Lecture 4: Overloading, Polymorphism and Virtual Functions

Introduction

Before we look at Polymorphism and its C++ association with classes we are first going to look at function overloading before moving on.

Overloading Functions

We have seen how to code a function where have a specific parameter list and a single return type. This does not allow for versatility. If we want a more versatile function, for instance one that can accept a different set of arguments. We may want a function that performs a 3D transformation on a set of vertices that are represented by floats; we may also want the function to work with ints as well. Rather than writing two separate functions with different names we could use function overloading so that a single function name could handle the different parameter lists. This way you can call one function and pass vertices as either floats or ints.

Coding the Overloaded Functions

To create an overloaded function we simply write the function definitions with the same name and different parameter lists.

```
int tripleItem(int number);
string tripleItem(string text);



int tripleItem(int number)
{
      return (number * 3);
}


string tripleItem(string text)
{
      return (text + text + text);
}
```

To implement function overloading you need to implement different function definitions – the parameter list needs to be different. If just the return type is different you will generate compile errors.

Calling Overloaded Functions

You can call an overloaded function the same way you call any other function. You use its name with valid arguments correctly. The compiler handles the rest, it determines, based on the function arguments, which definition to invoke.

For instance if we pass an `int` as an argument to the `tripleItem` function call then the compiler invokes the function definition that takes and returns an `int`.

Inlining Functions
Is basically away to make function calls quicker, when it can be used however must be carefully monitored.

When you call a function there is a small performance cost associated with it. Please note the word “small”. For tiny functions it may be possible to speed up program performance by inlining them.

Inlining a function causes the compiler to make a copy of the function everywhere its called. This means that the flow of control doesn’t need to jump to a different location each time the function is called. This should not be used for large functions as it can then have a performance related cost due to increase in memory consumption. Inlining should be used carefully.

Specifying a Function for Inlining
To specify a function for inlining you simply put the keyword `inline` before the function definition. You do not need to change the functions declaration.

```
int addtwonumbers(int x, int y);    // function declaration

//function definition
inline int addtwonumbers(int x, int y)
{
      Return (x + y);
}
```

Calling an Inlined Function
Calling an inlined function is no different than calling a non-inlined function is no different than calling an inlined function.

```
int number = addtwonumbers(5 + 7);
```

Assuming that the compiler grants the request for inlining, it sometimes may not like it, this code no longer results in a function call. Instead the compiler places the code to add the two numbers together right in the code.

Notes on Inlining
When you inline a function you make a request to the compiler which has the ultimate decision on whether or not to inline the function. If the compiler thinks inlining wont boost performance then it often wont inline the function.

Performance Boosts and Games
Games use a lot of resources and because of this many games programmers are obsessive about performance. There is a danger in being to performance obsessed – it can cause you to ignore good design that maybe a bit slower. To deal with this many games developers make sure there game works well before they then tweak for performance. Many developers analyse often by running a

utility program that profiles where the game program spends most its time/resources. They then go about optimising based on this information.

Friend Functions

A friend function is a function that is not a member of a class but has access to the class's private and protected members. Friend functions are not considered class members; they are normal external functions that are given special access privileges. Friends are not in the classes scope, and they are not called using the member-selection operators (. and ->) unless they are members of another class. A friend function is declared by the class that is granting access. The friend declaration can be placed anywhere in the class declaration. It is not affected by the access control keywords.

```
#include <iostream.h>
// Declaration of the function to be made as friend for the C++
// Tutorial sample

class CFriendExample
{
       int private_data;
       friend int AddToFriend(int x);
    public:
       CPP_Tutorial()
      {
          private_data = 5;
       }
};

int AddToFriend(int x);

int main()
{
       cout << "Added Result for this C++ tutorial: "<<
AddToFriend(4)<<endl;
}

int AddToFriend(int x)
{
       private_data += x;
       return private_data;
}
```

Creating a Friend Function

A friend function can access any member of a class of which it is a friend. You specify that a function is a friend of a class by listing the function prototype proceeded by the keyword `friend` inside the class definition.
That's what is happening inside the `CFriendExample` class:

```
friend int AddToFriend(int x);
```

This means that `AddToFriend(int x)` can access any member of `CFriendExample` even though its not a member function of the class.
In our example the function `AddToFriend(int x);` takes advantage of this by accessing the private member data `private_data`

Common Use
Perhaps the most common use of friend functions is overloading << and >> for I/O

Overloading Operators
Though this may sound at first like something you may want to avoid it is not. C++ operator overloading is the mechanism by which the language standard operators are used for customised operations of the classes.
For example if we are trying to write a string class, we would very simply ask for "+" operator to handle string concatenation. Or we could overload the * operator so when it's used with 3D matrices (objects instantiated from some class we have defined) the result in the matrices are multiplied.

