



Module 817

C++: Object-Oriented C

Aim

After working through this module you should be able to describe the major differences between C and C++, and identify the major (non object-oriented) additions to C that are found in C++

Learning objectives

After working through this module you should be able to:

1. Identify the major differences between C and C++
2. Use streams in C++
3. Create variable definitions in C++
4. Create compound types in C++
5. Use pointers in C++
6. Define and use functions and prototypes in C++
7. Manage parameters in C++
8. Use function name overloading in C++

Content

Transition from C to C++

Equivalent and improved constructs

Enhanced significance of prototypes in C++

Improved use of pointers and parameters in C++

Function overloading

Learning Strategy

Read the printed module and the assigned readings and complete the exercises as requested.

Assessment

Completion of exercises and the CML test at the end of the module.

Introduction

C++ was developed by Bjarne Stroustrup of AT&T to add object-oriented constructs to the C language. Because object-oriented methodology was relatively new at the time, and all existing implementations of object-oriented languages were rather slow and inefficient (most were interpreters), one goal of C++ was to maintain the efficiency of C.

C++ can be viewed as a procedural language with some additional constructs, some to support object-oriented programming and some for improved procedural syntax. A well-written C++ program will reflect elements of both object-oriented programming style and classic procedural programming. C++ is actually an *extensible* language since we can define new types in such a way that they act just like the predefined types of the standard language. C++ is clearly suited to large scale software development, but can equally be used for smaller applications.

The remaining Modules in this sequence (817-819) will assume a thorough knowledge of the C programming language as covered in Modules 813-816. Program development in C++ is virtually identical to the process described in Module 812. Some sections will be devoted to explaining the newer additions to C provided by the ANSI-C standard. As the ANSI-C standard was in development, many of the newer constructs from C++ were included as parts of C itself, so even though C++ is a derivation and extension of C, it would be fair to say that ANSI-C has some of its roots in C++. An example is prototyping, which was actually developed for C++ and later added to C.

The best way to learn C++ is by using it. Almost any valid C program is also a valid C++ program, and in fact the addition of about 12 keywords is the only reason that some C programs will not compile and execute as a C++ program.

The best way to learn C++ is to simply add to your present knowledge and use a few new constructs as you need them for each new program. It would be a mistake to try to use all of the new constructs in your first C++ program. It would be far better to add a few new constructs to your toolkit each day, and use them as needed while you gain experience with their use.

As an illustration of the portability of C to C++, all of the sample programs used in the earlier Modules should compile and execute correctly when compiled as C++ programs with no changes. None of the C++ programs will compile and execute correctly with any C compiler, however, if for no other reason than the use of the new style of C++ comments.

C++ Naming Conventions

There are primarily two standards for naming C++ files, one using the extension CPP and the other using the extension CXX. All sample files use the CPP extension for naming files. If your compiler requires the CXX extension it will be up to you to rename the files. When C++ was in its infancy, header files generally used the extension .HPP, but now .H is almost universally used for all header files. For that reason all header files used in these Modules use this convention.

Additional Reading

Margaret Ellis & Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990. This is the base document for the ANSI-C++ standard. Even though it is the definitive book on C++, it would be difficult for a beginner to learn the language from this book alone.

Objective 1: After working through this section you should be able to identify the major differences between C And C++

Improved comments

Our first C++ program [CONCOM.CPP] includes examples of several new things in C++. We will take the new constructs one at a time, beginning with the comments.

```
#include <iostream.h>
/* This is the stream definition file */
void print_it(const int data_value);
void main()
{
const int START = 3;
// The value of START cannot be changed
const int STOP = 9;
// The value of STOP cannot be changed
volatile int CENTER = 6;
/* The value of CENTER may be changed
by something external to this program */
int index; /* A normal C variable */
for (index = START ; index < STOP ; index++)
print_it(index);
} /* End of program */
void print_it(const int data_value)
{
cout << "The value of the index is " << data_value << "\n";
}
```

A comment in C++ begins with the double slash (`//`), starts anywhere on a line, and runs to the end of that line where it is automatically terminated. The old method of comment definition used with ANSI-C can also be used with C++, but the new method is the preferred method of comment definition because it is impossible to inadvertently comment out several lines of code. This can be done by forgetting to include the end of comment notation when using the older C method of comment notation. Good programming practice would be to use the new method for all comments and reserve the old method for use in commenting out a section of code during debugging, since the two methods can be nested.

CONST and VOLATILE

There are two new keywords used which were not part of the original Kernighan & Ritchie definition of C, but that are part of the ANSI-C standard. The keyword *const* is used to define a constant. In line 6 the constant is of type *int*, it is named *START*, and is initialised to the value 3. The compiler will not allow you to accidentally or purposefully change the value of *START* because it has been declared a constant. If you had another variable named *STARTS*, the system would not allow you to slightly misspell *STARTS* as *START* and accidentally change it. The compiler would give you an error message so you could fix the error. Since it is not permissible to change the value of a constant, it is

imperative that you initialise it when it is declared so it will have a useful value. The compiler does not *require* you to initialise it, however, and will not issue an error message if you do not.

Note that *const* is also used in the function header in line 17 to indicate that the formal parameter named `data_value` is a constant throughout the function. Any attempt to assign a new value to this variable will result in a compile error.

The keyword *volatile* is also part of the ANSI-C standard but was not included in the original Kernighan & Ritchie definition of C. Even though the value of a volatile variable can be changed by you, the programmer, there may be another mechanism by which the value could be changed, such as by an interrupt timer causing the value to be incremented. The compiler needs to know that this value may be changed by some external force when it optimises the code. Note that a constant can also be volatile, which means that you cannot change it, but the system can through some hardware function.

Line 3 illustrates *prototyping* and the modern method of function definition as defined by the ANSI-C standard, which will be discussed in greater detail in a later section of the Module. Prototyping is absolutely required in C++; programs without prototyping will generate linker errors.

The Scope Operator

The next program [SCOPEOP.CPP] illustrates another construct that is unique to C++; there is no corresponding construct in either Kernighan & Ritchie or ANSI-C. The *scope* operator allows access to the global variable named `index` even though there is a local variable of the same name within the main function. The use of the double colon in front of the variable name (in lines 7, 8 and 10) instructs the system that we are interested in using the global variable named `index`, defined in line 2, rather than the local variable defined in line 5.

```
#include <iostream.h>
int index = 13;
void main()
{
float index = 3.1415;
  cout << "The local index value is " << index << "\n";
  cout << "The global index value is " << ::index << "\n";
  ::index = index + 7; // 3 + 7 should result in 10
  cout << "The local index value is " << index << "\n";
  cout << "The global index value is " << ::index << "\n";
}
```

The use of this technique allows access to the global variable for any use. It could be used in calculations, as a function parameter, or for any other purpose. It is not really good programming practice to abuse this

construct, because it could make the code difficult to read. It would be best to use a different variable name instead of reusing this name, but the construct is available to you if you find that you need it sometime. The scope operator allows access to global variables even though hidden by a local variable.

