INTRODUCTION

In this week we cover the following topics

4.1 CLASSES

- Objects in C++ are created using classes. They have the following structure

```
class aclass
{
   private:
      // declare data and member functions
      int data;

   public:
      // declare data and member function

      aclass() {
         // constructor to set up object
      }
      aclass(int x):data(x) {
      }

      ~aclass() {
          // destructor
      }

      int getData() {
         return data;
      }
      void foo( arguments);
};

void aclass::foo(arguments)
{
   // code
}
```

- Data and member functions defined in the `private` section can only be accessed by member functions within the class. They cannot be accessed by external functions.

- Good object design says you should declare all data private and only allow controlled access via public member functions.

4.1.1 Scope Resolution Operator `::`

- The declarations of the member functions can be created outside of the main class definition.

- The scope operator is used to resolve which class a particular member function belongs to. For example

```
void aclass::foo(arguments)
{
   // code
}
```

  Without the `aclass::` the compiler wouldn't know which class `foo` was associated with.

- The scope operator is also used to determine which namespace an object is associated with. For example, if we didn't include `using namespace std;` we would have to use the `cout` object in the following manner.

```
std::cout << "not using namespace\n";
```

4.1.2 `this` pointer variable

- Every instance of a class automatically has a pointer variable associated with it called `this`.

- `this` contains the address of the instantiated class.

4.1.3 Function Overloading

- We can have functions of the same name, so long as their parameter list is different. E.g.

```
int foo(int x)
int foo(double x);
```

- Note in the previous definition of class `aclass` we have overloaded the constructors.

- You can overload normal functions and member functions of classes but you may not overload the `main` function.

4.1.4 Operator Overloading

- Just as you can overload functions, you can overload the operators in class definitions. This allows the operators to work on the newly defined class. For example

```
// overload the assignment operator
aclass& operator = (const aclass &abc) {
   data = abc.data;
   // any other code to make the
   // assignment for the class work
   return *this;
}
```

- We can now use the assignment operator with the class thusly

```
aclass a, b, c(3);
a = b = c;
```

4.1.5 Constructors and Destructors

- Member functions in the public section with the name of the class are *constructors*. They are used to initialise new class objects when they are created. The constructors can accept data in their argument lists and use this data to initialise the objects internal data.

- Even though the two constructer functions below end up doing the exact same thing, they do it slightly differently. The first initialises the `data` variable to `x` when the class is being created. The second creates the class object and then sets `data` to `x`. The 1st is more efficient as it requires less operations.

```
aclass(int x):data(x) {
}

aclass(int x) {
   data = x;
}
```

- A member function that has the same name as the class, but preceded by a ~, is a *destructor*. It gives instructions on how an object is to be released back to memory. If your class contains dynamically created memory then it is crucial that your class has instructions in the destructor to release this memory back to the os.

4.1.6 Shallow vs Deep Copying

- Consider the following code

```
class aclass
{
   private:
      int size, *ptr;

   public:
      aclass(int s) {
         size = s;
         ptr = new int[size];
      }
      ~aclass() {
         delete [] ptr;
      }
      void setArray(int i, int x) {
         ptr[i] = x;
      }

      /* more functions */
};

void foo(aclass xyz)
{
   /* some code */
}

int main()
{
   aclass abc(8);

   foo(abc);
   abc.setArray(0, 123);
   return 0;
}
```

- Note that `aclass` contains points to dynamically allocated memory.

- When `main` calls `foo` a copy of `abc` is made and put into `xyz`. This is called *shallow copying* and is very dangerous code.

- The reason it is dangerous is because the address in `abc.ptr` is copied into `xyz.ptr`. They are both pointing to the same section of memory. When `foo` is finished and `xyz` is released back into memory, it calls the destructor of `aclass`. This releases the declared array of `int`s back to memory. The call to `setArray` after the call to `foo` will cause the program to crash as the memory is no longer allocated to the object.