Using them can help put the users at ease. They need not worry about the internal handling of string operations or memory allocations. Instead they can start writing cleaner code concentrating only on the application functionality. Code Reusability becomes easier. Though the overloaded operators are very cleaner to use, the internals could be a little dirty.

To overload an operator you define a function called `operator` follow directly by the operator you wish to overload: `operator<<, operator+, operator-, etc`

The following code defines an operator for print out the internals of the class and demonstrates how that can be used.

```
ostream& operator<<(ostream& os, const Critter& aCritter)
{
      os << "\nCritter Object - ";
      os << "m_Name: " << aCritter.m_Name << std::endl;

      return os;
}
```

This function overloads the << operator so that when we send Critter objects with the << to cout the data member m_Name is displayed. It allows us to easily display Critter objects.

Dynamically Allocating Memory: New and Delete

Definition: A pool of memory used for dynamic memory allocation. Blocks of memory from this area are allocated for program use during execution using the new operator in C++ and the malloc function in C.

Dynamic storage

Refers to memory allocated and deallocated during program execution using the new operator and delete operator.
Dynamic data items are called dynamic because they are created and deleted during run time. So far we have declared variables and C++ has allocated the necessary memory for it, when the function that the variable was declared in ended C++ freed the memory. This memory in programming terms is known as the *stack*. We as the programmer can be in charge of another portion of memory known as the *heap*. As the programmer you are in charge of allocating and freeing this memory.

Why Use Dynamic storage?

One word: Efficiency.
Dynamic memory usage enables us to be efficient, only allocating when necessary – such as at the beginning of levels or when the user enters a new zone. It gives us as programmer's greater control over what we do.

The Memory Operators

There are two operators used to perform these functions.

new datatype
Used to allocate a dynamic variable.
e.g. *int *TmpPtr = new int;*

delete pointer
Used to deallocate a dynamic variable.
e.g. *delete TmpPtr;*

The new operator is used to create dynamic data variables in the so-called free store, available in memory for this purpose. Free space is also referred to as the heap. (Even guru's have been known to let their hair down sometimes.)

A couple of points to remember when you use the new statement:

You can't directly name a dynamic data variable as you can other regular variables. If you can't name it, then how do you reference it? The answer is that you do this with a pointer.

When the new operation is executed, it returns a pointer to the location of the variable space in the free store.

So let's look again at that example of using new.

int TmpPtr = new int;*

The first part, int* TmpPtr, declares an integer pointer named tmpPtr. The second part, new int, creates a space in the free store, and returns a pointer to that space. The returned pointer is assigned to TmpPtr. This is typical of C++, i.e. you can accomplish a lot in a single line of code.

You can declare arrays in free store as well as simple variables.
e.g. *int* WeightPtr = new int [3];*
To use this dynamic array element you could code:

WeightPtr[1] = 17; // Note: the "*" is not needed in an array reference.

We'll see more of this in the programming exercise for this week's lab exercise. The whole idea of using dynamic data is to economize on memory space. So when you've finished with a piece of dynamic data you should release the space with the delete statement.

delete TmpPtr;
delete [] WeightPtr;

This releases space in the free store, but does not delete the pointer. Be careful here! If you try to use the pointer again, after the delete statement, you don't know what address will be in the pointer. Beware the dreaded segmentation fault, core dump! (See the next section for details.)

To safeguard an inadvertent overwrite of a critical area in memory, it is advisable to set pointers to NULL after you delete the associated dynamic data space. e.g.

TmpPtr = NULL;
WeightPtr = NULL;

Obviously, new and delete operations do not only work with basic types. They do work with structs, unions, and most important, classes. This is one of the more powerful features of C++, the ability of creating and destroying objects dynamically. It is a key feature since it allows virtual functions, dynamic binding and polymorphism among others

The new operator may be used to allocate memory for single objects or arrays.

Advanced Pointers and Memory Issues
This section aims to teach you how to avoid the customary horrors of buffer overruns, memory leaks, and wild pointer errors that led to caffeine binges and marathon programming sessions.
It also aims to introduce to pointers in a more advanced way.

