

Intro to C++ What's wrong with just 'C'

In 'C', programs are constructed simply from a set of variables and a set of global functions. The variables hold the data that the program manipulates, while the functions contain the code and algorithms designed to manipulate that data.

This is OK when programs are relatively small, but as they get larger, it becomes more difficult to maintain, especially when several programmers work on the project because all the variables and functions due to the global nature become visible to everyone on the team e.g.

```
// global variables

int    a,b,c,d, i[10], x[5];
float  x, y, s[100];
...
...
int    myspeed, yourspeed, everyonesspeed ;

// global functions

void    myfunc1(void) { ... }
int     getmyspeed(void) { ... }
void    setmyspeed( int newspeed) { ... }
int     getyourspeed(void) { ... }
void    setyourspeed( int newspeed) { ... }
```

Sure, using the ideas of local variables and passing data between functions as arguments can help maintain some sanity, but it's still hard to manage for a team. For instance, as a programmer you have to keep inventing unique variable and function names for your code to avoid name conflicts with similar things created by other programmers in your team and it's difficult to keep track of which functions are related to (i.e. need access to) which variables. Everything ends up in one big giant melting pot of global names.

These problems only get worse as the program grows in size. In simple terms we need a way to manage the complexity and keep track of which code uses which variables so we can make changes more easily if the need arises.

Simple ways to handle the complexity

In the past, in 'C', this complexity was *kinda* managed in a limited but not wholly successful way using separate multiple source files. Here each programmer would isolate their own global variables and functions and put them together in their own source files.

Programmers could chose to hide the things that they didn't want other parts of the program (or other programmers) to have access to (i.e. give them **privacy**) by prefixing them with the word **static**. This meant that only the functions written within that source file could access **those** elements. Things *without* **static** could be accessed by the whole program i.e. were **public** to the outside world. Example

```
// variables
static int    myspeed;                // private to this source file

// functions
int    getmyspeed(void) { ... }        // public to the world
void    setmyspeed( int newspeed) { ... } // public to the world
static void myprivateFn1( void ) { ... } // private to this file
```

PaulsFile.c

```
// variables
static int    yourspeed;                // private to this source file

// functions
int    getyourspeed(void) { ... }        // public to the world
void    setyourspeed( int newspeed) { ... } // public to the world
static void myprivateFn1( void ) { ... } // private to this file
```

JoesFile.c

Now programmers could add variables and functions to their own source files without creating name conflicts that could occur if everything was “**global**” e.g. both above source files could now contain the private (**static**) function ‘myprivateFn1()’ without problem. The linker would take the resulting object files produced during compilation and keep the **static variables/fns** with the same names as separate variables.

Teams could now write and manage their own individual data and the functions that manipulated them much more easily, since their variables and the functions that manipulated them could be located within the same source file not scattered around a billion lines of code.

Hiding variables/functions away from programmers that did not need access to them (*making them private i.e. static*) also made it easier to change code, since the effects of any change were by definition limited to that source file. **Object Oriented Programming** (OOP) takes this privacy concept one step further.

Classes (Part I)

C++ *Classes* are a better way of grouping together related variables and functions into one place. Classes are defined using the keyword `class`, with the following example syntax:

```
class name {  
    private:  
        some private variables and functions ;  
  
    public:  
        some public variables and functions ;  
};
```

Explanation

Here `name` is the name of the class. The body of the class can contain `public` and `private` *members*, which can be either *variables* or *functions*.

`private` members of a class are accessible only to the functions that are members of that class. `public` members are accessible to everyone, including users of and designers of the class.

By default, all members of a class are `private` unless specified otherwise. For example:

```
class Rectangle {  
    int    width, height;  
    public:  
        void set_values (int x, int y) { ... }  
        int  get_area (void) { ... }  
};
```

declares a class i.e. a **new data type** (just like `int` and `floats` are data types) `Rectangle`. This class contains four member elements: two variables of type `int` (`width` and `height`) with `private`

access (because *private* is the default access level) and two member functions with *public* access: the functions `set_values()` and `get_area()`.

Because *width* and *height* are *private*, only the member functions of that class, i.e. `set_values()` and `get_area()` are allowed to access them, nobody else can even see those variables, certainly not users of that class. The member functions are public, so anybody can call or use them to indirectly get at and manipulate *width* and *height*.

Creating an Object from a Class

An *object* is an instance of a class. By analogy with traditional variables, a *class* introduces a new *data type*, while an *object* would be the *variable* of a data type. For example, in C we might say this to introduce instances of an integer variable:-

```
int    a, b, c;
```

In C++, now that we have classes, we can write something similar like this:-

```
Rectangle  r1, r2, r3;
```

Here *r1*, *r2* and *r3* are three instances of a Rectangle (i.e. they are Rectangle ‘*objects*’ or *variables of type Rectangle*). That is, *r1*, *r2* and *r3* each have hidden within them, their own unique member variables ‘*width*’ and ‘*height*’. In C++ we say that the class Rectangle *encapsulates* (or forms a *wrapper* around) its member variables and functions.

