposite of **const**. It indicates that the object may be modified; not only by you, but also by something outside of your program, such as an interrupt routine or an I/O port. Declaring an object to be **volatile** warns the compiler not to make assumptions concerning the value of the object while evaluating expressions containing it, since the value could (in theory) change at any moment. It also prevents the compiler from making the variable a register variable.

```
volatile  int  ticks;
interrupt   timer()
{
   ticks++;
}

wait(int interval)
{
   ticks = 0;
   while (ticks < interval);                                    /* Do nothing */
}
```

These routines (assuming **timer** has been properly associated with a hardware clock interrupt) will implement a timed wait of ticks specified by the argument *interval*. Note that a highly optimizing compiler might not load the value of *ticks* inside the **while** loop, since the loop doesn't change the value of *ticks*.

The cdecl and pascal Modifiers

Turbo C allows your programs to easily call routines written in other languages, and vice versa. When you mix languages like this, you have to deal with two important issues: *identifiers* and *parameter passing*.

When you compile a program in Turbo C, all the global identifiers in the program—that is, the names of functions and global variables—are saved in the resulting object code file for linking purposes. By default, those identifiers are saved in their original case (lower, upper, or mixed). Also, an underscore (_) is prepended to the front of the identifier, unless you have selected the -u- (Generate Underbars...Off) option.

Likewise, any external identifiers you declare in your program are presumed to have the same format. Linking is (by default) case sensitive, so

identifiers used in different source files must match exactly in both spelling and case.

pascal

In certain situations, such as referencing code written in other languages, this default method of saving names can be a problem.

So Turbo C gives you a way around the problem. You can declare any identifier to be of type **pascal**. This means that the identifier is converted to uppercase and that no underscore is stuck on the front. (If the identifier is a function, this also affects the parameter-passing sequence used; see "Function Type Modifiers" for more details.) It no longer matters what case is used in the source code; for linking purposes, it's considered uppercase only.

cdecl

You can make all the global identifiers in a source file of type **pascal** by compiling with the -p (Calling Convention... Pascal) option. However, you may then want to ensure that certain identifiers have their case preserved and keep the underscore on the front, especially if they're C identifiers from another file.

You can do so by declaring those identifiers to be **cdecl** (which also has an effect on parameter passing for functions).

You'll notice, for example, that all the functions in the header files (STDIO.H, and so on) are of type **cdecl**. This ensures that you can link with the library routines, even if you compile using -p.

See K&R Section 10.1.1 in this chapter, as well as Chapter 12, for more details.

The near, far, and huge modifiers

Turbo C has three modifiers that affect the indirection operator (*); that is, they modify pointers to data. These are **near**, **far**, and **huge**. The meaning of these keywords is explained in greater detail in Chapter 12, but here's a brief overview.

Turbo C allows you to compile using one of several memory models. The model you use determines (among other things) the internal format of pointers to data. If you use a small data model (tiny, small, medium), all data pointers are only 16 bits long and give the offset from the Data Segment (DS) register. If you use a large data model (compact, large, huge), all pointers to data are 32 bits long and give both a segment address and an offset.

Sometimes, when using one size of data model, you want to declare a pointer to be of a different size or format than the current default. You do so using the modifiers **near**, **far**, and **huge**.

A **near** pointer is only 16 bits long; it uses the current contents of the Data Segment (DS) register for its segment address. This is the default for small data models. Using **near** pointers limits your data to the current 64K data segment.

A **far** pointer is 32 bits long, and contains both a segment address and an offset. This is the default for large data models. Using **far** pointers allows you to refer to data anywhere in the 1-Mb address space of the Intel 8088/ 8086 processors.

A **huge** is also 32 bits long, again containing both a segment address and an offset. However, unlike **far** pointers, a **huge** pointer is always kept *normalized*. The details of this are given in Chapter 12, but here are the implications:

- Relational operators (==, !=, <, >, <=, >=) all work correctly and predictably with **huge** pointers; they do not with **far** pointers.
- Any arithmetic operations on a **huge** pointer affect both the segment address and the offset because of normalization; on a **far** pointer, only the offset is affected.
- A given **huge** pointer can be incremented through the entire 1-Mb address space; a **far** pointer will eventually wrap around to the start of its 64 Kb segment.
- Using **huge** pointers requires additional time because the normalization routines have to be called after any arithmetic operations on the pointers.

Structures and Unions (K&R Section 8.5)

Turbo C follows the K&R implementation of structures and unions and provides the following additional features.

Word Alignment

If you use the `-a` compiler option (Alignment...Word), Turbo C will pad the structure (or union) with bytes as needed for word alignment. This ensures three things:

- The structure will start on a word boundary (even address).
- Any non-**char** member will have an even offset from the beginning of the structure.
- A byte will be added (if necessary) at the end to ensure that the entire structure contains an even number of bytes.

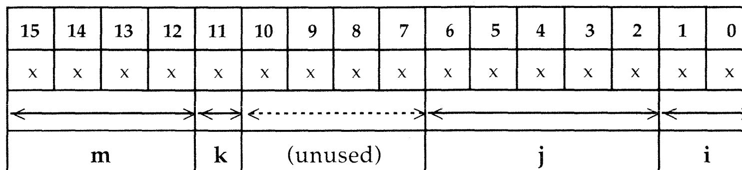
Bitfields

In Turbo C, a bitfield may be either a **signed** or **unsigned int** and may occupy from 1 to 16 bits. Bitfields are allocated from low-order to high-order bits within a word.

For example,

```
struct mystruct {
   int      i : 2;
   unsigned j : 5;
   int        : 4;
   int      k : 1;
   unsigned m : 4;
} a,b,c;
```

produces the following layout:



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
m				k	(unused)				j					i	

Integer fields are stored in two's complement form with the leftmost bit being the sign bit. For example, a **signed int** bitfield 1 bit wide (such as *a.k*) can only hold the values –1 and 0, since any nonzero value will be interpreted as –1.

Statements (K&R 9)

Turbo C implements all the statements described in K&R without exception and without modification.

External Function Definitions (K&R 10.1)

In Turbo C, **extern** declarations given inside a function obey proper block scope; they will not be recognized beyond the scope of the block in which they are defined. However, Turbo C will remember the declarations in order to compare them with later declarations of the same object.

Turbo C implements most of the ANSI enhancements to the K&R definition of functions. This includes additional function modifiers, as well as function prototypes. Turbo C also has a few enhancements of its own, such as functions of type **interrupt**.

Function Type Modifiers (K&R 10.1.1)

In addition to **extern** and **static**, Turbo C has a number of type modifiers specific to function definitions: **pascal**, **cdecl**, **interrupt**, **near**, **far**, and **huge**.

The pascal Function Modifier

The **pascal** modifier is specific to Turbo C and is intended for functions (and pointers to functions) that use the Pascal parameter passing sequence. This allows you to write C functions that can be called from programs written in other languages; likewise, it will allow your C programs to call external routines written in languages other than C. The function name is converted to all uppercase for linking purposes.

Note: Using the -p compiler option (Calling Convention... Pascal) will cause all functions (and pointers to those functions) to be treated as if they were of type **pascal**. Also, functions declared to be of type **pascal** can still be called from C routines, so long as the C routine sees that the function is of type **pascal**. For example, if you have declared and compiled the following function:

```
pascal putnums(int i, int j, int k)
{
   printf("And the answers are:  %d, %d, and %d\n",i,j,k);
}
```

another C program could then link it in and call it, given the following declarations:

```
pascal putnums(int i, int j, int k);

main()
{
    putnums(1,4,9);
}
```

Functions of type **pascal** cannot take a variable number of arguments, unlike functions such as **printf**. For this reason, you cannot use an ellipsis (...) in a **pascal** function definition. (See "Function Prototypes" for an explanation of using the ellipsis to define a function with a variable number of arguments.)

The cdecl Function Modifier

The **cdecl** modifier is also specific to Turbo C. Like the **pascal** modifier, it is used with functions and pointers to functions. Its purpose is to override the -p compiler directive and allow a function to be called as a regular C function. For example, if you were to compile the previous program with the -p option set, but wanted to use **printf**, you might do something like this:

```
extern cdecl printf();
putnums(int i, int j, int k);

cdecl main()
{
   putnums(1,4,9);
}

putnums(int i, int j, int k)
{
   printf("And the answers are:  %d, %d, and %d\n",i,j,k);
}
```

If a program is compiled with the -p option, all functions used from the run-time library will need to have **cdecl** declarations. If you look at the header files (such as STDIO.H), you'll see that every function is explicitly defined as **cdecl** in anticipation of this. Note that **main** must also be

declared as **cdecl**; this is because the C start-up code always tries to call **main** with the C calling convention.

The interrupt Function Modifier

The **interrupt** modifier is another one specific to Turbo C. **interrupt** functions are designed to be used with the 8086/8088 interrupt vectors. Turbo C will compile an **interrupt** function with extra function entry and exit code so that registers AX, BX, CX, DX, SI, DI, ES, and DS are preserved. The other registers of BP, SP, SS, CS, and IP are preserved as part of the C-calling sequence or as part of the interrupt handling itself. Here is an example of a typical **interrupt** definition:

```
void interrupt myhandler()
{
   ...
}
```

You should declare interrupt functions to be of type **void**. Interrupt functions may be declared in any memory model. For all memory models except huge, DS is set to the program data segment. For the huge model, DS is set to the module's data segment.

The near, far, and huge Function Modifiers

The **near**, **far**, and **huge** modifiers are specific to Turbo C. They can be combined with **cdecl** or **pascal**, but not **interrupt**.

A non-**interrupt** function may be declared to be **near**, **far**, or **huge**. This will override the default settings for a given memory model. A **near** function uses **near** calls, and a **far** or **huge** function uses **far** call instructions.

In the tiny, small, and compact memory models, an unqualified function defaults to type **near**. In the medium and large models, an unqualified function defaults to type **far**. In the huge memory model, it defaults to type **huge**.

A **huge** function is the same as a **far** function, except that the DS register is set to the data segment address of the source module when a **huge** function is entered, but left unset for a **far** function.

Functions of type **huge** are useful when you must interface with code in assembly language that doesn't use the same memory allocation as Turbo C.

Function Prototypes (K&R 10.1.2)

When you are declaring a function, K&R only allows a function declarator consisting of the function name, its type, and an empty set of parentheses. The parameters (if any) are declared only when you actually define the function itself.

The ANSI standard—and Turbo C—allow you to use function prototypes to declare a function. These are declarators that include information about the function parameters. The compiler uses that information to check function calls for validity. The compiler also uses that information to coerce arguments to the proper type. Suppose you have the following code fragment:

```
long  lmax(long v1, long v2);

main()
{
   int   limit = 32;
   char  ch    = 'A';
   long  mval;

   mval = lmax(limit,ch);
}
```

Given the function prototype for **lmax**, this program will convert *limit* and *ch* to **long** using the standard rules of assignment before they are placed on the stack for the call to **lmax**. Without the function prototype, *limit* and *ch* would have been placed on the stack as an integer and a character, respectively; in that case, the stack passed to **lmax** would not match in size or content what **lmax** was expecting, leading to problems. Since K&R, C does not do any checking of parameter type or number. Using function prototypes aids greatly in tracking down bugs and other programming errors.

Function prototypes also aid in documenting code. For example, the function **strcpy** takes two parameters: A source string and a destination string. The question is, which is which? The function prototype

```
char  *strcpy(char *dest, char *source);
```

makes it clear. If a header file contains function prototypes, then you can print that file to get most of the information you need for writing programs that call those functions.

A function declarator with parentheses containing the single word **void** indicates a function that takes no arguments at all:

```
f(void)
```

Otherwise, the parentheses contain a list of declarators separated by commas. The declarator may be in the form of a cast, as in

```
func(int *, long);
```

or it may include an identifier, as in

```
func(int * count, long total);
```

In the two lists of declarators just mentioned, the function **func** accepts two parameters: a pointer to **int** named *count* and a **long** (integer) named *total*. If an identifier is included, it has no effect except to be used in the diagnostic message, if and when a parameter-type mismatch occurs.

A function prototype normally defines a function as accepting a fixed number of parameters. For C functions that accept a variable number of parameters (such as **printf**), a function prototype may end with an ellipsis (...), like this:

```
f(int *count, long total,...)
```

With this form of prototype, the fixed parameters are checked at compile time, and the variable parameters are passed as if no prototype were present.

Here are some more examples of function declarators and prototypes.

```
int  f();                              /* A function returning an int with no
                                          information about parameters. This is
                                          the K&R "classic style." */

int  f(void);                          /* A function returning an int that
                                          takes no parameters. */

int  p(int,long);                      /* A function returning an int that
                                          accepts two parameters: the first, an
                                          int; the second, a long. */

int  pascal q(void);                   /* A pascal function returning an int
                                          that takes no parameters at all. */

char far * s(char *source, int kind);  /* A function returning a far pointer
                                          to a char and accepting two
                                          parameters: the first, a pointer to a
                                          char; the second, an int. */

int  printf(char *format,...);         /* A function returning an int and
                                          accepting a pointer to a char fixed
                                          parameter and any number of additional
                                          parameters of unknown type. */

int  (*fp)(int);                       /* A pointer to a function returning
                                          an int and accepting a single int
                                          parameter. */
```

Here is a summary of the rules governing how Turbo C deals with language modifiers and formal parameters in function calls, both with and without prototypes.

Rule #1: The language modifiers for a function definition must match the modifiers used in the declaration of the function at all calls to the function.

Rule #2: A function may modify the values of its formal parameters, but this has no effect on the actual arguments in the calling routine, except for interrupt functions. See "Interrupt Functions" in Chapter 12 for more information.

When a function prototype has not been previously declared, Turbo C converts integral arguments to a function call according to the integral widening (expansion) rules described in "Arithmetic Conversions." When a function prototype is in scope, Turbo C converts the given argument to the type of the declared parameter as if by assignment.

When a function prototype includes an ellipsis (...), Turbo C converts all given function arguments as in any other prototype (up to the ellipsis). The compiler will widen any arguments given beyond the fixed parameters, according to the normal rules for function arguments without prototypes.

If a prototype is present, the number of arguments must match (unless an ellipsis is present in the prototype). The types must be compatible only to the extent that an assignment can legally convert them. You can always use an explicit cast to convert an argument to a type that is acceptable to a function prototype.

The following example should clarify these points:

```
int   strcmp(char *s1, char *s2);                    /* Full prototype */
char  *strcpy();                                       /* No prototype */
int   samp1(float, int, ...);                        /* Full prototype */

samp2()
{
   char    *sx, *cp;
   double  z;
   long    a;
   float   q;

   if (strcmp(sx, cp))                                   /* 1. Correct */
      strcpy(sx, cp, 44);          /* 2. OK in Turbo C but not portable */

   samp1(3, a, q);                                       /* 3. Correct */
   strcpy(cp);                                     /* 4. Run-time error */
   samp1(2);                                        /* 5. Compile error */
}
```

The five calls (numbered by comment) in this example illustrate different points about function calls and prototypes.

In call #1, the use of **strcmp** exactly matches the prototype and everything is proper.

In call #2, the call to **strcpy** has an extra argument (**strcpy** is defined for two arguments, not three). In this case, Turbo C will waste a little time and code pushing an extra argument. However, there is no syntax error because the compiler has not been told about the arguments to **strcpy**. This call is not portable.

In call #3, the prototype directs that the first argument to **samp1** be converted to **float** and the second argument to **int**. The compiler will warn about possible loss of significant digits because a conversion from **long** to **int** chops the upper bits. (You can eliminate this warning with an explicit cast to **int**). The third argument, *q*, lines up with the ellipsis in the prototype, so it is converted to **double** according to the usual arithmetic conversions; the whole call is correct.

In call #4, **strcpy** is again called, but now with too few arguments. This will cause an execution error, and it may crash the program. The compiler will

say nothing (even though the number of parameters differs from that in a previous call to the same function!), since there is no function prototype for **strcpy**.

In call #5, **sampl** is called with too few arguments. Since **sampl** requires a minimum of two arguments, this statement is an error. The compiler will give a message about too few arguments in a call.

Important Note: If your function prototype does not match the actual function definition, Turbo C will detect this *if and only if* that definition is in the same file as the prototype. If you create a library of routines with a corresponding header file of prototypes, you might consider including that header file when you compile the library, so that any discrepancies between the prototypes and the actual definitions will be caught.

Scope Rules (K&R 11)

Turbo C is more liberal in allowing nonunique identifiers than K&R specifies a compiler need be. There are four distinct classes of identifiers in this implementation:

Variables, typedefs, and **enumeration members** must be unique within the block in which they are defined. Externally declared identifiers must be unique among externally declared variables.

Structure, union, and **enumeration tags** must be unique within the block in which they are defined. Tags declared outside of any function must be unique within all tags defined externally.

Structure and **union member** names must be unique within the structure or union in which they are defined. There is no restriction on the type or offset of members with the same member name in different structures.

Goto labels must be unique within the function in which they are declared.

Compiler Control Lines (K&R 12)

Turbo C supports all the control commands found in K&R. These preprocessor directives are source lines with an initial #, which may be preceded or followed by whitespace.

Token Replacement (K&R 12.1)

Turbo C implements the K&R definition of `#define` and `#undef` with the following additions.

- The following identifiers may not appear in a `#define` or `#undef` directive:

```
__STDC__
__FILE__
__LINE__
__DATE__
__TIME__
```

- Two tokens may be pasted together in a macro definition by separating them with ## (plus optional whitespace on either side). The preprocessor removes the whitespace and the ##, combining the separate tokens. This can be used to construct identifiers; for example, given the construct

```
#define VAR(i,j) (i ## j)
```

then `VAR(x,6)` would expand to (*x6*). This replaces the sometimes-used (but nonportable) method of using `(i/**/j)`.

- Nested macros mentioned in a macro definition string are expanded only when the macro itself is expanded, not when the macro is defined. This mostly affects the interaction of `#undef` with nested macros.
- The # symbol can be placed in front of a macro argument in order to *stringize* the argument (convert it to a string). When the macro is expanded, #*<formal arg>* is replace with "*<actual arg>*". So, given the following macro definition:

```
#define TRACE(flag) printf(#flag "=%d\n", flag)
```

then the code fragment

```
highval = 1024;
TRACE(highval);
```

becomes

```
highval = 1024;
printf("highval" "= %d\n", highval);
```

which, in turn, becomes

```
highval = 1024;
printf("highval=%d\n", highval);
```

- Unlike other implementations, Turbo C does *not* expand macro arguments inside strings and character constants.

File Inclusion (K&R 12.2)

Turbo C implements the `#include` directive as found in K&R but has the following additional feature: If the preprocessor can't find the include file in the default directory, assuming that you used the form

```
#include "filename"
```

then it searches the directories specified with the compiler option `-I`.

If you used the form `#include <filename>`, then only those directories specified with `-I` are searched. (Directories listed under the menu option **O**/**E**nvironment/**I**nclude Directories are equivalent to those given with the `-Ipathname` command-line option.)

You may construct the `#include` path name, including delimiters, using macro expansion. If the next line after the keyword begins with an identifier, the preprocessor scans the text for macros. However, if a string is enclosed in quotes or angle brackets, Turbo C will not examine it for embedded macros.

So, if you have the following:

```
#define   myinclude     "c:\tc\include\mystuff.h"
#include  myinclude
#include "myinclude.h"
```

then the first `#include` statement will cause the preprocessor to look for C:\TC\INCLUDE\MSTUFF.H, while the second will cause it to look for MYINCLUDE.H in the default directory.

Also, you may not use string literal concatenation and token pasting in macros that are used in an `#include` statement. The macro expansion must produce text that reads like a normal `#include` directive.

Conditional Compilation (K&R 12.3)

Turbo C supports the K&R definition of conditional compilation by replacing the appropriate lines with a line containing only whitespace. The lines thus ignored are those beginning with `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif` directives, as well as any lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun.

