Programming with Objects

Objects are just clever variables – variables which can have both data and functions associated to them. We can declare objects in much the same way as we would declare a variable and then that instance of the object has access to all of the member functions defined within it’s class – the class is the blueprint for creating an object, containing all the attributes (data) and behaviours (functions) that are associated to it.

**Structs**

An object is a variable which has member functions, and a class is a data type whose variables are objects. Therefore when defining a class, we have to define the data part (which kinds of values the variables can hold), and the function part (what the member functions are). The first step towards this is to look at ***structs*** (or structures). Structs can deal with the data part of an object, but not the function part. So, they are objects without members – a type of variable which can store multiple, different types of data. Defining a struct involves defining the different types of data within it.

A date has 3 elements – the day, the month and the year. All three of these elements can be stored as integers, so a date can be defined as a struct;

struct Date  
{  
int day;  
int month;  
int year;  
};

Note the semicolon at after the final brace.

Once the struct has been declared, an instance of it can be declared;

Date birthday;

‘birthday’ is now a variable which can be used to store data in much the same way as other variables. To assign a value to a struct you assign a value to each element in the struct;

birthday.day = 31;  
cout << birthday.day;

Notice again the ‘dot’ notation, indicating that ‘day’ is a member variable of the birthday – a Date type struct. Once the struct has been declared it can be used in the same way as the other pre defined data types, for instance;

|  |  |
| --- | --- |
| Date get\_values(); | For a functions return value type. |
| Date birthday{31,02,2005}; | Assigning values during type declaration. |
| struct Person  {  double height;  double weight;  Date birthday; }; | As a type within a new structure declaration! |

This final example gives us our first view of hierarchy- where one struct is a member of another struct. To assign a value in this case;

Person me;  
me.birthday.day = 31;

**Classes**

So, classes are the blueprints defining an object. Objects are variables which can have data values and member functions associated to them. We define the data values associated to an object in much the same way as the data values are defined for a struct. Last week we saw how to call a member function for an object (using the dot notation). All that is left is to see how to define the member functions of a class;

A class definition contains all its functions and variables;

class DayOfYear

{  
public:  
 void input();  
 void output();  
 int month;  
 int day;  
};

In this example, the name of the class is DayOfYear – it has 2 integer type member variables (month and day) and 2 member functions (input and output).

Instances of the object DayOf Year can be created within any other function;

DayOfYear today;

Values can be assigned to the integer variables included;

today.month = 3;

today.day = 13;

The member functions can be called by;

today.output();

But we still need to include a function definition to define output;

void DayOfYear::output()

{  
 cout << “Month = ” << month << “, day = ” << day << endl;  
}

This function definition can appear along with any other function definition, the ‘::’ operator defines the scope of the function – i.e. that it only exists in the DayOfYear object. While the ‘.’ (dot operator) is used with objects (or instances), the ‘::’ (scope resolution operator) is used with the class.

**Public & Private!**

Notice the keyword ‘Public:’ at the start of the class definition, an alternative keyword is ‘Private:’. Variables and functions which appear within the ‘private’ part of the class definition can only be accessed from within member functions – not the main program. It is a good idea to try and make all member variables private, so that they can only be accessed or changed when using member functions – in much the same way we recommend only using local variables within functions not global variables. Some functions can also be private functions – i.e. functions which are only called by other functions within the class. So an alternative class definition could look like;

class DayOfYear|  
{  
public:  
 void input();  
 void output();  
 void setDay(int new\_day);  
 void setMonth(int new\_month);  
 int getMonth();  
 int getDay();  
private:  
 void check\_date();  
 int month;  
 int day;  
};

Here the variables and one function have been made private, so only other functions can access them. So, when an instance (birthday) of ‘DayOfYear’ is declared it has 2 variables (month and day). In this case we aren’t able to manipulate them directly though;

birthday.month = 12 <– NOT ALLOWED!