Manipulating the Rectangle objects

Once we have a rectangle *object*, any of the *public* members of that *object* can be accessed as if they were normal functions or normal variables, by simply inserting a dot (.) between *object/instance/variable name* and *member name*. This follows the same syntax as accessing the members of ‘*struct*’ in C. For example we could say something like this in our code:

```
r1.set_values (3, 4);
r2.set_values (5, 6);

printf("r1's area is %d\n", r1.get_area( ) );
printf("r2's area is %d\n", r2.get_area( ) );
```

Here is a complete example of the class *Rectangle* and some code to illustrate using it:

```
#include <stdio.h>

class Rectangle {
    int width, height;

public:
    void set_values (int x, int y) {
        width = x;
        height = y;
    }

    int get_area() {
        return width * height;
    }
};

int main ()
{
    Rectangle    r1, r2;                // two rectangle objects

    r1.set_values (3, 4);                // initialise r1 with data for width and height
    r2.set_values (5, 6);                // initialise r2 with data for width and height

    printf("r1's area is %d\n", r1.get_area() );    // displays r1's area = 12
    printf("r2's area is %d\n", r2.get_area() );    // displays r2's area = 30

    return 0;
}
```

Notice that the call to `r1.get_area()` does not give the same result as the call to `r2.get_area()`. This is because `r1` and `r2` have their own variables `width` and `height` and that is what the functions are using, i.e. the functions `get_area()` and `set_values()` are manipulating the variables of the object that invoked them. In this case the invoking objects are `r1` and `r2` in turn – this is really **important**.

This means it was not necessary to explicitly pass `r1` or `r2` to the function `get_area()` to tell it which object to manipulate. Instead the compiler makes sure that `get_area()` and `set_values()` are implicitly given copies of `r1` and `r2` to work with at run time and thus have access to the variables '`width`' and '`height`' from their respective objects `r1` and `r2`. See the discussion on the pointer '`this`' later in these notes for explanation of how that works.

In the example above, we embedded the actual code for the functions `get_area()` and `set_values()` inside the class definition itself. This is acceptable here because that code is simple and small, but when those functions become large and there are lots of them, it becomes unwieldy to do it this way. In general we separate out the code like this:-

```
#include <stdio.h>

class Rectangle {
    int width, height;

public:
    void set_values (int x, int y) ;           // a declaration of the class function
    int get_area() ;                          // a declaration of the class function
};

void Rectangle::set_values (int x, int y) {    // a definition of the class function
    width = x;                               // width means width var of the invoking object
    height = y;                              // height means height var of the invoking object
}

int Rectangle::get_area() {                  // a definition of the class function
    return width * height;                   // return area of invoking object
}

int main ()
{
    Rectangle    r1, r2;                    // two rectangle objects

    r1.set_values (3, 4);                   // initialise r1 with data for width and height
    r2.set_values (5, 6);

    printf("r1's area is %d\n", r1.get_area() );    // displays 12
    printf("r2's area is %d\n", r2.get_area() );    // displays 30

    return 0;
}
```

This example also introduces the *scope resolution operator* (`::`) two colons. It is used to say that the functions `set_values()` and `get_area()` being defined above belong to the *Rectangle* class (as opposed to any other class that might also define functions with those same names).

The only difference between *defining* a member function *within* the class or to just include its *declaration* in the class and write the definition/code for it later outside the class, is that in the

first case the function is automatically considered an *inline* member function by the compiler, while in the second it is a normal (non-inline) class member function. This causes no differences in behavior, but only possible compiler optimizations such as size of compiled code – non-inline functions lead to smaller compiled code.

Separating classes into header and source files

Taking this one step further, C++ wisdom dictates that the parts of the class be separated out into a *Rectangle.cpp* source file (note the *.cpp* suffix) and a *Rectangle.h* header file. It's not essential but it's demonstrated below. You choose.

```
class Rectangle {  
    int    width, height;  
  
public:  
    void set_values (int x, int y) ;           // a declaration of the class function  
    int get_area() ;                          // a declaration of the class function  
};
```

Rectangle.h

```
void Rectangle::set_values (int x, int y) {    // a definition of the class function  
    width = x;  
    height = y;  
}  
  
int Rectangle::get_area() {                  // a definition of the class function  
    return width * height;  
}
```

Rectangle.cpp

```
#include <stdio.h>  
#include "Rectangle.h"           // IMPORTANT include header file for class  
  
int main ()  
{  
    Rectangle    r1, r2;         // two rectangle objects  
  
    r1.set_values (3, 4);  
    r2.set_values (5, 6);  
  
    printf("r1's area is %d\n", r1.get_area() );    // displays 12  
    printf("r2's area is %d\n", r2.get_area() );    // displays 30  
    return 0;  
}
```

MyProgram.cpp

What about the private members – can we access them?

Members `width` and `height` have (by default) `private` access. By declaring them `private`, access to them from outside the class is not allowed. This makes sense, since we have already defined the member functions `set_values()` and `get_area()` to manipulate those variables for us. Therefore, the rest of the main program does not *need* to have `direct` access to `width` and `height`. In fact if you tried, it would not be permitted. For example, if we wrote this, we'd get a compile error.

```
int main ()
{
    Rectangle    r1, r2;                // two Rectangle objects

    ...

    printf("r1's area is %d\n", r1.width * r1.height );    // Compiler ERROR
    printf("r2's area is %d\n", r2.width * r2.height );    // Compiler ERROR

    return 0;
}
```

Perhaps in such a simple example as this, it's difficult to see how restricting access to these variables may be useful, but in bigger projects it is important from a maintenance point of view that changes to code have *limited* and *isolated* effects.

Ideally then, classes present their external `interfaces` to the outside world, i.e. their `public member functions` that will allow the outside world (*you, me and joe public*) to `get` and `set` the members variables in ways that are appropriate e.g. `sort` and `search` functions etc. We don't want the users of the object to write code that is dependent upon how the object implements that algorithm.