Exercise 1

Describe the difference between *const* and *volatile* when applied to an identifier.

Objective 2: After working through this section you should be able to use streams in C++

The *iostream* library

Our first hint of object-oriented *messaging* appears in the next program [MESSAGE.CPP].

```
#include <iostream.h>
#include <string.h>
void main()
{
int index;
float distance;
char letter;
char name[25];
index = -23;
distance = 12.345;
letter = 'X';
strcpy(name, "John Doe");
cout << "The value of index is " << index << "\n";
cout << "The value of distance is " << distance << "\n";
cout << "The value of letter is " << letter << "\n";
cout << "The value of name is " << name << "\n";
index = 31;
cout << "The decimal value of index is " << dec << index << "\n";
cout << "The octal value of index is " << oct << index << "\n";
cout << "The hex value of index is " << hex << index << "\n";
cout << "The character letter is " << (char)letter << "\n";
cout << "Input a decimal value --> ";
cin >> index;
cout << "The hex value of the input is " << index << "\n";
}
```

In this program we define a few variables and assign values to them for use in the output statements illustrated in lines 13 to 16, and in lines 118 to 22. The new operator *cout* is the output function to the standard device, the monitor, but works a little differently from our familiar `printf()` function, because we do not have to tell the system what type we are outputting.

Like C, C++ has no input or output operations as part of the language itself, but defines the stream library to add input and output functions in a very elegant manner.

The operator `<<` (sometimes called the “put to” operator but more properly called the *insertion* operator) tells the system to output the variable or constant following it, but lets the system decide how to output the data. In line 13, we first tell the system to output the string, which it does by copying characters to the monitor, then we tell it to output the value of `index`. Notice, however, that we fail to tell it what the type is or how to output the value. Since we don't tell the system what the type is, it is up to the system to determine what the type is and to output the value accordingly. After the system finds the correct type, we also leave it up to the system to use the built in default as to how many characters should be used for this output. In this case, we find

that the system uses exactly as many as needed to output the data, with no leading or trailing blanks, which is fine for this output. Finally, the new line character is output, and the line of code is terminated with a semicolon.

When we called the `cout` output function in line 13, we actually called two different functions because we used it to output a string and a variable of type `int`. This is the first hint at object-oriented programming because we simply broadcast a *message* to the system to print out a value, and let the system find an appropriate function to do so. We are not required to tell the system exactly how to output the data, we only tell it to output it. This is a very weak example of object-oriented programming, and we will get into it in much more depth later.

In line 14 we tell the system to output a different string, followed by a floating point number, and another string of one character, the new line character. In this case, we told it to output a floating point number without telling it that it was a floating point number, once again letting the system choose the appropriate output means based on its type. We did lose a bit of control in the transaction, however, because we had no control over how many significant digits to print before or after the decimal point. We chose to let the system decide how to format the output data. The variable named `letter` is of type `char`, and is assigned the value of the uppercase `X` in line 11, then printed as a letter in line 15.

Because C++ has several other operators and functions available with streams, you have complete flexibility in the use of the stream output functions. You should refer to your compiler documentation for details of other available formatting commands. The `cout` and the `printf()` statements can be mixed in any way you want, as both statements result in output to the monitor.

The Stream library

The stream library was defined for use with C++ for increased execution efficiency. The `printf()` function was defined early in the development C and is meant to be all things to all programmers. As a result, it became a huge function with lots of extra baggage that is only used by a few programmers. By defining the smaller special purpose stream library, C++ allows the programmer to use a limited set of formatting capabilities, but which are still adequate for most programming jobs. If more elaborate formatting capabilities are required, the complete `printf()` library is available within any C++ program, and the two types of outputs can be freely mixed.

Lines 18 to 24 illustrate some of the additional features of the stream library which can be used to output data in a very flexible yet controlled

format. The value of `index` is printed out in decimal, octal, and hexadecimal format. When one of the special stream operators, `dec`, `oct`, or `hex`, is output, all successive output will be in that number base. Looking ahead to line 24, we find the value of `index` printed in hex format due to the selection of the hexadecimal base in line 20. If none of these special stream operators are output, the system defaults to decimal format.

The CIN operator

In addition to the `cout` operator, there is a `cin` operator which is used to read data from the standard input device, usually the keyboard. The `cin` operator uses the `>>` operator, usually called the ‘get from’ operator but properly called the *extraction* operator. It has most of the flexibility of the `cout` operator. A brief example of the use of the `cin` operator is given in line 23. The special stream operators, `dec`, `oct`, and `hex`, also select the number base for the `cin` stream separately from the `cout` stream. If none is specified, the input stream also defaults to decimal.

In addition to the `cout` operator and the `cin` operator there is one more standard operator, the `cerr`, which is used to output to the error handling device. This device cannot be redirected to a file as the output to the `cout` can be. The three operators (`cout`, `cin`, and `cerr`) correspond to the `stdout`, the `stdin`, and the `stderr` stream pointers of the programming language C.

File stream operations

The next example program [FSTREAM.CPP] includes examples of the use of streams with files.

```
#include <iostream.h>
#include <fstream.h>
#include <process.h>
void main()
{
    ifstream infile;
    ofstream outfile;
    ofstream printer;
    char filename[20];
    cout << "Enter the desired file to copy ----> ";
    cin >> filename;
    infile.open(filename, ios::nocreate);
    if (!infile) {
        cout << "Input file cannot be opened.\n";
        exit(1);
    }
    outfile.open("copy");
    if (!outfile) {
        cout << "Output file cannot be opened.\n";
        exit(1);
    }
    printer.open("PRN");
    if (!printer) {
        cout << "There is a problem with the printer.\n";
    }
}
```

```
        exit(1);
    }
    cout << "All three files have been opened.\n";
char one_char;
printer << "This is the beginning of the printed copy.\n\n";
while (infile.get(one_char)) {
    outfile.put(one_char);
    printer.put(one_char);
}
printer << "\n\nThis is the end of the printed copy.\n";
infile.close();
outfile.close();
printer.close();
}
```

A file is opened for reading, another for writing, and a third stream is opened to the printer to illustrate the semantics of stream operations on a file. The only difference between the streams in the last program and the streams in this program is the fact that in the last program, the streams were already opened for us by the system. You will note that the stream named printer is used in the same way we used the cout operator in the last program. Finally, because we wish to exercise good programming practice, we close all of the files we have opened prior to ending the program.

The standard file I/O library is available with ANSI-C; it is as easy to use as the stream library and very portable. For more information on the stream file I/O library, see Stroustrup's book or refer to your compiler documentation. When you compile and execute this program it will request a file to be copied; you can enter the name of any ASCII file in the current directory.