- The way to get around this is to code a *copy constructor*. The copy constructor for the previous `aclass` would be as follows

```
aclass(const aclass &other) {
   size = other.size;
   ptr = new int[size];

   for (int i=0; i<size; i++) {
      ptr[i] = other.ptr[i];
   }
}
```

  This way, when `foo` constructs `xyz` it will invoke the copy constructor, which will make a completely independent copy of the `aclass`. This is called *deep copying*.

- Along with the copy constructor you should also overload the *assignment operator*. For the `aclass` above this is how you could do it.

```
aclass& operator = (const &other) {
   size = other.size;
   ptr = new int[size];

   for (int i=0; i<size; i++) {
      ptr[i] = other.ptr[i];
   }
   return *this;
}
```

4.1.7 Static Data Declarations

- When a data declaration within a class is made `static` it means that only one version of the variable will exist and be shared by all instances of the class. For example

```
class xyz {
   private:
      static int x;
};

xyz a, b, c;
```

- This has the definition of class `xyz` and creates three instances of it – `a`, `b`, `c`. However, there will only be one copy of `x` and all the instances will share it. Also note that because `x` is private, nothing outside the `xyz` class can access it.

4.1.8 `const` Member Functions

- If you place a `const` after the parameter list in a member function it makes the function read only. It means the function cannot change any of the internal data of the class. For example, using the previous `aclass`

```
void setSize(int s) const {
   size = s;   // won't compile
               // because const
}

int getSize() const {
   return size;   // this is ok;
}
```

4.1.9 Classes and Pointers

- You access data and functions of a class using the dot operator. E.g

```
aclass xyz(5);
int x = xyz.getData();
```

- However this becomes a bit more messy when using a pointer. E.g.

```
aclass classPtr = new aclass(5);
int x = (*classPtr).getData();
```

- In order to simplify things C++ has the arrow operator which can only be used with pointers. E.g.

```
int x = classPtr->getData();
```

4.1.10 `struct`'s

- C++ has an equivalent structure to the `class` called the *`struct`*. It is used in exactly the same way as classes. Eg

```
struct myStruct {
   //etc
};
```

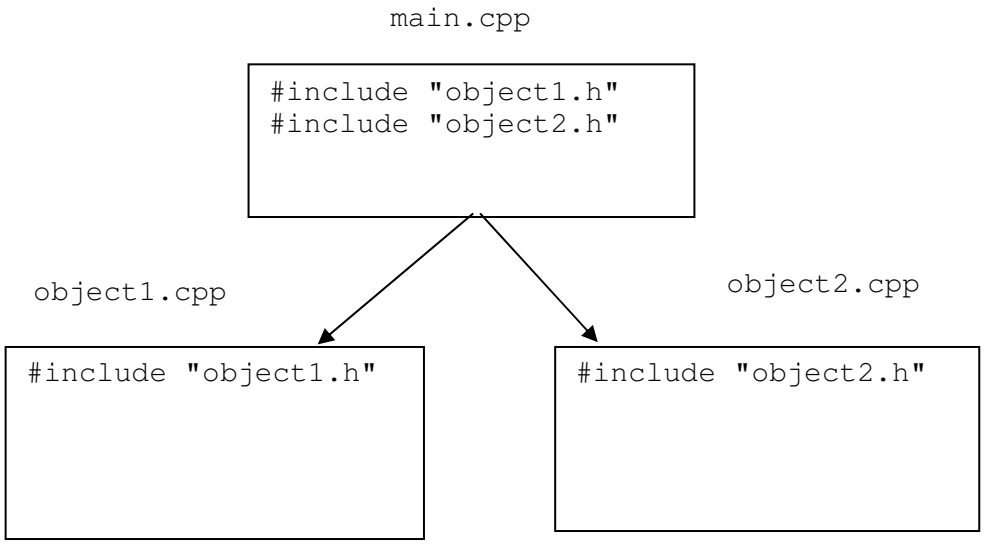
- The difference between a `class` and a `struct` is that, unless explicitly stated, data and functions declared in a `class` are automatically `private:` while data and functions declared in a `struct` are automatically `public:` in scope.