For example, just like driving a car, you use the interfaces provided by the car designer (brakes, accelerator, steering wheel etc.) and manipulate the car via those interfaces. You don't need to know how the car *works* to drive it, that implementation stuff is hidden and is only of concern to the car designer not the driver.

Furthermore, if all cars provide the same interfaces (*as they generally do*), then one car should be interchangeable with another as far as the user is concerned since they only manipulate its interfaces. The car designer is free to change things like the type of engine etc. and it shouldn't make any difference to the user (*you and I*), as we only see the interfaces not the implementation. This makes our software easier to maintain.

Exercise: Write 2 member functions to allow the outside world to `get` (i.e. return) the values of the variables `height` and `width`. Now write 2 member functions to allow them to `change` these two variables independently.

Constructors

What would happen in the previous example if we called the member function `get_area()` before having called `set_values()`? An undetermined result, since the object variables `width` and `height` would not have been assigned an initial value. As programmers, it would be our responsibility to make sure we called the function `set_values()` to initialise our rectangle as soon as possible after we create an instance of it. If we forget, then we are setting up problems for ourselves later.

In order to avoid that, a class can include a special function called a *constructor*, which is *automatically* called whenever a new object/instance of that class is created. The constructor can perform any actions you like e.g. initialise variables, open files, turn on LEDs in a small embedded system etc., but typically it is used to initialise the object variables. In a way, our `set_values()` function is already carrying out the task of a constructor, but it's not called automatically, we have to remember to call it ourselves.

A proper C++ constructor function is declared just like a regular member function, but with a name that is *identical* to the class name (*that's how the C++ compiler knows which of our many functions is the constructor for our class – it has the same name as the class*). Constructor functions do not have a return type, not even `void`.

The `Rectangle` class above can easily be improved by adding a proper constructor to replace the `set_values()` functions e.g.

```
#include <stdio.h>

class Rectangle {
    int width, height;

public:
    Rectangle (int x, int y) ;           // a declaration of the class constructor function
    int get_area() ;                     // a declaration of the class function
};

Rectangle::Rectangle (int x, int y) {    // the constructor function taking values to initialise
    width = x;                           // the member variables. Note NO void return type
    height = y;
}

int Rectangle::get_area() {              // the code to calculate the area
    return width * height;
}

int main ()
{
    Rectangle   r1(3,4), r2(5,6) ;       // 2 rectangle objects whose constructors
                                         // are called to properly initialise them

    printf("r1's area is %d\n", r1.get_area() ) ; // displays 12
    printf("r2's area is %d\n", r2.get_area() ) ; // displays 30
    return 0;
}
```

Notice how arguments are passed to the constructor at the moment at which the objects of this class are created e.g. `Rectangle r1(3,4)`. The '3' becomes the argument 'width' and '4' the argument 'height' in the constructor function.

Now, class `Rectangle` has no member function `set_values`, but instead has a **constructor** that performs a similar action. It initialises the values of `width` and `height` with the arguments passed to it.

Summary: Constructors are executed automatically when a new object/instance of that class is created. They should be declared with `public:` access (see above and verify this) to allow them to be called. Constructors *never* return values, they simply initialise the object.

Overloading Constructors

Like any other C++ function (*and unlike functions in simple C*), a constructor can also be "*overloaded*". This means that you can have several functions in your class with the *same* name as long as they have different numbers of arguments (*or the same number, but different types of arguments*). The compiler will automatically call the correct constructor to match the arguments used when the object is created. For example:

```
#include <stdio.h>

class Rectangle {
    int width, height;

public:
    Rectangle () ;                // a default constructor taking no argument
    Rectangle (int x, int y) ;    // another constructor taking 2 int argument
    int get_area() ;              // a class function to return the area
};

Rectangle::Rectangle (int x, int y) {    // a constructor function taking 2 int's
    width = x;
    height = y;
}

Rectangle::Rectangle () {                // a 2nd, default constructor taking no arguments
    width = 0;
    height = 0;
}

int Rectangle::get_area() {              // the area code
    return width * height;
}

int main ()
{
    Rectangle    r1(3,4);                // constructor with 2 int arguments called
```

```

Rectangle r2;                                // default constructor called

printf("r1's area is %d\n", r1.get_area() );    // displays 12
printf("r2's area is %d\n", r2.get_area() );    // displays 0
return 0;
}

```

In the above example we also introduced a special kind of constructor: the *default constructor*. The *default constructor* is the constructor that takes *no argument*, and it is special because it is called when an object is introduced but is not initialised with any arguments.

In the example above, the *default constructor* is called for *r2*. Note how *r2* is not even constructed with an empty set of parentheses - in fact, empty parentheses cannot be used to invoke the default constructor:

```

Rectangle r2;           // ok, default constructor called
Rectangle r2();         // ERROR, default constructor NOT called – looks like a function prototype

```

This is because the empty set of parentheses above would make *r2* look like a function that takes no arguments and returns a value of type *Rectangle*.

Member variable initialisation in Constructors

When a constructor is used to initialise the member variables of a class, the variables can be initialised directly within the body of the constructor. For example, consider a class with the following declaration:

```

class Rectangle {
    int width, height;                // private member variables

public:
    Rectangle(int x, int y);          // public member functions
    int area() { return width * height; } // constructor function
};

```

The constructor for this class could be defined, as:

```

Rectangle::Rectangle (int x, int y)
{
    width = x;
    height = y;
}

```

But it could also be defined like this using a member initialisation list

```
Rectangle::Rectangle (int x, int y) : width(x), height(y)
{
}
}
```

Note how in this above case, the constructor does nothing more than initialise its member variables, hence it has an empty function body, but you could put other code inside the body if you wished.