Turbo C also supports the ANSI operator `defined(symbol)`. This will evaluate to 1 (*true*) if the symbol has been previously defined (using

#define) and has not been subsequently undefined (using #undef); otherwise, it evaluates to 0 (*false*). So the directive

```
#if defined(mysym)
```

is the same as

```
#ifdef mysym
```

The advantage is that you can use defined repeatedly in a complex expression following the #if directive, such as

```
#if defined(mysym) || defined(yoursym)
```

Finally, Turbo C (unlike ANSI) allows you to use the **sizeof** operator in a preprocessor expression. Thus, you can write the following

```
#if (sizeof(void *) == 2)
#define SDATA
#else
#define LDATA
#endif
```

Line Control (K&R 12.4)

Turbo C supports the K&R definition of #line. Macros are expanded in #line as they are in the #include directive.

Error Directive (ANSI C 3.8.5)

Turbo C supports the #error directive, which is mentioned (but not explicitly defined) in the ANSI standard. The format is

```
#error errmsg
```

and the message issued is

```
Fatal: filename line#  Error directive: errmsg
```

Typically, programmers include this directive in a preprocessor conditional that catches some undesired compile-time condition. In the normal case, that condition won't be true. In the event that the condition is true, you want the compiler to print an error message and stop the compile. You do this by putting an #error directive within a conditional that is true for the undesired case.

For example, suppose you `#define` *MYVAL*, which must be either 0 or 1. You could then include the following conditional in your source code to test for an incorrect value of *MYVAL*:

```
#if (MYVAL != 0 && MYVAL != 1)
#error MYVAL must be defined to either 0 or 1
#endif
```

The preprocessor scans the text to remove comments but displays any remaining text without looking for embedded macros.

Pragma Directive (ANSI C 3.8.6)

Turbo C supports the `#pragma` directive, which (like `#error`) is vaguely defined in the ANSI standard. Its purpose is to permit implementation-specific directives of the form:

```
#pragma <directive name>
```

With `#pragma`, Turbo C can define whatever directives it desires without interfering with other compilers that support `#pragma`. Why? Because, by definition, if the compiler doesn't recognize the directive name, it ignores the `#pragma` directive.

#pragma inline

Turbo C recognizes three `#pragma` directives. The first is

```
#pragma inline
```

This directive is equivalent to the `-B` compiler option. It tells the compiler that there is inline assembly language code in your program (see Chapter 12). This is best placed at the top of the file, since the compiler restarts itself with the `-B` option when `#pragma inline` is encountered. Actually, you can leave off both the `-B` option and the `#pragma inline` directive, and the compiler will restart itself anyway as soon as it encounters **asm** statements; the purpose of the option and the directive is to save some compile time.

#pragma warn

The second `#pragma` directive is

```
#pragma warn
```

This directive allows you to override specific -wxxx command-line options (or specific **D**isplay Warnings...On options).

For example, if your source code contains the directives

```
#pragma warn +xxx
#pragma warn -yyy
#pragma warn .zzz
```

the *xxx* warning will be turned *on* (even if on the **O/C/E**rrors menu it was toggled to *off*), the *yyy* warning will be turned *off*, and the *zzz* warning will be restored to the value it had when compilation of the file began.

A complete list of the three-letter abbreviations and the warnings to which they apply is given in Appendix C in the *Turbo C Reference Guide*.

#pragma saveregs

This pragma guarantees that a **huge** function will not change the value of any of the registers when it is entered. This directive is sometimes needed for interfacing with assembly language code. The directive should be placed immediately before the function definition. It applies to that function alone.

Null Directive (ANSI C 3.7)

For the sake of completeness, the ANSI standard and Turbo C recognize the null directive, which simply consists of a line containing the character #. This directive is always ignored.

Predefined Macro Names (ANSI C 3.8.8)

The ANSI standard requires five predefined macros to be made available by the implementation. Turbo C implements all five. Note that each of these starts and ends with two underscore characters (__).

__LINE__ The number of the current source-file line being processed, a decimal constant. The first line of a source file is defined to be 1.

__FILE__ The name of the current source file being processed, a string literal.

This macro changes whenever the compiler processes an `#include` directive or a `#line` directive, or when the include file is complete.

__DATE__ The date the preprocessor began processing the current source file, a string literal.

Each inclusion of __DATE__ in a given file is guaranteed to contain the same value, regardless of how long the processing takes. The date appears in the format *mmm dd yyyy*, where *mmm* equals the month (Jan, Feb, etc.), *dd* equals the day (1...31, with the first character of *dd* a blank if the value is less than 10), and *yyyy* equals the year (1986, 1987, etc.).

__TIME__ The time the preprocessor began processing the current source file, a string literal.

Each inclusion of __TIME__ is guaranteed to contain the same value, regardless of how long the processing takes. It takes the format *hh:mm:ss*, where *hh* equals the hour (00...23), *mm* equals minutes (00...59), and *ss* equals seconds (00...59).

__STDC__ The constant 1, if you compile with the ANSI compatibility (`-A`) flag (ANSI Keywords Only...ON); otherwise, the macro is undefined.

Turbo C Predefined Macros

The Turbo C preprocessor defines several additional macros for your use. As with the ANSI-prescribed macros, each starts and ends with two underscore characters.

__TURBOC__ Gives the current Turbo C version number, a hexadecimal constant. Version 1.0 is 0x0100; version 1.2 is 0x0102; and so on.

__PASCAL__ Signals `-p` flag; set to the integer constant 1 if `-p` flag is used; undefined otherwise.

__MSDOS__ The integer constant 1 for all compiles.

__CDECL__ Signals that the `-p` flag was not used (Calling Convention...C): set to the integer constant 1 if `-p` was not used; undefined otherwise.

The following six symbols are defined based on the memory model chosen at compile time. Only one is defined for any given compilation; the others, by definition, are undefined. For example, if you compile with the small model, __SMALL__ is defined and the rest are not, so that the directive `#if defined(__SMALL__)` will be *true*, while `#if defined(__HUGE__)` (or any of the others) will be *false*. The actual value for any of these defined macros is 1.

__TINY__	The tiny memory model selection options
__SMALL__	The small memory model selection options
__MEDIUM__	The medium memory model selection options
__COMPACT__	The compact memory model selection options
__LARGE__	The large memory model selection options
__HUGE__	The huge memory model selection options

Anachronisms (K&R 17)

None of the anachronisms mentioned in K&R exist in Turbo C.

C H A P T E R

12

Advanced Programming in Turbo C

We knew you'd get around to this chapter sooner or later. You've undoubtedly worked through the earlier chapters at an alarming rate, absorbing knowledge like a sponge absorbs water. And now you want to explore new and more rarified realms. Glad to have you here.

We'll cover three major topics in this chapter. First, we'll talk about memory models, from tiny to huge. We'll tell you what they are, how to choose one, and why you would (or would not) want to use a particular memory model. Next, we'll discuss the issues in mixed-language programming. You've seen that some already in Chapter 10, which talked about mixing Turbo C and Turbo Prolog. Here, we'll be talking about how to mix with other languages, including Pascal and assembly language. After that, we'll look at three aspects of low-level programming in Turbo C: inline assembly code, pseudo-variables, and interrupt-handling. Finally, we'll look at floating-point issues. So let's get started.

Memory Models

What are memory models, and why do you have to worry about them? To answer that question, we have to take a look at the computer system you're working on. Its central processing unit (CPU) is a microprocessor belonging to the Intel iAPx86 family; probably an 8088, though possibly an 8086, an 80186, an 80286, or an 80386. For now, we'll just refer to it as an 8086.

General Purpose Registers

AX	AH	AL	accumulator (math
BX	BH	BL	base
CX	CH	CL	count (loops, etc.)
DX	DH	DL	data (holding

Segment Address

CS		code segment pointer
DS		data segment
SS		stack segment
ES		extra segment

Special Purpose

SP		stack pointer
BP		base
SI		source
DI		destination

Figure 12.1: 8086 Registers

Figure 12.1 shows the registers found in the 8086 processor, with a brief description of what each is used for. There are two more registers—IP (instruction pointer) and the flag register—but Turbo C can't access them, so they aren't shown here.

General-Purpose Registers

The general-purpose registers are the ones used most often to hold and manipulate data. Each has some special functions that only it can do. For example,

- Many math operations can only be done using AX.
- BX can be used to hold the offset portion of a far pointer.
- CX is used by some of the 8086's LOOP instructions.
- DX is used by certain instructions to hold data.

But there are many operations that all these registers can do; in many cases, you can freely exchange one for another.

Segment Registers

The segment registers hold the starting address of each of the four segments. As described in the next section, the 16-bit value in a segment register is shifted left 4 bits (multiplied by 16) to get the true 20-bit address of that segment.

Special Purpose Registers

The 8086 also has some special purpose registers.

- The SI and DI registers can do many of the things the general-purpose registers can, plus they are used as index registers. They're also used by Turbo C for register variables.
- The SP register points to the current top-of-stack and is an offset into the stack segment.
- The BP register is a secondary stack pointer, usually used to index into the stack in order to retrieve parameters.

The base pointer (BP) register is used in C functions as a base address for arguments and automatic variables. Parameters have positive offsets from BP, which vary depending on the memory model and the number of

registers saved on function entry. BP always points to the saved previous BP value. Functions that have no parameters and declare no arguments will not use or save BP at all.

Automatic variables are given negative offsets from BP, with the first automatic variables having the largest magnitude negative offset.

Memory Segmentation

The Intel 8086 microprocessor has a *segmented memory architecture.* It has a total address space of 1 Mb, but it is designed to directly address only 64K of memory at a time. A 64K chunk of memory is known as a segment; hence the phrase, "segmented memory architecture."

Now, how many different segments are there, where are they located, and how does the 8086 know where they're located?

- The 8086 keeps track of four different segments: *code, data, stack,* and *extra.* The code segment is where the machine instructions are; the data segment, where information is; the stack is, of course, the stack; and the extra segment is used (usually) for extra data.
- The 8086 has four 16-bit segment registers (one for each segment) named CS, DS, SS, and ES; these point to the code, data, stack, and extra segments, respectively.
- A segment can be located anywhere in memory—at least, almost anywhere. For reasons that will become clear as you read on, a segment must start on an address that's evenly divisible by 16 (in base 10).

Address Calculation

Okay, so how does the 8086 use these segment registers to calculate an address? A complete address on the 8086 is composed of two 16-bit values: the segment address and the offset. Suppose the data segment address—the value in the DS register—is 2F84 (base 16), and you want to calculate the actual address of some data that has an offset of 0532 (base 16) from the start of the data segment; how is that done?

Address calculation is done as follows: Shift the value of the segment register 4 bits to the left (equivalent to one hex digit), then add in the offset.

The resulting 20-bit value is the actual address of the data, as illustrated here:

```
DS register (shifted):  0010 1111 1000 0100 0000  =  2F840
Offset:                      0000 0101 0011 0010  =  00532
-----------------------------------------------------------
Address:                0010 1111 1101 0111 0010  =  2FD72
```

The starting address of a segment is always a 20-bit number, but a segment register only holds 16 bits—so the bottom 4 bits are always assumed to be all zeros. This means—as we said—that segments can only start every 16 bytes through memory, at an address where the last 4 bits (or last hex digit) are zero.

So, if the DS register is holding a value of 2F84, then the data segment actually starts at address 2F840. By the way, a chunk of 16 bytes is known as a *paragraph*, so you could say that a segment always starts on a paragraph boundary.

The standard notation for an address takes the form *segment:offset*; for example, the previous address would be written as 2F84:0532. Note that since offsets can overlap, a given segment:offset pair is not unique; the following addresses all refer to the same memory location:

```
0000:0123
0002:0103
0008:00A3
0010:0023
0012:0003
```

One last note: Segments can (but do not have to) overlap. For example, all four segments could start at the same address, which means that your entire program would take up no more than 64K—but that's all the space you would have for your code, your data, and your stack.

Near, Far, and Huge Pointers

What do pointers have to do with memory models and Turbo C? A lot. The type of memory model you choose will determine the default type of pointers used for code and data. However, you can explicitly declare a pointer (or a function) to be of a specific type, regardless of the model being used. Pointers come in three flavors: *near* (16 bits), *far* (32 bits) and *huge* (also 32 bits); let's look at each.

Near Pointers

A 16-bit (near) pointer relies on one of the segment registers to finish calculating its address; for example, a pointer to a function would add its 16-bit value to the left-shifted contents of the code segment (CS) register. In a similar fashion, a near data pointer contains an offset to the data segment (DS) register. Near pointers are easy to manipulate, since any arithmetic (such as addition) can be done without worrying about the segment.

Far Pointers

A far (32-bit) pointer contains not only the offset within the segment, but also (as another 16-bit value) the segment address, which is then left-shifted and added to the offset. By using far pointers, you can have multiple code segments; that, in turn, allows you to have programs larger than 64K. Likewise, with far data pointers you can address more than 64K worth of data.

When you use far pointers for data, you need to be aware of some potential problems in pointer manipulation. As explained in the section on address calculation, you can have many different segment:offset pairs refer to the same address. For example, the far pointers 0000:0120, 0010:0020, and 0012:0000 all resolve to the same 20-bit address. However, if you had three different far pointer variables—*a*, *b*, and *c*—containing those three values respectively, then all the following expressions would be *false*:

```
if (a == b) ...
if (b == c) ...
if (a == c) ...
```

A related problem occurs when you want to compare far pointers using the >, >=, <, and <= operators. In those cases, only the offset (as an **unsigned**) is used for comparison purposes; given that *a*, *b*, and *c* still have the values previously listed, the following expressions would all be *true*:

```
if (a > b) ...
if (b > c) ...
if (a > c) ...
```

The equals (==) and not-equals (!=) operators use the 32-bit value as an **unsigned long** (not as the full memory address). The comparison operators (<=, >=, <, and >) use just the offset.

The == and != operators need all 32 bits, so the computer can compare to the NULL pointer (0000:0000). If you used only the offset value for equality

checking, any pointer with 0000 offset would be equal to the NULL pointer, which is not what you want.

One more thing you should be aware of: If you add values to a far pointer, only the offset is changed. If you add enough to cause the offset to exceed FFFF (its maximum possible value), the pointer just wraps around back to the beginning of the segment. For example, if you add 1 to 5031:FFFF, the result would be 5031:0000 (not 6031:0000). Likewise, if you subtract 1 from 5031:0000, you would get 5031:FFFF (not 5030:000F).

If you want to do pointer comparisons, it's safest to use either near pointers—which all use the same segment address—or huge pointers, described next.

Huge Pointers

Huge pointers are also 32 bits long and, like far pointers, contain both a segment address and an offset. Unlike far pointers, however, they are *normalized*, to avoid the problems described in "Far Pointers."

What is a normalized pointer? It is a 32-bit pointer which has as much of its value in the segment address as possible. Since a segment can start every 16 bytes (10 in base 16), this means that the offset will only have a value from 0 to 15 (0 to F in base 16).

How do you normalize a pointer? Simple: Convert it to its 20-bit address, then use the right 4 bits for your offset and the left 16 bits for your segment address. For example, given the pointer 2F84:0532, we convert that to the absolute address 2FD72, which we then normalize to 2FD7:0002. Here are a few more pointers with their normalized equivalents:

```
0000:0123    0012:0003
0040:0056    0045:0006
500D:9407    594D:0007
7418:D03F    811B:000F
```

Now you know that huge pointers are always kept normalized. Why is this important? Because it means that for any given memory address, there is only one possible huge address—segment:offset pair—for it. And that means that the == and != operators return correct answers for any huge pointers.

In addition to that, the >, >=, <, and <= operators are all used on the full 32-bit value for huge pointers. Normalization guarantees that the results there will be correct also.

Finally, because of normalization, the offset in a huge pointer automatically wraps around every 16 values, but—unlike far pointers—the segment is adjusted as well. For example, if you were to increment 811B:000F, the result would be 811C:0000; likewise, if you decrement 811C:0000, you get 811B:000F. It is this aspect of huge pointers that allows you to manipulate data structures greater than 64K in size.

There is a price for using huge pointers: additional overhead. Huge pointer arithmetic is done with calls to special subroutines. Because of this, huge pointer arithmetic is significantly slower than that of far or near pointers.

Turbo C's Six Memory Models

Avoiding overhead—except when you want it—is just what Turbo C allows you to do. There are six different memory models you can choose from: tiny, small, medium, compact, large, and huge. Your program requirements determine which one you pick. Here's a brief summary of each:

Tiny: As you might guess, this is the smallest of the memory models. All four segment registers (CS, DS, SS, ES) are set to the same address, so you have a total of 64K for all of your code, data, and arrays. Near pointers are always used. Use this when memory is at an absolute premium. Tiny model programs can be converted to .COM format by linking with the `/t` option.

Small: The code and data segments are different and don't overlap, so you have 64K of code and 64K of static data. The stack and extra segments start at the same address as the data segment. Near pointers are always used. This is a good size for average applications.

Medium: Far pointers are used for code, but not for data. As a result, static data is limited to 64K, but code can occupy up to 1 Mb. This is best for large programs that don't keep much data in memory.

Compact: The inverse of medium: Far pointers are used for data, but not for code. Code is then limited to 64K, while data has a 1-Mb range. This choice is best if your code is small but you need to address a lot of data.

Large: Far pointers are used for both code and data, giving both a 1-Mb range. It is needed only for very large applications.

Huge: Far pointers are used for both code and data. Turbo C normally limits the size of all static data to 64K; the huge memory model sets aside that limit, allowing static data to occupy more than 64K.

The following illustrations (Figures 12.2 through 12.7) show how memory in the 8086 is apportioned for the six Turbo C memory models.