4.2 MULTI-FILE PROGRAMS

- C++ programs get far too large to fit in a single file. They are difficult to edit and they take a very long time to compile. They are therefore split into numerous files.

- Each file can then be separately compiled before the program is linked together. If only one file has changed then that is the only part of the program that has to be re-compiled.

- The following diagram illustrates how a simple program might be split.




```
rerun% cat main.cpp

#include "object1.h"
#include "object2.h"

int main()
{
   /* code */
   return 0;
}

// ########################################
```

```
rerun% cat object1.h

#ifndef OBJECT1_H
#define OBJECT1_H

class object1
{
   private:
      /* some data */


   public:
      /* constructors, destructors */
      int foo();
};

#endif

// ########################################

rerun% cat object1.cpp

#include "object1.h"

int object1::foo()
{
   /* code */
}

// ########################################
```

```
rerun% cat object2.h

#ifndef OBJECT2_H
#define OBJECT2_H

class object2
{
   private:
      /* some data */
   public:
      /* constructors, destructors */
      int foo();
};

#endif

// ########################################

rerun% cat object2.cpp

#include "object2.h"

int object2::foo()
{
   /* code */
}
```

4.2.1 `makefile`

- In order to compile a multi-file program we need to use the `make` utility on UNIX.

- In order for `make` to work we need a file called `makefile`, which contains the compiling instructions `make` will use to compile all the separate files to create the final program.

- The `makefile` will tell `make` how to compile each separate file into a `.o` file. If the file hasn't been changed then `make` won't enact that bit of `makefile`. The final part of the `makefile` tells `make` how to link all the `.o` files to become the final program.

- An example of a `makefile` would be

```
CC = g++
prog: main.o object1.o object2.o
  $(CC) main.o object1.o object2.o -o program
main.o: main.cpp object1.h object2.h
  $(CC) -c main.cpp
object1.o: object1.cpp object1.h
  $(CC) -c object1.cpp
object2.o: object2.cpp object2.h
  $(CC) -c object2.cpp
```

- Note that the first character on each command line *must* be a tab.

- We compile the program by typing `make` while in the directory containing the `makefile`.

4.3 EXCEPTION HANDLING

- C++ has a system for handling errors called exception handling.

- You test for some error and, if there is an error you `throw` an exception error to be caught elsewhere in your code. The `throw` must happen from within a `try` block which has an associated set of `catch`'s. For example

```
try {
   // some code
   if (test_fails) {
      throw out_of_range("error message");
   }
} catch (out_of_range &ex) {
   cout << "Error out of range\n" <<
            ex.what();
} catch (...) {
   cout << Undefined exception thrown\n";
}
```

- `out_of_range` is one of a number of standard exceptions found in `stdexcept`, so you have to `#include <stdexcept>` in order to use them. The book lists the others.

- `catch(...)` is for those exceptions thrown where there isn't a specific `catch` set up for it. It is like the `default:` in a `switch` statement.

- If you make a `throw` and it doesn't appear to be within a `try` block it will exit the function and see if it is within a `try` block in the calling function. It will keep going back until it gets to the `main` function. If it still isn't within a `try` block the program will abort.

