

PROVIDING CLASS FUNCTIONALITY

1. FUNCTION DECLARATIONS

The syntax and importance of function prototyping has been met by those familiar with ANSI C. Function prototyping is almost identical in C++.

NOTE: Function prototypes first appeared in early versions of C++ and were added to C in the late 1980s.

In C++, all functions must be prototyped before they can be called. If you try and call a function without specifying its prototype beforehand, the compiler complains about an error “undefined function”. In C, the compiler would assume an *int* return and argument checking would be suppressed.

Functions in C and C++ are only allowed to return a single value. If the function has no return value, the void keyword should be used. C++ functions can return built-in data types such as *int* or user-defined types such as a class. It is also possible to return pointer values, as in C.

The argument list contains a list of comma separated parameters. Functions in C++ can take built-in data types or user-defined data types. Pointers are often used for greater efficiency.

An empty argument list can be specified in two ways in C++ :

```
void clearscreen (void);
```

```
void clearscreen ();
```

This highlights another difference between C and C++. Both have the same interpretation for the former but the latter is interpreted as an unknown argument list in C.

An alternative to pointers in C++ is that of reference variables, which will be discussed later. For example, if the function was required to receive a *Course* object and modify its contents, the function could be written as follows :

```
// Takes a pointer to a Course object  
void Course_Function ( Course *Course_Pointer);
```

```
// Takes a reference to a Course object  
void Course_Function ( Course &Course_Reference);
```

Variadic functions (using the ... notation) are only used in exceptional circumstances where a function takes different numbers of arguments from one call to the next. The C macros *va_start*, *va_arg* and *va_end* must be used to gain access to these arguments (see *stdarg.h*).

An example is :

```
int printf (char *, ... );
```

NOTE : Many organisations discourage their use because of their specialist nature.

2. DEFAULT ARGUMENTS

C++ allows default values to be specified for some or all of its arguments in a function prototype. If the function is called with fewer than the declared number of arguments, the default values are used instead.

For example, a function *Show ()* is used to show a window on the screen. If most windows have a height of 25 rows and a width of 80 columns, the function can provide these dimensions as default values :

```
void Show ( char * Title, int Height = 25, int Width = 80 );
```

The function can now be called with a differing numbers of values :

```
Show ( "Task Window" ); equates to Show ( "Task Window", 25, 80);
```

```
Show ( "Task Window", 6 ); equates to Show ( "Task Window", 6, 80);
```

```
Show ( "Task Window", 5, 11 ); equates to Show ( "Task Window", 5, 11);
```

If default arguments are defined, the arguments must be arranged so that the default arguments appear at the end of the argument list. The following are illegal :

```
double power (double pow = 1.0, double number );
```

```
FILE *Open_File (char *name, char *mode = "r", int share );
```

These restrictions are designed to prevent function calls such as the following, which were considered hard to read and likely to cause confusion later in the development cycle of the program :

```
power ( , 4.0 );
```

```
Open_File ( "FILE.EXT", , 0);
```

NOTE: Both function calls cause compiler errors.

It is also illegal to redefine a default argument in a subsequent function, even if the same value is being assigned :

```
// First declaration  
void Paint_Window ( int Back = 0, int Fore = 7 );
```

```
//Error : Redefined value of "Back"  
void Paint_Window ( int Back = 1, int Fore );
```

```
// Error : Redefined value of "Fore" even though its the same value  
void Paint_Window ( int Back, int Fore = 7 );
```

Function declarations can add default values for previously unassigned arguments. This allows a function to be extended to take arguments, without having to modify each function call to supply the extra arguments.

```
void Resize ( int x, int y, int height = 10, int width = 40);
```

```
void Resize ( int x, int y = 0, int height, int width );
```

```
void Resize ( int x, int y, int height, int width, int border = 1 );
```

```
Resize ( 2 ); equates to Resize ( 2, 0, 10, 40, 1 );
```

```
Resize ( 0, 0, 25, 80 ); equates to Resize ( 0, 0, 25, 80, 1 );
```

```
Resize ( 0, , , 0 ); ERROR - must supply values.
```

Default arguments should be defined in the prototyping in the header file as follows :

```
//CURSOR.HPP  
  
void Move ( int xstep = 1, int ystep = 1);
```

```
//CURSOR.CPP  
  
#include "cursor.hpp"  
  
void Move ( int xstep, int ystep )  
{  
    // Function Body  
}
```

```
//CLIENT.CPP  
  
#include "cursor.hpp"  
  
void main ( )  
{  
    Move ( 40, 2 );  
}
```

3. FUNCTION OVERLOADING

C++ allows functions to be overloaded i.e. different functions can have the same name, provided each function has an unique signature. In other words, each function must take different numbers of arguments or different data types, so that the compiler can tell them apart.

NOTE : It is not possible to have overloaded functions where only the return types are different. There must be some difference in the argument list.

