FUNCTIONS : basics, overloading, command line arguments, C and C++, ...

FUNCTION BASICS

Almost all programs use functions to break-up large programs into manageable and reusable chunks of code. Related functions are usually grouped into separate files which are independently compiled. When accessing global variables from a separate file or library, you must use the "extern" keyword followed by the variable declaration to tell the compiler that the particular variable is not part of the file. The following is an example of how to call functions from a separate file with a shared global variable:

```
// file1.cp
#include <iostream.h>

// global variable in external file
extern float result;

// function prototypes (external)
float myInput( char *prompt );
void myMultiply( float a, float b );
void myPrintResult( float result );

main()

   float a,b;

   a = myInput( "Input A: " );
   b = myInput( "Input B: " );
   myMultiply( a, b );
   myPrintResult( result );

   return 0;

// end file1.cp


// file2.cp
#include <iostream.h>

float result;

// function prototypes (local)
float myInput( char *prompt );
void myMultiply( float a, float b );
void myPrintResult( float result );

float myInput( char *prompt )

   float input;

   cout << prompt;
   cin >> input;

   return input;


void myMultiply( float a, float b )

   result = a * b;
```

```
void myPrintResult( float result )

   cout << "The result is ";
   cout << result << endl;

// end file2.cp
```

Although not required, you can insert the extern keyword in front of the function prototypes in 'file1.cp' to identify that these functions are located in an external file. Instead of duplicating the function prototypes in both files as done above, you could place these prototypes in a separate header file, preferably called 'file2.h', and #include this file at the beginning of 'file1.cp' and 'file2.cp'.

FUNCTION OVERLOADING

In C++, you can create more than one function with the same name. However, the parameter list of these "overloaded" functions must differ. When asked to call a function, the compiler will compare the parameters of the called function to the list of functions which share the same name. When a match is determined, the proper function is called.

```
// overload.cp
#include <iostream.h>

void DoAdd( int a, int b );
void DoAdd( float a, float b );

main()

   int i1    =    5;
   int i2    =   10;
   float f1 = 2.5;
   float f2 = 4.1;

   DoAdd( i1, i2 );
   DoAdd( f1, f2 );

   return 0;


void DoAdd( int a, int b )

   int result;

   result = a + b;
   cout << a << " + " << b << " = " << result << endl;


void DoAdd( float a, float b )

   float result;

   result = a + b;
   cout << a << " + " << b << " = " << result << endl;

// end overload.cp
```

COMMAND LINE ARGUMENTS

Although the Macintosh doesn't utilize a command line prompt (at least not yet), you may need to write programs which use command line arguments. The following code should work with most Macintosh compilers:

```
// command.cp
#include <iostream.h>
#include <console.h>                    // compiler-specific

int main( int argc, char *argv[] )   // full prototype for main

   int i;

   argc = ccommand( &argv );            // compiler-specific

   for( i = 1; i < argc; i++ )
      cout << argv[i] << " ";
   cout << endl;

   return 0;

// end command.cp
```

CALLING C FUNCTIONS FROM C++

C++ performs "name mangling" in order to perform such feats as function overloading. Basically, this means that C++ renames your functions behind the scenes. Therefore, if you need to call a C function (this usually means calling a function from an object code library which has been compiled by a straight C compiler), you need to let the C++ compiler know not to perform any name mangling. This is performed by using the following statement:

```
extern "C" function_prototype;
```

If there is a list of functions, you can use:

```
extern "C"

   function_prototype1;
   function_prototype2;
   function_prototype3;
   //...
```

To enclose a complete library, you can use:

```
extern "C"

   #include <header_file.h>
```

DEFAULT ARGUMENT LIST

You can supply default values for function arguments in C++ as follows:

```
// default.cp
#include <iostream.h>

void myPrint( int val = 0, int times = 1 );

main()

   myPrint();
   myPrint(5);
```

```
   myPrint(7,3);

   return 0;


void myPrint( int val, int times )

   for( int i=1; i<=times; i++ )

      if( i==1 )
         cout << "Output Value: " << val << endl;
      else
         cout << "          Copy " << i << ": " << val << endl;


// end default.cp
```

VARIABLE-LENGTH ARGUMENT LIST

Although not recommended, you may need to create a function which has a variable argument list (such as printf and scanf). Here's how to define and use variable-length argument lists in functions:

```
// list.cp
#include <iostream.h>
#include <stdarg.h>

void myPrintList( char *format, ... );   // ellipsis must be last item

main()

   int      int1   = 2;
   int      int2   = 7;
   double real1 = 3.14;

   myPrintList( "List:%d%d%f", int1, int2, real1 );

   return 0;


void myPrintList( char *format, ... )

   va_list argPtr;
   char      *p;
   int       intItem;
   double   realItem;

   va_start( argPtr, format );
   for( p = format; *p; p++ )   // search for end of string

      if( *p != '%' )

         cout << *p; // print char
         continue;     // get next letter

      cout << endl;
      switch( *++p )

         case 'd':
            intItem = va_arg( argPtr, int );
            cout << intItem;
```

```
            break;
         case 'f':
            realItem = va_arg( argPtr, double );
            cout << realItem;
            break;
         default:
            cout << "%" << *p;
            break;


   va_end( argPtr ); // clean up

// end list.cp
```