4.4 stringlist.cpp

```
rerun% cat stringlist.h

/*********************************************\
   Definitions of members of stringlist class
\*********************************************/

#include <iostream>

using namespace std;

class stringlist
{
   private:
      int numstrings;
      string *strings;

   public:
      stringlist() : numstrings(0), strings(NULL)
      {}

      // copy constructor
      stringlist(const stringlist &other) :
         numstrings(other.numstrings),
         strings(new string[other.numstrings]) {
         for (int i=0; i<numstrings; i++) {
            strings[i] = other.strings[i];
         }
      }

      // destructor
      ~stringlist() {
         if (numstrings > 0) delete [] strings;
      }

      void printStrings();
      const string& getString(int num);
      void setString(int num,
                     const char *astring);
```

```
      void setString(int num,
                     const string &astring);

      int getNumstrings() {
         return numstrings;
      }
      void setNumstrings(int num);
      void swapStrings(stringlist &other) ;

      // overloaded operators
      stringlist& operator =
                      (const stringlist &other) {
         stringlist temp(other);
         swapStrings(temp);
         return *this;
         // destructor for temp will be
         // called at end of this function
      }
};

// ###############################################

rerun% cat stringlist.cpp

/*********************************************\
   Definitions of members of stringlist class
\*********************************************/

#include <iostream>
#include <stdexcept>
#include <algorithm>

#include "stringlist.h"

using namespace std;

void stringlist::printStrings()
{
   for (int i=0; i<numstrings; i++) {
      cout << strings[i] << '\n';
   }
}
```

```
const string& stringlist::getString(int num)
{
   if (num < 0 || num >= numstrings) {
      throw
         out_of_range("stringlist::getString");
   }
   return strings[num];
}

void stringlist::setString(int num,
                           const char *astring)
{
   if (num < 0 || num >= numstrings) {
      throw
          out_of_range("stringlist::setString");
   }
   strings[num] = astring;
}

void stringlist::setString(int num,
                           const string &astring)
{
   if (num < 0 || num >= numstrings) {
      throw
         out_of_range("stringlist::setString");
   }
   strings[num] = astring;
}
```

```
void stringlist::setNumstrings(int num)
{
   if (numstrings != 0) {
      throw
      runtime_error("stringlist::setNumstrings -
       numstrings not 0");
   }
   if (num < 0) {
      throw
         out_of_range("stringlist::setNumstrings
                       - num < 0");
   }
   numstrings = num;
   strings = new string[numstrings];
}

void stringlist::swapStrings(stringlist &other)
{
   swap(numstrings, other.numstrings);
   swap(strings, other.strings);
}

// ###############################################

rerun% cat main.cpp

/*********************************************\
   program to print entered arguments to screen
   using stringlist class
\*********************************************/

#include <iostream>
#include <stdexcept>

#include "stringlist.h"

using namespace std;

void loadStrings(stringlist &mystrings);
```

```
void loadStrings(stringlist &mystrings)
{
    int numstrings;
    string astring;

    cout << "Please enter number of
                 strings to be entered\n";
    cin >> numstrings;
    mystrings.setNumstrings(numstrings);

    cout << "Please enter the strings\n";
    for (int i=0; i<numstrings; i++) {
       cin >> astring;
       mystrings.setString(i, astring);
    }
}

int main()
{
   stringlist mystrings, *stringsPtr = NULL;
   int numString;

   try {
      loadStrings(mystrings);

      // dynamically create a new stringlist and
      // copy mystrings into it
      stringsPtr = new stringlist;
      *stringsPtr = mystrings;

      // change new string
      numString = stringsPtr->getNumstrings();
      stringsPtr->setString(numString/2,
                            "NEWSTRING");

      // swap the two strings around
      stringsPtr->swapStrings(mystrings);
```

```
      // print both the strings
      cout << "\nFirst stringlist\n";
      mystrings.printStrings();
      cout << "\nDynamically created list\n";
      stringsPtr->printStrings();

      delete stringsPtr;
   } catch(out_of_range &ex) {
      cout << "\nERROR - Out of Range Exception
                thrown\n" << ex.what() << "\n";
      exit(1);
   } catch (runtime_error &ex) {
      cout << "\nERROR - Runtime Exception
              thrown\n" << ex.what() << "\n";
      exit(1);
   }

   return 0;
}

// ###############################################

rerun% cat makefile

CC = g++
prog: main.o stringlist.o
    $(CC) main.o stringlist.o -o strings
main.o: main.cpp stringlist.h
    $(CC) -c main.cpp
stringlist.o: stringlist.cpp stringlist.h
    $(CC) -c stringlist.cpp

// ###############################################
```

```
rerun% ./strings
Please enter number of strings to be entered
3
Please enter the strings
big
bad
wolf

First stringlist
big
NEWSTRING
wolf

Dynamically created list
big
bad
wolf

rerun% ./strings
Please enter number of strings to be entered
-2

ERROR - Out of Range Exception thrown
stringlist::setNumstrings - num < 0
```