For members of fundamental types e.g. `ints`, `floats`, `chars` etc. it makes no difference which of the two styles of constructor initialisation you chose, however member variables whose type is another class should be initialised using the member initialization list. For example:

```
// member initialization
#include <stdio.h>

class Circle {                                // a circle class
    double radius;

public:
    Circle(double r) : radius(r)              // constructor using a member initialization list
    {}

    double get_area() {
        return radius*radius*3.14159265;
    }
};

class Cylinder {                               // a cylinder class
    Circle base;                               // a member variable 'base' which is a circle
    double height;

public:
    Cylinder (double r, double h) : base (r), height(h) // note constructor to initialise base (a circle)
    {}                                                // this has to be done using initialisation list

    double get_volume() {
        return base.get_area() * height;           // get the area of the base variable (a circle)
    }
};

int main ()
{
    Cylinder c1 (10, 20);                          // call cylinder constructor

    printf("c1's volume = %f\n", c1.get_volume() );
    return 0;
}
```

The output from this program is: `c1's volume = 6283.185`

In this example, class `Cylinder` has a member variable 'base' whose type is another class (a `Circle`). Because objects of class `Circle` can only be constructed with a parameter (i.e. the only available circle constructor is one that requires an `int` argument), `Cylinder's` constructor needs to call `base's` constructor, and the only way to do this is in the *member initialiser list*.

Copy Constructors and References

A copy constructor is used to make copies of other class objects when a new one is created and you want its member variables to be copies of the values of another existing object of the same type. As an example of their use, suppose we had a cylinder object `c1` in our code constructed like this:-

```
Cylinder c1 (10, 20);           // call cylinder constructor
```

Now suppose we wanted to create another cylinder `c2` with exactly the same values as `c1`. Remember `c1` could have been changed by calling a member function after it was constructed, but we don't care, we simply want to copy its current value as it is now.

One way to create `c2` as an identical copy of `c1`, is to create `c2` (*with or without some default values*) and then read the current values of `c1` using functions like `getbase()` and `getHeight()` – *assuming they exist*, and copy them to `c2` using functions like `setBase()` and `setHeight()` – again *assuming they exist*. However, what if those function don't exist, or what if the variables inside the new cylinder were marked as "**const**" meaning their values could not be changed after default construction. If any of these were the case we would not be able to copy an existing object when creating a new one. This is exactly what a copy constructor is designed to overcome.

Note if you do not supply a copy constructor explicitly in your code, the compiler will create one for you automatically, and it will literally perform copying of all member variables in the source object into their identical counterparts in the destination object, i.e. **member wise copying** as it is called. Generally this default copy constructor works ok, but what if the object we were copying had **pointers** in it, this default copy constructor would end up copying the **pointer values** from the source object to the destination object – this may be what we wanted but generally it's not, since both objects would end up having pointers pointing to the same "**things**". If one of those objects were to destroy the "thing" in the future, what would the other one be left pointing to?

How do we write a Copy Constructor?

A copy constructor for the `Cylinder` class would have the following signature and code. That is *because it's a constructor, it will still have the same name as the class, i.e. `Cylinder`*, it does not return anything (i.e. no `void` return type) and it takes a single argument which is a reference (the `'&'` bit) to an existing cylinder object `'theSourceObject'`.

```
Cylinder::Cylinder (Cylinder &theSourceObject ) :  
    base (theSourceObject.base),    // initialise new base with existing source object's base  
    height (theSourceObject.height) // initialise new height with existing source object's height  
{  
    // anything else you want to do, put it here  
}
```

Now, assuming the existence of a `Cylinder` object `c1`, we could now write the following code, to “clone” that existing cylinder when creating a new cylinder “`c2`”

```
Cylinder c2( c1 ) ;
```

The copy constructor is called because the argument `'c1'` is a `Cylinder` and that's the argument we used when we constructed `c2`, i.e. construct `c2` based on a copy of `c1`.

Note that it's also possible to write the copy constructor like this (provided the variables we are copying to are not constant), where the copying is done inside the body of the constructor. This is a little more readable.

```
Cylinder::Cylinder (Cylinder &theObjectToCopy )  
{  
    base = theObjectToCopy.base;    // initialise new base with other object's base  
    height = theObjectToCopy.height; // initialise new height with other object's height  
}
```

References: The **&** operator in variable declarations

If you are unfamiliar with references (*they don't exist in C, only in C++*), then think of them as being very similar like pointers e.g. as if they were declared as **Cylinder *C** in the constructor above – except that you cannot write the copy constructor to use a pointer in its signature, you must use a reference '**&**' not '*****' – the language makes this a requirement for copy constructors.

References are just like pointers except that when you use pointers, you have to use '**&**' to make the pointer **point** to something and remember to use '*****' every time you want the "**something**" pointed to by the pointer e.g. taking a simple example

```
char c ;           // a simple char
char *cptr         // a pointer to char (i.e. a char pointer)
cptr = &c;         // initialise pointer to point to 'c'

*cptr = 5;         // set the value of 'c' to 5 using the pointer with the '*'
```

Using **references** is sometimes more convenient (*or confusing if you like pointers*)

```
char c ;           // a simple char
char &cref          // a char reference
cref = c;          // initialise reference to refer to 'c' (note NO '&' in initialisation)

cref = 5;          // set the value of 'c' to 5 using the reference (note NO '*')
```

Destructors

A C++ class can also have a **destructor** function. It is declared with the same name as the class but with a '**~**' in front of its name. A destructor is *automatically* called whenever an object is destroyed (*e.g. it goes out of scope*) or is *deliberately* destroyed.