Because of this, further functions have been declared, the set and get functions. These are known as ‘***mutator***‘ and ***‘accessor***‘ functions respectively. Mutator functions ‘mutate’ or change the data stored within a variable and normally (but not necessarily) have the word ‘set’ in their name. Accessor functions ‘access’ or get the data stored within a variable and normally (but not necessarily) have the word ‘get’ in their name. So we could expect the get\_month() function (when defined) to return an integer value for the corresponding month of the year;

month = get\_month(); <– This would work, while

month = birthday.month; <—would NOT!

It is normal for the public member functions for objects to be accessor and mutator type functions – so that the data stored in an object can be got and manipulated, rather than using public member variables.

|  |
| --- |
| **Classes summary**  1) Classes have both member variables and member functions.  2) A member (either member variable or member function) may be either public or private.  3) Normally, all the member variables of a class are labeled as private members.  4) A private member of a class cannot be used except within the definition of another member function of the same class.  5) The name of a member function for a class may be overloaded, just like the name of an ordinary function. (remember overloading?)  6) A class may use another class as the type for a member variable.  7) A function may have formal parameters whose types are classes.  8) A function may return an object; that is, a class may be the type for the value returned by a function. |

**Constructor Function**

As well as accessor and mutator type functions, another common type of function is a constructor function. A ‘***constructor***‘ is a member function of a class that has the same name as the class. A constructor is called automatically when an object of the class is declared. Constructors are used to initialize objects. A constructor must have the same name as the class of which it is a member. For a constructor function, no return type is give – not even ‘void’. Constructors can’t be called in the same way as the other functions that have been discussed, they are only called when a new instance of the object is declared.

Consider the DayOfYear object, we can include a DayOfYear constructor function;

class DayOfYear

{  
public:  
 DayOfYear(int day, int month);  
 ….

When declaring a new instance of this class now, we would have to assign values to the day and month variables;

DayOfYear birthday(31, 1);

This declaration would then call the actual constructor function, which could be;

DayOfYear::DayOfYear(int day, int month)  
{  
 if ((day > 31) || (month > 12))  
 {  
 cout << “Illegal value for day or month”;  
 exit(1);  
 }  
}

Alternatively we can overload the DayOfYear constructor function to allow for occasions when no initial values are given for the date.

class DayOfYear  
{  
public:  
 DayOfYear(int day, int month);  
 DayOfYear();  
 ….

And

DayOfYear::DayOfYear() : day(1), month(1)  
{  
 // Body intentionally empty  
}

When an instance of the object is declared, one of the two constructor functions is called depending on what values are given. The second constructor is the default constructor, which will create the object for the 1st of January. This way the programmer guarantees that a value has been given for all objects. It is a good idea to always include a default constructor.

### Friend Functions

A class declares the ***member functions*** and ***variables*** for an object, these ***members*** can either be ***public***  
or ***private***. Member variables are placed in the private section of the class as it allows the programmer to control access to them, making sure they are always manipulated in a safe, predictable way, only by other member functions. Accessor, mutator and constructor functions are then used to manipulate these member functions. Sometimes it is useful to write functions using objects, which aren’t member functions, for example if we wanted to test if two different objects were the same, such as if two dates were the same (or two fractions were the same!).

Remember the date class we wrote? If we write a compare function to check if today’s date matched  
the birthday input. There are several ways of doing this – firstly, we could write a member function;

void DayOfYear::compare(DayOfYear date)  
{  
 if(month = = date.month && day = = date.day)  
 cout << “Happy Birthday!\n”;  
 else  
 cout << “Happy Unbirthday!\n”;  
}

This function can then be called by one object, with the second object as a parameter. This isn’t ideal as it means arbitrarily choosing one of the dates as the calling date – (should we test if today is the same as birthday, or birthday is the same as today). Sometimes it isn’t clear which object should take precedence, particularly when dealing with equality. An alternative implementation would be to use a non-member function where both objects are sent as parameters.

bool equal(DayOfYear today, DayOfYear birthday)  
{  
 return (today.get\_day() == birthday.get\_day() && today.get\_month() == birthday.get\_month());  
}

In this case we make use of the member accessor functions get\_day() and get\_month(), as part of a boolean expression.

