Introduction

Welcome to the world of QuickC programming. If you're like most programmers who have been programming with QuickC and are now interested in expanding your C programming skills and learning how to develop some useful tools and PC applications, then you've come to the right place. Microsoft's QuickC is a powerful language and programming environment for serious software development for the IBM PC. It provides all of the necessary basic programming tools that you'll need to master the PC. From the fast and easy-to-use integrated environment to the sophisticated programming support features, such as the QuickC librarian or the debugger, QuickC offers something for all programmers, beginning to advanced.

Whom This Book Is For

This book assumes that you're familiar with the basics of C programming, and it does not explain C programming fundamentals at the level you'll need if you're a new hand at C programming. If you have some experience working with the C language, you'll benefit from the material covered in this book. Our major concern is to show you how to get the most out of QuickC. Therefore, we're not going to waste your time discussing the same tired basic C programming examples that you'll find in numerous other books.

How This Book Is Different

*Advanced Programming with Microsoft QuickC* emphasizes practical and useful programming examples. Most C programming books only present simple toy programs that serve no purpose other than to provide a vehicle for
learning programming concepts. This book covers the advanced programming features of both the IBM PC and QuickC.

We’ll spend most of our time discussing software development techniques for constructing powerful C programming tools. We’ll also look at clever ways of using the tools that we develop for creating useful programs that take advantage of the PC’s unique hardware. Most professional quality software is designed using both the software tools approach and techniques of stepwise development. Thus, we’ll emphasize both these principles in the tools and programs that we’ll develop. For example, the tools that we develop in earlier chapters, such as data structure development tools, will be used and extended in later chapters. These examples will give you some insight into the importance of designing portable and powerful software tools.

What about Standards?

QuickC supports the latest proposed ANSI C standard which we’re going to emphasize throughout this book. The proposed standard provides many powerful features such as function prototypes and void and enumerated types. By using the proposed standard features, we can better guarantee that our code will be reliable and portable.

What You’ll Need

To get the most out of the material presented in this book, you should have access to QuickC. You’ll also need an IBM PC, XT, AT, or a compatible computer system. All of the code that we’ll develop throughout the book is designed for use with the QuickC compiler, but because we’ll emphasize the portable features of QuickC, much of the code can be easily adapted for other C compilers.

How This Book Is Organized

Advanced QuickC Programming is divided into five sections. In Section 1 (Chapters 1-2), we’ll introduce some important topics that will be emphasized throughout the book, such as the proposed ANSI standard features, tips and techniques about C programming style, working with the C preprocessor, and
tips for using pointers and managing memory allocation tasks. If you’re an experienced C programmer, but relatively new to QuickC, you’ll find this material to be a good starting point. Section 2 (Chapters 3-5) presents data structures. In these chapters, we’ll discuss techniques for constructing useful and reliable data structures from linked lists to binary trees. Section 3 (Chapters 6-7) covers the complete C I/O system. Here we’ll take an in-depth look at the many tools that QuickC provides for accessing files and other I/O devices. Section 4 (Chapters 8-10) explains the techniques for interacting with DOS and the special features of QuickC. In these chapters, we’ll learn how to access the PC’s hardware, such as the screen, and we’ll develop a library of powerful tools that we’ll use for developing multifunctional user interfaces. We’ll even discuss how to interface with the mouse.

The final section, Section 5 (Chapters 11-13), presents the tools and techniques for developing QuickC-like user interfaces. Here we’ll investigate topics such as pop-up windows, pop-up menus, and pull-down menus. The code presented in these final chapters can be used to enhance all of the QuickC programs that you develop from now on.
QuickC is much more than just a fast C compiler with a good user interface. It is a complete integrated programming environment that can greatly increase your programming productivity. The QuickC package provides a wide assortment of development tools from a command line compiler, linker, make utility, and library manager to a powerful integrated compiler that provides C debugging features. Since you've more than likely been working with QuickC, you're probably familiar with most of the basic features that QuickC provides; however, you've now picked up this book to learn how to get more out of both the QuickC environment and the C programming language.