Exercise 2

Write a C++ program to read a name and address from the keyboard, and write them ten times on the monitor, using the cin and cout operators.

Write a C++ program to copy any text file (requested by the user) by using file streams.

Objective 3: After working through this section you should be able to create variable definitions in C++

The following program [VARDEF.CPP] has a few more additions to the C++ language which aid in writing a clear and easy-to-understand program.

```
#include <iostream.h>
int index;
void main()
{
int stuff;
int &another_stuff = stuff; // A synonym for stuff
stuff = index + 14; //index was initialised to zero
cout << "stuff has the value " << stuff << "\n";
stuff = 17;
cout << "another_stuff has the value " << another_stuff << "\n";
int more_stuff = 13; //not automatically initialised
cout << "more_stuff has the value " << more_stuff << "\n";
for (int count = 3;count < 8;count++) {
cout << "count has the value " << count << "\n";
char count2 = count + 65;
cout << "count2 has the value " << count2 << "\n";
}
static unsigned goofy; //automatically initialised to zero
cout << "goofy has the value " << goofy << "\n";
}
```

In C++, as in ANSI-C, global and static variables are automatically initialised to zero when they are declared. The variables named `index` in line 2, and `goofy` in line 18 are therefore automatically initialised to zero.

Of course, you can still initialise either to some other value if you so desire. Global variables are sometimes called *external* since they are external to any functions. *Automatic* variables - those declared inside of any function - are not automatically initialised but will contain the value that happens to be in the location where they are defined, which must be considered a garbage value. The variable named `stuff` in line 5, therefore, does not contain a valid value, but some garbage value which should not be used for any meaningful purpose. In line 7 it is assigned a value based on the initialised value of `index` and it is then displayed on the monitor for your examination.

The reference variable

The ampersand (&) in line 6 defines `another_stuff` as a *reference* variable, which is a new addition to C++. In fact, the reference variable should not be used very often, if at all, in this context. In order to be complete however, we will discuss its operation.

The reference variable is not quite the same as any other variable because it operates like a self dereferencing pointer. Following its

initialisation, the reference variable becomes a synonym for the variable `stuff`, and changing the value of `stuff` will change the value of `another_stuff` because they are both actually referring to the same variable. The synonym can be used to access the value of the variable for any legal purpose in the language. It should be pointed out that a reference variable must be initialised to reference some other variable when it is declared or the compiler will respond with an error. Following initialisation, the reference variable cannot be changed to refer to a different variable.

The use of the reference variable in this way can lead to very confusing code, but it has another use where it can make the code very clear and easy to understand. We will study this use later.

Definitions as executable statements

Anywhere it is legal to put an executable statement, it is also legal to declare a new variable, because a data declaration is defined as an executable statement in C++. In line 11 we define the new variable named `more_stuff` and initialise it to the value of 13. It has a scope from the point where it was defined to the end of the block in which it is defined, so it is valid throughout the remainder of the main program. The variable named `goofy` is declared even later in line 18.

It is very significant that the variable is declared near its point of usage. This makes it easier to see just what the variable is used for, since it has a much more restricted scope of validity. When you are debugging a program, it is convenient if the variable declaration is located in close proximity to where you are debugging the code.

DEFINITION and DECLARATION

The words *definition* and *declaration* refer to two different things in ANSI-C and C++. A declaration provides information to the compiler about the characteristics of something such as a type or a function but it does not actually define any code to be used in the executable program, and you are permitted to make as many declarations of the same entity as you desire. A definition, on the other hand, actually defines something that will exist in the executable program, either some useful variables, or some executable code, and you are required to have one and only one definition of each entity in the program. In short, a declaration introduces a name into the program and a definition introduces some code.

If we declare a struct, we are only declaring a pattern to tell the compiler how to store data later when we define one or more variables of that type. But when we define some variables of that type, we are actually declaring their names for use by the compiler, and defining a

storage location to store the values of the variables. Therefore, when we define a variable, we are actually declaring it and defining it at the same time.

We will refer to these definitions many times throughout the course of this tutorial.

A better FOR loop

The for loop defined in line 13 is a little clearer than the for loop in ANSI-C, because the loop index is defined in the for loop itself. The scope of this loop index is from its declaration to the end of the enclosing block. In this case its scope extends to line 17 since the closing brace in line 17 corresponds to the most recent opening brace prior to the declaration of the variable. Since the variable is still available, it can be used for another loop index or for any other purpose which an integer type variable can legally be used for. The variable named `count2` is declared and initialised during each pass through the loop because it is declared within the block controlled by the for loop. Its scope is only the extent of the loop so that it is automatically deallocated each time the loop is completed. It is therefore declared, initialised, used and deallocated five times, once for each pass through the loop.

You will notice that the variable `count2` is assigned a numerical value in line 15 but when it is printed out, a character value is actually output. This is because C++ (version 2.0 and later) is careful to use the correct type.

Finally, as mentioned earlier, the static variable named `goofy` is declared and automatically initialised to zero in line 18. Its scope is from the point of its declaration to the end of the block in which it is declared, line 20.

Operator precedence

Operator precedence is identical to that defined for ANSI-C, but there is a small difference when some operators are *overloaded* which we will learn to do later in this Module. Some of the operators act slightly different when overloaded than the way they operate with elements of the predefined language.

Exercise 3

Describe the difference between *definition* and *declaration* in C++.

Objective 4: After working through this section you should be able to create compound types in C++

Examine the next program [ENUM.CPP] for an example of the enumerated type variable. The enumerated type is used in C++ in exactly the same way it is used in ANSI-C with one small exception: the keyword *enum* is not required to be used again when defining a variable of that type, but it can be used if desired. It may be clearer to you to use the keyword when defining a variable in the same manner that it is required to be used in C, and you may choose to do so.

```
#include <iostream.h>
enum game_result {win, lose, tie, cancel};
void main()
{
game_result result;
enum game_result omit = cancel;
for (result = win; result <= cancel; result++) {
if (result == omit)
cout << "The game was cancelled\n";
else {
cout << "The game was played ";
if (result == win)
cout << "and we won!";
if (result == lose)
cout << "and we lost.";
cout << "\n";
}
}
}
```

The example program uses the keyword `enum` in line 6, but omits it in line 5 to show that it is indeed optional.

A Simple Structure

The next program [STRUCTUR.CPP] illustrates how to use a very simple structure. This structure is no different from that used in ANSI-C except for the fact that the keyword `struct` is not required to be used again when defining a variable of that type.

```
#include <iostream.h>
struct animal {
int weight;
int feet;
};
void main()
{
animal dog1, dog2, chicken;
animal cat1;
struct animal cat2;
dog1.weight = 15;
dog2.weight = 37;
chicken.weight = 3;
dog1.feet = 4;
dog2.feet = 4;
chicken.feet = 2;
cout << "The weight of dog1 is " << dog1.weight << "\n";
cout << "The weight of dog2 is " << dog2.weight << "\n";
}
```

```
cout << "The weight of chicken is " << chicken.weight << "\n";
}
```

Lines 8 and 9 illustrate the declaration of variables without the keyword, and line 10 indicates that the keyword *struct* can be included if desired. It is up to you to choose which style you prefer to use in your C++ programs.