Wouldn’t it be easier though, if we didn’t need to use the accessor functions? If we could just access the day and month directly? A simpler and more efficient definition for equal could be;

bool equal(DayOfYear today, DayOfYear birthday)  
{  
 return (today.day = = birthday.day && today.month = = birthday.month)  
}

However, this won’t work as the equal function doesn’t have access to the private member variables. That is unless we make the function a ***friend function***. Friend functions are not member functions of a class, but they do have access to the private member variables. To make a function a friend, add the function declaration to the class definition preceded by the keyword ‘friend’.

class DayOfYear  
{  
public:  
 friend bool equal(DayOfYear today, DayOfYear birthday)  
 …  
}

Aren’t friends great? Why don’t we make all functions friends and then there is no need for accessors, mutators etc.? True, but not practical – if you want to reuse your object for some other purpose you will still need accessors to get to the member variables (for instance if you wanted to calculate how many months before your birthday, you would need to get the current month).

So why bother with friends? We could just use the accessors to get to the private members? Also true, but sometimes using friends can make a function definition simpler and more efficient.

So when to use friends, and when to use accessors etc.? A brief rule of thumb is to use nonmember (friend) functions when the task involves using more than one object (for instance the equal function). This is ok, but not always sufficient – member functions can still be useful for functions involving more than one object – if one object if the task is more related to one object. As you can see, either option works, and so either option is correct – in time you’ll decide when each is more appropriate.

### More on ‘const’

When working with functions, we have been using call by reference and call by value parameters. The call by reference parameter is more efficient than a call by value parameter – call by value parameters declare a local variable within the function, meaning there are often 2 copies of the value. With call by reference, there is still only one copy. Two copies of a simple data type is not that serious a problem, but when we send more complex datatypes such as objects as parameters in functions the duplication can become important.

So far we have used call by value for functions which don’t change the value of the parameter, for example we have used call by value for functions which display output. We could instead just use a call by reference parameter to save making a copy of the value;

output(date& birthday)  
{  
 cout << day << month;  
}

This would cause no problems for C++. However, by sending call by reference parameters, it means there is a possibility that values could be changed (perhaps by poor coding). So, to ensure that values aren’t changed the keyword const can be added to make a ***constant call by reference parameter***.

output(const date& birthday)  
{  
 cout << day << month;  
}

Notice this declares a constant of type date, in the same way as the expression;

const pi = 3.1415

The const modifier can also be used with member functions. If a member function shouldn’t be allowed to change the value of the calling object the function can be marked const. This time const appears just before the semi colon. In this member function declaration the output function is not able to change any values of its calling object.

void output(ostream& outs) const;

Using const is simply a way of automatic error checking – making sure that there are no problems with changing values that shouldn’t be changed. However, it is an all or nothing option – if you want to use const to prevent possible problems, you have to use const for all functions and all parameters which shouldn’t be changed. Otherwise one function with a constant parameter may call another function which allows the parameter to change.

### Overloading Operators

The function we looked at earlier tested if two dates were the same, and it was called by;

if(equal(today, birthday))

However, wouldn’t it be nicer if we could just do;

if (today == birthday)

For basic datatypes we can test if two values are equal by using the “==” operator. We can do this as well for testing equality between more complex datatypes such as objects, but first we need to overload the “==” operator. Similarly we could write a function to add two fractions – add(f1, f2), which would be better as (f1 + f2). By overloading the “+” operator this is possible.

So, within the class definition we may want a friend function to compare two dates, our new function declaration would look like this;

friend bool operator == (const date today, const date birthday);

The definition for this function however can stay the same. Notice the operator keyword. Similarly in the fraction class a function declaration might look like;

friend fraction operator +(const fraction f1, const fraction f2);

There are some rules for overloading operators though;

* At least one argument must be of a class type.
* An overloaded operator can be, but doesn’t have to be a friend of a class.
* You can’t create a new operator, only use the existing operators, such as ‘+’, ‘-‘, ‘\*’, ‘/’ etc.
* You can’t change the number of arguments an operator takes (+ always takes 2, ++ always takes 1)
* You can’t change the precedence order of an operator.
* You can’t overload the (.) or (::) operators.