We'll start our exploration in Section 1 with a quick tour of the important proposed ANSI C standard features that QuickC provides. If you're an experienced C programmer but fairly new to the proposed C standard, you'll really benefit from the material presented in Chapter 1. We'll also look at the useful global variables included with the QuickC environment, and then we'll end the chapter with a discussion of some important C programming style tips. In Chapter 2, we'll turn our attention to some programming techniques that should help you improve your C programming skills. Here we'll present some interesting topics including the QuickC preprocessor, command line arguments, and pointer and memory allocation techniques.
We'll begin our discussion of QuickC by looking at the main features that are part of the new proposed ANSI C standard. We'll cover topics such as function prototypes, variable length arguments, generic pointers, pointer and function modifiers, enumerated types, nested comments, and other important topics that are part of the proposed standard. QuickC also provides some important features that we'll discuss such as unique global variables that are not defined by the proposed standard. We'll also include examples to show you how these variables can be used to access the PC's operating system and improve the performance of the programs we write. The last section of this chapter is devoted to the issues relating to style in C programs, including writing portable and readable code.

**Working Towards a New Standard**

The C language has been evolving since its introduction in the 1970s. Many of the new features that are being added are helping to make the language more portable, more efficient and reliable, and easier to use. Fortunately, QuickC supports the Draft-Proposed ANSI C standard. This means that most of the code you develop with QuickC should work with other C compilers that adhere to the proposed ANSI standard. Of course, there will always be some language features that can't be ported such as the low-level routines provided for communicating with the operating system. Throughout this book we'll point out
the features that might not be compatible with other C compilers. Keep in mind, however, that many of the example programs that we’ll present are written to take advantage of the features of the PC; thus, if you want to use the programs on other computer systems, you will have to make the necessary changes to support other hardware.

In this section we’ll discuss some of the important trends that are influencing the C language. Since QuickC supports both the old style of C programming, which was introduced to the world by Kernighan and Ritchie in their classic text *The C Programming Language*, and the modern style adapted by the new proposed standard, you can use the features of either style. We’ll emphasize the modern style, however, because of the new benefits it provides.

**Function Prototypes**

Function prototypes are one of the more useful features of the proposed C standard. They help the compiler determine if the correct arguments are supplied whenever a function is defined or called. A function prototype looks like a function definition without the function body. The basic form is

\[ \text{type function-name (arg-type arg1, arg-type arg2, \ldots)}; \]

In this case, type refers to the data type of the value returned by the function and function-name is the function’s unique name. The function prototype also allows us to specify the names and types of the arguments used by the function. These are represented by the terms arg-type, arg1, arg2, etc. For example, the following prototype

\[ \text{int linelen(char *str, int max)}; \]

declares a function called *linelen* which returns an integer value and uses two arguments—a pointer to a character and an integer type.

Although a function prototype looks very much like a function declaration, don’t confuse the two, because they are different. The function prototype acts as a template to tell the compiler the form of each defined function. Let’s look at some more examples to illustrate how the function prototype is used.

In the old style of C programming, functions are defined using the form shown here:

\[ \text{int avg(x, y)} \]
\[ \text{int x, y;} \]
When function prototypes are used, a function is defined as

```c
int avg(int x, int y)
{
    return ( (x + y) / 2);
}
```

Here note that the definitions of the arguments are moved up into the function’s argument list. A prototype for this function can now be written as

```c
int avg(int x, int y);
```

This prototype tells the compiler the types of the arguments used by the function. The names of the arguments are actually ignored by the compiler; however, they are useful for documenting the names of arguments used in a function. Of course, since they are ignored, we can obtain the same results by writing the prototype as

```c
int avg(int, int);
```

Throughout this book, we’ll use the first style in the code that we present because of its built-in documentation features. If a function is used differently from the way it is defined by the prototype, the compiler will issue a warning message. For example, if we call the function `avg` with the statement

```c
c = avg(x, r, p);
```

where the variables `c` and `r` are defined as a character and a floating point, the compiler will inform us of this incompatibility. In addition, the compiler will tell us that the function call contains too many arguments. In general, if there is a type mismatch between a function prototype declaration and a function call, QuickC will issue a warning message. Of course, we can ignore this warning message, and the compiler will convert the arguments of the calling function to the corresponding type of those declared in the prototype. Let’s look at an example to see how this is done.