You should take note of the fact that this is a valid ANSI-C program except for the fact that it uses the stream library, the C++ comments, and the absence of the keyword *struct* in two of the lines. Compile and execute this program after studying it carefully, because the next example program is very similar but introduces a construct not available in standard C, the *class*.

A Simple Class

Our first example of a *class* in C++ is to be found in the next program [CLASS1.CPP]. The class is the major reason for using C++ over ANSI-C (or perhaps some other programming language).

```
#include <iostream.h>
class animal {
public:
    int weight;
    int feet;
};
void main()
{
    animal dog1, dog2, chicken;
    animal cat1;
    class animal cat2;
    dog1.weight = 15;
    dog2.weight = 37;
    chicken.weight = 3;
    dog1.feet = 4;
    dog2.feet = 4;
    chicken.feet = 2;
    cout << "The weight of dog1 is " << dog1.weight << "\n";
    cout << "The weight of dog2 is " << dog2.weight << "\n";
    cout << "The weight of chicken is " << chicken.weight << "\n";
}
```

You will notice the keyword *class* is used in line 2, in exactly the same way that the keyword *struct* was used in the last program, and they are in fact very similar constructs. There are definite differences, as we will see, but for the present time we will be concerned more with their similarities.

The word *animal* in line 2 is the name of the class, and when we declare variables of this type in lines 9 to 11, we can either omit the keyword *class* or include it if we desire as illustrated in line 11. In the last program, we declared 5 variables of a structure type, but in this program we declare 5 *objects*. They are called objects because they are of a class type. The differences are subtle, and in this case the

differences are negligible, but as we proceed we will see that the class construct is important and valuable. The class was introduced here only to give you a glimpse of what is to come later.

The class is a type which can be used to declare objects in much the same way that a structure is a type that can be used to declare variables. Your dog named King is a specific instance of the general class of dogs, and in a similar manner, an object is a specific instance of a class. It would be well to take note of the fact that the class is such a generalised concept that there will be libraries of prewritten classes available in the marketplace soon. You will be able to purchase classes which will perform some generalised operations such as managing stacks, queues, or lists, sorting data, managing windows, etc. This is because of the generality and flexibility of the class construct. In fact, a few class libraries are available now.

The new keyword `public` in line 3, followed by a colon, is necessary in this case because the variables in a class default to a private type and we could not access them at all without making them public.

Free UNION

The next program [UNIONEX.CPP] is example of a *free union*. In ANSI-C, all unions must be named in order to be used, but this is not so in C++.

```
#include <iostream.h>
struct aircraft {
    int wingspan;
    int passengers;
    union {
        float fuel_load;    // for fighters
        float bomb_load;    // for bombers
        int pallets;        // for transports
    };
} fighter, bomber, transport;
void main()
{
    fighter.wingspan = 40;
    fighter.passengers = 1;
    fighter.fuel_load = 12000.0;
    bomber.wingspan = 90;
    bomber.passengers = 12;
    bomber.bomb_load = 14000.0;
    transport.wingspan = 106;
    transport.passengers = 4;
    transport.pallets = 42;
    transport.fuel_load = 18000.0;
    fighter.pallets = 4;
    cout << "The fighter carries "
         << fighter.pallets << " pallets.\n";
    cout << "The bomber bomb load is " << bomber.bomb_load << "\n";
}
```

In C++ we can use a *free union*, which is a union without a name. The union is embedded within a simple structure and you will notice that

there is not a variable name following the declaration of the union in line 9. In ANSI-C we would have to name the union and give a triple name (three names dotted together) to access the members. Since it is a free union, there is no union name, and the variables are accessed with only a doubly dotted name.

You will recall that a union causes all the data contained within the union to be stored in the same physical memory locations, such that only one variable is actually available at a time. This is exactly what is happening here. The variable named `fuel_load`, `bomb_load`, and `pallets` are stored in the same physical memory locations and it is up to the programmer to keep track of which variable is stored there at any given time. You will notice that the transport is assigned a value for `pallets` in line 21, then a value for `fuel_load` in line 22. When the value for `fuel_load` is assigned, the value for `pallets` is corrupted and is no longer available since it was stored where `fuel_load` is currently stored. The observant student will notice that this is exactly the way the union is used in ANSI-C except for the way components are named.

C++ Type Conversions

Examine the next program [TYPECONV.CPP] for a few examples of type conversions in C++. The type conversions are done in C++ in exactly the same manner as they are done in ANSI-C, but C++ gives you another way of doing the conversions.

```
#include <iostream.h>
void main()
{
  int a = 2;
  float x = 17.1, y = 8.95, z;
  char c;
  c = (char)a + (char)x;
  c = (char)(a + (int)x);
  c = (char)(a + x);
  c = a + x;
  z = (float)((int)x * (int)y);
  z = (float)((int)x * (int)y);
  z = (float)((int)(x * y));
  z = x * y;
  c = char(a) + char(x);
  c = char(a + int(x));
  c = char(a + x);
  c = a + x;
  z = float(int(x) * int(y));
  z = float(int(x) * int(y));
  z = float(int(x * y));
  z = x * y;
}
```

Lines 8 to 15 of this program use the familiar ‘cast’ form of type conversions used in ANSI-C, and there is nothing new to the experienced C programmer. You will notice that lines 8 to 11 are actually all identical to each other. The only difference is that we are coercing the compiler to do the indicated type conversions prior to

doing the addition and the assignment in some of the statements. In line 10 the int type variable will be converted to type float prior to the addition, then the resulting float will be converted to type char prior to being assigned to the variable c. Additional examples of type coercion are given in lines 11 to 14 and all four of these lines are essentially the same.

The examples given in lines 15 to 18 are unique to C++ and are not valid in ANSI-C. In these lines the type coercions are written as though they are function calls instead of the 'cast' method illustrated earlier. Lines 15 to 18 are identical to lines 7 to 10.

You may find this method of type coercion to be clearer and easier to understand than the cast method and in C++ you are free to use either, or to mix them if you so desire, but your code will be very difficult to read if you mix them indiscriminately.

Exercise 4

Modify the ENUM.CPP program to include a *forfeit* type, including appropriate changes to program logic and messages. [Note: in this — and all subsequent exercises the involve modifying one of the example programs — you should either work on a second copy of the source code disk, or copy and rename the file to be modified first. In other words, leave the original intact for comparison purposes.]

Modify CLASS1.CPP by extending the class to include an identifier for height, of type float, and to display some of the results.

