# Tute 3: Classes

## Review: Classes

**What is the difference between a class and an object?**

A class defines a data type in a similar way to how a struct does in C, it allows you to store data and provide a set of methods to act on that data. An object is an instance of a class with its own address and space in memory. You can think of a class as a blueprint used to create objects.

**What is method overloading?**

Method overloading involves creating multiple methods with the same name, but accepting different data types as arguments (or different numbers of arguments). The compiler will decide which method to call based on what arguments are supplied. Note that a method cannot be overloaded based on its return type.

**What is a default argument?**

A default argument is a value that is given to a function argument that the compiler will pass to that function when the argument is not supplied by the caller. Note that default arguments should only be supplied in the function declaration, not in the function definition and if a default value is given for an argument, all subsequent arguments must also have default values supplied.

**How are arguments passed to methods?**

Arguments can be passed by reference, by value or by pointer. In the case of pass-by-value, a copy of the original variable is made and the copy is passed to the method, modifying this copy will not modify the original variable. When passing by pointer, a pointer to the variable is passed to the method, modifying the data this points to will modify the original variable. Passing by reference is the same as passing by pointer with the difference being that you cannot change what a reference points to and it cannot be null.

- void method(int i); // pass by value
- void method(int *i); // pass by pointer
- void method(int &i); // pass by reference

**What is auto?**

Note: this is a C++11 feature.

auto is a keyword used for automatic type inference, ie the compiler will decide what type to assign to a variable. There is no loss of performance in using the keyword auto, and the compiler will always choose the most appropriate data type to assign to a variable, ie:

- auto i = 10; // this will be an int
- auto f = 10.0f; // this will be a float

If it is potentially ambiguous as to what data type should be used for a given variable, you should not use the auto keyword. auto cannot be used as an argument in a function declaration, or as a return type (however it can be used as a return type in the next C++ version, C++14).

**What is nullptr?**

Note: this is a C++11 feature.

The nullptr keyword represents the null pointer value (similar to NULL in C). It is used to represent a pointer that does not point to anything. It's main advantage over the use of the NULL keyword is that NULL can be used as an int, whereas nullptr cannot, so in the case of:

- void method(char *c);
- void method(int c);

a call to method(NULL) will actually call method(int c), instead of method(char *c), whereas a call to method(nullptr) will call method(char *c) instead of method(int c).

## Programming Concepts: Structs vs Classes

Look at structs.c and constructors.h/cpp from the Week2 examples folder as a reference.

**What is a struct in C?**

A struct is a custom user defined data type used to aggregate multiple data types together so they can be accessed via a single variable. Data types declared inside a struct are typically stored contiguously in memory in the memory location reserved for the struct, however if the struct contains pointers, the data that these point to will usually be stored elsewhere.

**What is a function?**

A function is a piece of code written to perform a specific task that can be called multiple time. Functions are named, ca have a return type declared (meaning that they must return a variable of that type when finished) and they can accept variables as arguments. Functions are stored somewhere within a program’s memory and therefore have their own memory address which can be accessed using a pointer to that memory location. Whenever the function is called, the program will follow the pointer to that memory location in order to execute it.

**What is a function pointer?**

A function pointer is simply a pointer to the memory location of a function. Whenever a function is called that function’s pointer is implicitly used. Custom function pointers can be declared and used as any other pointer, ie, passed between functions as arguments, assigned to different memory addresses, function pointers can even be used to override existing functions so that the compiler will call a different function at runtime.

**Why is it necessary to set function pointers when new objects are created?**

Referring to structs.c, each EventLog "object" has its own set of function pointers, however these are pointing to junk memory when the EventLog object is first created (the same way any pointer would be) and must be told where to find the correct functions. In languages like C++ and Java which support methods inside classes the constructor still needs to do this, however it is handled behind the scenes by the compiler.

**Why is it necessary to pass an object pointer to each function?**

As the EventLog struct stores an int count, whenever an EventLog "method" is called on a particular EventLog "object", it needs to work with its own count variable and not one owned by a different object, therefore the appropriate object needs to be passed to the function. In languages like Java and C++ the object does not need to be passed to the method, as the compiler takes care of this implicitly, however in these languages the same thing happens behind the scenes, and the EventLog object would be accessible using the "this" variable.

**How is this functionality replicated in C++?**

C++ declares an EventLog class instead of a struct. The methods can be declared inside the class without the need for function pointers and separate functions, and no EventLog pointer is required to be passed to them. The EventLog itself is still used in almost exactly the same way.