Pointers and Strings

Here we look at a pointer declaration:

char *str_ptr = "MARK";

This declaration has the effect of places the string "MARK" into an array pointed to by *str_ptr*, and automatically appends the null terminator '\0' to the end.
So what is occurs is:
– An array of 4 characters is allocated
– *str_ptr* points to the start of this array

If we then printed out *str_ptr via the following:

```
cout << *str_ptr << endl;
```

We get the following output:

```
M
```

Just the same as if we had used the following:

```
cout << str_ptr[0] << endl;
```

This is because *str_ptr is actually a pointer to the first element of an array; in this case a 1D character arrays with a NULL terminator.

As we have seen above with pointers and arrays to print the whole character array you can do:

```
cout << str_ptr << endl;
```

Void Pointers

A void pointer may be used when we do not wish to be explicit about the type of data being referenced:

```
    void *ptr;
```

Any normal pointer type will automatically convert to a void pointer if required:

```
float number;
void *ptr = &number;
```

The advantage of void pointers is that they may be used to write *generic* functions, which will process data of different types (provided the function is written so as to allow this). The disadvantages are that a void pointer always has to be cast into some specific pointer type before you can de-reference it and use it, and that no type-checking takes place when automatic conversion to void* occurs:

```
void DoSomething(void *ptr)
{
    // cast ptr to a specific pointer type
    // before doing something with it.
}
```

You can pass any type of pointer to this function but you will get unpredictable results (probably a run-time crash) if it's an inappropriate type.

Void pointers are a left-over from the C programming language from which C++ is derived. C++ provides much safer and more effective methods of doing the sort of things that void pointers were used for in C, so you should only have to use void pointers occasionally, usually when calling some functions in legacy code.

Invalid Pointer Values

A pointer value may be invalid, either because it has not been initialised—the most common cause, or because, through later modifications to allocated memory (see the later section on the heap), it does not point to any properly allocated memory location. Invalid pointers are a frequent source of run-time crashes or other incorrect program behaviour. What actually happens may vary from one run of the program to another and may occur some time after the invalid pointer has been used, so these problems can be very hard to track down.

Null pointer values

NULL is a valid pointer value, which may be used to partly avoid the problems of pointer variables having invalid values. A pointer variable of any type, including a void pointer, may be assigned the value NULL when it is not required to address an actual memory location.

These are valid NULL pointer declarations:

```
char *cPtr = 0;     // Initialise to the NULL address
int  *iPtr = NULL;
```

```
int  *kPtr = 0;
```

The NULL (or 0) address is a special value that can be assigned to a pointer to indicate that it is not currently pointing to any particular memory location.

This is **not** the same as the pointer variable being undefined. i.e. having no particular value.

If you attempt to de-reference a pointer with value NULL, you will get the same run-time problems that would occur by de-referencing an invalid pointer value. However, you can test if a pointer value is, or is not NULL with expressions **ptr == NULL** or **ptr != NULL**, and hence can take steps to avoid de-referencing a NULL pointer. This is not possible with a genuinely invalid pointer value. Also, most library functions, that take pointer parameters, check whether or not a pointer value is NULL before attempting to de-reference it, thus avoiding unexpected run-time errors.

The name NULL is usually defined as 0, and you may see 0 used instead of NULL in some legacy code. But, NULL is preferable to 0, if only because it is clearer what it means as a pointer value.

Uses for NULL pointers:

We can use the NULL memory address to see whether the pointer is defined before using it:

```
int  *iPtr = NULL;
// …. pointer manipulation code
 if (iPtr == NULL)
// iPtr is not pointing to a memory location
else
// iPtr is pointing to a memory location
```

Pointer Arguments to functions

In the original C language, function parameters are always passed as value parameters, i.e. a copy of the data is made and the function deals with the copy. This means that in C the function cannot alter the data it was passed. However, if we pass a pointer as a value parameter, the function can de-reference the pointer and modify the data it points to. This gives the same effect as reference parameters in C++, but the notation is a bit less convenient.

Example

The following function swaps the values of two integer variables by passing copies of their addresses as pointer value parameters.

```
    void swap(int *px, int *py)
```

```
{
    int temp = *px;

    *px = *py;
    *py = temp;
}
```

It is often a good idea to prefix the names of pointer variables or parameters by ptr or p, as was done in this function, to remind oneself when one is using pointer variables or parameters.

Allocating memory on the Heap

Definition: A pool of memory used for dynamic memory allocation. Blocks of memory from this area are allocated for program use during execution using the new operator in C++ and the malloc function in C.

Dynamic storage - refers to memory allocated and deallocated during program execution using the new operator and delete operator.

So far we have learnt two ways to allocate memory for variables and arrays.