This is not allowed in C where each and every function must have a different name.

Overloaded functions are designed to make a class easier to understand for the class user. For example consider a *String* class which allows the user to concatenate text at the end of the *String* in one of three ways - append a single character, an array of characters or another *String* object. With function overloading it is possible to write three separate functions called *Append_To_String* (), which cater for these three possibilities (each of the functions have the same logical effect, but has a different algorithm internally).

```
class String
{
public :
    //Append one character - Function 1
    void Append_To_String ( char Append_Char );

    // Append an array of characters - Function 2
    void Append_To_String ( char Char_Array [ ] );

    // Append another String object - Function 3
    void Append_To_String ( String Another_String );

    // Rest of class declaration

};
```

The compiler will decide which of the three *Append_To_String* () functions should be invoked depending on the argument specified in the function call :

String First, Second;

First.Append_String ('A'); // Calls Function 1

First.Append_String ("Message"); // Calls Function 2

First.Append_String (Second); // Calls Function 3

4. RESOLVING OVERLOADED FUNCTION CALLS

When an overloaded function is called, the compiler compares the actual arguments against the formal arguments for each overloaded function, searching for an exact match. If no match exists, standard conversions are attempted between the built-in data types to identify which function is the best match.

Consider the following function overloading :

```
void Func ( int, int );           // Function 1
```

```
void Func ( float, float );      // Function 2
```

```
Func ( 100, 200 );
```

```
Func ( 250F, 10.5F );
```

```
Func ( 100, 'X' );
```

```
Func ( 2.54F, 100 );
```

In the above example, the first call to *Func ()* is unambiguous because the arguments match function 1 exactly. Similarly, the second call to *Func ()* results in an exact match with that of function 2.

In the third and fourth call of *Func ()*, the supplied arguments do not match exactly either of the function prototypes. Thus the compiler tries to achieve a match by type conversion.

The result is :

```
Func ( 100, (int) 'X' );         // Call function 1
```

```
Func ( 2.54F, (float) 100 );    // Call function 2
```

However with this ability to overload functions names comes the possibility of confusing the compiler. The problem exists not when the compiler sees two similar-looking overloaded functions, but when the user tries to call one of these functions. If the compiler cannot work out which of the overloaded functions to call, an ambiguity error exists.

Consider the following example :

```
void Bad ( int, double );       // Function 3
```

```
void Bad ( long, float );       // Function 4
```

```
Bad ( 100L, 2.9 );
```

```
Bad ( 'c', 3 );
```

The first call of the function *Bad ()* results in the possible results of type conversion :

```
Bad ( (int)100L, 2.9 );    // Matches function 3
```

```
Bad ( 100L, (float)2.9 ); // Matches function 4
```

The compiler cannot decide which is the best fit and thus generates an ambiguity error.

A similar situation exists with the second call of *Bad ()*. The possible conversions result in :

```
Bad ( (int)'c', (double) 3 ); // Matches function 3
```

```
Bad ( (long) 'c', (float) 3 ); // Matches function 4
```

Which function does the compiler call ?

Ambiguities can proliferate if function overloading, default arguments and variadic functions are used in combination. The following examples demonstrate the problems :

```
// Ambiguities arising from use of default arguments
```

```
void Move ( int Number, int Direction = 1 ); // Function 1
```

```
void Move ( int Absolute_Column );          // Function 2
```

```
Move ( 1, -1 );                            //Unambiguous - call function 1
```

```
Move ( 2 );                                //Ambiguous - which function call ?
```

```
// Ambiguities arising from use of variadic functions
```

```
void Print ( int Num_Args, ... );          //Function 3
```

```
void Print ( int Record_ID );             //Function 4
```

```
Print ( 5, 10, 20, 30 );                  //Unambiguous - call function 3
```

```
Print ( 100 );                            //Ambiguous - which function call ?
```

The arbitration rules for overloaded functions are extremely intricate and hard to understand. Rather than trying to learn and digest these rules, a better approach is to avoid the problem altogether. Do not use overloaded functions and default arguments in combination.

5. ANONYMOUS ARGUMENTS

Another useful facility in C++ is the ability to leave out the name of the argument in the definition of the function if the argument is not used in the function. This signals to the compiler that the argument is intentionally unused, rather than being an oversight and the 'unreferenced

formal parameter' warning message is not generated.

Anonymous arguments are not a particularly object-oriented feature, but they are useful. For example, a skeleton function can be provided (where the interface to the routine has been decided) for testing until the function is fully completed at a later date (i.e. the functional processing is finalised). The calling program is still obliged to supply the full set of arguments for the function, even though the function might not yet be ready to use the arguments.

```
class Window
{
public :

    void Resize ( int x, int y, int height, int width );

};

//Anonymous arguments to inhibit compiler warning

void Window::Resize ( int, int, int, int )
{
    MessageBox ("Initial version of Window::Resize ");
}
```

They are also useful when a function is expected to require extra parameters in the future and when a particular argument is no longer required, but there are still many calls to the function that the new interface has still to be implemented.