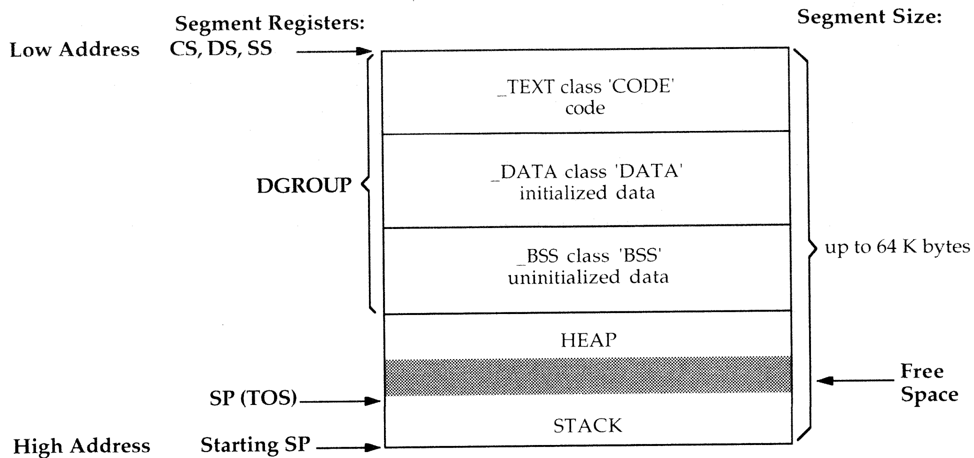


Figure 12.2: Tiny Model Memory Segmentation

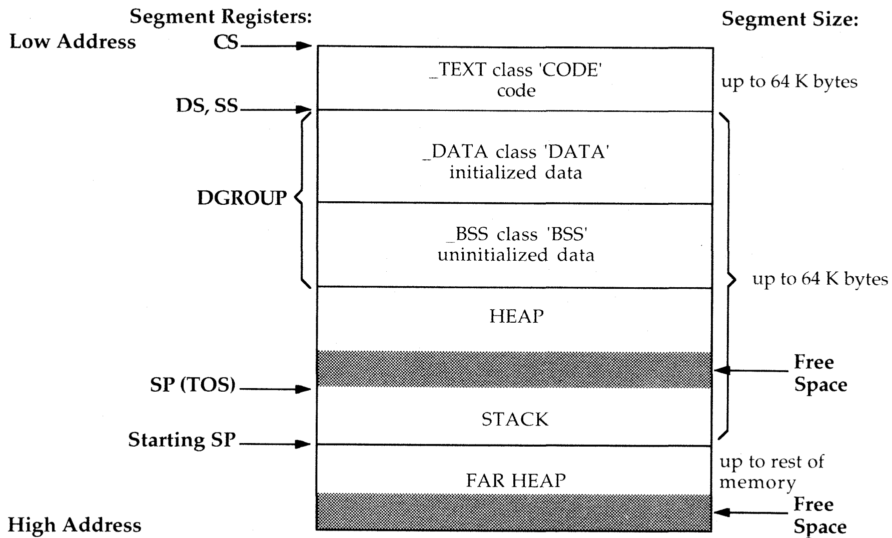


Figure 12.3: Small Model Memory Segmentation

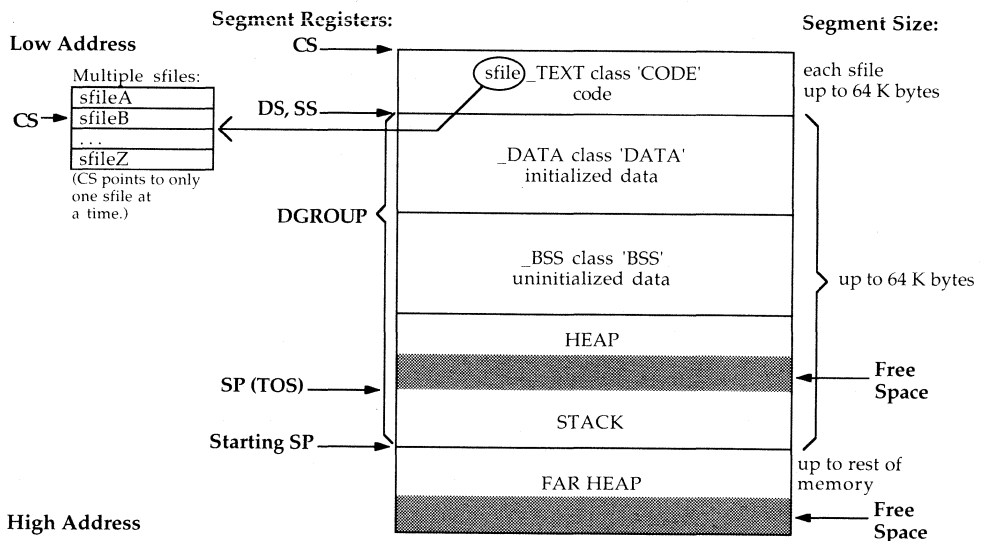


Figure 12.4: Medium Model Memory Segmentation

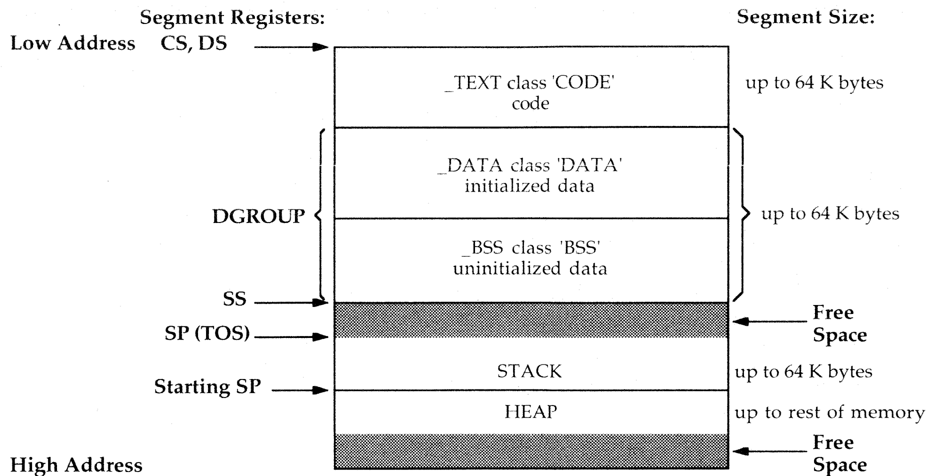


Figure 12.5: Compact Model Memory Segmentation

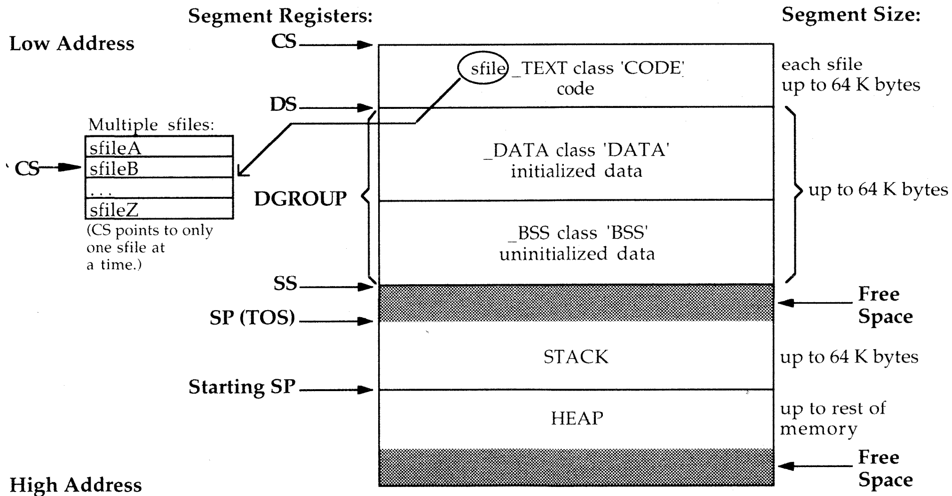


Figure 12.6: Large Model Memory Segmentation

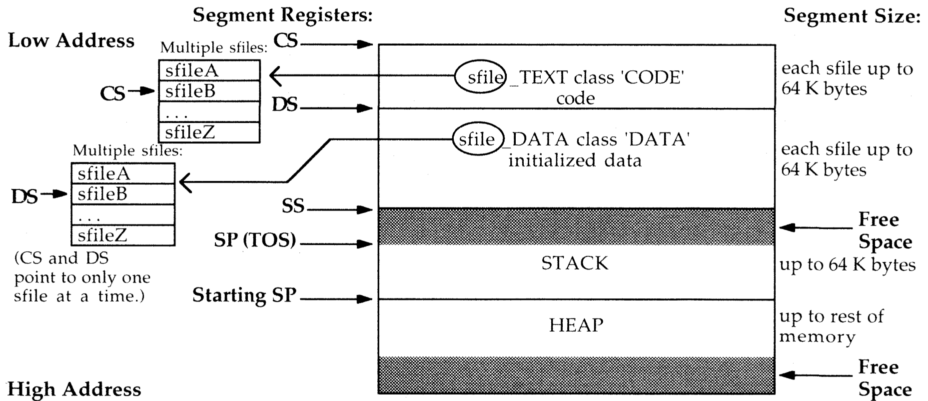


Figure 12.7: Huge Model Memory Segmentation

Table 12.1 summarizes the different models and how they compare to one another. The models are often grouped according to whether their code or data models are *small* (64K) or *large* (1 Mb); these groups correspond to the rows and columns in Table 12.1. So, for example, the models tiny, small, and compact are known as *small code models* because, by default, code pointers are near; likewise, compact, large, and huge are known as *large data models* because, by default, data pointers are far. Note that this is also true for the huge model—the default data pointers are far, not huge. If you want huge data pointers, you must declare them explicitly as huge.

Table 12.1: Memory Models

Data Size	Code Size: 64K	Code Size: 1 Mb
64K	Tiny (data, code overlap; total size = 64K)	
64K	Small (no overlap; total size = 128K)	Medium (small data, big code)
1 Mb	Compact (big data, small code)	Large (big data, code)
1 Mb		Huge (same as large but static data > 64K)

Important Note: When you compile a module (a given source file with some number of routines in it), the resulting code for that module cannot be greater than 64K, since it must all fit inside of one code segment. This is true even if you're using a large code model (medium, large, huge). If your module is too big to fit into one (64K) code segment, you must break it up into different source code files, compile each file separately, then link them together. Similarly, even though the huge model permits static data to total more than 64K, it still must be less than 64K in *each module.*

Mixed-Model Programming: Addressing Modifiers

Turbo C introduces seven new keywords not found in standard (Kernighan and Ritchie or ANSI) C: **near**, **far**, **huge**, **_cs**, **_ds**, **_es**, **_ss**. These can be used as modifiers to pointers (and in some cases, to functions), with certain limitations and warnings.

In Turbo C, you can modify functions and pointers with the keywords **near**, **far**, or **huge**. We explained **near**, **far**, and **huge** data pointers earlier in

this chapter. **near** functions are invoked with near calls and exit with near returns. Similarly, **far** functions are called far and do far returns. **huge** functions are like **far** functions, except that **huge** functions can set DS to a new value, while **far** functions cannot.

There are also four special **near** data pointers: **_cs**, **_ds**, **_ss**, and **_es**. These are 16-bit pointers that are specifically associated with the corresponding segment register. For example, if you were to declare a pointer to be

```
char _ss *p;
```

then *p* would contain a 16-bit offset into the *stack segment*.

Functions and pointers within a given program will default to near or far, depending on the memory model you select. If the function or pointer is near, then it is automatically associated with either the CS or the DS register.

Table 12.2 shows just how this works. Note that the size of the pointer corresponds to whether it is working within a 64K memory limit (near, within a segment) or inside the general 1 Mb memory space (far, has its own segment address).

Table 12.2: Pointer Results

Memory Model	Function Pointers	Data Pointers
Tiny	near, _cs	near, _ds
Small	near, _cs	near, _ds
Medium	far	near, _ds
Compact	near, _cs	far
Large	far	far
Huge	far	far

Declaring Functions to Be Near or Far

On occasion, you'll want (or need) to override the default function type of your memory model shown in Table 12.1 (page 353).

For example, suppose you're using the large memory model, but you've got a recursive (self-calling) function in your program, like this:

```
double  power(double x,int exp)
{
   if (exp <= 0)
      return(0);
   else
      return(x*power(x,exp-1));
}
```

Every time **power** calls itself, it has to do a far call, which uses more stack space and clock cycles. By declaring power as **near**, you eliminate some of the overhead by forcing all calls to that function to be near:

```
double  near power(double x,int exp)
```

This guarantees that **power** is callable only within the code segment in which it was compiled, and that all calls to it are near calls.

This means that if you are using a large code model (medium, large, or huge), you can only call **power** from within the module where it is defined. Other modules have their own code segment and thus cannot call **near** functions in different modules. Furthermore, a near function must be either defined or declared before the first time it is used, or the compiler won't know it needs to generate a near call.

Conversely, declaring a function to be far means that a far return is generated. In the small code models, the far function must be declared or defined before its first use to ensure it is invoked with a far call.

Look back at the **power** example. It is wise to also declare **power** as static, since it should only be called from within the current module. That way, being a static, its name will not be available to any functions outside the module. Since **power** always takes a fixed number of arguments, you could optimize further by declaring it **pascal**, like this:

```
static double near pascal power(double x, int exp)
```

Declaring Pointers to Be Near, Far, or Huge

You've seen why you might want to declare functions to be of a different model than the rest of the program. Why might you want to do the same thing for pointers? For the same reasons given in the preceding section: either to avoid unnecessary overhead (declaring **near** when the default

would be **far**) or to reference something outside of the default segment (declaring **far** or **huge** when the default would be **near**).

There are, of course, potential pitfalls in declaring functions and pointers to be of nondefault types. For example, say you have the following small model program:

```
void myputs(s)
char *s;
{
   int  i;
   for (i=0; s[i] != 0; i++) putc(s[i]);
}

main()
{
   char  near *mystr;

   mystr = "Hello, world\n";
   myputs(mystr);
}
```

This program works fine, and, in fact, the **near** declaration on *mystr* is redundant, since all pointers, both code and data, will be near.

But what if you recompile this program using the compact (or large or huge) memory model? The pointer *mystr* in **main** is still near (it's still a 16-bit pointer). However, the pointer *s* in **myputs** is now far, since that's the default. This means that **myputs** will pull two words out of the stack in an effort to create a far pointer, and the address it ends up with will certainly not be that of *mystr*.

How do you avoid this problem? The solution is to define **myputs** in modern C style, like this:

```
void  myputs(char *s);
{
   /*body of myputs*/

}
```

Now when Turbo C compiles your program, it knows that **myputs** expects a pointer to **char**; and since you're compiling under the large model, it knows that the pointer must be **far**. Because of that, Turbo C will push the data segment (DS) register onto the stack along with the 16-bit value of *mystr*, forming a far pointer.

How about the reverse case: Parameters to **myputs** declared as **far** and compiling with a small data model? Again, without the function prototype, you will have problems, since **main** will push both the offset and the

segment address onto the stack, but **myputs** will only expect the offset. With the prototype-style function definitions, though, **main** will only push the offset onto the stack.

Moral: If you're going to explicitly declare pointers to be of type **far** or **near**, then be sure to use function prototypes for any functions that might use them.

Pointing to a Given Segment:Offset Address

How do you make a far pointer point to a given memory location (a specific segment:offset address)? You can use the built-in library routine MK_FP, which takes a segment and an offset and returns a far pointer. For example:

```
MK_FP(segment_value, offset_value)
```

Given a far pointer, *fp*, you can get the segment component with FP_SEG(*fp*) and the offset component with FP_OFF(*fp*). For more information about these three Turbo C library routines, refer to the *Turbo C Reference Guide*.

Building Proper Declarators

A *declarator* is a statement in C that you use to declare functions, variables, pointers, and data types. And C allows you to build very complex declarators. This section gives you some examples of declarators so that you can get some practice at designing (and reading) them; it'll also show you some pitfalls to avoid.

Traditional C programming has you build your complete declarator in place, nesting definitions as needed. Unfortunately, this can make for programs that are difficult to read (and write).

Consider, for example, the declarators in Table 12.3, assuming that you are compiling under the small memory model (small code, small data).

Table 12.3: Declarators without Typedefs

Declarator	Description
`int f1();`	function returning int
`int *p1;`	pointer to int
`int *f2();`	function returning pointer to int
`int far *p2;`	far pointer to int
`int far *f3();`	near function returning far pointer to int
`int * far f4();`	far function returning near pointer to int
`int (*fp1)(int);`	pointer to function returning int and accepting int parameter
`int (*fp2)(int *ip);`	pointer to function returning int and accepting pointer to int
`int (far *fp3)(int far *ip)`	far pointer to function returning int and accepting far pointer to int
`int (far *list[5])(int far *ip);`	array of five far pointers to functions returning int and accepting far pointers to int

```
int (far *gopher(int (far * fp[5])(int far *ip)))(int far *ip);
near function accepting array of five far pointers to functions returning int
and accepting far pointers to int, and returning one such pointer
```

These are all valid declarators; they just get increasingly hard to understand. However, with judicious use of **typedef**, you can improve the legibility of these declarators.

Here are the same declarators, rewritten with the help of **typedef** statements:

Table 12.4: Declarators with Typedefs

Declaration	Meaning
`int     f1();`	function returning int
`typedef int *intptr;`	
`intptr  p1;`	pointer to int
`intptr  f2();`	function returning pointer to int
`typedef int far *farptr;`	
`farptr  p2;`	far pointer to int
`farptr  f3();`	near function returning far pointer to int
`intptr  far f4();`	far function returning near pointer to int
`typedef int (*fncptr1)(int);`	
`fncptr1 fp1;`	pointer to function returning int and accepting int parameter
`typedef int (*fncptr2)(intptr);`	
`fncptr2 fp2;`	pointer to function returning int and accepting pointer to int
`typedef int (far *ffptr)(farptr);`	
`ffptr   fp3;`	far pointer to function returning int and accepting far pointer to int
`typedef ffptr   ffplist[5];`	
`ffplist list;`	array of five far pointers to functions returning int and accepting far pointers to int
`ffptr   gopher(ffplist);`	near function accepting array of five far pointers to functions returning int and accepting far pointers to int, and returning one such pointer

As you can see, there's a big difference in legibility and clarity between this **typedef** declaration of *gopher* and the previous one. If you'll use **typedef** statements and function prototypes wisely, you'll find your programs easier to write, debug, and maintain.

Using Library Files

Turbo C offers a version of the standard library routines for each of the six memory models. Running in the Integrated Environment (TC), Turbo C is smart enough to link in the appropriate libraries in the proper order, depending on which model you've selected. Likewise, running as a stand-alone compiler (TCC), Turbo C is smart enough to link automatically.

If, however, you're using TLINK (the Turbo C linker) directly (as a stand-alone linker), you need to specify which libraries to use. If you're not going to use all six memory models, then you only need to copy (to your working disk or your hard disk) the files for the model(s) you are using. Here's a list of the library files needed for each memory model:

Tiny	C0T.OBJ, MATHS.LIB, CS.LIB
Small	C0S.OBJ, MATHS.LIB, CS.LIB
Compact	C0C.OBJ, MATHC.LIB, CC.LIB
Medium	C0M.OBJ, MATHM.LIB, CM.LIB
Large	C0L.OBJ, MATHL.LIB, CL.LIB
Huge	C0H.OBJ, MATHH.LIB, CH.LIB

Note that the tiny and small models use the same libraries, but have different startup files (C0T.OBJ vs. C0S.OBJ). Also, if your system has an 8087/80287 math coprocessor, then you'll need the file FP87.LIB; if instead you want to emulate the 8087/80287, you'll need the file EMU.LIB.

Here are some example TLINK command lines:

```
tlink c0m a b c, prog, mprog, fp87 mathm cm
tlink c0c d e f, plan, mplan, emu mathc cc
```

The first will produce an executable program called PROG.EXE, with the medium-model libraries and the 8087/80287 support library linked in. The second command line will yield PLAN.EXE, compiled as a compact-model program that emulates the 8087/80287 floating-point routines if a coprocessor is not available at run time.

Note: The order of objects and libraries is very important. You must always put the C start-up module (C0*x*.OBJ) first in the list of objects. The library list should contain, in this specific order:

- your own libraries (if any)
- FP87.LIB or EMU.LIB, followed by MATH*x*.LIB (only necessary if you are using floating point)
- C*x*.LIB (standard Turbo C run-time library file)

(The *x* in C0*x*, MATH*x*, and C*x* refers to the letter specifying the memory model: *t*, *s*, *m*, *c*, *l*, or *h*.)

Linking Mixed Modules

What if you compiled one module using the small memory model, and another module using the large model, then wanted to link them together? What would happen?

The files would link together fine, but the problems you would encounter would be similar to those described in "Declaring Functions to Be Near or Far." If a function in the small module called a function in the large module, it would do so with a near call, which would probably be disastrous.

Furthermore, you could face the same problems with pointers as described in "Declaring Pointers to Be Near, Far, or Huge," since a function in the small module would expect to pass and receive **near** pointers, while a function in the large module would expect **far** pointers.

The solution, again, is to use function prototypes. Suppose that you put **myputs** into its own module and compile it with the large memory model. Then create a header file called MYPUTS.H (or some other name with an .H extension), which would have the following function prototype in it:

```
void far myputs(char far *s);
```

Now, if you put **main** into its own module (called MYMAIN.C), set things up like this:

```
#include <stdio.h>
#include "myputs.h"

main()
{
   char  near *mystr;

   mystr = "Hello, world\n";
   myputs(mystr);
}
```

When you compile this program, Turbo C reads in the function prototype from MYPUTS.H and sees that it is a **far** function that expects a **far** pointer. Because of that, it will generate the proper calling code, even if it's compiled using the small memory model.

What if, on top of all this, you need to link in library routines? Your best bet is to use one of the large model libraries and declare everything to be **far**. To do this, make a copy of each header file you would normally include (such as STDIO.H), and rename the copy to something appropriate (such as FSTDIO.H).

Then edit each function prototype in the copy so that it is explicitly **far**, like this:

```
int far cdecl printf(char far * format, ...);
```

That way, not only will **far** calls be made to the routines, but the pointers passed will be **far** pointers as well. Modify your program so that it includes the new header file:

```
#include <fstdio.h>

main()
{
   char  near *mystr;
   mystr = "Hello, world\n";
   printf(mystr);
}
```

Compile your program with TCC, then link it with TLINK, specifying a large model library, such as CL.LIB. Mixing models is tricky, but it can be done; just be prepared for some difficult bugs if you do things wrong.