One is to define variables and arrays globally, outside the body of any function, or qualified by the keyword static inside a function body. In this case the memory for them is allocated when the program starts executing and remains allocated in the same place until the program finishes executing (static allocation). Global variables should only be used rarely, as they give more chance for programming errors. It is also more difficult to reuse functions that reference global variables in other programs.

The second method is to define variables and arrays locally inside the body of a function. In this case the memory for them is allocated when the function starts to execute and is de-allocated when the function returns on completing its execution (automatic allocation). Because the allocation and de-allocation of automatic memory follows an orderly sequence, the memory area set aside for automatic memory allocation is called the stack. Local variables should always be used for values that are only required inside a particular function.

Pointer variables give the means of a third method of allocating and using memory. Consider the following:

```
int   *piVal, iVal;

piVal = new int;
```

The new operator allocates sufficient memory to store a value of the type given as its operand and returns the starting address of this piece of memory as the value of the expression. Thus the above code allocates sufficient memory to store a single int value and returns the address of this int value, which is stored in the piVal pointer variable. This can then be used to reference the memory. For example:

```
*piVal = 123;
// store the value 123 in the memory.
ival = *piVal;
// copy the value 123 into the variable iVal.
```

Note that the piece of memory allocated by new has no name of its own and can only be accessed through the pointer variable in which its address is stored.

Memory allocated by new remains allocated until the programmer de-allocates it with the delete operator:

```
delete piVal;
// de-allocate the memory addressed by piVal.
```

The memory does not have to be de-allocated in the same function in which it was allocated, so the lifetime of the data in memory allocated by new is completely under the control of the programmer. This allows advanced programming techniques, not possible with static or automatic memory allocation, which we may use later.

Also, if we have allocated several pieces of memory with the new operator, we do not have to de-allocate them with delete in the same order, nor in the reverse order to that in which they were allocated. We can delete them in any order we choose. Over a period of time, this leads to an untidy allocation of memory, so the memory area used for this type of memory allocation is called the heap.

We can also use new to allocate memory for arrays. For example:

```
char  *pcName;

pcName  = new char[20];
```

This allocates memory for an array of 20 char values and stores the address of the first byte in the pointer pcName. This technique can be use to dynamically allocate memory for character strings read in, for example, from the keyboard or from a file, without the programmer knowing in advance how many strings there are or how long they are.

To de-allocate array memory from the heap, we must use the delete operator as follows:

```
delete pcName[];
```

Note that you must not put the array size in the brackets with the delete operator. On the other hand, forgetting to put the brackets after the pointer variable when deleting an array from the heap, or putting brackets after a pointer to a single variable on the heap, will lead to unpredictable and incorrect program behaviour.

Memory Leaks

If you do not deallocate memory that you have previously reserved, and the program exits, then the memory will remain allocated – even after the program has ended.

Such a program is said to have a *memory leak*.
If you repeatedly run the program (which allocates memory but does not unallocate it), then more and more of your computers free memory will be reserved. Eventually, your computer will run out of free memory, and the program will not be able to run at all.
Memory leaks are the sign of sloppy or careless work.
Always ensure your programs have no memory leaks.

Safe Memory Management

This next section looks at safe memory management in C++

Use std::string instead of char * or char []

Character arrays are the only way to encapsulate string data in C. They're quick and easy to use, but unfortunately their use can be fraught with peril. Let's look at some of the more common errors that occur with character pointers and arrays. Keep in mind that most if not all of these problems will go undetected by the compiler.

Avoid the headaches associated with character arrays and pointers and use std::string. For legacy functions that expect a character pointer, you can use std::string's c_str() member function.

Use standard containers instead of homegrown containers

Besides std::string, the standard library provides the following container classes that you should prefer over your homegrown alternatives: vector, deque, list, set, multiset, map, multimap, stack, queue, and priority_queue. It is beyond the scope of this article to describe these in detail, however you can probably ascertain what most of them are by their names. For a proper treatment of the subject, I highly recommend the book by Josuttis listed in my references.

An important feature of the standard containers is that they are all template classes. This is a powerful concept. Templates let you define lists (or stacks or vectors) of *any* data type. The compiler generates type-safe code for each type of list you create. With C, you either needed a list for each type of data it would hold (e.g. IntList, MonsterList, StringList) or the list would hold a void * that pointed to data in each node; somewhat the antithesis of type-safety.

In addition to providing generic, type-safe containers for any data type, these classes also provide multiple ways to search and iterate, and like std::string, they manage their own memory - a huge win over rolling these things yourself. I can't stress enough how important it is to familiarize yourself with the standard containers.