A destructor can be used to "**tidy up**" any "loose ends" after an object is destroyed. For instance, it could be used to close a file, or turn off an LED in an embedded applications. It could also be used to release any memory back to the operating system that might have been allocated. It really is up to you what you ask your destructor to do, you don't have to have one at all.

Destructor functions do not have a return type; not even **void**. Neither do they take any arguments. For example, a destructor for the rectangle class might look like this:-

```
class Rectangle {
    int width, height;

public:
    Rectangle(int, int);
    ~Rectangle();
    int area() { return x*y; }
};

Rectangle::~Rectangle() // code for destructor (note placement of '~')
{
    printf("Rectangle Destructor Called....\n") ;
}
```

Pointers to Class Objects

Once a class has been introduced a class name becomes a new valid type, so we can create pointers to that type of data. For example, the following statement declares that `p1` is a pointer to an object of class `Rectangle`. That is, it will eventually be initialised to point to an object in memory which is a `Rectangle`.

```
Rectangle *p1;
```

As we have seen with plain data structures in 'C', the members of a **structure** can be accessed via a pointer using the arrow operator (`->`). The same principle applies when a pointer is used to access the members of a class. Here is an example with some possible combinations:

[illegible]


```
Rectangle *p1;                                // create a pointer to a rectangle

p1 = &r1;                                       // make p1 point to object r1

printf("r1's area = %d\n", r1.get_area() );    // print area using r1 directly
printf("r1's area = %d\n", p1->get_area() );    // print area indirectly using pointer p1

return 0;
}
```

Problem: Write the above code in main() to use a reference to a Rectangle instead of pointer.

The pointer “**this**”

We mentioned earlier that a class is created by defining its member variables and member functions, however it's important to realise that only the variables are allocated storage for each new object. For example, referring back to our Rectangle class

```
class Rectangle {  
    int width, height;  
  
public:  
    void set_values (int x, int y) {  
        width = x;  
        height = y;  
    }  
  
    int get_area() {  
        return width * height;  
    }  
};
```

Here we see the class contains two variable and two member functions. When we write this statement

```
Rectangle r1, r2, r3;
```

The only things created at that time are the variables `width` and `height` for each of the 3 rectangles. The functions `set_values()` and `get_area()` are “shared” by all rectangle objects, that is, there is only 1 copy of these two functions, but 3 copies of the `width` and `height` variables. When we write this code

```
r1.set_values(3, 4)
```

how does the function `set_values()` know that it has to set the variables of ‘r1’? There’s some “magic” that goes on here via the hidden point “**this**”. In essence the statement above is *actually* translated into by the C++ compiler into the C style statement

```
set_values( &r1, 3, 4);
```

That is, a *hidden pointer* to the invoking object (i.e. the address of or pointer to `r1`) is secretly passed to the function `set_values()` as the first parameter. This means it can use that pointer to manipulate the variables of the invoking object (`r1`). This is how a member function can be shared by all objects of that class (*which is just as well as making separate copies of the member function code for each object would result in massive code bloat*)

As a consequence of the above observation, it also means that all member functions have a hidden first parameter, one we don’t declare, but which the compiler magically inserts for us. Thus the function `set_values(int x, int y)` above is actually translated into

```
void set_values (Rectangle *this, int x, int y) {
    this->width = x;
    this->height = y;
}
```

That is, the compiler introduces a pointer called “*this*” which is initialised to point to the invoking rectangle object and uses/dereferences that pointer to get at the width and height variables of the invoking object.

Now even though the above explanation describes the *hidden* mechanics of how classes work, we can nevertheless use the pointer “*this*” explicitly ourselves, e.g. as shown below, where we could write either version of the function

```
void set_values (int x, int y) {
    this->width = x;
    this->height = y;
}
```

```
void set_values (int x, int y) {
    width = x;
    height = y;
}
```

Dynamic Memory Allocation: operators *new* and *delete*

In the ‘C’ language we could dynamically allocate storage for memory using library functions such [malloc](#), [calloc](#), [realloc](#) and [free](#), defined in the header file `<stdlib.h>` in C. The functions are also available in C++ and can also be used to allocate and deallocate dynamic memory. But there is a nicer way to do this in C++ using operators *new* and *delete*.

For example in C++, if we wanted to dynamically allocate memory for some variable, we could call the operator *new* to make that allocation. This operator returns a *pointer* to the data that we allocated storage for. As an example, we could allocate storage for 1 *new int* like this

```
int *p1 = new int;
```

We could allocate storage for an array of 20 floats like this

```
float *p2 = new float [20];
```

We could then use the pointers to manipulate the data as we would have done in ‘C’, e.g.

```
*p1 = 5; // set the integer that p1 points to, to the value 5

for(i = 0; i < 20; i++)
    p2[i] = (float)(i); // set all element in the array pointed to by p2 equal to 'i'
```

When we have finished with the variables, we could deallocate their storage using operator *delete* (previously we used *free* in C). For example, in C++ we could release the storage we allocated above like this:-

```
delete p1; // delete the single element pointed to by p1
```

```
delete [] p2;           // delete the array of elements pointed to by p2
```