Mixed-Language Programming

Turbo C eases the way for your C programs to call routines written in other languages and, in return, for programs written in other languages to call your C routines. In this section, we make it clear how easy interfacing Turbo C to other languages can be; we also provide support information for such interface.

We will talk first about the two major sequences for passing parameters, and then get on with showing you how to write your own assembly language module.

Parameter-Passing Sequences: C and Pascal

Turbo C supports two methods of passing parameters to a function. One is the standard C method, which we will explain first; the other is the Pascal method.

C Parameter-Passing Sequence

Suppose you have declared the following function prototype:

```
void  funca(int p1, int p2, int p3);
```

By default, Turbo C uses the C parameter-passing sequence, also called the C calling convention. When this function (**funca**) is called, the parameters are pushed on the stack in right-to-left order (*p3*, *p2*, *p1*), following which the return address is pushed on the stack. So, if you make the call

```
main()
{
   int   i,j;
   long  k;
   ...
   i = 5; j = 7; k = 0x1407AA;
   funca(i,j,k);
   ...
}
```

the stack will look like this (just before the return address is pushed):

```
SP + 06:  0014
SP + 04:  07AA  k = p3
SP + 02:  0007  j = p2
SP:       0005  i = p1
```

(Remember that, on the 8086, the stack grows from high memory to low memory, so that *i* is currently at the *top* of the stack.) The routine being called doesn't need to know (and, for that matter, can't know) exactly how many parameters have been pushed onto the stack. All it assumes is that the parameters it expects are there.

Also—and this is very important—the routine being called should not pop parameters off the stack. Why? Because the calling routine will. For example, the assembly language that the compiler produces from the C source code for this main function looks something like this:

```
mov   word ptr [bp-8],5                          ; Set i = 5
mov   word ptr [bp-6],7                          ; Set j = 7
mov   word ptr [bp-2],0014h               ; Set k = 0x1407AA
mov   word ptr [bp-4],07AAh
push  word ptr [bp-2]                    ; Push high word of k
push  word ptr [bp-4]                     ; Push low word of k
push  word ptr [bp-6]                                ; Push j
push  word ptr [bp-8]                                ; Push i
call  near ptr funca                 ; Call funca (push addr)
add   sp,8                                     ; Adjust stack
```

Note carefully that last instruction: `add sp,8`. The compiler knows at that point exactly how many parameters have been pushed onto the stack; it also knows that the return address was pushed by the call to **funca** and was already popped off by the `ret` instruction at the end of **funca**.

Pascal Parameter-Passing Sequence

The other approach is the standard Pascal method for passing parameters (also known as the Pascal calling convention). Note that this does *not* mean you can call Turbo Pascal functions from Turbo C: You can't. This sequence pushes the parameters on left-to-right, so that if **funca** is declared as

```
void  pascal funca(int p1, int p2, int p3);
```

then, when this function is called, the parameters are pushed on the stack in left-to-right order (*p1*, *p2*, *p3*), following which the return address is pushed on the stack. So, if you make the call

```
main()
{
   int   i,j;
   long  k;
   ...
   i = 5; j = 7; k = 0x1407AA;
   funca(i,j,k);
   ...
}
```

the stack will look like this (just before the return address is pushed):

```
SP + 06:  0005   i = p1
SP + 04:  0007   j = p2
SP + 02:  0014
SP:       07AA   k = p3
```

So, what's the big difference? Well, besides switching the order in which the parameters are pushed, the Pascal parameter-passing sequence assumes that the routine being called (**funca**) knows how many parameters are being passed to it and adjusts the stack accordingly. In other words, the assembly language for the call to **funca** now looks like this:

```
push  word ptr [bp-8]                    ; Push i
push  word ptr [bp-6]                    ; Push j
push  word ptr [bp-2]       ; Push high word of k
push  word ptr [bp-4]        ; Push low word of k
call  near ptr funca      ; Call funca (push addr)
```

Note that there is no `add sp,8` instruction after the call. Instead, **funca** uses the instruction `ret 8` at termination to clean up the stack before returning to **main**.

By default, all functions you write in Turbo C use the C method of parameter passing. The only exception is when you use the `-p` compiler option (Calling Convention...Pascal), in which case all functions use the

Pascal method. In that situation, you can force a given function to use the C method of parameter passing by using the modifier **cdecl**, as in

```
void  cdecl funca(int p1, int p2, int p3);
```

That overrides the -p compiler directive.

Now, why would you want to use the Pascal calling convention at all? There are three major reasons.

- You may be calling existing assembly language routines that use that calling convention.
- You may be calling routines written in another language.
- The calling code produced is slightly smaller, since it doesn't have to clean up the stack afterwards.

What problems might arise from using the Pascal calling convention?

First, it's not as robust as the C calling convention. You cannot pass a variable number of parameters (as you can with the C convention), since the routine being called has to know how many parameters are being passed and clean up the stack accordingly. Passing either too few or too many parameters will almost certainly lead to serious problems, whereas doing so to a C-convention routine usually has no ill effects (beyond, possibly, wrong answers).

Second, if you use the -p compiler option, then you must be sure to include any header files for standard C functions that you call. Why? Because if you don't, Turbo C will use the Pascal calling convention to call each of those functions—and your program will surely crash because (1) the parameters will be in the wrong order, and (2) nobody will clean up the stack.

The header files declare each of those functions as **cdecl**, so if you #include them, the compiler will see that and use the C calling convention instead. However, because **cdecl** identifiers are underscored while **pascal** identifiers are not, you will get lots of link errors—unless you selected Generate Underbars...Off and linked with no case-sensitivity. Then you're in big trouble.

The upshot is this: If you're going to use the Pascal calling convention in a Turbo C program, be sure to use function prototypes as much as possible, with each function explicitly declared as **cdecl** or **pascal**. It's useful in this case to enable the "prototype required" warning option to ensure that every function called has a prototype.

Assembly Code Interface

Now you know how each of the calling conventions work, which tells you what the Turbo C compiler does. What do you do in the routine being called? Take a look now at how to write assembly language routines that you can call from Turbo C.

Note: In this section, we assume that you know how to write 8086 assembly language routines and how to define segments, data constants, and so on. If you are unfamiliar with these concepts, read the *Turbo Assembler Reference Guide* for more information.

Setting Up to Call .ASM from Turbo C

You should write assembly language routines as modules to be linked into your C programs. However, there are certain conventions that you must follow to (1) ensure that the linker can get the necessary information, and (2) ensure that the file format jibes with the memory model used for your C program. The general layout is as follows:

Identifier	Name	File Name
< text >	SEGMENT	BYTE PUBLIC 'CODE'
	ASSUME	CS: < text >, DS: < dseg >
	<................. code segment	.................>
< text >	ENDS	
< dseg >	GROUP	_DATA, _BSS
< data >	SEGMENT	WORD PUBLIC 'DATA'
	<.......... initialized data segment	...........>
< data >	ENDS	
_BSS	SEGMENT	WORD PUBLIC 'BSS'
	<........ uninitialized data segment	..........>
_BSS	ENDS	
	END	

The identifiers *<text>*, *<data>*, and *<dseg>* in this layout have specific replacements, depending on the memory model being used; Table 12.5 (page 367) shows what you should use for each model. *filename* in Table 12.5 is the name of the module; it should be used consistently in the NAME directive and in the identifier replacements.

Note that with the huge memory model, there is no _BSS segment, and the GROUP definition is dropped completely. In general, _BSS is optional; you only define it if you will be using it.

The best way to create an assembly language template is to compile an empty program to .ASM (using the TCC option -S) and look at the generated assembly code.

Table 12.5: Identifier Replacements and Memory Models

Model	Identifier Replacements	Code and Data Pointers
Tiny, Small	<code> = _TEXT <data> = _DATA <dseg> = DGROUP	Code: DW _TEXT:xxx Data: DW DGROUP:xxx
Compact	<code> = _TEXT <data> = _DATA <dseg> = DGROUP	Code: DW _TEXT:xxx Data: DD DGROUP:xxx
Medium	<code> = filename_TEXT <data> = _DATA <dseg> = DGROUP	Code: DDxxx Data: DW DGROUP:xxx
Large	<code> = filename_TEXT <data> = _DATA <dseg> = DGROUP	Code: DDxxx Data: DD DGROUP:xxx
Huge	<code> = filename_TEXT <data> = filename_DATA <dseg> = filename_DATA	Code: DDxxx Data: DDxxx

Defining Data Constants and Variables

Memory models also affect how you define any data constants that are pointers to code, data, or both. Table 12.5 shows what those pointers should look like, where *xxx* is the address being pointed to.

Note carefully that some definitions use DW (Define Word), while others use DD (Define Doubleword), indicating the size of the resulting pointer. Numeric and text constants are defined normally.

Variables are, of course, defined just the same as constants. If you want variables that are not initialized to specific values you can declare them in the _BSS segment, entering a question mark (?) where you would normally put a value.

Defining Global and External Identifiers

Now you have created a module, but that isn't going to do you much good unless your Turbo C program knows what functions it can call and what variables it can reference. Likewise, you may want to be able to call your Turbo C functions from within your assembly language routines, or you may want to be able to reference variables declared within your Turbo C program.

When making these calls, you need to understand something about the Turbo C compiler and linker. When you declare an external identifier, the compiler automatically sticks an underscore (_) on the front before saving that identifier in the object module. This means that you should put an underscore on the front of any identifiers in your assembly language module that you want to reference from your C program. Pascal identifiers are treated differently than C identifiers—they are uppercased and are *not* prefixed with an underscore character.

Underscores for C identifiers are optional, but *on* by default. They can be turned *off* with the `-u-` command-line option. However, if you are using the standard Turbo C libraries, you will encounter problems unless you rebuild the libraries. (To do this, you will need another Turbo C product—the source code to the run-time libraries; contact Borland International for more information.)

If any **asm** code in your source file references any C identifiers (data or functions), those identifiers must begin with underscore characters.

The Turbo Assembler (TASM) is not case-sensitive; in other words, when you assemble a program, all identifiers are saved as uppercase only. The `/mx` switch to TASM makes it case-sensitive for externals. The Turbo C linker does the same thing with **extern** identifiers, so things should match up fine. You'll notice that in our examples, we put keywords and directives in uppercase, and all other identifiers and opcodes in lowercase; this matches the style found in the TASM reference manual. You are free to use all uppercase (or all lowercase), or any mixture thereof, as you please.

To make the identifiers (names of routines and variables) visible outside of your assembly language module, you need to declare them as being PUBLIC.

So, for example, if you were to write a module that had the integer functions *max* and *min*, and the integer variables MAXINT, *lastmax* and *lastmin*, you would put the statement

```
PUBLIC  _max, _min
```

in your code segment, and the statements

```
PUBLIC    _MAXINT, _lastmax, _lastmin
_MAXINT   DW  32767
_lastmin  DW  0
_lastmax  DW  0
```

in your data segment.

Setting Up to Call Turbo C from .ASM

You use the EXTRN statement to let your assembly language module reference functions and variables that are declared in your Turbo C program.

Referencing Functions

To be able to call a C function from an assembly language routine, you must declare it in your module with the statement

```
EXTRN <fname> : <fdist>
```

where *<fname>* is the name of the function, and *<fdist>* is either near or far, depending on whether the C function is near or far. If *<fdist>* is near, then the EXTRN statement must appear within the code segment of your module; if it's far, then the EXTRN statement should appear outside of any segment. So you could have the following in your code segment:

```
EXTRN  _myCfunc1:near, _myCfunc2:far
```

allowing you to call *myCfunc1* and *myCfunc2* from within your assembly language routines.

Referencing Data

To reference variables, you should place the appropriate EXTRN statement(s) inside of your data segment, using the format

```
EXTRN  <vname> : <size>
```

where <*vname*> is the name of the variable, and <*size*> indicates the size of the variable.

The possible values for <*size*> are as follows:

- BYTE (1 byte)
- WORD (2 bytes)
- DWORD (4 bytes)
- QWORD (8 bytes)
- TBYTE (10 bytes)

Arrays must use the size of the array elements for <*size*>. Structures should be declared with the most frequently used size in the structure substituted for <*size*>.

So, if your C program had the following global variables:

```
int   i,jarray[10];
char  ch;
long  result;
```

you could make them visible within your module with the following statement:

```
EXTRN_i:WORD,_jarray:WORD,_ch:BYTE,_result:DWORD
```

Last Important Note: If you're using the huge memory model, the EXTRN statements must appear outside of any segments. This applies to both procedures and variables.

Defining Assembly Language Routines

Now that you know how to set everything up, it's a good time to look at how to actually write a function in assembly language. There are some important things to consider: parameter passing, returning values, and register conventions.

Suppose you want to write the function **min**, which you can assume has the following function prototype in C:

```
int  extern min(int v1, int v2);
```

You want **min** to return the minimum of the two values passed to it. The overall format of **min** is going to be

```
      PUBLIC  _min
_min  PROC    near
      .
      .
      .
_min  ENDP
```

This assumes, of course, that **min** is going to be a near function; if it were a far function, you would substitute **far** for **near**. Note that we've added the underscore to the start of **min**, so that the Turbo C linker can correctly resolve the references.

Passing Parameters

Your first decision is which parameter-passing convention to use; barring an adequate reason to use it, you should avoid the Pascal convention and go with the C method instead. This means that when **min** gets called, the stack is going to look like this:

```
SP + 04:  v2
SP + 02:  v1
SP:       return addr
```

You want to get to the parameters without popping anything off the stack, so you'll save the base pointer (BP), move the stack pointer (SP) into the base pointer, then use that to index directly into the stack to get your values. Note that when you push BP onto the stack, the relative offsets of the parameters will increase by two, since there will now be two more bytes on the stack.

Handling Return Values

Your function returns an integer value; where do you put that? For 16-bit (2-byte) values (**char**, **short**, **int**, **enum**, and **near** pointers), you use the AX register; for 32-bit (4-byte) values (including **far** and **huge** pointers), you use the DX register as well, with the high-order word (segment address for pointers) in DX and the low-order word in AX.

float, **double**, and **long double** values are returned in the 8087/80287 top-of-stack (TOS) register, ST(0); if the 8087/80287 emulator is being used, then the value is returned in the emulator TOS register.

Structure values are returned by placing the value in a static data location, then returning a pointer to that location (AX in the small data models, DX:AX in the large data models).

The calling function must then copy that value to wherever it's needed. Structures that are 1 or 2 bytes long are returned in AX (like any normal **int**), while 4-byte structures are returned in AX and DX.

For the **min** example, all you're dealing with is a 16-bit value, so you can just place the answer in AX.

Here's what your code looks like now:

```
        PUBLIC  _min
_min    PROC    near
        push    bp                              ; Save bp on stack
        mov     bp,sp                           ; Copy sp into bp
        mov     ax,[bp+4]                       ; Move v1 into ax
        cmp     ax,[bp+6]                       ; Compare with v2
        jle     exit                            ; If v1 > v2
        mov     ax,[bp+6]                       ; Then load ax with v2
exit:   pop     bp                              ; Restore bp
        ret                                     ; And return to C
_min    ENDP
```

What if you declare **min** as a **far** function—how will that change things? The major difference is that the stack on entry will now look like this:

```
SP + 06:  v2
SP + 04:  v1
SP + 02:  return segment
SP:       return offset
```

This means that the offsets into the stack have increased by two, since two extra bytes (for the return segment) had to be pushed onto the stack. Your far version of **min** would look like this:

```
        PUBLIC  _min
_min    PROC    far
        push    bp                              ; Save bp on stack
        mov     bp,sp                           ; Copy sp into bp
        mov     ax,[bp+6]                       ; Move v1 into ax
        cmp     ax,[bp+8]                       ; Compare with v2
        jle     exit                            ; If v1 > v2
        mov     ax,[bp+6]                       ; Then load ax with v2
exit:   pop     bp                              ; Restore bp
        ret                                     ; And return to C
_min    ENDP
```

Note that all the offsets for *v1* and *v2* increased by two, to reflect the additional bytes on the stack.

Now, what if (for whatever reason) you declare **min** as a **pascal** function; that is, you decide to use the Pascal parameter-passing sequences.

Your stack on entry will now look like this (assuming **min** is back to being a **near** function):

```
SP + 04:  v1
SP + 02:  v2
SP:       return addr
```

In addition, you will need to follow Pascal conventions for the identifier **min**: uppercase and no underscore.

Besides having swapped the locations of *v1* and *v2*, this convention also requires **min** to clean up the stack when it leaves, by specifying in the RET instruction how many bytes to pop off the stack. In this case, you have to pop off four additional bytes for *v1* and *v2* (the return address is popped off automatically by RET).

Here's what the modified routine looks like:

```
        PUBLIC  MIN
MIN     PROC    near                          ; Pascal version
        push    bp                            ; Save bp on stack
        mov     bp,sp                         ; Copy sp into bp
        mov     ax,[bp+6]                     ; Move v1 into ax
        cmp     ax,[bp+4]                     ; Compare with v2
        jle     exit                          ; If v1 > v2
        mov     ax,[bp+4]                     ; Then load ax with v2
exit:   pop     bp                            ; Restore bp
        ret     4                             ; Clear stack and return
MIN     ENDP
```

Here's one last example to show you why you might want to use the C parameter-passing sequence. Suppose you redefined **min** as follows:

```
int  extern min(int count, int v1, int v2,...);
```

min can now accept any number of integers and will return the minimum value of them all. However, since **min** has no way of automatically knowing how many values are being passed, make the first parameter a *count value*, indicating how many values follow it.

For example, you might use it as follows:

```
i = min(5, j, limit, indx, lcount, 0);
```

assuming *i, j, limit, indx,* and *lcount* are all of type **int** (or a compatible type). The stack upon entry will look like this:

```
SP + 08:  (etc.)
SP + 06:  v2
SP + 04:  v1
SP + 02:  count
SP:       return addr
```

The modified version of **min** now looks like this:

```
         PUBLIC  _min
_min     PROC    near
         push    bp                        ; Save bp on stack
         mov     bp,sp                     ; Copy sp into bp
         mov     ax,0                      ; Set ax = 0
         mov     cx,[bp+4]                 ; Move count into cx
         cmp     cx,ax                     ; Compare with 0
         jle     exit                      ; If <= 0, then exit
         mov     ax,[bp+4]                 ; Move first value into ax
         jmp     ltest                     ; And test loop
compare: cmp     ax,[bp+6]                 ; Compare with next value
         jle     ltest;                    If next value is lower
         mov     ax,[bp+6]                 ; Then load ax with next value
ltest:   add     bp,2                      ; Move to new value
         loop    compare                   ; Then loop back
exit:    pop     bp                        ; Restore bp
         ret                               ; And return to C
_min     ENDP
```

Note that this version correctly handles all possible values of *count*.

- If *count* <= 0, **min** returns 0.
- If *count* = 1, **min** returns the first value in the list.
- If *count* >= 2, **min** makes successive comparisons to find the lowest value in the parameter list.

Register Conventions

You used several registers (BP, SP, AX, BX, CX) in **min**; were you able to do so safely? What about any registers that your Turbo C program might be using?

As it turns out, you wrote this function correctly. Of those you used, the only register that you had to worry about was BP, and you saved that on the stack on entry, then restored it from the stack on exit.

The other two registers that you have to worry about are SI and DI; these are the registers Turbo C uses for any register variables. If you use them at all within an assembly language routine, then you should save them (probably on the stack) on entering the routine, and restore them on leaving. However, if you compile your Turbo C program with the `-r-` option (Use Register Variables...Off), then you don't have to worry about saving SI and DI.

Note: You must use caution if you use the `-r-` option. Refer to Appendix C in the *Turbo C Reference Guide* for details about this register variables option.