5. SCOPING CONFLICTS

The scoping rules in C++ are a direct extension of the existing rules in C, in which they are relatively simple. There are only two different scopes; that of *local* or *global*. When the compiler sees an identifier inside a function, it first checks to see if there is a local identifier with this name; if not, the compiler checks to see if there is a global identifier with this name. If it does not find either a local or global match, it raises an “undefined identifier” error message.

In C++, the situation is a little more complicated because C++ also has the notion of *class* scope. C++ member functions have three possible scopes that have to be scanned by the compiler when trying to resolve an identifier within the member function.

Local Scope - variables can be declared anywhere in the function and are visible from the point of definition to the end of the enclosing block.

Global Scope - variables are visible to the whole file from the point of definition. If a local variable and global variable have the same name, the global variable is hidden by the local variable.

Class Scope -a member function enters into the scope of its class, meaning that the data members of the class are visible to the member functions belonging to that class.

The compiler first checks to see if there is a local identifier with the specified name. If not, it then looks at the class definition to see if the class has a member with the required name. Finally it checks if the identifier is defined at global scope. Final failure results in an error message as before.

The following example illustrates the various different scopes that are possible :

```
int Total;                                // Global scope (Total)

class Example
{
public :
    void Abs_Sum ( int, int );

private :
    int Element [5];                        // Class scope (Element)
};

void Example::Abs_Sum (int First, int Second ) // Function scope
// (First, Second)
{
    Total = 0;

    for ( int i = First; i < Second; i++) // Function scope ( i)
    {
        int Temp;                        //Block scope (Temp)

        if ( Element [ i ] > 0 )
            Temp = + Element[ i ];
        else
            Temp = - Element[ i ];

        Total += Temp;
    }
}
```

Now consider the following :

```
class Course
{
public :
    void Reserve (int); //Reserve places on course
    int Max_Size (); // Get maximum number of students

private :
    int Count; // Bookings on course
};

long Count; // Total bookings (on all courses)
```

```

void Course::Reserve (int Count)
{
    if ( Count + Count <= Max_Size( ) )
    {
        Count += Count;           //Update global count
        Count += Count;           //Update count for this course
    }
}

```

The example above causes no compilation problems. Unfortunately it does not give the correct results due to there being numerous different items called *Count* that are within scope in the member function *Course::Reserve ()*.

The function receives a parameter called *Count*, which indicates the number of places that the user is trying to reserve on the course. The function attempts to perform a simple calculation that there are sufficient places still available. The expression (*Count + Count*) is intended to add the local *Count* value to the data member *Count*, but this is not what happens. All references to *Count* inside the function will be resolved by compiler as references to the local parameter of that name.

The same problem occurs within the selection body. The intention is to update the *Count* data member and the global variable *Count*, but once again all references are resolved to the local *Count* variable.

This is poor programming style. The class developer controls the naming of class data members and local variables and thus scoping conflicts should not exist.

6. SCOPE OPERATOR

The problems highlighted in the above section can be avoided by the use of the scope operator (::). It acts as a qualifier and tells the compiler where to find the identifier definition.

A better approach is to choose different identifier names but this is not possible when using inheritance.

The scope operator has two modes of operation :

- Unary operator

indicates that the identifier is a global item.

e.g. `::Count` is resolved as a global variable. The compiler ignores any local item(s) or class members with the same name.

NOTE : Some companies have a clause in their C++ coding standards stating that calls to C functions should be qualified with the scope operator to avoid possible confusion.

```
::printf ("Enter the daily rate of pay \n "); // Call a global function.
```

- Binary operator

indicates that the operator on the right (of the scope operator) is a class member. The name of the class is specified on the left of the scope operator.

e.g. `Course::Count` where `Count` is a member of the `Course` class.

The scope operator is also useful for resolving between member functions in a base class and a derived class in an inheritance hierarchy.

7. SUMMARY

Class functions is provided by a set of functions whose prototypes are placed in the class definitions.

The code within a member function has unrestricted access to all the members in the class, including the private members. Within a member function, the compiler resolves the scope of the identifiers in the order of the local scope, the class scope and finally global scope. The scope resolution operator `::` can be used to specify explicitly the scope of the identifier.

Functions in C++ can be overloaded as long as the overloaded functions have different argument lists. The compiler keeps a tight check on the rules when trying to select the appropriate function, and an ambiguity error is raised if no unique function match is found.

Arguments may be given default values in the function prototype, in which case the calling function does not have to specify a value for the argument, unless a different value is required.

Another facility of C++ is the ability to leave out the name of a parameter in the function definition. This signals to the compiler that the argument is not used during the function.