The following simple program includes a prototype for the `avg` function:

```c
int avg(int x, int y);
```
main()
{
    int x;
    char c;
    unsigned result;

    x = 20;
    c = '0';
    result = avg(x, c);
    printf("The average is %d", result);
}

When the function avg is called, both arguments \( x \) and \( c \) are stored on the stack as integers as shown in Figure 1.1. The argument \( c \) is converted from a character to an integer because of the prototype. If the prototype for \( \text{avg} \) is omitted, this argument would be placed on the stack as a character which is shown in Figure 1.2. In this case, the prototype guarantees that the correct types of the arguments are passed to and from a function.

In general, function prototypes increase the reliability of programs, and they should be used whenever possible. Usually, they are placed at the beginning of a program or in a header file which is included in a C source file. Putting the function prototypes in a separate header file is a good programming technique because it allows you to share the prototypes between different source files.

![Figure 1.1. Arguments stored on stack with prototype conversion](image1)

![Figure 1.2. Arguments stored on stack without prototype](image2)
We’ll use this technique in most of the code that we develop throughout this book.

If you have C programs and libraries that have been compiled with other C compilers, you might want to write prototypes for the functions used in your code and compile them with QuickC. Often when this is done, the QuickC compiler will find errors that were not detected by other compilers. Of course, such errors are usually not easy to locate. The important lesson here is to use function prototypes to help guarantee that your code is as reliable as possible.

Prototypes with Variable Arguments

QuickC allows us to use prototypes to define functions that use a variable number of arguments. The basic syntax for a prototype with variable arguments is illustrated by the following function argsum:

```c
int argsum(int nargs, int arg1, ...);
```

The ellipsis indicates that this function uses a variable number of arguments. In this case, `argsum` must contain at least two arguments. The compiler will check to make sure that these arguments are supplied for any call to `argsum`; however, the additional arguments are not checked. For example, if we called `argsum` with

```c
i = argsum(count, x, y, z);
```

the compiler would test the arguments `count` and `x` for compatibility with the prototype and would ignore the type checking for the other arguments.

Functions constructed with variable arguments can provide us with more general software tools. If you’ve never used this feature, you should examine the program shown next:

```c
#include <stdio.h>
#include <stdarg.h>

int argsum(int nargs, int arg1, ...);

main()
{
    int result;

    result = argsum(5, 4, 2, 1, 7, 10); /* sum arguments 2 to 5 */
    printf("\nThe sum of arguments is %d\n", result);
}
```
int argsum(int nargs, int arg1, ...) {
    va_list arg_ptr;
    int total, curarg;

    va_start(arg_ptr, arg1);
    total = arg1;
    while (-arg1 > 1) {
        curarg = va_arg(arg_ptr, int);
        total = total + curarg;
    }
    va_end(arg_ptr);
    return(total);
}

This program sums up the values of all the arguments passed to argsum except
the first argument. The first argument is used to keep track of the number of
arguments passed to argsum. To access the variable arguments in a function,
QuickC provides three predefined macros, which are va_start, va_arg, and
va_end. The first, va_start is a macro declared as

    void va_start(va_list argptr, lastarg);

This macro sets a pointer to the first variable argument. In this case, the
argument argptr is used to store the pointer, and lastarg refers to the name of
the last fixed argument contained in the variable argument list. The term va_list
is a predefined type that QuickC provides for processing variable length
arguments. If you examine the call to va_start in our sample program

    va_start(arg_ptr, arg1);

you’ll notice that we set the pointer arg_ptr to point to the first variable
argument. Note here that argl is the name of the last fixed argument in our
definition of sumargs. Once this pointer is initialized, we can access the variable
arguments by calling va_arg, which is another macro that QuickC provides.
This macro is declared as

    arg_type va_arg(va_list argptr, arg_type);

Note here that two arguments are required. The first consists of the argument
pointer, and the second represents the argument data type to be accessed. When
va_arg is called, it returns the value of the variable argument currently referenced by the argument pointer and it also increments the argument pointer so that it points to the next argument in the list.