This new technique for allocating and releasing memory is useful when we introduce classes. For example, we can easily dynamically allocate storage for class objects using a syntax like this. The advantage is the constructor for the object gets called when we "new" it

```
int main()
{
    Rectangle *p1 = new Rectangle(3,4);           // create storage for a rectangle, call its constructor
                                                    // to initialise it with the values 3, 4 and then make
                                                    // p1 point to it
    Rectangle *p2;                                 // just a pointer at the moment

    printf("area = %d\n", p1->get_area() );
    delete p1;                                     // free up the storage

    p2 = new Rectangle [10];                       // allocate storage for an array of 10 rectangles
                                                    // default constructor called for each. You can
                                                    // initialise them all individually using set_values() function
    ...
    delete [] p2;                                  // free up the storage allocated to the array.
    return 0;
}
```

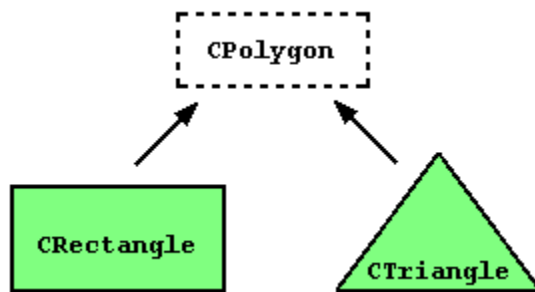
Class Inheritance

One of the main powers of classes is the ability to extend them, creating new classes by basing them on existing ones. This process, known as *inheritance*, involves a **base class** and a **derived class**. The derived class inherits all the member **variables** and **functions** of the base class (i.e. as if you had written them all again yourself in the derived class). You can also add new member variables and functions to the derived class. This is important because it leads to "re-use" of existing code. We can take something that somebody else may have written and add to it rather than *re-inventing the wheel* from scratch each time.

Inheritance also allows us to *override* (or modify) something we inherited from the base class. For example, we could write a new member function in our derived class which has the same name as the function in the base class, but which will do something *different*, i.e. we can tailor the behaviour of our new derived class to make it more suitable for our application.

For example, let's imagine a simple series of classes to describe two kinds of polygons:- **triangles** and **rectangles**. These two polygons have certain common properties, such as the values needed to calculate their **areas**. They both could be described simply with a **height** and a **width** (or **base**) variable.

This could be represented in the world of classes with a class **Polygon** from which we would derive the two other shapes: **Triangle** and **Rectangle**:



The Polygon (our **base**) class would contain the member variables and functions that are common for both rectangles and triangles, in our case the variables **width** and **height**.

Rectangle and Triangle would be new **derived** classes, with specific features that are unique to each type of polygon.

Classes that are derived from others *inherit* all the members of the base class. That means that if a base class like Polygon includes a member variable or function **A** and we derive a new class from it and then add a new member called **B**, to that derived class, then the derived class will contain both member **A** and member **B**. Nothing can be taken out during inheritance, it is not selective; you inherit everything from the base class. However, you can **add** to what you inherit.

Specifying Inheritance in C++

The new derived classes Rectangle and Triangle can be created by deriving them from the base class Polygon using the following example syntax:

```
class Rectangle : public Polygon {  
    ...  
};  
  
class Triangle : public Polygon {  
    ...  
}
```

For this course, we'll mostly stick with public inheritance (though if you look through the course library, you will see others). This simply means that any method or variable that was public in Polygon will remain public in both Rectangle and Triangle. The following is a definition for the base class:

```

class Polygon {
    int width, height;    // private

public:
    Polygon(int w, int h) : width(w), height(h) {}    // constructor
    int getWidth() { return width; }                // `getters'
    int getHeight() { return height; }
    void setWidth(int w) { width=w; }                // `setters'
    void setHeight(int h) { height=h; }
};

```

Now let's create two new classes **Rectangle** and **Triangle** by deriving from the base class **Polygon** and add a **get_area()** function to each

```

class Rectangle : public Polygon {
public:
    Rectangle( int x, int y ) : Polygon(x,y) {}    // calls base constructor
    int getArea() { return getWidth()*getHeight(); } // call inherited functions
};

class Triangle : public Polygon {
public:
    Triangle( int x, int y ) : Polygon(x,y) {}    // calls base constructor
    int getArea() { return getWidth()*getHeight()/2; } // call inherited functions
};

```

Notice how the **Rectangle** and **Triangle** constructors have to call the base class **Polygon** constructor to initialize its members.

Note that since both **width** and **height** were declared as **private** (the default), *only* **Polygon** has direct access to them. Even though both **Rectangle** and **Triangle** both have width and height variables internally, the two classes can only get access to them through **Polygon**'s **set()** and **get()** type functions. Derived classes are constructed from the bottom-up, initializing base class members first, so the base class constructor should be called in the member initialization list.

Even though we never explicitly wrote a **getWidth()** or **getHeight()** function in either **Triangle** or **Rectangle**, both classes have these methods. They were inherited from the **Polygon** class, allowing us to re-use them, which is the whole point. We don't want to have to re-invent the wheel every time.

Let's see how we can use them in an example program

```
int main ()
{
    Rectangle    r1(4, 5);
    Triangle     t1(5, 6);

    printf("Area of rectangle r1 = %d\n", r1.get_area()) ;
    printf("Area of triangle t1 = %d\n", t1.get_area()) ;
    return 0;
}
```

Notice how the `get_area()` function for both `Rectangle` and `Triangle` call upon the `get_width()` and `get_height()` functions from the `Polygon` class.