The registers CS, DS, SS, and ES may have corresponding values, depending on the memory model being used. Here are the relationships:

Tiny	CS = DS = SS; ES = scratch
Small, Medium	CS != DS, DS = SS; ES = scratch
Compact, Large	CS != DS != SS; ES = scratch (one CS per module)
Huge	CS != DS != SS; ES = scratch (one CS and one DS per module)

Calling C Functions from .ASM Routines

Yes, you can go the other way: You can call your C routines from within your assembly language modules. First, though, you have to make the C function visible to your assembly language module. We've already discussed briefly how to do this: Declare it as EXTRN, with either a **near** or a **far** modifier. For example, say you've written the following C function:

```
long docalc(int *fact1, int fact2, int fact3);
```

For simplicity, assume **docalc** is a C function (as opposed to Pascal). Assuming you're using the tiny, small, or compact memory model, you'd declare it as this in your assembly module:

```
EXTRN  _docalc:near
```

Likewise, if you were using the medium, large, or huge memory models, you'd declare it as `_docalc:far`.

docalc is to be called with three parameters:

- the address of a location named *xval*
- the value stored in a location named *imax*
- a third constant value of 421 (base 10)

You should also assume that you want to save the result in a 32-bit location named *ans*. The equivalent call in C would then be

```
ans = docalc(&xval,imax,421);
```

You'll need to push 421 on the stack first, then *imax*, then the address of *xval*, and then call **docalc**. When it returns, you'll need to clean up the stack, which will have six extra bytes on it, and then move the answer into *ans* and *ans*+2.

Here's what your code will look like:

```
mov   ax,421          ; Get 421, push onto stack
push  ax
push  imax            ; Get imax, push onto stack
lea   ax,xval         ; Get &xval, push onto stack
push  ax
call  _docalc         ; Call docalc
add   sp,6            ; Clean up stack
mov   ans,ax          ; Move 32-bit result into ans
mov   ans+2,dx        ; Including high-order word
```

What if **docalc** used the Pascal parameter-passing sequence instead? Then you would have to reverse the order of the parameters, and you wouldn't have to worry about cleaning up the stack upon return, since the routine would (should) have done that for you. Also, you would need to spell **docalc** in the assembly source using Pascal conventions (uppercase and no underscore).

The EXTRN statement is then

```
EXTRN  DOCALC:near
```

and the code to call **docalc** is

```
lea   ax,xval         ; Get &xval, push onto stack
push  ax
push  imax            ; Get imax, push onto stack
mov   ax,421          ; Get 421, push onto stack
push  ax
call  DOCALC          ; Call docalc
mov   ans,ax          ; Move 32-bit result into ans
mov   ans+2,dx        ; Including high-order word
```

That's all you need to know to get started interfacing other languages with Turbo C.

Low-Level Programming: Pseudo-Variables, Inline Assembly, and Interrupt Functions

What if you want to do some low-level work, but don't want to go to all the trouble of setting up a separate assembly language module? Turbo C still has the answer for you—three answers, in fact: pseudo-variables, inline assembly, and interrupt functions. Take a look at the rest of this chapter to see how each of these can help you get the job done.

Pseudo-Variables

The CPU in your system (the 8088/8086/80186/80286 processor) has a number of registers, or special storage areas, that it uses to manipulate values. Each register is 16 bits (2 bytes) long; most of them have some special purpose, though several can be used for general purposes as well. See "Memory Models" at the beginning of this chapter for specific details on these CPU registers.

Sometimes in low-level programming, you may want to directly access these registers from your C program.

- You might want to load values into them before calling a system routine.
- You might want to see what values they currently hold.

Turbo C makes it very easy for you to access these registers through *pseudo-variables*. A pseudo-variable is simply an identifier that corresponds to a given register: You can use it as if it were a variable of type **unsigned int** or **unsigned char**.

Table 12.6 shows a complete list of the pseudo-variables you can use, their types, the registers they correspond to, and what those registers are usually used for.

Table 12.6: Turbo C Pseudo-Variables

Pseudo-variable	Type	Register	Purpose
_AX	unsigned int	AX	General/accumulator
_AL	unsigned char	AL	Lower byte of AX
_AH	unsigned char	AH	Upper byte of AX
_BX	unsigned int	BX	General/indexing
_BL	unsigned char	BL	Lower byte of BX
_BH	unsigned char	BH	Upper byte of BX
_CX	unsigned int	CX	General/counting and loops
_CL	unsigned char	CL	Lower byte of CX
_CH	unsigned char	CH	Upper byte of CX
_DX	unsigned int	DX	General/holding data
_DL	unsigned char	DL	Lower byte of DX
_DH	unsigned char	DH	Upper byte of DX
_CS	unsigned int	CS	Code segment address
_DS	unsigned int	DS	Data segment address
_SS	unsigned int	SS	Stack segment address
_ES	unsigned int	ES	Extra segment address
_SP	unsigned int	SP	Stack pointer (offset to SS)
_BP	unsigned int	BP	Base pointer (offset to SS)
_DI	unsigned int	DI	Used for register variables
_SI	unsigned int	SI	Used for register variables

Why would you even want to directly access these variables from Turbo C?

You might need to set registers to certain values before calling low-level system routines. For example, you can call certain routines in your computer's ROM by executing the `INT (interrupt)` instruction, but first you have to put the necessary information into certain registers, like this:

```
void readchar(unsigned char page, unsigned char *ch, unsigned char *attr);
{
   _AH = 8;                              /* Service code: read char, attribute */
   _BH = page;                                  /* Specify which display page */
   geninterrupt(0x10)                               /* Call INT 10h services */
   *ch = _AL;                              /* Get ASCII code of character read */
   *attr = _AH;                             /* Get attribute of character read */
}
```

As you can see, the service code and the display page number are both being passed to the `INT 10h` routine; the values returned are copied over into *ch* and *attr*.

The pseudo-variables can be treated just as if they were regular global variables of the appropriate type (**unsigned int**, **unsigned char**). However, since they refer to the CPU's registers, rather than some arbitrary location in memory, there are some restrictions and concerns you must be aware of.

- You cannot use the address-of operator (&) with a pseudo-variable, since a pseudo-variable has no address.
- Since the compiler is constantly generating code that uses the registers (after all, that's what most of the 8086's instructions do), you have absolutely no guarantee that values you place in pseudo-variables will be preserved for any length of time.

  This means you must assign values right before using them and read values right after obtaining them, as in **readchar** (previous example). This is especially true of the general-purpose registers (AX, AH, AL, and so on), since the compiler freely uses these for temporary storage. On top of that, the CPU changes them in ways you might not expect; for example, using CX when it sets up a loop or does a shift operation, or using DX to hold the upper word of a 16-bit multiply.
- You can't rely on pseudovariables' values remaining the same across a function call. As an example of this, take the following code fragment:

  ```
  _CX = 18;
  myFunc();
  i = _CX;
  ```

  Not all registers are saved during a function call, so you have no guarantee that *i* will get assigned a value of 18. The only registers that you can count on having the same values before and after a function call are _CS, _BP, _SI, and _DI.
- You need to be very careful modifying certain registers, since this could have unexpected and untoward effects. For example, directly storing values to _CS, _SS, _SP, or _BP can (and almost certainly will) cause your program to behave erratically, since the machine code produced by the Turbo C compiler uses those registers in various ways.

Using Inline Assembly Language

You've already seen how to write separate assembly language routines and link them in to your Turbo C program. But Turbo C also lets you write assembly language code right inside your C program. This is known as *inline assembly*.

To use inline assembly in your C program, you can use the -B compiler option. If you don't, and the compiler encounters inline assembly, it (the compiler) will issue a warning and restart itself with the -B option. You can avoid this with the #pragma inline statement in your source, which in effect enables the -B option for you when the compiler encounters it.

You must have a copy of Turbo Assembler (TASM). The compiler first generates an assembly file, and then invokes TASM on that file to produce the .OBJ file.

Of course, you also need to be familiar with the 8086 instruction set and architecture. While you're not writing complete assembly language routines, you still need to know how the instructions you're using work, how to use them, and how not to use them.

Having done all that, you need only use the keyword **asm** to introduce an inline assembly language instruction. The format is

```
asm  <opcode>  <operands> <; or newline>
```

where

- *<opcode>* is a valid 8086 instruction (several tables of allowable *opcodes* will follow).
- *<operands>* contains the operand(s) acceptable to the *<opcode>*, and can reference C constants, variables, and labels.
- < ; or *newline>* is a semicolon or a newline, either of which signals the end of the **asm** statement.

A new **asm** statement may be placed on the same line, following a semicolon, but no **asm** statement can continue to the next line.

Semicolons may not be used to start comments (as they may in TASM). When commenting **asm** statements, use C-style comments, like this:

```
asm mov ax,ds;                                /* This comment is OK */
asm pop ax; asm pop ds; asm iret;              /* This is legal too */
asm push ds                                  ;THIS COMMENT IS INVALID!!
```

Note that the last line will generate an error, since (as it declares) the comment there is invalid.

The *<opcode>* *<operand>* pair is copied straight to the output, embedded in the assembly language that Turbo C is generating from your C instructions. Any C symbols are replaced with appropriate assembly language equivalents.

The inline assembly facility is not a complete assembler, so many errors will not be immediately detected. TASM will catch whatever errors there might

be. However, TASM might not identify the location of errors, particularly since the original C source line number is lost.

Each **asm** statement counts as a C statement. For example,

```
myfunc()
{
   int  i;
   int x;

   if  (i > 0)
      asm  mov  x,4
   else
      i = 7;
}
```

This construct is a valid C **if** statement. Note that no semicolon was needed after the `mov x,4` instruction. **asm** statements are the only statements in C that depend on the occurrence of a newline. This is not in keeping with the rest of the C language, but this is the convention adopted by several UNIX-based compilers.

An assembly statement may be used as an executable statement inside a function, or as an external declaration outside of a function. Assembly statements located outside any function are placed in the DATA segment, and assembly statements located inside functions are placed in the CODE segment.

Here is an inline assembly version of the function **min** (introduced in "Handling Return Values" earlier in this chapter).

```
int min (int V1, int V2)
{
   asm  mov  ax,V1
   asm  cmp  ax,V2
   asm  jle  minexit
   asm  mov  ax,V2
   minexit:
   return (_AX);
}
```

This example demonstrates why using inline assembly with Turbo C is more versatile and powerful than calling .ASM routines. This one inline assembly example works for modules compiled with large code, small code, Pascal calling convention, or C calling convention.

The .ASM equivalent must always be changed, depending on the memory model and the calling convention (C or Pascal). In the .ASM equivalent of

min, you must always account for parameter offsets and the spelling of the identifier (*_min* or *MIN*); not so with this inline assembly version.

Note: There is a new feature called **__emit__** that allows Turbo Pascal style inline coding. For more information on **__emit__**, see the entry in Chapter 2 of the *Turbo C Reference Guide*.

Any of the 8086 instruction opcodes may be included as inline assembly statements. There are four classes of instructions allowed by the Turbo C compiler:

- normal instructions—the regular 8086 opcode set
- string instructions—special string-handling codes
- jump instructions—various jump opcodes
- assembly directives—data allocation and definition

Note that all operands are allowed by the compiler, even if they are erroneous or disallowed by the assembler. The exact format of the operands is not enforced by the compiler.

Opcodes

The following is a summary list of the opcode mnemonics that may be used as normal instructions:

Table 12.7: Opcode Mnemonics

aaa	fcom	fldl2t	fsub	or
aad	fcomp	fldlg2	fsubp	out
aam	fcompp	fldln2	fsubr	pop
aas	fdecstp**	fldpi	fsubrp	popa
adc	fdisi	fldz	ftst	popf
add	fdiv	fmul	fwait	push
and	fdivp	fmulp	fxam	pusha
bound	fdivr	fnclex	fxch	pushf
call	fdivrp	fndisi	fxtract	rcl
cbw	feni	fneni	fyl2x	rcr
clc	ffree**	fninit	fyl2xp1	ret
cld	fiadd	fnop	hlt	rol
cli	ficom	fnsave	idiv	ror
cmc	ficomp	fnstcw	imul	sahf
cmp	fidiv	fnstenv	in	sal
cwd	fidivr	fnstsw	inc	sar
daa	fild	fpatan	int	sbb
das	fimul	fprem	into	shl
dec	fincstp**	fptan	iret	shr
div	finit	frndint	lahf	stc
enter	fist	frstor	lds	std
f2xm1	fistp	fsave	lea	sti
fabs	fisub	fscale	leave	sub
fadd	fisubr	fsqrt	les	test
faddp	fld	fst	mov	wait
fbld	fld1	fstcw	mul	xchg
fbstp	fldcw	fstenv	neg	xlat
fchs	fldenv	fstp	not	xor
fclex	fldl2e	fstsw		

Note: When using 80186 instruction mnemonics in your inline assembly statements, you must include the `-1` command-line option. This forces appropriate statements into the assembly language compiler output so that Turbo Assembler will expect the mnemonics. Also, if you are using an older assembler, these mnemonics may not be supported at all.

Another Note: If you are using inline assembly in routines that use floating-point emulation (the TCC option `-f`), the opcodes marked with ** are not supported.

String Instructions

In addition to the listed opcodes, string instructions given in the following table may be used alone or with repeat prefixes.

Table 12.8: String Instructions

cmps	insw	movsb	outsb	scasw
cmpsb	lods	movsw	outsw	stos
cmpsw	lodsb	msb	scas	stosb
ins	lodsw	outs	scasb	stosw
insb	movs			

Repeat Prefixes

The following repeat prefixes may be used:

```
rep        repe        repne        repnz        repz
```

Jump Instructions

Jump instructions are treated specially. Since a label cannot be included on the instruction itself, jumps must go to C labels (discussed in "Using Jump Instructions and Labels"). The allowed jump instructions are given in Table 12.9:

Table 12.9: Jump Instructions

ja	jge	jnc	jnp	js
jae	jl	jne	jns	jz
jb	jle	jng	jnz	loop
jbe	jmp	jnge	jo	loope
jc	jna	jnl	jp	loopne
jcxz	jnae	jnle	jpe	loopnz
je	jnb	jno	jpo	loopz
jg	jnbe			

Assembly Directives

The following assembly directives are allowed in Turbo C inline assembly statements:

```
db        dd        dw        extrn
```

Inline Assembly References to Data and Functions

You can use C symbols in your **asm** statements; Turbo C will automatically convert them to appropriate assembly language operands and will tack underscores onto identifier names. Any symbol can be used, including automatic (local) variables, register variables, and function parameters.

In general, a C symbol can be used in any position where an address operand would be legal. Of course, a register variable can be used wherever a register would be a legal operand.

If the assembler encounters an identifier while parsing the operands of an inline assembly instruction, it searches for the identifier in the C symbol table. The names of the 8086 registers are excluded from this search. Either uppercase or lowercase forms of the register names may be used.

Inline Assembly and Register Variables

The two most frequently used register declarations in a function are treated as register variables, and all other register declarations are treated as automatic (local) variables. If the keyword **register** occurs in a declaration that cannot be a register, the keyword is ignored.

Only **short**, **int** (or the corresponding **unsigned** types), or 2-byte pointer variables may be placed in a register. SI and DI are the 8086 registers used for register variables. Inline assembly code may freely use SI or DI as scratch registers if no register declarations are given in the function. The C function entry and exit code automatically saves and restores the caller's SI and DI.

If there is a register declaration in a function, inline assembly may use or change the value of the register variable by using SI or DI, but the preferred method is to use the C symbol in case the internal implementation of register variables ever changes.

Inline Assembly, Offsets, and Size Overrides

When programming, you don't need to be concerned with the exact offsets of local variables. Simply using the name will include the correct offsets.

However, it may be necessary to include appropriate WORD PTR, BYTE PTR, or other size overrides on assembly instruction. A DWORD PTR override is needed on LES or indirect far call instructions.

Using C Structure Members

You can, of course, reference structure members in an inline assembly statement in the usual fashion, that is, *<variable>.<member>*. In such a case, you are dealing with a variable, and you can store or retrieve values. However, you can also directly reference the member name (without the

variable name) as a form of numeric constant. In this situation, the constant equals the offset (in bytes) from the start of the structure containing that member. Consider the following program fragment:

```
struct myStruct {
   int a_a;
   int a_b;
   int a_c;
} myA ;

myfunc()
{
   ...
   asm  mov  ax, myA.a_b
   asm  mov  bx, [di].a_c
   ...
}
```

We've declared a structure type named *myStruct* with three members, *a_a*, *a_b*, and *a_c*; we've also declared a variable *myA* of type *myStruct*. The first inline assembly statement moves the value contained in *myA.a_b* into the register AX. The second moves the value at the address `[di]+offset(a_c)` into the register BX (it takes the address stored in DI and adds to it the offset of *a_c* from the start of *myStruct*). In this sequence, these assembler statements produce the following code:

```
mov  ax, DGROUP : myA+2
mov  bx, [di+4]
```

Why would you even want to do this? If you load a register (such as DI) with the address of a structure of type *myStruct*, then you can use the member names to directly reference the members. The member name actually may be used in any position where a numeric constant is allowed in an assembly statement operand.

The structure member must be preceded by a dot (.) to signal that a member name, rather than a normal C symbol, is being used. Member names are replaced in the assembly output by the numeric offset of the structure member (the numeric offset of *a_c* is 4), but no type information is retained. Thus members may be used as compile-time constants in assembly statements.

However, there is one restriction. If two structures that you are using in inline assembly have the same member name, you must insert between the dot and the member name the structure type in parentheses, as if it were a cast. For example:

```
asm   mov    bx,[di].(struct tm)tm_hour
```

Using Jump Instructions and Labels

You may use any of the conditional and unconditional jump instructions, plus the loop instructions, in inline assembly. They are only valid inside a function. Since no labels can be given in the **asm** statements, jump instructions must use C **goto** labels as the object of the jump. Direct far jumps cannot be generated.

Indirect jumps are also allowed. To use an indirect jump, you can use a register name as the operand of the jump instruction. In the following code, the jump goes to the C **goto** label *a*.

```
int     x()
{
a:                                          /* This is the goto label "a" */
   ...
   asm  jmp  a                                   /* Goes to label "a" */
   ...
}
```

Interrupt Functions

The 8086 reserves the first 1024 bytes of memory for a set of 256 far pointers—known as interrupt vectors—to special system routines known as *interrupt handlers*. These routines are called by executing the 8086 instruction

```
int  <int#>
```

where *<int#>* goes from 0h to FFh. When this happens, the computer saves the code segment (CS), instruction pointer (IP), and status flags, disables the interrupts, then does a far jump to the location pointed to by the corresponding interrupt vector. For example, one interrupt call you're likely to see is

```
int  21h
```

which calls most DOS routines. But many of the interrupt vectors are unused, which means, of course, that you can write your own interrupt handler and stick a **far** pointer to it into one of the unused interrupt vectors.

To write an interrupt handler in Turbo C, you must define the function to be of type **interrupt**; more specifically, it should look like this:

```
void  interrupt myhandler(bp, di, si, ds, es, dx,
                          cx, bx, ax, ip, cs, flags, ... );
```

As you can see, all the registers are passed as parameters, so you can use and modify them in your code without using the pseudo-variables discussed earlier in this chapter. Also note that you can have additional parameters (*flags, ...*) passed to the handler; those should be defined appropriately.

A function of type **interrupt** will automatically save (in addition to SI, DI, and BP) the registers AX through DX, ES, and DS. These same registers are restored on exit from the interrupt handler.

Interrupt handlers may use floating-point arithmetic in all memory models. Any interrupt handler code that uses an 8087/80287 must save the state of the chip and restore it on exit from the handler.

An interrupt function may modify its parameters. Changing the declared parameters will modify the corresponding register when the interrupt handler returns. This may be useful when you are using an interrupt handler to act as a user service, much like the DOS INT 21 services. Also, note that an interrupt function exits with an IRET (return from interrupt) instruction.

So, why would you want to write your own interrupt handler? For one thing, that's how most memory-resident routines work. They install themselves as interrupt handlers. That way, whenever some special or periodic action takes place (clock tick, keyboard press, and so on), these routines can intercept the call to the routine handling the interrupt and see what action needs to take place. Having done that, they can then pass control on to the routine that was there.