Objective 5: After working through this section you should be able to use pointers in C++

Because pointers are so important in C and C++, this section will review some of the more important topics concerning pointers. Even if you are extremely conversant in the use of pointers, you should not completely ignore this section because some new material (unique to C++) is presented here.

Pointer review

Look at the next program [POINTERS.CPP] for a simple example of the use of pointers. This is a pointer review and if you are comfortable with the use of pointers from earlier Modules you can skip this example program completely.

```
#include <iostream.h>
void main()
{
  int   *pt_int;
  float *pt_float;
  int   pig = 7, dog = 27;
  float x = 1.2345, y = 32.14;
  void *general;
  pt_int = &pig;
  *pt_int += dog;
  cout << "Pig now has the value of " << *pt_int << "\n";
  general = pt_int;
  pt_float = &x;
  y += 5 * (*pt_float);
  cout << "y now has the value of " << y << "\n";
  general = pt_float;
  const char *name1 = "John";    // Value cannot be changed
  char *const name2 = "John";    // Pointer cannot be changed
}
```

A pointer in ANSI-C or C++ is declared with an asterisk preceding the variable name. The pointer is then a pointer to a variable of that one specific type and should not be used with variables of other types. Thus `pt_int` is a pointer to an integer type variable and should not be used with any other type. Of course, any experienced C programmer knows that it is simple to coerce the pointer to be used with some other type by using a cast, but she must assume the responsibility for its correct usage.

In line 9 the pointer named `pt_int` is assigned the address of the variable named `pig` and line 10 uses the pointer named `pt_int` to add the value of `dog` to the value of `pig` because the asterisk dereferences the pointer in exactly the same manner as standard C. The address is used to print out the value of the variable `pig` in line 11 illustrating the use of a pointer with the stream output operator `cout`. Likewise, the pointer to float named `pt_float` is assigned the address of `x`, then used in a trivial calculation in line 14.

If you are not completely comfortable with this trivial program using pointers, you should review the use of pointers earlier in this tutorial before proceeding, because we will assume that you have a thorough knowledge of pointers from now on. It is not possible to write a C program of any significant size or complexity without the use of pointers.

Constant pointers and pointers to constants

The definition of C++ allows a pointer to a constant to be defined such that the value to which the pointer points cannot be changed but the pointer itself can be moved to another variable or constant. The method of defining a pointer to a constant is illustrated in line 17. In addition to a pointer to a constant, you can also declare a constant pointer, one that cannot be changed; line 18 illustrates this. Note that neither of these pointers are used in illustrative code.

Either of these constructs can be used to provide additional compile time checking and improve the quality of your code. If you know a pointer will never be moved due to its nature, you should define it as a constant pointer. If you know that a value will not be changed, it can be defined as a constant and the compiler will tell you if you ever inadvertently attempt to change it.

Pointers to void

The pointer to void is actually a part of the ANSI-C standard but widespread use of it is relatively new. A pointer to void can be assigned the value of any other pointer type. You will notice that the pointer to void named `general` is assigned an address of an `int` type in line 12 and the address of a `float` type in line 16 with no cast and no complaints from the compiler. This is a relatively new concept in C and C++. It allows a programmer to define a pointer that can be used to point to many different kinds of things to transfer information around within a program. A good example is the `malloc()` function which returns a pointer to void. This pointer can be assigned to point to any entity, thus transferring the returned pointer to the correct type.

A pointer to void is aligned in memory in such a way that it can be used with any of the simple predefined types available in C++, or in ANSI-C for that matter. They will also align with any compound types the user can define since compound types are composed of the simpler types.

Dynamic allocation and deallocation

The next program [NEWDEL.CPP] contains the first example of the *new* and *delete* operators. These perform dynamic allocation and

deallocation in much the same manner that malloc() and free() do in a C implementation.

```
#include <iostream.h>
struct date {
    int month;
    int day;
    int year;
};
void main()
{
    int index, *point1, *point2;
    point1 = &index;
    *point1 = 77;
    point2 = new int;
    *point2 = 173;
    cout << "The values are " << index << " " <<
        *point1 << " " << *point2 << "\n";
    point1 = new int;
    point2 = point1;
    *point1 = 999;
    cout << "The values are " << index << " " <<
        *point1 << " " << *point2 << "\n";
    delete point1;
    float *float_point1, *float_point2 = new float;
    float_point1 = new float;
    *float_point2 = 3.14159;
    *float_point1 = 2.4 * (*float_point2);
    delete float_point2;
    delete float_point1;
    date *date_point;
    date_point = new date;
    date_point->month = 10;
    date_point->day = 18;
    date_point->year = 1938;
    cout << date_point->month << "/" << date_point->day << "/" <<
        date_point->year << "\n";
    delete date_point;
    char *c_point;
    c_point = new char[37];
    delete c_point;
    c_point = new char[sizeof(date) + 133];
    delete c_point;
}
```

During the design of C++ it was felt that, since dynamic allocation and deallocation are such a heavily used part of the C programming language and would also be heavily used in C++, they should be a part of the language, rather than a library add-on. The new and delete operators are actually a part of the C++ language and are operators, much like the addition operator or the assignment operator. They are therefore very efficient and easy to use.

Lines 10 and 11 illustrate the use of pointers in the tradition of C and line 12 illustrates the use of the new operator. This operator requires one modifier which must be a type as illustrated here. The pointer named point2 is now pointing at the dynamically allocated integer variable which exists on the heap, and can be used in the same way that any dynamically allocated variable is used in ANSI-C. Line 14 illustrates displaying the value on the monitor which was assigned in line 13.

Line 16 allocates another new variable and line 17 causes point2 to refer to the same dynamically allocated variable as point1 is pointing to. In this case, the reference to the variable that point2 was previously pointing to has been lost and it can never be used or deallocated. It is lost on the heap until we return to the operating system when it will be reclaimed for further use, so this is obviously not good practice. Note that point1 is deallocated with the delete operator in line 21, and point2 can not actually be deleted. Since the pointer point1 itself is not changed, it is actually still pointing to the original data on the heap. This data could probably be referred to again using point1, but it would be terrible programming practice since you have no guarantee what the system will do with the pointer or the data. The data storage is returned to the free list to be allocated in a subsequent call, and will soon be reused in any practical program.

Since the delete operator is defined to do nothing if it is passed a NULL value, it is legal to ask the system to delete the data pointed to by a pointer with the value of NULL, but nothing will actually happen; this is wasted code. The delete operator can only be used to delete data allocated by a new operator. If the delete is used with any other kind of data, the operation is undefined and anything can happen. According to the ANSI standard, even a system crash is a legal result of this illegal operation, and can be defined as such by the compiler writer.

In line 22 we declare some floating point variables; in C++ the variables do not have to be declared at the beginning of a block. A declaration is an executable statement and can therefore appear anywhere in a list of executable statements. One of the float variables is allocated within the declaration to illustrate that this can be done. Some of the same operations are performed on these float type variables as were done on the int types earlier. Some examples of the use of a structure are given in the last four lines and should be self explanatory.