Overriding members and Polymorphism

If a base class `X` had a member variable or function `A`, and if we derive a new class `Y` from `X` and add its own new member `A`, then the derived class will actually contain two members called `A`. The one actually used depends on the specific type of the reference or pointer. Consider the following. If we use the derived object directly (or a reference or a pointer to the derived object) to invoke a function, then it is the one for the object that gets invoked. If we used a base class pointer or reference, then the function for the base class is involved. See example below

```
/* *****
 * Example of Overriding base class member function (no Polymorphism)
 * ***** */
#include <stdio.h>

// Base class
class Animal {
public:
    void speak() { printf("grunt\n"); }
};

// derived class
class Dog : public Animal {
public:
    void speak() { printf("woof\n"); }
};

// derived class
class Cat : public Animal {
public:
    void speak() { printf("meow\n"); }
};

int main() {
    Dog dog;      // creates a new derived class dog
    dog.speak();  // use derived object to speak. Result = "woof"
}
```

```

Cat cat;      // creates a new derived class cat
cat.speak();  // use derived object to speak. Result = "meow"

Animal &a1 = dog;      // base class reference to a derived class dog
a1.speak();            // use derived class reference to speak result = "grunt"

return 0;
}

```

Polymorphism

Polymorphism refers to the ability to permanently redefine methods for a derived class. This would let the dog bark, for example, regardless of whether we treat it as a Dog or an Animal. To accomplish this, we introduce the keyword **virtual**. A **virtual function** is completely redefined in the derived class. This ensures that the derived class's version is **ALWAYS** called for an object regardless of the expression used. Modifying our previous example slightly,

```

/*****
 * Example of Overriding using Virtual functions (WITH Polymorphism)
 *****/
#include <stdio.h>

// Base Class
class Animal {
public:
    virtual void speak() { printf("grunt\n"); } // NOTE virtual function
};

// Derived Class
class Dog : public Animal {
public:
    void speak() { printf("woof\n"); }          // overrides Animal's copy
};

// derived class
class Cat : public Animal {
public:
    void speak() { printf("meow\n"); }          // overrides Animal's copy
};

int main() {
    Dog dog;      // creates a new dog
    Cat cat;      // creates a new cat

    dog.speak();  // "woof"
    cat.speak();  // "meow"

    Animal &a1 = dog;      // Animal type reference to derived class dog
    a1.speak();           // "woof"

    a1 = cat;           // Animal type reference to derived class cat
    a1.speak();          // "meow"

    return 0;
}

```


What **important** information do we take away from this use of inheritance and virtual functions?

1. A *base class pointer* (or reference) can be initialised to point to either *a base class object* or *any object that is derived* from the same base class.
2. When we use a *base class pointer* (or reference) to invoke a *base class function*, it will be the derived class function that will be called. If the derived class function does not redefine its own version of the base class function, then the base class function will be called.

This can be **INCREDIBLY** useful. For example, suppose we had a linked list of **Animal pointers**. With such a list we could add either Dogs or Cats to it, since both are derived from – or can be thought of as kinds of – animal, more to the point we could add other kinds of animals we create in the future, e.g. rabbits, tigers etc

Afterwards, we could then *iterate* through the list asking each animal within it to “speak” (without knowing what kind of animal it was – only that it was in fact some kind of animal). The dogs in the list would say “woof”, the cats would say “meow”.

Virtual Destructors

The same principle of declaring virtual functions should also be applied to class **destructors**. For example, consider this code

```
/* *****  
 * Example of Overriding using Virtual functions (WITH Polymorphism)  
 * ***** */  
#include <stdio.h>  
  
// Base Class  
class Animal {  
public:  
    virtual void speak() { printf("grunt\n"); } // NOTE virtual function  
    virtual ~Animal() { printf("Animal Destructor\n"); } // NOTE virtual destructor  
};  
  
// Derived Class  
class Dog : public Animal {  
public:  
    void speak() { printf("woof\n"); } // overrides Animal's copy  
    ~Dog() { printf("Dog Destructor\n"); }  
};  
  
// derived class  
class Cat : public Animal {  
public:  
    void speak() { printf("meow\n"); } // overrides Animal's copy  
    ~Cat() { printf("Cat Destructor\n"); }  
};  
  
int main() {  
    Animal *a1 = new Dog(); // creates a new dog  
    Animal *a2 = new Cat(); // creates a new cat
```

```


    a1->speak();
    a2->speak();

    delete a1;
    delete a2;

    getchar();
    return 0;
}

```

When this program is run, the output looks like this

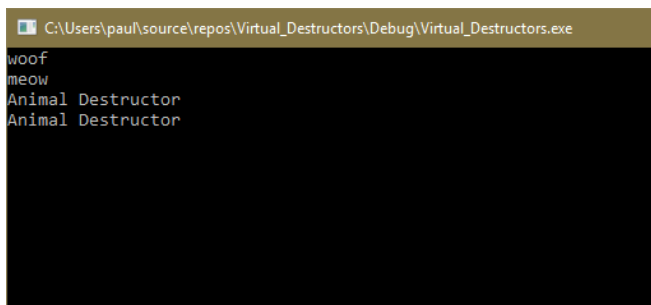


```

C:\Users\paul\source\repos\Virtual_Destructors\Debug\Virtual_Destructors.exe
woof
meow
Dog Destructor
Animal Destructor
Cat Destructor
Animal Destructor

```

Notice how the derived class destructor is called first, followed by the base class destructor even though the cats and dogs were “**deleted**” using only **animal** pointers. Without the “**virtual**” keyword in the base class destructor, the output would look like this:-



```

C:\Users\paul\source\repos\Virtual_Destructors\Debug\Virtual_Destructors.exe
woof
meow
Animal Destructor
Animal Destructor

```

Notice how the derived class destructors for cats and dogs were not called. So as a **rule of thumb**, if you **know** your classes are going to serve as **base classes** for **derived classes in the future**, then declare the base class member functions as **virtual**. There will be small penalty in size of code and execution speed for doing this, but nothing significant.