Using Low-Level Practices

You've already seen a few examples of how to use these different low-level practices in your code; now it's time to look at a few more. For starters, you will write an actual interrupt handler that does something harmless yet visible (or, in this case, audible): It will beep whenever it's called.

First, you need to write the function itself. Here's what it would look like:

```
#include        <dos.h>

void  interrupt  mybeep(unsigned bp, unsigned di, unsigned si,
                        unsigned ds, unsigned es, unsigned dx,
                        unsigned cx, unsigned bx, unsigned ax)
{
   int      i, j;
   char     originalbits, bits;
   unsigned char   bcount = ax >> 8;

   /* Get the current control port setting */
   bits = originalbits = inportb(0x61);

   for (i = 0; i <= bcount; i++){

      /* Turn off the speaker for awhile */
      outportb(0x61, bits & 0xfc);
      for      (j = 0; j <= 100; j++)
         ; /* empty statement */

      /* Now turn it on for some more time */
      outportb(0x61, bits | 2);
      for      (j = 0; j <= 100; j++)
         ; /* another empty statement */
      }

   /* Restore the control port setting */
   outportb(0x61, originalbits);
}
```

Next, you need to write a function to install your interrupt handler. You will pass it the address of the function and its interrupt number (0...255 or 0x00...0xFF). The function must do three things:

- Disable interrupts so that nothing funny happens while it is updating the vector table
- Store the function address passed into the appropriate location
- Enable interrupts so that everything is working fine again

Here's what your installation routine looks like:

```
void  install(void interrupt (*faddr)(), int inum)
{
   setvect(inum, faddr);
}
```

Finally, you will want to call your beep routine to test it out. Here's a function to do just that:

```
void  testbeep(unsigned char bcount, int inum)
{
   _AH = bcount;
   geninterrupt(inum);
}
```

Your **main** function will look like this:

```
main()
{
   char  ch;

   install(mybeep,10);
   testbeep(3,10);
   ch = getch();
}
```

Using Floating-Point Libraries

There are two types of numbers you work with in C: integer (**int**, **short**, **long**, etc.) and floating point (**float**, **double**). Your computer's processor is set up to easily handle integer values, but it takes more time and effort to handle floating-point values.

However, the iAPx86 family of processors has a corresponding family of math coprocessors, the 8087 and the 80287.

The 8087 and 80287 (both of which we refer to here as "the coprocessor") are special hardware numeric processors that can be installed in your PC. They execute floating-point instructions very quickly. If you use floating point a lot, you'll probably want a coprocessor. The CPU in your computer interfaces to the 8087/80287 via special interrupts.

Turbo C is designed to help you adapt your program to your computer and to your needs.

- If you don't need to use floating-point values at all, you can tell the compiler that.
- If you do need to use floating-point values but your computer doesn't have a math coprocessor (8087/80287), you can tell Turbo C to link in special routines to make it look as though you do have one. In that case, if your program is run on a system with a coprocessor, the chip will be used automatically, and your program runs much faster.
- If you're writing programs only for systems that have a math coprocessor, you can instruct the Turbo C compiler to produce code that always uses the 8087/80287 chip.

The following TCC and TLINK examples assume that the TURBOC.CFG file exists with the correct `-L` and `-I` paths set, and that the library and start-up object files are stored in a subdirectory named \LIB.

Emulating the 8087/80287 Chip

What if you want to use floating point, but your computer doesn't have a math coprocessor? Or what if you have to write a program for computers that might or might not have one? Relax; Turbo C handles that situation well.

With the emulation option, the compiler will generate code as if the 8087/80287 were present, but will link in the emulation library (EMU.LIB). When the program runs, it will use the 8087/80287 if it is present; if no coprocessor is present at run time, the program will use special software that *emulates* the 8087/80287.

The emulation library works like this:

- When your program starts to run, the C start-up code will determine if an 8087/80287 is present or not.
- If the coprocessor is there, the program will allow the special interrupts for the 8087/80287 to be passed straight through to the 8087/80287 chip.
- If the coprocessor is not there, the program causes the interrupts to be intercepted and diverted to the emulation routines.

Suppose you modify RATIO.C to look like this:

```
main()
{
   float   a,b,ratio;

   printf("Enter two values:  ");
   scanf("%f %f",&a,&b);
   ratio = a/b;
   printf("The ratio is %0.2f\n",ratio);
}
```

If you are using TC (the integrated environment), you need to go to the Options menu, choose Compiler, choose Code Generation, then toggle the Floating-point item until the field following it reads *Emulation*. When you compile and link your program, Turbo C will automatically select the proper options and libraries for you.

If you're using TCC (the stand-alone compiler), your command line should look like this:

```
tcc -mX ratio
```

If you link the resulting code manually, you must specify both the appropriate math library (depending on the model size) and the EMU.LIB file. The emulation option (`-f`) is on by default, so you don't need to give it unless your TURBOC.CFG file contains one of the other floating-point switches (`-f-` or `-f87`).

Your invocation of TLINK should look like this:

```
tlink lib\c0X ratio, ratio, ratio, lib\emu.lib
      lib\mathX.lib lib\cX.lib
```

where *X* is a letter indicating the proper model library.

Note: The `tlink` command is given all on one line.

Also remember that *the order of the libraries is very important.*

Using the 8087/80287 Math Coprocessor Chip

If you are absolutely sure your program will be run only on systems that have an 8087 or 80287 chip, you can create programs that will take advantage of that chip. At the same time, your resulting .EXE files will be smaller, since Turbo C won't have to include the 8087/80287 emulation routines (EMU.LIB).

If you are using TC (the integrated environment), you need to go to the **O**ptions menu, choose **C**ompiler, choose **C**ode Generation, then toggle the **F**loating-point item until the field following it says *8087/80287*. When you compile and link your program, Turbo C will automatically select the proper options and libraries for you.

If you're using TCC (the stand-alone compiler), you need to use the `-f87` option on your command line, like this:

```
tcc -f87 -mX ratio
```

This tells Turbo C to generate inline calls to the 8087/80287 chip. When TLINK is invoked, the files FP87.LIB and MATH*x*.LIB are linked in.

If you manually link the resulting code, you must specify both the appropriate math library (depending on the model size) and the FP87 library, like this:

```
tlink lib\c0X ratio, ratio, ratio, lib\fp87.lib
      lib\mathX.lib lib\cX.lib
```

where, as always, *X* is a letter indicating the proper model library.

If You Don't Use Floating Point...

If your program doesn't use any floating-point routines, the linker will not link in any of the floating-point libraries (EMU.LIB or FP87.LIB, along with MATH*x*.LIB) at link time, even if you listed them on the command line. You can optimize the link step by omitting these libraries from the linker command line (if, as we said, your program uses no floating point).

Suppose you want to compile and link the following program (saved as RATIO.C):

```
main()
{
   int    a,b,ratio;

   printf("Enter two values:  ");
   scanf("%d %d",&a,&b);
   ratio = a/b;
   printf("The ratio is %d\n",ratio);
}
```

Since this program uses no floating-point routines, you can choose to compile it with floating-point emulation on, or with no floating point at all.

If you are using TC (the integrated environment) and choose to compile with emulation on, just choose **C**ompile to OBJ from the **C**ompile menu. (Emulation *On* is the default.) The linker will include the floating-point libraries at the link step, but none will actually be linked.

If you want to speed up the linking process, you can specify "no floating point." Go to the **O**ptions menu, choose **C**ompiler, choose **C**ode Generation, then choose the **F**loating Point toggle.

Repeatedly pressing *Enter* at this command cycles you through three options: None, Emulation, and 8087/80287. You want the None option. You can then press *Esc* three times to get back to the menu bar (or just press *F10*).

When you compile and link this program with **F**loating point set to None, Turbo C does not attempt to link in any floating-point math routines.

If you're using TCC (the stand-alone compiler), you need to use the `-f-` option on your command line, like this:

```
tcc -f- -mX ratio.c
```

This tells Turbo C that you have no floating-point instructions in your program. It also says that you used the *x* memory model, where *x* is a letter indicating the desired model (t = tiny, s = small, c = compact, m = medium, l = large, h = huge).

Since RATIO.C is a stand-alone program, TCC will automatically invoke TLINK, linking in C0*x*.OBJ and C*x*.LIB, and producing RATIO.EXE.

If you used the "compile only" (-c) option on the TCC command line, you need to manually link the resulting code. In this situation, you don't need to (and shouldn't) specify any math library; your invocation of TLINK should look like this:

```
tlink lib\c0x ratio, ratio, ratio, lib\cx.lib
```

This links together C0*x*.OBJ and RATIO.OBJ, uses the library C*x*.LIB, and produces the files RATIO.EXE and RATIO.MAP.

The 87 Environment Variable

If you build your program with 8087/80287 emulation (in other words, you choose Floating point...Emulation from the menus or you include the -f option on the TCC command line), the C0*x*.OBJ start-up module will use 8087/80287 auto-detection logic when you run the program. This means that the start-up code will automatically check to see if an 8087/80287 is available.

If the 8087/80287 is available, then the program will use it; if it is not there, the program will use the emulation routines.

There are some instances in which you might want to override this default auto-detection behavior. For example, your own run-time system might have an 8087/80287, but you need to verify that your program will work as intended on systems without a coprocessor. Or your program may need to run on a PC-compatible system, but that particular system returns incorrect information to the auto-detection logic (saying that a nonexistent 8087/80287 is available, or vice versa).

Turbo C provides an option for overriding the start-up code's default auto-detection logic; this option is the 87 environment variable.

You set the 87 environment variable at the DOS prompt with the SET command, like this:

```
C> SET 87=N
```

or like this:

```
C> SET 87=Y
```

Setting the 87 environment variable to N (for No) tells the start-up code that you do not want to use the 8087/80287 (even though it might be present in the system).

Conversely, setting the 87 environment variable to Y (for Yes) means that the coprocessor is there, and you want the program to use it. Let the Programmer beware!! If you set 87 = Y when, in fact, there is no 8087/80287 available on that system, your program will crash and burn in a logical inferno.

The 87 environment variable is able to override the default auto-detection logic because, when you start to run your program, the start-up code first checks to see if the 87 variable has been defined.

- If the 87 variable has been defined, the start-up code looks no further, and your program runs in the prescribed mode.
- If the 87 variable has not been defined, the start-up code goes through its auto-detection logic to see if an 8087/80287 chip is available, and the program runs accordingly.

If the 87 environment variable has been defined (to any value) but you want to undefine it, enter the following at the DOS prompt:

```
C> SET 87=
```

(That is, press *Enter* immediately after typing the equal sign.)

Registers and the 8087/80287

There are a couple of points concerning registers that you should be aware of when using floating point.

First, in 8087/80287 emulation mode, register wraparound is not supported.

Second, if you are mixing floating point with inline assembly, you may need to take special care when using registers. This is because the 8087/80287 register set is emptied before Turbo C calls a function. You might need to pop and save the 8087/80287 registers before calling functions that use the coprocessor, unless you are sure that enough free registers exist.

Using matherr with Floating Point

When an error is detected in one of the floating-point routines during execution of a program, that routine automatically calls **_matherr** with several arguments. **_matherr** then stuffs an exception structure (defined in math.h) with its arguments and calls **matherr** with a pointer to that structure.

The **matherr** routine is a hook that you can use to write your own error-resolution routine. By default, **matherr** does nothing but return 0. However, you can modify **matherr** to deal with floating-point routine errors in any way you desire. Such a modified **matherr** then returns nonzero if the error was resolved, or 0 if it was not.

For more information about **matherr** and **_matherr** refer to the **matherr** description in Chapter 2 of the *Turbo C Reference Guide*.

Caveats and Tips

Turbo C's Use of RAM

Turbo C does not generate any intermediate data structures to disk when it is compiling (Turbo C writes only .OBJ files to disk); instead it uses RAM for intermediate data structures between passes. Because of this, you might encounter the message `OUT OF MEMORY...` if there is not enough memory available for the compiler.

The solution to this problem is to make your functions smaller, or to split up the file that has large functions. You might also delete any RAM-resident programs you have installed to free up more memory for Turbo C to use.

Should You Use Pascal Conventions?

No—not unless you have read and really understood this chapter.

Remember, if you are compiling your main file with Pascal calling conventions, make sure to declare **main** as a C function:

```
cdecl main(int argc, char * argv[], char * envp[])
```

Summary

You've seen how to use all three aspects of low-level programming in Turbo C (pseudo-variables, inline assembly, interrupt functions); you've learned about interfacing with other languages, including assembly; you've been introduced to some of the details of using floating-point routines; and you've discovered how the different memory models on the 8086 interact. Now it's up to you to use these techniques to gain complete control of your computer; best of luck.

Bibliography

Alonso, Robert. *Turbo C DOS Utilities*, John Wiley and Sons.

Bernap, Steve. *Turbo C for Beginners*, COMPUTE! Publications.

Davis, Stephen R. *Turbo C: The Art of Program Design, Optimization and Debugging*, M&T Publishing.

Derman, Bonnie (editor) and Strawberry Software. *Complete Turbo C*, Scott, Foresman & Co.

Edmead, Mark. *Illustrated Turbo C*, WordWare Publishing.

Harbison, Samuel P., and Guy L. Steel. *C: A Reference Manual*, Prentice-Hall.

Kelly-Bootle, Stan. *Mastering Turbo C*, Sybex.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language* (First Edition), Prentice-Hall.

LaFore, Robert. *Turbo C Programming for the IBM*, Howard W. Sams & Co.

Porter, Kent. *Stretching Turbo C*, Brady Books.

Schildt, Herbert. *Advanced Turbo C*, Osborne/McGraw-Hill.

Schildt, Herbert. *Turbo C: The Complete Reference*, Osborne/McGraw-Hill.

Schildt, Herbert. *Using Turbo C*, Osborne/McGraw-Hill.

Stevens, Al. *Turbo C: Memory Resident Utilities, Screen Input and Output*, MIS: Inc.

Young, Michael. *Systems Programming with Turbo C*, Sybex.

Zimmerman, Beverly and Scott. *Programming with Turbo C*, Scott, Foresman & Co.

Index

8086 341
  address segment:offset notation 345
  registers 342-344
8087 390
80287 390
8087/80287
  auto-detection of 394
  emulation 118
  floating-point emulation 360
  inline code 118
  math coprocessor 360
8088/8086 instruction set 117
87 environment variable 394
!= operator 172, 347
&& operator 173
++ operator 166
-- operator 166
<< operator 167
<= operator 172
== operator 172, 347
>= operator 172
>> operator 167
?: operator 206
| | operator 173
! operator 173
% operator 166
& operator 167, 169, 189
* operator 166, 169, 189
+ operator 166
- operator 166
/ operator 166
< operator 172
= operator 166, 174
> operator 172
^ operator 167
| operator 167
~ operator 167
# symbol 334
80x86 instruction set 117
–A compiler option 325
.ASM source files 38
–B compiler option 337
.COM files 348
  Pascal 285
#define (preprocessing directive) 211
.EXE files
  building 19
  making 109
  named from project file 30
  naming by Project-Make 110
_fmode (global variable 208
–I compiler option 40, 335
–K compiler option 314, 317
–L compiler option 40
.OBJ files 111
  compiling to 109
  naming by Project-Make 109
–P compiler option 326
#pragma inline statement 380

A

active window 222
adapters, video 221
  graphics, compatible with Turbo C 233
Add Watch command 66, 73, 100, 140
addition operator (+) 166
address, mailing, Borland 7
address, passing by 266
address-of operator 257, 266
address-of operator (&) 169, 189
address operators 169
addresses 188
  calculation 343, 344-345
  passing 170, 192
  space, pointers and 324
advanced programming 341-397
_AH pseudo-variable 378
AH register 378
_AL pseudo-variable 378
AL register 378
alignment, structures 324
Alignment toggle 118
alloc_gstack (Prolog function) 295
anachronisms 340
AND operator (&) 167
ANSI C standard 3, 5, 311
  violations 124
ANSI Keywords Only toggle 122
argc (identifier) 285
arguments
  command-line 38, 285

to function main 105, 133
Arguments setting 105, 133
argv (identifier) 285
arithmetic, pointer 192
arithmetic conversions 318
arithmetic operations
pointers and 324
arrays 194, 270
and pointers 195
character 163, 171
declaration 194
multi-dimensional 196, 273, 290
passing 198
passing 197
pointers vs. 290
Turbo C vs. Pascal 272
artificial intelligence 293
asm (keyword) 380
aspect ratio 237
Assembler, Turbo 380
assembly code
inline 379
assembling 38
C structure members in 385
calling functions 384
debugging with integrated debugger 77
floating point in 395
goto in 387
jump instructions in 387
referencing data in 384
register variables in 385
restriction on structure member names 386
size overrides in 385
variable offsets in 385
interfacing with 328, 366-376
layout of source files 366
routines 366
calling C functions from 369, 375
example of 370
passing parameters to 371
referencing C data from 369
referencing C functions from 369
register conventions in 374
return values in 371
template 367
assignment 288
operator 255, 288
stacked 166
statements, value of 174
assignment operator (=) 166, 174
attributes
cell 222, 229
blink 230
colors 229
control functions 226
screen, controlling 226
Auto Dependencies toggle 113
autodependency checking 35, 113
Autoindent mode 149
Autoindent toggle 149
auxiliary port 209
_AX pseudo-variable 378
AX register 343, 371, 378

B

–B compiler option 380
background
color 226, 229, 242
setting 226
Backup (source) Files toggle 129
backup files, automatically created 129
backward pair matching 152
bar chart (example program) 308
_BH pseudo-variable 378
BH register 378
binary mode 208
binary operators 166
binary streams 208
BIOS, calls to 230
bit 188
bit-mapped fonts 239
bitfields, in structures 325
bitwise operators 167
_BL pseudo-variable 378
BL register 378
blink enable bit 226
block statements 258
Boolean data type 255
Borland
CompuServe Forum 7

license statement 6, 9
mailing address 7
technical support 7
bottom-up debugging 77
boundary conditions 75
_BP pseudo-variable 378
BP register 343, 371, 378
break (keyword) 202, 203, 204
Break Make On menu 32, 112
break statement 260
Break/Watch menu 91, 139
breakpoints 46, 98, 139
cancelled 65
deleting 141, 142
inserting 141
lost track of by TC 65, 142
moving cursor to 142
setting 51
sticking, from one debug session to another 64
buffered streams 208
Build All command 110
builds, vs makes 110
BUILTINS.MAK 42
_BX pseudo-variable 378
BX register 343, 378
byte 188
alignment 118
BYTE (assembler) 370