**What is the meaning of private and public?**

These are keywords used to facilitate encapsulation in C++ and represent the visibility of a class member. They work the same as in Java, ie anything marked private can only be used within the EventLog class and its methods, whereas anything marked public can be used anywhere. While visibility modifiers need to be declared for every class member in Java, in C++ they need only be declared once, where they apply to all class members listed after them until the next visibility modifier is declared, or until the end of the class. In addition C++ provides a protected modifier which means only visible within that class and its subclasses. There is no package level protection in C++ as exists in Java.

**How is the constructor implemented and then used? How are multiple constructors used?**

A constructor is a method that is used to instantiate a class object. When the constructor is called, memory is allocated for the object and all code in the constructor is then executed. A constructor is called when an object is created on the stack:

```
Object object;
```

in which case it will be destroyed automatically when it goes out of scope. Objects that are created on the heap:

```
Object *object = new Object;
```

Will not have their memory freed until their destructor is called. Note that you can create an array of objects in this manner:

```
Object objects[50];
Object *objects = new Object[50];
```

The constructor is implemented as a method with no return type and with the same name as the class. Note that a constructor with no arguments is called a default constructor, and a class will be given one by default if no other constructor is declared.

The constructor can be called directly:

```
Object *o = new Object(args);
Object o(args);
```

or indirectly (if it takes no arguments):

```
Object *o = new Object; // same as new Object();
Object o;
```

objects can also be constructed implicitly (without the constructor):

```
class Object
{
    int i, int j;
};

…

Object o { 1, 2 };
```

## Compilation: Preprocessor

**What does the compiler do before checking the syntax of a program?**

The compiler will first run the preprocessor, which checks the given code for any preprocessor directives and handles them accordingly. Preprocessor directives include and #defines or #includes or #ifs in the code. For a #include, the compiler will lookup the file being included, and simply copy everything in that file into the location it is being included. Note that there is a difference between #include “filename” and #include <filename>. In the first case, the compiler will look in a relative path, and the second case, the compiler will look at any location stored in your include path variable. For a #define, the compiler will look at any place the define is used, and replace it with whatever it has been defined as.

**How can we compile to check the output of the preprocessor?**

Preprocessor output can be checked with the compile flag -E, eg:

```
g++ foo.cpp -E
```

Note that this will output to the terminal, so it can be useful to pipe the output to a file for easier viewing.

**Makefile: write a makefile to compile the solution the linked list exercise from last week.**
**CMake: write a cmakefile to compile the solution from last week.**

Samples for each can be found here:
https://github.com/muaddib1971/cpt323/tree/master/chat_examples/week3/intlist

## Errors: None this week

## Debugging: GDB

**What is gdb?**

gdb is the GNU Project Debugger. It allows you to see inside your program to determine what it is doing during execution and also to determine what went wrong if your program crashes for any reason (ie a seg fault). It is the default debugging program that we will be using for this subject.

**How can we compile programs to use gdb?**

In order to use gdb, programs need to be compiled specifically to use it with the -g flag, ie:

```
g++ code.cpp -g
```

This will insert appropriate debugging flags/info into your executable that gdb can recognise and can use when running your program.

You can use this flag in conjunction with optimisation flags such as -O2, however the -O3 flag can significantly alter the compiled executable to the point that debugging flags may no longer give correct information.

**How can we run a program using gdb?**

To run a program in gdb, you must first compile the program, and then type into a terminal:

```
gdb exe
```

This will open the given executable using gdb. The exe itself will not yet be running and must be manually started from within the gdb environment using the run command (you can also provide any command line args here):

```
r
```

We will cover how to use gdb to diagnose and fix a program that has crashed in tute 4 when we examine segmentation faults.

## Exercises:

**What is a binary search tree? What characteristics does it have?**

A binary search tree is a tree where each node has up to two children and has this special characterisitic: all nodes to the left of the current node hold a value less than the current node and all values to the right have a value greater than the current node.

**What is a standard way to implement these?**

```
Class tree
{
        Node * root;
}

Class node
{
        Node * left;
        Node * right;
        Int data;
};

node::traverse(node * current)
{
        if(current.left)
                traverse(current.left);
        Print (current.data);
        if(current.right)
                traverse(current.right);
}
```

**What is a binary heap data structure?**

A binary heap is an array based representation of a binary tree that is guaranteed to be balanced. We represent it in memory something like the following:

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** | **19** |

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | | |

The children of the current node are the nodes at array offsets 2*indexof(current_node) and 2 * indexof(current_node) + 1. The guarantee we provide on implementation is that (for a minheap), the value at parent is less than both children and we maintain this arrangement throughout the heap.

**Why would a binary heap be faster as a tree structure in comparison with a binary search tree?**

This has to do with the contiguity of data. If an element is near to another element in memory that has also been accessed recently, it has likely been prefetched which means the time to access it will only be the time to access the element in cpu cache. If we maintain consistent memory access we can get close to cpu cache performance for all our memory accesses due to the service of this prefetching mechanism.