Finally, since the new operator requires a type to determine the size of the dynamically allocated block, you may wonder how you can allocate a block of arbitrary size. This is possible by using the construct illustrated in line 37 where a block of 37 char sized entities, which will be 37 bytes, is allocated. A block of 133 bytes greater than the size of the date structure is allocated in line 39. It is therefore clear that the new operator can be used with all of the flexibility of the malloc() function which you are familiar with.

The standard functions which you have been using in C for dynamic memory management, malloc(), calloc(), and free(), are also available for use in C++ and can be used in the same manner they were used in C. The new and delete operators should not be intermixed with the older function calls since the results may be unpredictable. If you are updating code with the older function calls, continue to use them for any

additions to the code. If you are designing and coding a new program you should use the newer constructs because they are a built in part of the language rather than an add on and are therefore more efficient.

Pointers to functions

The next program [FUNCPNT.CPP] illustrates the use of a pointer to a function. It must be pointed out that there is nothing new here: the pointer to a function is available in ANSI-C as well as in C++ and works in the manner described here for both languages. But, as it is not regularly used by most C programmers, it is defined here as a sort of refresher. If you are comfortable with the use of pointers to functions, you can skip this discussion entirely.

```
#include <stdio.h>
void print_stuff(float data_to_ignore);
void print_message(float list_this_data);
void print_float(float data_to_print);
void (*function_pointer)(float);
void main()
{
float pi = 3.14159;
float two_pi = 2.0 * pi;
print_stuff(pi);
function_pointer = print_stuff;
function_pointer(pi);
function_pointer = print_message;
function_pointer(two_pi);
function_pointer(13.0);
function_pointer = print_float;
function_pointer(pi);
print_float(pi);
}
void print_stuff(float data_to_ignore)
{
printf("This is the print stuff function.\n");
}
void print_message(float list_this_data)
{
printf("The data to be listed is %f\n", list_this_data);
}
void print_float(float data_to_print)
{
printf("The data to be printed is %f\n", data_to_print);
}
```

There is nothing unusual about this program except for the pointer to a function declared in line 5. This declares a pointer to a function which returns nothing (*void*) and requires a single formal parameter, a float type variable. You will notice that all three of the functions declared in lines 2 to 4 fit this profile and are therefore candidates to be called with this pointer.

Observe that in line 10 we call the function `print_stuff()` with the parameter `pi` and in line 11 we assign the imaginatively-named function pointer named `function_pointer` the value of `print_stuff()` and use the function pointer to call the same function again in line 13. Lines 11 and

13 are therefore identical in what is accomplished because of the pointer assignment in line 11. In lines 15 to 18, a few more illustrations of the use of the function pointer are given. You will be left to study these on your own.

Since we assigned the name of a function to a function pointer, and did not get an assignment error, the name of a function must be a pointer to that function. This is exactly the case. A function name is a pointer to that function, but it is a pointer constant and cannot be changed. This is exactly the case with arrays in ANSI-C; an array name is a pointer constant to the first element of the array.

Since the name of the function is a constant pointer to that function, we can assign the name of the function to a function pointer and use the function pointer to call the function. The only caveat is that the return value and the number and types of parameters must be identical. Most C and C++ compilers will not (and in fact cannot) warn you of type mismatches between the parameter lists when the assignments are made. This is because the assignments are done at runtime when no type information is available to the system, rather than at compile time when all type information is available.

Exercise 5

Modify NEWDEL.CPP by printing out the value of point1 after it has been deleted by duplicating the cout statement just before point1 is deleted. Describe and explain the results.

Modify FUNCPNT.CPP by adding a function that has a single integer as a parameter; attempt to call it by using the function pointer. Describe the result.

Objective 6: After working through this section you should be able to define and use functions and prototypes in C++

This section discusses enhancements to functions in C++. These changes make programming more convenient and permit the compiler to do further checking for errors. A fair amount of time is also spent in this section teaching the modern form of function definition and prototyping.

Prototyping allows the compiler to do additional type checking for your function calls which can detect some programming errors. The first two example programs in this section are designed to teach prototyping and what it will do for you. Prototyping is a relatively new addition to C, so even some experienced C programmers are not familiar with it.

Prototypes

Our first look at a prototype is in the next program [PROTYPE1.CPP]. The prototyping used in C++ is no different to that used in ANSI-C. Actually, many C programmers take a rather dim view of prototyping and seem reluctant to use it, but with C++ it is considerably more important and is in much heavier use. In fact, prototyping is required to be used in most modern C++ compilers.

```
#include <iostream.h>
void do_stuff(int wings, float feet, char eyes);
void main()
{
    int arm = 2;
    float foot = 1000.0;
    char lookers = 2;
    do_stuff(3, 12.0, 4);
    do_stuff(arm, foot, lookers);
}
void do_stuff(int wings, float feet, char eyes)
{
    cout << "There are " << wings << " wings." << "\n";
    cout << "There are " << feet << " feet." << "\n";
    cout << "There are " << (int)eyes << " eyes." << "\n\n";
}
```

A prototype is a limited model of a more complete entity to come later. In this case, the full function is the complete entity to come later and the prototype is illustrated in line 2. The prototype gives a model of the interface to the function that can be used to check the calls to the function for the proper number of parameters and the correct types of parameters. Each call to the function named `do_stuff()` must have exactly three parameters or the compiler will give an error message. In addition to the correct number of parameters, the types must be compatible or the compiler will issue an error message. Notice that when the compiler is working on lines 8 and 9, the type checking can be done based on the prototype in line 2 even though the function itself is

not yet defined. If the prototype is not given, the number of parameters will not be checked, nor will the types of the parameters be checked. Even if you have the wrong number of parameters, you will get an apparently good compile and link, but the program may do some very strange things when it is executed.

To write the prototype, simply copy the header from the function to the beginning of the program and append a semicolon to the end as a signal to the compiler that this is not a function but a prototype. The variable names given in the prototype are optional and act merely as comments to the program reader since they are completely ignored by the compiler. You could replace the variable name `wings` in line 2 with your first name and there would be no difference in compilation. Of course, the next person that had to read your program would be somewhat baffled with your choice of variable names. In this case, the two function calls to this function, given in lines 8 and 9, are correct so no error will be listed during compilation.

Even though we wish to use the `char` type for eyes in the function, we wish to use it as a number rather than as a character. The cast to `int` in line 15 is required to force the printout of the numerical value rather than an ASCII character. The next example program is similar but without the cast to `int`.

Compatible types

Compatible types complete our discussion of prototyping. Compatible types are any simple types that can be converted from one to another in a meaningful way. For example, if you used an integer as the actual parameter and the function was expecting a float type as the formal parameter, the system would do the conversion automatically, without mentioning it to you. This is also true of a float changing to a `char`, or a `char` changing to an `int`. There are definite conversion rules which would be followed. These rules are given in great detail in the ANSI-C standard and are also given in the second edition of the Kernighan and Ritchie.

