

Osborne 

**Covers Draft
ANSI C++**

C++

**I N S I D E
& O U T**

BRUCE ECKEL

*Guides You Quickly From Fundamentals to
Advanced Concepts So You Gain a Thorough
Understanding of C++*

Osborne **McGraw-Hill**
2600 Tenth Street
Berkeley, California 94710 U.S.A.

For information on translations and book distributors outside of the U.S.A., please write to Osborne **McGraw-Hill** at the above address.

C++ Inside & Out

Copyright © 1993 by McGraw-Hill, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

1234567890 DOC 99876543

ISBN 0-07-881809-5

Publisher

Kenna S. Wood

Acquisitions Editor

Jeffrey M. Pepper

Associate Editor

Emily Rader

Project Editor

Erica Spaberg

Copy Editor

Paul Medoff

Technical Editor

Jeffrey Hsu

Proofreaders

Audrey Johnson

K. D. Sullivan

Mick Arellano

Indexers

Matthew Spence

Bruce Eckel

Computer Designer

Stefany Otis

Illustrator

Marla Shelasky

Cover Designer

Bay Graphics Design, Inc.

Mason Fong

Information has been obtained by Osborne **McGraw-Hill** from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, Osborne **McGraw-Hill**, or others, Osborne **McGraw-Hill** does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from use of such information.

```
float A = (float)200;
```

sizeof()—An Operator by Itself

The **sizeof()** operator stands alone because it satisfies an unusual need—it gives you information about the amount of memory allocated for data items. As described earlier in this chapter, **sizeof()** tells you the number of bytes used by any particular variable. It can also give the size of a data type (with no variable name):

```
printf("sizeof(double) = %d\n", sizeof(double));
```

The **sizeof()** operator can also give you the sizes of user-defined data types. This is used later in the book.

Creating Functions in C and C++

Most modern languages have an ability to create named subroutines or subprograms, and C and C++ are not exceptions. In C/C++, a subprogram is called a *function*. All functions have return values (although that value can be "nothing"), so functions in C are very similar to functions in Pascal. (The Pascal *procedure* is the specialized case of a function with no return value. From that perspective, it hardly seems worth the brouhaha to give it a separate name....)

Function Prototyping

Function prototyping is another improvement invented in C++ that propagated back into ANSI C. You have been seeing function prototyping in this book described as "telling the compiler that a function exists and how it is called." Now it's time for more details.

In old (pre-ANSI) C, you could call a function with any number or type of arguments, and the compiler wouldn't complain. Everything seemed

fine until you ran the program. You got mysterious results (or worse, the program crashed) with no hints as to why. The lack of help with argument passing and the enigmatic bugs that resulted are probably prime reasons why C has been dubbed a "high-level assembly language."

C programmers just adapted to it. When Stroustrup designed C++, he decided that something must be done about the problem, so he added *function prototyping*. With function prototyping, you always use a *prototype*, an example, when declaring and defining a function. When the function is called, the compiler uses the prototype to ensure that the proper arguments are passed in and that the return value is treated correctly. If the programmer makes a mistake when calling the function, the compiler catches the mistake.

Telling the Compiler How Arguments Are Passed

In a function prototype, the *argument list* (which follows the name and is surrounded by parentheses) contains the types of arguments that must be passed to the function and (optionally for the declaration) the names of the arguments. The order and type (but not the names) of the arguments must match in the declaration, definition, and function call. Here's an example of a function prototype in a declaration:

```
int translate(float x, float y, float z);
```

You cannot use the same form when defining variables in function argument lists as you do in ordinary variable definitions, for example, **float x, y, z**. You must indicate the type of each argument. In a function declaration, the following form is also acceptable,

```
int translate(float, float, float);
```

since the compiler doesn't do anything but check for types when the function is called.

In the function definition, names are required because the arguments are referenced inside the function:

```
int translate(float x, float y, float z) {  
    x = y = z;  
}
```

```
// ...  
)
```

The only exception to this rule occurs in C++ as follows: an argument may be unnamed in the argument list of the function definition. Since it is unnamed, you cannot use it in the function body, however. The reason unnamed arguments are allowed is to give the programmer a way to "reserve space in the argument list." You must still call the function with the proper arguments, but you can use the argument in the future without modifying any of the other code. This option of ignoring an argument in the list is possible if you leave the name in, but you will get an obnoxious warning message about the value being unused every time you compile the function. The warning is eliminated if you remove the name.

ANSI C and C++ have two other ways to declare an argument list. If you have an empty argument list, you can declare it as **foo()** in C++. Remember this only means an empty argument list in C++. In ANSI C it means "an indeterminate number of arguments." In both ANSI C and C++, the declaration **foo(void)**; means an empty argument list. The **void** keyword means "nothing" in this case (it can also mean "no type" when applied to certain variables).

The other option for argument lists occurs when you don't know how many arguments or what type of arguments you will have; this is called a *variadic function*. This "uncertain argument list" is represented by ellipses (...). Defining a variadic function is significantly more complicated than a plain function. You can use a variadic function declaration for a function that has a fixed set of arguments if (for some reason) you want to disable the error checks of function prototyping. Handling variadic argument lists is described in the library section of your local ANSI C guide.

Function Return Values

A function prototype may also specify the return value of a function. The type of this value precedes the function name. If no type is given, the return value type defaults to **int** (most things in C default to **int**). If you want to specify that no value is returned, as in a Pascal procedure, the **void** keyword is used. This will generate an error if you try to return a value from the function. Here are some complete function prototypes:

```
foo1(void); // returns an int, takes no arguments
foo2(); // like foo2() in C++ but not in ANSI C!
float foo3(float, int, char, double); // returns a float
void foo4(void); // takes no arguments, returns nothing
```

At this point, you may wonder how to specify a return value in the function definition. This is done with the **return** statement. The **return** keyword exits the function back to the point right after the function call. If **return** has an argument, that becomes the return value of the function. You can have more than one **return** statement in a function definition, as in this example:

```
//: RETURN.CPP—Use of "return"
#include <iostream.h>
```

```
char cfunc(const int i) {
    if(i == 0)
        return 'a';
    if(i == 1)
        return 'g';
    if(i == 5)
        return 'z';
    return 'c';
}
```

```
void main() {
    cout << "type an integer: ";
    int val;
    cin >> val;
    cout << cfunc(val) << endl;
}
```

The code in **cfunc()** acts like an **if-else** statement. The **else** is unnecessary because the first **if** that evaluates true causes an exit of the function via the **return** statement. Notice that a function declaration is not necessary because the function definition appears before it is used in **main()**, so the compiler knows about it. Arguments and return values are covered in detail in Chapter 9.

Using the C Function Library

All the functions in your local C function library are available while you are programming in C++. You should look hard at the function library