C

C calling sequence 117
C Reference Manual 311
C0*x*.OBJ 360
call stack 70, 137
displaying executing line of function from 71
returning to execution bar from 71
Call Stack command 71, 73, 100, 137
Calling Convention toggle 117, 323
calling sequences
C 117
Pascal 117
calloc (function), Turbo Prolog and 295
case (keyword) 202, 259
Case-sensitive Link toggle 127
case sensitivity 165, 280, 312
in Turbo Assembler 368
linking with no 365
case statement 259
cdecl (keyword) 323, 327, 365
CDECL (macro) 339
cells
attributes 222, 229
blink 230
colors 229
characters in 222
screen 222
CGA, color control on
high resolution 244
low resolution 242
_CH pseudo-variable 378
CH register 378
Change Dir command 17, 104
char (keyword) 164, 271, 317
char declarations, signed vs unsigned 118
character codes, hexadecimal 314
character constants, hexadecimal 213, 314
character pointer 164
characters 162
array of 163, 171
in screen cells 222
pointers to 164, 171
CL.LIB, Turbo Prolog and 294
_CL pseudo-variable 378
CL register 378
classic C style 183, 184
Clear All Breakpoints command 65, 142
Clear Breakpoints command 73, 100
Clear Project command 33, 113
clearing Watch window 141
clipping 240
Code Generation menu 116
code segment 344
colors
background 226, 229, 242
control
functions 241
on CGA 242

on EGA/VGA 244
drawing 242
foreground 226, 229
screen 241
COM1 209
combined operators 168
comma operator 174, 179
command line
arguments 19, 38, 133, 285
compiling and linking from 37, 38
configuration file 143
file names on 38
format 38
options 38
order of evaluation 41
turning off 38
running programs from 42
switches 82
automated build 83
configuration file 19, 82, 145
dual monitor 83
floating-point 392, 393, 394
floating-point emulation 392
make 83
syntax 39
Turbo C 37, 38, 143, *See also* TCC
commands *See also* menu commands
control flow 203
debugging, table of 72, 74, 99, 101
editing 93
pair-matching 151, 152
backward 152
forward 152
comments 185
delimiters 152
pair matching 154
nested 122, 312
commutative operators 319
COMPACT (macro) 340
compact memory model 348, 356
comparison operator 288
Compile/Build All command 50
Compile menu 19, 90, 108
compile time
debugging 128
error messages 96
Compile to OBJ command 109
compiler-linker options, in configuration file 144
Compiler menu 115
compiler options 143
compiling
for debugging 46, 98, 105, 106, 138
from command line 37, 38
from the integerated development environment 26
to an .EXE file 108
to an .OBJ file 108, 109
Compiling window 19
compound statement 175
CompuServe Forum, Borland 7
conditional compilation 335
conditional execution 159, 258
conditional operator (?:) 206
conditional statements 172, 206
Config Auto Save toggle 129, 146
configuration files 18, 143, 148
automatic save 129
changing 145
command-line 40, 143
creating 40, 144
data
compiler-linker options 144
pick file name 144
project name 144
directory 132
integrated development environment 143
loading 19, 133
menu settings saved in 143, 145
naming 129
overridden by command-line options 40
precedence over TCINST settings 145
saving 133
TC 143
user-specified 144
conglomerate data structures 199
console I/O functions 223
const (keyword) 321
const variable 282

constants 181
  character 314
  floating-point 315
  hexadecimal character 213
  integer 313
  naming restrictions 165
  pointer 291
  string 316
  Turbo C vs. Pascal 281
  typed 281
context-sensitive help 80
contexts, changing 146
continue (keyword) 203, 205
control flow commands 203
conventions
  calling 117
  menu-naming in TC 89
  typographic 6
conversion pointers 318
conversions 281
  arithmetic 318
  char 317
  enum 317
  int 317
coordinates, screen 223
  in text mode 222
  origin 223, 227
coprocessor
  8087/80287 math 360
  chip, floating-point 390
copyright law 6
CPINIT.OBJ 307
CPU 341, 377
  target, specifying 117
_cs (keyword) 353
_CS pseudo-variable 378
CS register 344, 346, 378
Current Pick File setting 132, 147, 148
cursor, running to line with 69
C*x*.LIB 360
_CX pseudo-variable 378
CX register 343, 378

D

data
  constants, defining in assembly code routines 367
  range 316
  segments 344
  size (bits) 316
  structures 188, 199, 271, 277
    conglomerate 199
    dynamic 200
  types 158, 161, 162, 254, 316
    conversions 281, 317
    signed 214
DATE (macro) 339
DD statement (assembler) 367
Debug menu 91, 133
debugger 45
  integrated 29, 45, 46, 98, 133, 139
    used on inline assembly code 77
  source level 46
debugging 45
  & vs && 64
  | vs || 64
  bottom-up 77
  boundary conditions 75
  cancelling session 106
  commands, table of 72, 74, 99, 101
  compile-time 128
  compiling a program for 46, 98, 105, 106, 138
  desk checking 62
  evaluating expressions 53, 54, 134
  example (WORDCNT) 48
  functions accessible to debugger 69
  guidelines 74
  infinite loop 55
  initiating a session 106, 107
  large source files 70
  modifying value of expressions 134
  moving cursor to next breakpoint 65
  multi-file programs 70, 71
  process, steps in 46
  qualifying variable names 57
  recompiling during 98
  reevaluating expressions 56
  restarting session 51
  run-time 46

running to cursor 69
starting a run 51
stepping over function calls 52
tracing into functions 107
declarations
function 180, 183, 210
global 183
improving legibility 357
Turbo C vs. Pascal 271
void functions 212
declarators 357, 358
decrement operator 257
decrement operator (--) 166
Default Char Type toggle 118, 314
default data pointers 352
Default Libraries toggle 126
define directive 334
defined (operator) 335
Defines setting 116
definitions
enumerated types 211
function 180, 184
strings 163
Delete Watch command 67, 73, 100, 141
delimiters
directional 152
levels 153
nestable 153
nondirectional 152
unmatched 154
dependencies
checking, automatic 113
explicit 34
file, checked by MAKE 42
implicit 34
_DH pseudo-variable 378
DH register 378
_DI pseudo-variable 378
DI register 343, 374, 378
diagnostic messages 96, 123
direct video output 230
directional delimiters 152
directional pair matching 152
directives
conditional 335
define 334
elif 335
else 335
endif 335
error 336
if 335
ifdef 335
ifndef 335
include 335
line 336
null 338
pragma 337
preprocessor 116, 333
undef 334
directories
changing 104
configuration file 132
help file 132
include file 143
choosing 17
library file 143
choosing 17
Directories menu 130
Directory command 104
Display Swapping toggle 69, 138
Display Warnings toggle 124
distribution disks 1
backing up 7, 9
division, integer 162
division operator (/) 166
_DL pseudo-variable 378
DL register 378
do...while loops 179, 262
do (keyword) 179
DOS
commands, MODE 83
exiting to 85, 104
shelling to 104
double (keyword) 162, 315
drawing color 242
drawing functions 236
_ds (keyword) 353
_DS pseudo-variable 378
DS register 344, 346, 378
dual monitor mode 83, 85, 104, 139
duplicate symbols linker warning 127

DW statement (assembler) 367
DWORD (assembler) 370
_DX pseudo-variable 378
DX register 343, 371, 378
dynamic data structures 200
dynamic memory allocation 188, 190, 283
  Turbo Prolog and 295, 302

E

Edit Auto Save toggle 129
Edit command 90, 91, 104
Edit Watch command 67, 73, 100, 141
Edit window 51, 53, 82, 91, 94
editing commands 93
editing keys
  assignment 155
  combinations 155
editing modes, displayed in status line 92
Editor 91
EGA, color control on 244
EGA/VGA setting 130
elif directive 335
else (keyword) 175, 200, 259
else directive 335
EMU.LIB 360, 391, 393
  Turbo Prolog and 294
emulation
  8087/80287 118
  8087/80287 floating-point 360
  floating-point 391
  option *See* –f emulation option
endif directive 335
entry (keyword) 313
entry codes, function 118
enum (keyword) 211, 317, 320
enumerated data types 255, 320
  definition 211
env (identifier) 285
environment
  variable 285
  working 17
Environment menu 127, 146
equal to operator (==) 172
errors 22
  common 124
  directive 336
  functions for handling, graphics 245
  less common 124
  messages 123
    compile-time 27, 31, 96
    graphics 245
    linker 28
  run-time, correcting 29
  syntax 27, 31, 32
    correcting 28
  tracking 32, 96, 289
    in a multi-file program 31
    linker errors 28
    syntax errors 27
Errors: Stop After setting 123
Errors menu 123
_es (keyword) 353
_ES pseudo-variable 378
ES register 344, 378
escape sequences 160, 213, 214, 315
Evaluate command 53, 54, 56, 73, 100, 134
Evaluate field 53, 134
Evaluate window 57
executable files 20
  named by TCC 39
execution
  conditional 258
  iterative 262
execution bar 51, 99
execution position 51, 99
Execution screen 53, 68
exit codes
  displayed 111
  function 118
explicit dependencies 34
expressions 166, 174
  default in Expression field 53, 56
  evaluating during debugging 53, 54, 134
  invalid for evaluating 57
  modifying value of during debugging 134
  reevaluating during debugging 56
  repeat 135

watch 66, 139
deleting from Watch window 67, 141
editing 67, 141
inserting in Watch window 140
scrolling 68
extensions, Turbo C 213
extern (keyword) 326
external identifiers 368
extra segment 344
EXTRN statement (assembler) 369, 375

F

factorial (function), Turbo Prolog and 300
far (keyword) 324, 345, 353, 361
far functions 354
far pointers 346
arithmetic on 347
comparing 346, 347
fdopen (function) 208
fflush (function) 209
fields
multiple 279
width 160
specifiers 168
FILE (macro) 338
File menu 90, 102, 146, 147
FILE object 207
files
.COM 348
Pascal 285
.EXE
building 19
making 109
named from project file 30
naming by Project-Make 110
.OBJ 111
compiling to 109
naming by Project-Make 109
backup, automatically created 129
configuration 18, 143, 148
automatic save 129
changing 145
command-line 40, 143
creating 40, 144
integrated development environment 143
loading 19, 133
menu settings saved in 143, 145
naming 129
overridden by command-line options 40
precedence over TCINST settings 145
saving 133
TC 143
user-specified 144
dependencies, checked by MAKE 42
executable 20
named by TCC 39
font, registering 240
graphics driver, linking 234
I/O 286
including 335
information in dependency checks 35, 113
library
external 36
run-time 131
names, on command line 38
object 20
external 36
startup 131
out-of-date, recompiled 34
pick 146
contains Editor information 147
contains file data 147
contains pick list 147
creating 132, 147
current 132
loading 132
name saved in configuration file 132
saved by Turbo C 148
project 29, 30, 112
graphics library listed in 231
README 10
source 20
.ASM 38
automatic save 129

creating 94
loading 18, 19, 95
loading multiple into Editor 33
multiple 29, 30, 32
name of 111
overwriting 95
saving 95
size of 111
working with in Edit window 94
writing to disk 24, 95
standard
include 131
standard library, overriding 37
fill patterns 237
filling functions 236
Find Function command 70, 73, 100, 137
flag register 343
float (keyword) 162, 181, 315
floating constants 315
floating point
arithmetic
interrupt functions and 388
emulation 391
error detection in 396
expressions, order of evaluation in 319
libraries 390
numbers 161, 162
programs that don't use 393
Floating Point toggle 118, 391, 392, 393, 394
flow of control commands 203
flow patterns, Turbo Prolog and 295, 302
flushall (function) 209
flushing stream buffers 209
font files, loading and registering 240
fonts, bit-mapped vs stroked 239
fopen (function) 208, 286
for (keyword) 178
for loops 178, 263
foreground
color 226, 229
setting 226
format commands 159
format specifications 159
format specifiers 57, 135
format strings 159
fortran (keyword) 313
forward declarations 267
forward pair matching 152
forward statement 267
FP87.LIB 360, 393
FP_OFF 357
FP_SEG 357
fprintf (function) 23
free (function)
Turbo Prolog and 295
free union variant record 279
freopen (function) 208, 210
fseek (function) 209
functions 180
accessible to debugger 69, 107
attribute control 226
calling in inline assembly code 384
cdecl type 327
color control 241
console I/O 223
declaration 180, 183, 210
declarator 329
declaring 264
declaring as near or far 354
definitions 180, 182, 184, 210, 326
drawing 236
entry codes 118
error checking 266
error-handling, graphics 245
exit codes 118
far 354
fdopen 208
fflush 209
filling 236
flushall 209
fopen 208
fprintf 23
freopen 208, 210
fseek 209
getch 172
gets 172
graphics system control functions 232

image manipulation 238
input 182
interrupt type 328
main 182
malloc 191, 217
mode control 225
naming restrictions 165
near 354
nested 185, 270
output 182
parameter lists 211
parentheses and 289
pixel manipulation 238
printf 22, 159
prototypes 183, 211, 267, 291, 329, 356, 357, 361
with Pascal-calling convention 365
putchar 161
puts 161
recursive 354
scanf 22, 170, 171
screen manipulation 238
setbuf 209
setmode 210
setvbuf 209
state query 227, 246
strcpy 164, 217
text manipulation 224
text output
graphics mode 239
text mode 224
Turbo C vs. Pascal 264
type modifiers for 326
viewport manipulation 238
void 212
window control 225

G

generate underbars option 365
Generate Underbars toggle 118, 322
Turbo Prolog and 294
Get Info command 111
getch (function) 172, 257
gets (function) 172, 257, 275
global declarations 183
global identifiers
defining in assembly code routines 368
global stack (Turbo Prolog) 302
global variables 252, 283
_fmode 208
Go to Cursor command 69, 72, 99, 106
goto (keyword) 203, 206, 387
graphics
drivers
linking 234
loading and selecting 234
mode *See* screen operating mode, *See* operating mode of screen
system control 232
Turbo Prolog and 308
GRAPHICS.H 231
GRAPHICS.LIB 231
Graphics Libraries toggle 127
greater than operator (>) 172
greater than or equal to operator (>=) 172
guidelines, debugging 74

H

hardware tabs 149
header files 267, 276
Hello, world program 21
help
file directory 132
getting 80, 93, 96, 97
index 80
screens 93
exiting 81
invoking 80
keywords 80
on library functions 81
hexadecimal character codes 314
hexadecimal character constants 213
hot keys 84, 85, 92
add watch expression 66, 73, 97, 100
change wild card mask 104
change window contents 68, 74, 84, 95, 101
compile to .OBJ file 27, 70, 109, 110

debugging 72, 74, 99, 101
delete watch expression 67, 97
edit watch expression 97
evaluate expression 54, 73, 100, 137
exiting to DOS 20, 85, 104
get help 84
go to cursor 69, 72, 99, 107
go to Editor 96, 105
insert watch expression 141
invoke help screen 93
load a file 18, 94
main menu choices 84
make .EXE file 93, 96, 110
make and run program 24, 31, 74, 101, 162
make program 19, 31, 32
next error 28, 33
pick file to load 95
previous error 28, 33
program reset 106
reset program 72, 99
save file 95, 103, 162
set/clear breakpoint 51, 73, 100
show call stack 73, 100
step over functions 53, 72, 93, 96, 99, 108
swap screens 20, 24, 53, 68, 69, 74, 85, 101
disabled 104
switch windows 28, 67, 68, 74, 84, 93, 96, 97, 99, 101
table of 86
toggle breakpoint 142
trace into functions 50, 62, 72, 93, 96, 99, 107
window/menu toggle 84, 93, 96, 105
zoom windows 68, 74, 84, 93, 101
Message window 96
Watch window 66, 97
huge (keyword) 324, 345, 353
HUGE (macro) 340
huge memory model 348, 356
huge pointers 347
comparing
!= operator 347
== operator 347
overhead of 348

I

I/O 286
stream 207, 209
Identifier Length setting 122, 312
identifiers
case 312, 322
defining in assembly code routines 368
global 322
length 312
naming restrictions 165
nonunique 333
Pascal-type vs. C-type 323
if...else statements 175
if (keyword) 173, 175
if directive 335
if statement 173
if/then/else statement 258
ifdef directive 335
ifndef directive 335
Ignore Case keystroke commands 156
implicit dependencies 34
include directive 335
Include Directories setting 17, 131
include files 276
directories 143
choosing 17
standard 131
increment operator 257
increment operator (++) 166
index
help 80
range error 257
variable 178
indexing 290
indirection operator 323
indirection operator (*) 169, 189
infinite loop 179
INIT.OBJ, Turbo Prolog and 294
initialization
module 294
variables 281, 282
Initialize Segments toggle 126

inline assembly code 379, *See* assembly code, inline
input 158, 182
  from keyboard 170, 172
  functions 170
  interactive 170
  Turbo C vs. Pascal 257
INSTALL 11
Install Editor screen 155
installing Turbo C 10, 11
  on a laptop system 11
Instruction Set toggle 117
int (keyword) 163, 317
INT instruction 387
integers 161
  constants 313
  division of 162
  values compatible with 202
integrated debugger 29, 45, 46, 98, 133, 139
integrated development environment 143
  configuration files 143
intensity, setting 226
interfacing
  with assembly code 366-376
  with other languages 326
interrupt (keyword) 326, 328, 387
interrupts
  functions 387
    example of 388
    floating-point arithmetic in 388
  handlers 387
  vectors 328, 387
invoking
  help screens 80
  main menu 105
IP (instruction pointer) register 343
iterations 159, 176
iterative execution 262

J

Jump Optimization toggle 121
jumps
  eliminating redundant 121
  instructions in inline assembly code 387

K

Keep Messages toggle 33, 128
Kernighan and Ritchie 3, 5, 21, 311
keyboard input 257
keys, rebinding 155
keystroke commands
  Ignore Case 156
  Verbatim 156
  WordStar-like 156
keystroke-editing mode 155
keystrokes, primary and secondary 155
keywords 313
  _cs 353
  _ds 353
  _es 353
  _ss 353
  ANSI 122
  asm 380
  cdecl 365
  far 345, 353, 361
  goto 387
  help screen 80
  huge 345, 353
  interrupt 387
  near 345, 353
  typedef 358

L

labels
  in inline assembly code 387
LARGE (macro) 340
large code models 352
large data models 352
large memory model 348, 356
legends 179
less than operator (<) 172
less than or equal to operator (<=) 172
Library Directories setting 17, 131
library files
  directories 143
    choosing 17
  external 36
  for memory models 360

run-time 131
using 359
library functions, help screens about 81
license statement, Borland 6, 9
LINE (macro) 338
line directive 336
Line Numbers toggle 119
line style 237
link, case sensitive 127
Link EXE File command 110
linked lists 200
linker 125
error messages 28
options, in configuration file 143
Turbo Prolog and 298
using directly 359
Linker menu 125
linking 108
from command line 37, 38
mixed modules 360
Turbo C and Turbo Prolog 293
without a make 110
Load command 18, 94, 95, 103, 147
load operations, suppressing redundant 121
loading
source files into TC 18
TC 16
logical AND operator (&&) 173
logical NOT operator (!) 173
logical operators 172, 173, 257, 263
logical OR operator (| |) 173
long (keyword) 162, 163
longwords 188
loop reorganizations 121
loops 159, 176, 262
do...while 179
for 178
infinite 179
repeat...until (Pascal) 180
while 176
low-level operations 167
low-level programming 377-390

M

macros
CDECL 339
COMPACT 340
converting to strings 334
DATE 339
defining 116
expanding 334
FILE 338
HUGE 340
LARGE 340
LINE 338
MEDIUM 340
MSDOS 339
nested 334
PASCAL 339
predefined 338, 339
SMALL 340
STDC 339
TIME 339
TINY 340
TURBOC 339
main (function) 182
when to declare with cdecl 328
main menu 82, 102
bar 90
options 105
Make EXE File command 19, 29, 109, 110
makes 102, 108
.OBJ file 109
projects 34
stopping 31, 112
vs builds 110
malloc (function) 191, 217
Turbo Prolog and 295
map file, object 119
Map file menu 126
masking
with Load command 103
math library, Turbo Prolog and 299
matherr (function), with floating point 396
MATHL.LIB 299
Turbo Prolog and 294
MATH*x*.LIB 360