If we supplied a pointer to an integer as the actual parameter and expected an integer as the formal parameter in the function, the conversion would not be made because they are two entirely different kinds of values. Likewise, a structure would not be converted automatically to a long float, an array, or even to a different kind of structure, they are all incompatible and cannot be converted in any meaningful manner. The entire issue of type compatibility as discussed earlier in the tutorial applies equally well to the compatibility of types when calling a function. Likewise, the type specified as the return type,

in this case void, must be compatible with the expected return type in the calling statement, or the compiler will issue a warning.

This is your chance to try prototyping for yourself and see how well it works and what kinds of error messages you get when you do certain wrong things. Change the actual parameters in line 8 of the program above to read (12.2, 13, 12345) and see what the compiler says about that change. It will probably say nothing because they are all type compatible. If you change it to read (12.0, 13), it will issue a warning or error because there are not enough arguments given.

Likewise you should receive an error message if you change one of the parameters in line 9 to an address by putting an ampersand in front of one of the variable names. Finally, change the first word in line 3 from void to int and see what kind of error message is given. You will first be required to make the function header in line 9 agree with the prototype, then you will find that there is no variable returned from the function. You should get the (correct) impression that prototyping is doing something useful after making these changes.

More prototyping

The next example program [PROTYPE2.CPP] includes a little more information on prototyping.

```
#include <iostream.h>
void do_stuff(int, float, char);
void main()
{
    int arm = 2;
    float foot = 1000.0;
    char lookers = 65;
    do_stuff(3, 12.0, 67);
    do_stuff(arm, foot, lookers);
}
void do_stuff(int wings,    // Number of wings
              float feet,  // Number of feet
              char eyes)   // Number of eyes
{
    cout << "There are " << wings << " wings." << "\n";
    cout << "There are " << feet << " feet." << "\n";
    cout << "There are " << eyes << " eyes." << "\n\n";
}
```

This program is identical to the last one except for a few small changes. The variable names have been omitted from the prototype in line 2 merely as an illustration that they are interpreted as comments by the C++ compiler. The function header is formatted differently to allow for a comment alongside each of the actual parameters. This should make the function header a little more self explanatory. However, you should remember that comments should not be used to replace careful selection of variable names. In this particular case, the comments add essentially nothing to the clarity of the program.

The cost of prototyping

Prototyping is essentially free because it costs absolutely nothing in terms of run time size or speed of execution. Prototyping is a compile time check and slows down the compile time a negligible amount because of the extra checking that the compiler must do. If prototyping finds one error for you that you would have had to find with a debugger, it has more than paid for itself for use in an entire project.

The only price you pay to use prototyping is the extra size of the source files because of the prototypes, and the extra time for the compiler to read the prototypes during the compilation process, but both costs are negligible. When you compile and execute this example program you will find that it is identical to the last example program.

Exercise 6

Modify PROTOTYPE1.CPP by changing the type of the identifier *wings* to float in the prototype only. Describe and explain the resulting compiler message.

Modify PROTOTYPE1.CPP by changing the type of the identifier *wings* to float in both the prototype and the function, and without changing the calls in lines 9 and 10. Describe and explain the results of running the modified program.

Change the three function names in OVERLOAD.CPP to unique names and re-compile. Compare the size of the resulting program (executable) with that of the original OVERLOAD.CPP compiler program.

Objective 7: After working through this section you should be able to manage parameters in C++

Pass by reference

The next file [PASSREF.CPP] includes an example of a *pass by reference*, a construct which is not available in ANSI-C. The reference variable was mentioned earlier and it was recommended there that you don't use it in the manner illustrated there. This example program illustrates a situation where it can be used to your advantage. The pass by reference allows the passing of a variable to a function and returning the changes made in the function to the main program. In ANSI-C the same effect can be seen when a pointer to a variable is passed to a function, but use of a reference variable is a little cleaner.

```
#include <iostream.h>
#include <stdio.h>
void fiddle(int in1, int &in2);
void main()
{
  int count = 7, index = 12;
  cout << "The values are ";
  printf("%3d %3d\n", count, index);
  fiddle(count, index);
  cout << "The values are ";
  printf("%3d %3d\n", count, index);
}
void fiddle(int in1, int &in2)
{
  in1 = in1 + 100;
  in2 = in2 + 100;
  cout << "The values are ";
  printf("%3d %3d\n", in1, in2);
}
```

Observe the prototype in line 3 where the second variable has an ampersand in front of the variable name. The ampersand instructs the compiler to treat this variable just like it were passed a pointer to the variable since the actual variable from the main program will be used in the function. In the function itself, in lines 15 to 18, the variable `in2` is used just like any other variable but we are using the variable passed to this function from the main program not a copy of it. The other variable named `in1` is treated just like any other normal variable in ANSI-C. In effect, the name `in2` is a synonym for the variable named `index` in the main program.

If you prefer to omit the variable names in the prototypes, you would write the prototype as:

```
void fiddle(int, int&);
```

As a Pascal programmer you will recognise that the variable named `in1` is treated just like a normal parameter in a Pascal call, a *call by value*.

The variable named `in2` however, is treated like a variable with the reserved word `VAR` used in front of it, usually referred to as a *call by reference*. The reference variable is actually a self dereferencing pointer which refers to, or points to, the original value.

When you compile and execute this program, you will find that the first variable got changed in the function but was returned to its original value when we returned to the main program. The second variable however, was changed in the function and the new value was reflected back into the variable in the main program which we can see when the values are listed on the monitor.

Default parameters

Our next program [DEFAULT.CPP] has an example of the use of default parameters in C++. This program really looks strange since it contains default values for some of the parameters in the prototype, but these default values are very useful as we will see shortly.

```
#include <iostream.h>
#include <stdio.h>
int get_volume(int length, int width = 2, int height = 3);
void main()
{
    int x = 10, y = 12, z = 15;
    cout << "Some box data is " << get_volume(x, y, z) << "\n";
    cout << "Some box data is " << get_volume(x, y) << "\n";
    cout << "Some box data is " << get_volume(x) << "\n";
    cout << "Some box data is ";
    cout << get_volume(x, 7) << "\n";
    cout << "Some box data is ";
    cout << get_volume(5, 5, 5) << "\n";
}
int get_volume(int length, int width, int height)
{
    printf("%4d %4d %4d  ", length, width, height);
    return length * width * height;
}
```

This prototype says that the first parameter named `length` must be given for each call of this function because a default value is not supplied. The second parameter named `width`, however, is not required to be specified for each call, and if it is not specified, the value 2 will be used for the variable `width` within the function. Likewise, the third parameter is optional, and if it is not specified, the value of 3 will be used for `height` within the function.