**Using the void Type**

In our discussion of function prototypes, all of the functions defined used arguments and returned values. If we define functions that do not use arguments or do not return values, we can use the reserved word void to inform the compiler. The first statement illustrates how a function prototype is written for a function without arguments:

```c
int process(void);
```

If the function does not return a value, it can be declared as

```c
void skipblanks(char *str);
```

In both cases the void type acts as a null type. If it is omitted for the return value in a function definition, the compiler assumes that the function returns an integer type by default. For example, if the function skipblanks is defined as

```c
skipblanks(char *s)
{
    if (strchr(s, ' ')) ... ;
    ...
}
```

the QuickC compiler will treat the function as if it returns an integer value.

The void type can also be used to create a new type of pointer called the generic pointer. This pointer is often used by functions that allocate memory and return back pointers. We’ll examine generic pointers in more detail in the next section.

**Generic Pointers**

In traditional C programming, all functions that return a pointer must return one of a standard type such as int, char, struct, etc. If the type of a pointer is not specified, the compiler assumes the pointer is of type char. With the proposed
ANSI standard, we can now use and return pointers that are truly generic. A generic pointer is one that can be used to reference any type of data object. Because they are untyped, the value of a generic pointer can be assigned to any other type of pointer. For example, if we have two pointers declared as

```c
void *ptr1;
char *ptr2;
```

the following assignment statement is valid:

```c
ptr2 = ptr1;
```

Of course, since the type of the `void` pointer is not defined, we cannot use the `*` operator to retrieve the contents of the pointer. For example, the assignment

```c
val = *ptr1;
```

is illegal.

Many of the QuickC library functions, such as those used for memory allocation and low-level support, return generic or `void` pointers. As an example, look at the following declaration of `memcpy`:

```c
void * memcpy (void *destin, const void *source, size_t n);
```

Since this function returns a generic pointer, it is up to us to decide how the pointer is to be used. This feature increases the flexibility of the function.

In most cases, when a `void` pointer is returned, it is a good idea to typecast the returned pointer. This means that we tell the C compiler how we expect to use the pointer. For example, if we want to use the returned pointer from `memcpy` as a character pointer, the typecasting operation is coded as

```c
chptr = (char*) memcpy (dest, src, sze);
```

Here the expression `(char*)` informs the compiler that the returned value of `memcpy` should be converted into the type character pointer. If you’re concerned about the portability of your QuickC programs, you should use typecasting whenever you work with functions that return `void` pointers.
Special Pointers

In addition to the void or generic pointer, QuickC provides a set of special pointers that are declared using the addressing modifiers near and far. With these modifiers, we can create pointers that override the size of the default pointers provided with each of the four QuickC memory models. Table 1.1 lists the sizes for the default data pointers for each memory model.

<table>
<thead>
<tr>
<th>Memory Model</th>
<th>Data Pointer Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>16 bits</td>
</tr>
<tr>
<td>medium</td>
<td>16 bits</td>
</tr>
<tr>
<td>compact</td>
<td>32 bits</td>
</tr>
<tr>
<td>large</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

The first two memory models listed in the table use only the current value of the Data Segment (DS) register to access data. Therefore, only a 16 bit pointer is needed to reference the offset of the data inside the Data Segment. The other two memory models, compact and large, use both a segment address, which is different than the Data Segment, and an offset address. Because of this, a 32 bit pointer is needed to support each of these memory models.

Now you might be wondering, how do the modifiers near and far affect these memory models? When a pointer variable is declared, one of these modifiers can be used to direct the compiler to treat the pointer differently than the default size specified by the memory model in use. The descriptions of these modifiers are discussed next.

**near Modifier**

The near modifier is used to create a 16 bit pointer. The syntax for its usage is

```c
near data-type *pointer;
```

As an example, the declaration

```c
near char *chptr;
```