MEDIUM (macro) 340
medium memory model 348
member access operator 278
memory
  addressing 115
  allocation 191, 283
    dynamic 188, 190
    explicit 195, 217
    explicit for structures 200
    for arrays 195
    graphics system 235
    Turbo Prolog and 295
  available 111
  dump 59, 136
  RAM 188
  segmentation 344
memory models 116, 324, 328, 348, 356, 341-362
  illustrations 349-352
  library files for 360
  startup module for 360
  switches 115
  Turbo Prolog and 297
memory-resident routines 388
menu commands 87
  Add Watch 66, 73, 100, 140
  Build All 50, 110
  Call Stack 71, 73, 100, 137
  Change Dir 17, 104
  Clear All Breakpoints 65, 142
  Clear Breakpoints 73, 100
  Clear Project 33, 113
  Compile to OBJ 109
  Delete Watch 67, 73, 100, 141
  Edit 90, 91, 104
  Edit Watch 67, 73, 100, 141
  Evaluate 53, 54, 56, 73, 100, 134
  Find Function 70, 73, 100, 137
  Get Info 111
  Go to Cursor 69, 72, 99, 106
  Link EXE File 110
  Load 18, 94, 95, 103, 147
  Make EXE File 19, 29, 98, 109, 110
  New 94, 103
  Next Breakpoint 73
  OS Shell 83, 104, 146
  Program Reset 51, 72, 99, 106
  Quit 85, 104, 146
  Refresh Display 69, 139
  Remove All Watches 67, 73, 100, 141
  Remove Messages 33, 74, 101, 114
  Retrieve Options 18, 133, 145
  Run 51, 74, 101, 105, 146
  Save 95, 103
  Save Options 18, 129, 132, 133, 144, 146
  Step Over 52, 72, 98, 99, 107
  Toggle Breakpoint 51, 73, 100, 141
  Trace Into 50, 62, 72, 99, 107
  User Screen 20, 24, 53, 68, 69, 85, 104
  View Next Breakpoint 65, 100, 142
  Write To 95, 104
menu settings 87
  Arguments 105, 133
  Current Pick File 132, 147, 148
  Defines 116
  Errors: Stop After 123
  Identifier Length 122
  Include Directories 17, 131
  Library Directories 17, 131
  Output Directory 17, 132
  Pick File Name 132, 147
  Primary C File 110
  Project Name 31, 112
  Tab Size 129, 149, 150, 151
  Turbo C Directory 132
  Warnings: Stop After 123
menu toggles 87
  Alignment 118
  ANSI Keywords Only 122
  Auto Dependencies 113
  Backup (source) Files 129
  Calling Convention 117, 323
  Case-sensitive Link 127
  Config Auto Save 129, 146
  Default Char Type 118
  Default Libraries 126
  Display Swapping 69, 138
  Display Warning 124
  Edit Auto Save 129

Floating Point 118, 391, 392, 393, 394
Generate Underbars 118, 322
Graphics Libraries 127
Initialize Segments 126
Instruction Set 117
Jump Optimization 121
Keep Messages 33, 128
Line Numbers 119
Merge Duplicate Strings 118
Message Tracking 32, 110, 128
Nested Comments 122, 154
OBJ Debug Information 50, 72, 98, 100, 107, 119, 137
Optimization For 120
Register Optimization 121
Source Debugging 46, 50, 98, 105, 106, 137, 138
Stack Warning 127
Standard Stack Frame 70, 71, 72, 100, 118, 138
Test Stack Overflow 119
Use Register Variables 120
Warn Duplicate Symbols 127
Zoomed Windows 130
menus
Break Make On 32, 112
Break/Watch 91, 139
choosing from 84
Code Generation 116
Compile 19, 90, 108
Compiler 115
Debug 91, 133
Directories 130
Environment 127, 146
Errors 123
exiting 84
File 90, 102, 146, 147
Linker 125
main 82, 90, 102
Map file 126
Model 115
Names 124
naming conventions in TC 89
Optimization 120
Options 90, 114, 144, 145, 146
Pick 95, 103, 146
Project 90, 111
pulldown, moving through 84
Run 90, 105, 146
Screen Size 130
Source 122
structure of system 87, 88
Merge Duplicate Strings toggle 118
Message Tracking toggle 32, 110, 128
Message window 19, 22, 27, 28, 31, 33, 82, 96
clearing 128
syntax errors in 109
mixed-language programming 362
mixed modules
linking 360
modern C style 183, 184
modifiers
cdecl 323, 327
const 321
far 323
function type 326
huge 323
interrupt 328
near 323
pointer 323
signed 321
volatile 322
modules
linking mixed 360
size limit 353
modulus operator (%) 166
MSDOS (macro) 339
multi-dimensional arrays 196, 273
passing 198
multi-source programs 29
building 30
multiple fields 279
multiple operators 168
multiple source files 32
loading into Editor 33
multiple string units 316
multiple types 279
multiplication operator (*) 166

N

Names Menu 124

naming conventions for TC menus 89
near (keyword) 324, 345, 353
near functions 354
near pointers 346
negation operators 319
negative offsets 344
nestable delimiters 153
nested comments 122
Nested comments command-line
  option 312
Nested Comments toggle 122, 154
nested functions 185, 270
nested macros 334
nested subexpressions
  pair matching 151
New command 94, 103
New Value field 53, 56, 134
newline character 214
Next Breakpoint command 73
nondirectional delimiters 152
nondirectional pair matching 152
normalized pointers 324, 347
not equal to operator (!=) 172
NOT operator (~) 167
null character 164
null directive 338
null string 285
null terminator (strings) 164

O

OBJ Debug Information toggle 50, 72,
  98, 100, 107, 119, 137
object code 20
object files 20
  external 36
  map 119
  startup 131
offsets 189, 346
  component of a pointer 193, 357
opcode mnemonics for inline
  assembly 382
operating mode of screen
  defining 222
  graphics mode 222, 231
    setting 234
  selecting 234
  text mode 222
    restoring 234
    setting 229
operations 158
  low level 167
operators 166, 167, 319
  addition (+) 166
  address 169, 266
  address-of 257
  address-of (&) 169, 189
  AND (&) 167
  assignment 255, 288
  assignment (=) 166, 174
  binary 166
  combined 168
  comma 174, 179
  commutative 319
  comparison 288
  conditional (?:) 206
  decrement 257
  decrement (--) 166
  division (/) 166
  equal to (==) 172
  greater than (>) 172
  greater than or equal to (>=) 172
  increment 257
  increment (++) 166
  indirection 323
  indirection (*) 169, 189
  less than (<) 172
  less than or equal to(<=) 172
  logical 172, 173, 257, 263
    AND (&&) 173
    NOT (!) 173
    OR (| |) 173
  member access 278
  modulus (%) 166
  multiplication (*) 166
  negation 166, 319
  NOT(~) 167
  not equal to (!=) 172
  OR (|) 167
  order of precedence 255
  relational 172, 263
    pointers and 324
  shift left (<<) 167
  shift right (>>) 167

short-circuit 257
subtraction (–) 166
ternary (?:) 206
Turbo C vs. Pascal 255
unary 166
plus 319
unary minus (–) 166
unary plus (+ 166
XOR (^) 167
Optimal fill mode 149, 150
examples 150
Optimization For toggle 120
Optimization menu 120
optimizing code 120
for size 118, 120
for speed 118, 120
options, command-line 38
–I 40
–L 40
order of evaluation 41
turning off 38
Options menu 90, 114, 144, 145, 146
OR operator (|) 167
order, row-column 196
OS Shell command 83, 104, 146
outdenting 150
output 159, 161, 182, 230
functions 224
to screen 159
Turbo C vs. Pascal 253
Output Directory setting 17, 132
overflow, stack 119
overhead 348
overriding standard library files 37

P

pair matching 151, 152
angle brackets 151
backward 152
braces 151
commands 151
comment delimiters 151, 154
directional 152
double quotes 151
examples 154, 155
forward 152
nested subexpressions 151
nondirectional 152
parentheses 151
single quotes 151
square brackets 151
palettes 241
paragraphs 188, 345
boundary 345
parameters
lists 211
order on stack 362
passing 322
passing sequence
C 362
C vs. Pascal 362-365
Pascal 364, 373, 376
parentheses with functions 289
Pascal *See* Turbo Pascal
calling conventions 396
calling sequence 117
parameter-passing sequence 323, 364, 373, 376
pascal (keyword) 323
function type modifier 326
identifiers of type 313
PASCAL (macro) 339
passing by address 266
passing by value 266
passing by var 266
path names
in project file 30
Pick File Name setting 132, 147
pick files 146
contents
Editor information 147
file data 147
pick list 147
creating 132, 147
current 132
loading 132
name 147
name saved in configuration file 132
saved by Turbo C 148
pick lists 103, 146, 147
Pick menu 95, 103, 146
pitfalls in C programming 214

= vs == 217
array indexing from 0 218
for Pascal programmers 288
function calls 289
misuse of pointers 215
misuse of strings 215
passing by address 219
the break in switch statements 218
using backslash in path names 214
pixels, setting color of 241
plus, unary 166
pointer arithmetic 192
pointers 162, 188, 271
and arrays 195
and FILE objects 207
and structures 200
arithmetic on 347, 348
arrays vs. 290
character 163, 164, 171
comparing 346, 347
!= operator 347
== operator 347
constant 291
conversion 318
declarations 271
declaring as near, far, or huge 355-357
default data 352
far 324, 346
huge 324, 347
overhead of 348
manipulation 346
near 324, 346
normalized 324, 347
offsets of 193
relational operators and 324
string manipulation and 271
Turbo C vs. Pascal 271
void type and 320
portability
of nested comments 122
of predefined streams 209
warnings 124
positive offsets 343
pragma directive 337
inline 337
saveregs 338
warn 337
predefined streams 209
portability of 209
redirected 210
prefix opcodes
repeat 384
preprocessor 339
directives 116, 292, 333
primary .C file 110, 111
Primary C File setting 110
primary keystrokes 155
printf (function) 22, 159, 177, 253
Turbo Prolog and 295
Program Reset command 51, 72, 99, 106
programming
basic elements 158
in Turbo C 157
Turbo C vs. Pascal) 250
programs
multi-source 29
building 30
running 20
structure, C vs. Pascal 250
project files 29
Project-Make 29, 34, 102, 105, 109
Project menu 90, 111
Project Name setting 31, 112
projects 111
clearing 33
files 29, 30, 112
graphics library listed in 231
making 34
Prolog *See* Turbo Prolog
prototypes, function 183, 211
advantages of using 219
pseudo-variables 377
putc (function)
Turbo Prolog and 295
putchar (function) 161, 254
puts (function) 161, 254

Q

qualifying variable names 57
Quick-Ref Line 82, 91, 92, 96, 97

Quick Reference Line 82, *See also* Quick-Ref Line
Quit command 85, 104, 146
quitting Turbo C integrated development environment 85
QWORD (assembler) 370

R

RAM memory 188, 396
  Turbo C's use of 396
random access streams 209
Read (Pascal function) 257
reading streams 209
Readln (Pascal function) 257
README file 10
real mode 117
real numbers 161
rebinding keys 155
recompiling during debugging 98
records 272
  in Pascal 277
recursive functions 354
recursive structures, Turbo Prolog and 306
redirecting predefined streams 210
redirection 133
referencing data in inline assembly code 384
Refresh Display command 69, 139
Register Optimization toggle 121
registers
  8086 342-344
    illustrations 342
  8087/80287 top-of-stack 371
  AH 378
  AL 378
  allocation, Turbo Prolog and 294
  AX 343, 371, 378
  BH 378
  BL 378
  BP 343, 371, 378
  BX 343, 378
  CH 378
  CL 378
  conventions 374
  CS 344, 346, 378
  CX 343, 378
  DH 378
  DI 343, 374, 378
  DL 378
  DS 344, 346, 378
  DX 343, 371, 378
  ES 344, 378
  flag 343
  IP (instruction pointer) 343
  optimizing use of 121
  segment 343, 344
  SI 343, 374, 378
  SP 343, 371, 378
  SS 344, 378
  variables 120, 322
    in inline assembly code 385
relational operators 172, 263
  pointers and 324
Remove All Watches command 67, 73, 100, 141
Remove Messages command 33, 74, 101, 114
repeat...until loops (Pascal) 180, 262
repeat count 58
repeat expression 135
repeat prefix opcodes 384
reserved words 313
resolution, screen 222
restrictions on calling Turbo Prolog from other languages 306
Result field 53, 134
Retrieve Options command 18, 133, 145
return (keyword) 203, 204
return statement 265
retyping 281
routines, assembly code 366
row-column order 196
Run command 51, 74, 101, 105, 146
Run menu 90, 105, 146
run time
  errors, correcting 29
  library files 131
running a program 20

S

Save command 95, 103

Save Options command 18, 129, 132, 133, 144, 146
saved User screen buffer 85
scaling factor 237
scanf (function) 22, 170, 171, 257, 275
scope rules 333
Screen Size menu 130
screens
  attributes
    controlling 226
  colors 241
  coordinates 223
    in text mode 222
  help 93
  operating mode
    controlling 225
    defining 222
    graphics mode 222, 231, 234
    selecting 234
    text mode 222, 229, 234
  resolution 222
  swapping, smart 138
  TC 85
  User 85
scrolling watch expressions 68
secondary keystrokes 155
segment:offset address notation 345
  making far pointers from 357
segmented memory architecture 344
segments 189, 345, 348
  component of a pointer 357
  initializing 126
  memory 344
  naming 124
  registers 343, 344
semicolons 258, 291
setbuf (function) 209
setmode (function) 210
settings *See also* menu settings
  43/50 Lines 130
  25 Lines 130
  EGA/VGA 130
  environment, saved in configuration file 143, 145
  standard display 130
setvbuf (function) 209
shelling to DOS 104
shift left operator (<<) 167
shift right operator (>>) 167
short (keyword) 163
short-circuit operators 173, 257
shortcuts 85, *See* hot keys
_SI pseudo-variable 378
SI register 343, 374, 378
sign extension 317
signed (keyword) 214, 321
size overrides in inline assembly code 385
sizeof (keyword) 191, 193, 336
small
  code models 352
  data models 352
  memory model 348, 356
SMALL (macro) 340
smart screen swapping 138, 139
soft tabs 149
software interrupt instruction 387
software tabs 149
source code 20
Source Debugging toggle 46, 50, 98, 105, 106, 107, 137, 138
source files 20
  .ASM 38
  automatic save 129
  creating 94
  loading 18, 19, 95
  multiple 29, 30, 32
    loading into Editor 33
  name of 111
  overwriting 95
  saving 95
  size of 111
  working with in Edit window 94
  writing to disk 95
source-level debugger 46
Source menu 122
_SP pseudo-variable 378
SP register 343, 371, 378
_ss (keyword) 353
_SS pseudo-variable 378
SS register 344, 378

stack
call 70, 137
displaying executing line of function from 71
returning to execution bar from 71
frame, standard 70, 118
global (Turbo Prolog) 302
overflow 119
segment 344
Stack Warning toggle 127
stand-alone utilities
configuration file converter (TCCONFIG.EXE) 41
program manager (MAKE) 42
standard display setting 130
standard files
include 131
library, overriding 37
standard stack frame 70
Standard Stack Frame toggle 70, 71, 72, 100, 118, 138
startup modules for memory models 360
startup object files 131
state queries 227, 246
statements 326
assignment
value of 174
block 258
break 260
case 259
return 265
switch 259
static variables 282
status line 92
STDC (macro) 339
Step Over command 52, 72, 98, 99, 107
strcpy (function) 164, 217, 329
streams 207
binary 208
buffered 208
buffers, flushing 209
I/O 207, 208, 209
opening 208
predefined 209
portability 209
redirecting 210
random access 209
text 208
strings 162, 195
arrays of 163
concatenation 316
defining 163
merging duplicate 118
multiple 316
null terminated 164
passing 171
pointers and 271
Turbo C vs. Pascal 273
stroked fonts 239
struct (keyword) 199, 212, 272, 279
structure of menu system 87, 88
structures
additions to K&R alignment 324
and pointers 200
bitfields 325
data 199, 271
declaration 199
member access 199, 200
operator 200
recursive 306
Turbo C vs. Pascal records 277
style, C programming, classic vs modern 183, 184, 210
subroutines 159, 180, 264
subtraction operator (–) 166
sum function
Turbo Prolog and 300
switch (keyword) 201
switch statement 259
switches, command-line 82
automated build 83
configuration file 19, 82, 145
dual monitor 83
floating-point 392, 393, 394
floating-point emulation 392
make 83
syntax
command-line 39
errors 27, 31, 32

correcting 28
system control, graphics 232
system requirements, Turbo C 2

T

Tab Size setting 129, 149, 150, 151
tabs 149
hardware 149
soft 149
software 149
target CPU, specifying 117
TASM 38, 77, 380
TBYTE (assembler) 370
TC 79, 143, *See also* Turbo C
integrated development
environment
configuration files, creating 144
screen 85
values specific to
environment options 144
pick file name 144
project name 144
TC screen 82
TCC 37, 38, 143, *See also* command-line Turbo C
TCCONFIG.EXE 41
TCCONFIG.TC 18, 133, 143, 144, 145, 146
conversion to TURBOC.CFG 41
directory of 145
TCINST 145, 148, 149, 155
Autoindent toggle 149
settings, precedence of configuration file over 145
technical support, Borland 7
template, assembly code 367
ternary operators 206
Test Stack Overflow toggle 119
text
data type 162
manipulation
and output 224
functions 224
mode 208, *See* screen operating mode, *See* operating mode of screen
stream 208
then (Pascal keyword) 259
TIME (macro) 339
TINY (macro) 340
tiny memory model 348
TLINK
Turbo Prolog and 298
using directly 359
Toggle Breakpoint command 51, 73, 100, 141
toggles *See also* menu toggles
Autoindent 149
tokens
pasting 312
replacement 334
TOS register 371
Trace Into command 50, 62, 72, 99, 107
Turbo Assembler 38, 77, 380
Turbo C
calling Turbo Prolog 304
command-line 143
installing 10, 11
on a laptop system 11
integrated development environment 79, 143, *See also* TC
loading 16, 19, 82
interactive Editor 91
structure members in inline assembly code 385
system requirements 2
Turbo Pascal vs. 249
working environment 127
Turbo C Directory setting 132
Turbo Pascal 249
Turbo Prolog
calling from Turbo C 304
interfacing with 293
linking with 293
TURBOC (macro) 339
TURBOC.CFG 40, 143, 391
conversion to TCCONFIG.TC 41
tutorial, Turbo C 157
type-casting 191, 281
type mismatch 289
typed constants 281
typedef (keyword) 199, 211, 212, 358

types
  const 321
  enum 320
  enumeration 319
  long double 319
  modifiers 319
  multiple 279
  signed 321
  specifiers 319
  unsigned char 319
  unsigned long 319
  unsigned short 319
  void 319, 320
  volatile 322
typographic conventions 6

U

unary operators 166
  minus (–) 166
  plus 319
  plus (+) 166
undef directive 334
undefined routines, searched by TC compiler 126
underscores 322
  leading, in assembly code, routines 368
Unindent mode 150
union (keyword) 212
unions 279, 324
unmatched delimiters 154
unsigned (keyword) 163, 321
Use Register Variables toggle 120
User screen 85
User Screen command 20, 24, 53, 68, 69, 85, 104
user-specified configuration file 144
utilities, standalone *See* standalone utilities

V

value, passing by 266
var, passing by 266
variables
  const 282
  defining in assembly code routines 367
  global 252, 283
  index 178
  initialization 281, 282
  names, qualifying 57
  naming restrictions 165
  offsets in inline assembly code 385
  register 322
  static 282
  storage 283
Verbatim keystroke commands 156
VGA, color control on 244
video adapters 221
  graphics, compatible with Turbo C 233
View Next Breakpoint command 65, 100, 142
viewports 223
violations, ANSI 124
void (keyword) 180, 212, 264, 320, 329
  interrupt functions and 328
void functions 212
volatile (keyword) 322

W

Warn Duplicate Symbols toggle 127
warnings 22, 123, 124, 289
  portability 124
Warnings: Stop After setting 123
watch expressions 66, 96, 139
  default 66
  deleting from Watch window 67, 141
  editing 67, 141
  inserting in Watch window 66, 140
  scrolling 68
Watch window 57, 66, 96, 139
  clearing 141
while (keyword) 176, 179
while loops 176, 205, 262
whitespace 170
windows
  active 91
  Compiling 19
  controlling 225
  Edit 51, 53, 82, 91, 94
  Evaluate 57

Message 19, 22, 27, 28, 82, 96
switching 93
text 222, 227
creating 228
Watch 57, 66, 96, 139
zooming 27, 93, 94
with statement 278
WORD (assembler) 370
WORDCNT 48
words 188
alignment 118, 325
WordStar-like keystroke commands 156
working environment 17, 127
wrch (Prolog function), Turbo Prolog and 295
Write (Pascal function) 253
Write To command 95, 104
Writeln (Pascal function) 253
writing
files to disk 24, 104
streams to disk 209

X

XOR operator (^) 167

Z

Zoomed Windows toggle 130
zwf (Prolog function), Turbo Prolog and 295