Replacing printf() and scanf() with cout and cin

Of course there’s lots of other stuff related to C++ that we haven’t covered here, as this was only intended to be a quick “**get you up to speed**” tutorial on object oriented programming in C++, not a complete course.

For instance we didn't use `cin`, `cout` and the "`iostream`" header file as a replacement for `scanf()`, `printf()` and `stdio.h` (see http://www.cplusplus.com/doc/tutorial/basic_io/). This is very easy to use and you might want to experiment with this in conjunction with "namespaces".

For more details on things such as how to set field width and field precision (analogous to using a formatter such `%6.2f` for example) take a look here <http://www.cplusplus.com/reference/iolibrary/>

Namespaces

A namespace is similar to the concept of a package in Java. It allows for the possibility of having two global variables or functions with the same name but with each declared inside their own namespace. For example `cin` and `cout` are in the standard library but within the sub-library/namespace "`std`" inside the header file "`iostream`". To gain access to them, we have to declare that we are using namespace "`std`" as shown below

1st approach – add a "`using namespace std`" declaration to your source files

```
// i/o example

#include <iostream>
using namespace std;           // use global fns/vars from "std" namespace

int main ()
{
    int i;
    cout << "Please enter an integer value: ";      // cout and cin in "std"
    cin >> i;

    cout << "The value you entered is " << i;
    cout << " and its double is " << i*2 << ".\n";

    return 0;
}
```

Some programmers *object* to the "`using namespace std`" declaration in their code as it means we end up bringing in ALL the names from the standard library and, by definition, once we have told the compiler we are using that namespace, other namespaces become "invisible".

2nd approach – prefix all names from the standard library with `std::`

Here we add a prefix all individual names from a library with their namespace. For example we would have to prefix every use of `cin` and `cout` with "`std::`" as shown below.

```
// i/o example

#include <iostream>
// NOTE no namespace declared
```

```
int main ()
{
    int i;
    std::cout << "Please enter an integer value: ";
    std::cin >> i;

    std::cout << "The value you entered is " << i;
    std::cout << " and its double is " << i*2 << ".\n";

    return 0;
}
```

NOTE For the sake of brevity and because real-estate on a power point slide is limited, I'm going to use the 1st approach where we explicitly include a **"using namespace std"** declaration

Why bother with cin and cout?

Because it now provides an extensible way to use the same "<<" and ">>" operators to output and input to our user defined types e.g. That is, we can provide a meaning for all of C++ operators for our new classes

For example, if we had a Rectangle class as declared previously, e.g.

```
Rectangle r1(5, 6);
```

We could NOT at the moment write

```
cout << r1 << "\n";
```

because C++ does NOT know how to interpret and apply the << operator to user defined classes. However, we could write a function that tells the compiler how to interpret that operator. An example is shown below based on the use of a friend function which is a global function that can access the private members of a class, but is **NOT** a member of that class. Note the name of the function is called **"operator<<"**. We could provide an implementation of this operator (and indeed many other C++ operators: e.g. ++, --) for any of our user defined classes. This is an example of a topic generally called **"operator overloading"**.

```
friend ostream &operator<<( ostream &o, const Rectangle &r )
{
    o << "Width = " << r.width, << ", Height = " << r.height;
    return o;
}
```

Here, because "<<" in C++ is a *binary* operator taking two operands, called with a syntax such as **cout << i;** we need a **"<<"** function that takes two parameters. The first will be an instance of an output stream, the second will be an instance of a variable that we want to output to the stream.

Thus when the compiler see the statement `cout << r1;`, the variable `cout` would be passed to the function (by reference – see function code above) and become the parameter “o”, while “r1” would become the 2nd parameter “r”. The function should also return the original ostream object (cout in this example usage) to allow *daisy chaining* of the << operator e.g. `cout << r1 << r2` etc.

The result of `cout << r1;` is that we would see the following printed.

```
Width = 5, Height = 6
```

Further Reading

Template classes

Lastly template classes offer a very powerful way to specialise a class at compile time, which is particularly useful with things like linked lists and other dynamic data structures. For example we could say

```
List<int>      L1
List<Rectangle> L2
```

Where “<xxx>” is the C++ syntax for creating specialised (typed) versions of a generic List class. In this case we created L1 to be a List of *int* variables, but L2 to be a List of Rectangle’s. The trick here is that the compiler uses the <T> bit (where T is a type: int or Rectangles in this example) to replace our class variables and member function parameters with the “T” e.g. a basic List class which might, without the aid of templates, have declared variables such as `void *head`, could now be declared as `T *head`, that is, a specialised type.

When the compiler see the introduction of the variable L2 above, it substitutes `Rectangle` in place of `T` in the class code and creates/compiles a whole new class to deal with Rectangles. You can read about it here <http://www.cplusplus.com/doc/tutorial/templates/>

The Standard Template Library

Using templates classes is so useful that a library of standard template classes has been created (the STL or standard template library) that can be used in C++ programs and contains such things as dynamically sizeable string, lists, vectors etc as well as algorithms on sorting and searching etc. A tutorial on using this library in your programs can be found on Connect beneath this lecture.

Don't get hung up if some of these more advanced concepts sound a little complicated, this whole document gives quite a bit of the essential background (even though it's not complete) to C++, we don't use half as much of this in CPEN 333 as you might think but it is useful to know

the background (and be able to put it on a resume), so read it, but don't agonise over it if you don't get it 1st, 2nd or even 3rd time through.