In line 7 of this program all three parameters are specified so there is nothing unusual about this call from any other function call we have made. Only two values are specified in line 8 however, so we will use the default value for the third parameter and the system acts as if we called it with `get_value(x, y, 3)` since the default value for the third value is 3. In line 9 we only specified one parameter, which will be used for the first formal parameter, and the other two will be defaulted. The

system will act as if we had called the function with `get_volume(x, 2, 3)`. Note that the output from these three lines is reversed; this will be explained shortly.

There are a few rules which should be obvious but will be stated anyway. Once a parameter is given a default value in the list of formal parameters, all of the remaining must have default values also. It is not possible to leave a hole in the middle of the list, only the trailing values can be defaulted. Of course, the defaulted values must be of the correct types or a compiler error will be issued. The default values can be given in either the prototype or the function header, but not in both. If they are given in both places, the compiler must not only use the default value, but it must carefully check to see that both values are identical. This could further complicate an already very complicated problem, that of writing a C++ compiler.

As a matter of style, it is highly recommended that the default values be given in the prototype rather than in the function. The reason will be obvious when we begin studying the object-oriented structures found in C++.

Scrambled output

When the compiler finds a `cout` statement, the complete line of code is initially scanned from right to left to evaluate any functions, then the data is output field by field from left to right. Thus in line 7 `get_volume()` is evaluated with its internal output displayed first. Then the fields of the `cout` are displayed from left to right with "Some box data is" displayed next. Finally, the result of the return from `get_volume()` is output in `int` format, the type of the returned value. The end result is that the output is not in the expected order when lines 7 to 9 are executed. (The output is not what you would intuitively expect to happen so appears to be a deficiency in the language. Borland International verified that this is operating correctly.)

Lines 10 to 13 are similar to any two of the lines of code in lines 7 to 9, but are each separated into two lines so the output is in the expected order.

Variable number of arguments

The next program [VARARGS.CPP] shows how to use a variable number of arguments in a function call.

```
#include <iostream.h>
#include <stdarg.h>
        // Declare a function with one required parameter
void display_var(int number, ...);
void main()
{
```

```
int index = 5;
int one = 1, two = 2;
display_var(one, index);
display_var(3, index, index + two, index + one);
display_var(two, 7, 3);
}
void display_var(int number, ...)
{
va_list param_pt;
va_start(param_pt,number);          // Call the setup macro
cout << "The parameters are ";
for (int index = 0 ; index < number ; index++)
    cout << va_arg(param_pt,int) << " ";    // Extract a parameter
cout << "\n";
va_end(param_pt);                  // Closing macro
}
```

We have gone to a lot of trouble to get the compiler to help us by carefully checking how many parameters we use in the function calls and checking the types of the parameters. On rare occasion, we may wish to write a function that uses a variable number of parameters. The `printf()` function is a good example of this. ANSI-C has a series of three macros available in the "stdarg.h" header file to allow the use of a variable number of arguments. These are available for use with C++ also, but we need a way to eliminate the strong type checking that is done with all C++ functions. The three dots illustrated in line 4 will do this for us. This prototype says that a single argument of type `int` is required as the first parameter, then no further type checking will be done by the compiler.

You will note that the main program consists of three calls to the function, each with a different number of parameters, and the system does not balk at the differences in the function calls. In fact, you could put as many different types as you desire in the calls. As long as the first one is an `int` type variable, the system will do its best to compile and run it for you. Of course the compiler is ignoring all type checking beyond the first parameter, so it is up to you to make sure you use the correct parameter types in this call.

In this case the first parameter gives the system the number of additional parameters to look for and handle. In this simple program, we simply display the numbers on the monitor to illustrate that they really did get handled properly.

You will realise that using a variable number of arguments in a function call can lead to very obscure code and should be used very little in a production program, but the capability exists if you need it.

Exercise 7

Remove the default value from the prototype for `height` only. Describe and explain the resulting compiler error.

Objective 8: After working through this section you should be able to use function name overloading in C++

The next program [OVERLOAD.CPP] has examples of how function names can be *overloaded*. This is not possible in ANSI-C, but is perfectly legal and in fact used regularly in C++. At first this will seem a bit strange, but it is one of the keystones of object oriented programming. You will soon see its purpose and usefulness.

```
#include <iostream.h>
overload do_stuff; // This is optional
int do_stuff(const int); // This squares an integer
int do_stuff(float); // This triples a float & returns int
float do_stuff(const float, float); // This averages two floats
void main()
{
    int index = 12;
    float length = 14.33;
    float height = 34.33;
    cout << "12 squared is " << do_stuff(index) << "\n";
    cout << "24 squared is " << do_stuff(2 * index) << "\n";
    cout << "Three lengths is " << do_stuff(length) << "\n";
    cout << "Three heights is " << do_stuff(height) << "\n";
    cout << "The average is " << do_stuff(length,height) << "\n";
}
int do_stuff(const int in_value) // This squares an integer
{
    return in_value * in_value;
}
int do_stuff(float in_value) // Triples a float & return
int
{
    return (int)(3.0 * in_value);
}
// This averages two floats
float do_stuff(const float in1, float in2)
{
    return (in1 + in2)/2.0;
}
```

You will notice that there are three functions, in addition to the main function, and all three have the same name. Your first question is likely to be “Which function do you call when you call `do_stuff()`?” That is a valid question and the answer is, the function that has the correct number of formal parameters of the correct types. If `do_stuff()` is called with an integer value or variable as its actual parameter, the function beginning in line 18 will be called and executed. If the single actual parameter is of type float, the function beginning in line 21 will be called, and if two floats are specified, the function beginning in line 26 will be called.

It should be noted that the return type is not used to determine which function will be called. Only the formal parameters are used to determine which overloaded function will be called.

The keyword *overload* used in line 2 tells the system that you really intend to overload the name `do_stuff`, and the overloading is not merely an oversight. This is only required in compilers compatible with C++ version 1.2. C++ version 2.0 does not *require* the keyword *overload*, but its optional use allows the existing body of C++ code to be compatible with newer compilers. It is not necessary to use this keyword because, when overloading is used in C++, it is generally used in a context in which it is obvious that the function name is overloaded.

The actual selection of which function to actually call is done at compile time, not at execution time so the program is not slowed down. If each of the overloaded function names were changed to different names, each being unique, there would be no difference in execution size or speed of the resulting program.

Note the use of the keyword `const` used in some of the function prototypes and headers. Once again, this prevents the programmer from accidentally changing the formal parameter within the function. In a function as short as these, there is no real problem with an accidental assignment. In a real function that you occasionally modify, you could easily forget the original intention of the use of a value and attempt to change it during an extended debugging session.

Exercise 8

Change the three function names in `OVERLOAD.CPP` to unique names and re-compile it. Compare the size of the resulting program (executable) with that of the original `OVERLOAD.CPP` compiler program. Explain the results.