Suppose we were to write a fraction class and attempt to do this;

f2 = f1 + 2

Here f2 is a fraction, and f1 is a fraction which already has a value. The “+” operator has now been overloaded so it can deal with adding two fractions together, but hasn’t been told how to deal with a fraction and an integer – so how will it work? Firstly the computer checks to see if “+” has been overloaded to deal with a fraction and an integer, if not it then looks to see if there is a constructor for the fraction class which can deal with a single integer. As our fraction class includes this constructor, the computer will then convert the type of the second input (2) from integer to fraction (2/1) based upon the constructor. Therefore we can use constructors to automatically convert types.

**Separate Compilation**

We should by now be comfortable with writing classes for our own objects, such as the fraction class. As our programs get bigger, and combine more objects (more data types), then the code is likely to get more complicated. Using objects has allowed us to break the code down into manageable chunks – the object is now a ***black box***which a programmer can use, without needing to look at how it works – the programmer just needs to know what the object can do. This means we can move object definitions to a new file and compile it separately.

Consider the <fstream> library. The <fstream> library defines the stream objects we need to use file input and output (ifstream and ofstream). We can find out about what these streams can do, by looking in a book or helpfile, but we never see how they do it – we have no need to. What we can see by looking in a book etc. is actually the class definition (with the function declarations and comments) but not the function definitions. We can then choose to include this class in our project if it will do what we need it to.

So now we have 3 levels or files,

Our ***Project*** ***File*** (which can include the objects we need)

an ***Interface File*** (which describes what the object can do)

and an ***Implementation File*** (which actually defines how it does it)

C++ provides us with some existing files, and we can include this in our Project File, and we can look at the Interface File so we know how to use it, but we never need to see the Implementation File.

When we write our own objects, we can break them down into appropriate files too. The Implementation File contains the function definitions, the Interface File contains the class definition and the Project File has an ‘include’ statement to include the Interface file in our project.

1) When writing a class, it is normal to begin by writing the definition (to define which data members and which member functions the object should have). This should be saved as a ‘***header***‘ file, with the file suffix ‘***.h***‘. Consider putting the class definition for the rectangle class in a file called ‘***fraction.h***‘. This file should be well commented to explain what each function does and how to use them, as the actual function definitions don’t appear in this file.

2) Next, write the implementation file where all functions (including friends) and overloaded operators that are declared in the header file can be written. This file should begin with an ‘include’ statement, instructing the computer to use the appropriate header file, so ‘***#include “fraction.h”***‘. Notice that the include function contains the full name of the header file, and also that the header file is surrounded by quotes ‘***“***‘ and not ‘***<***‘ as we have used thus far, this is for user defined header files (or libraries). This file could be saved with the extension ‘***.cpp***‘. Traditionally, but not necessarily, the implementation file and the interface file have the same filename, with different file extensions. Consider moving all the function definitions to a file called ***fraction.cpp***‘ and adding the appropriate include statement.

3) Finally we can write the Project file (or application file). In this file we just need to have the include directive ‘***#include “fraction.h”***‘ and then we can have access to create the new datatype, and use it’s member functions.

When you then compile and run your program each of the included libraries are compiled separately from the main project file. Before too long, we can write very complex programs which use many libraries and many objects, now stored in many files. If our program contains many files, it is very possible that we may have more than one file which uses the same library. In this case we may have more than one compiler directive – more than one ‘***#include “rectangle.h”***‘ in our file. Each time the compiler encounters this command it is instructed to compile the appropriate interface and implementation files. Unfortunately C++ doesn’t  
allow the same class to be defined more than once, even if the definitions are identical. So to avoid this we can use the ‘***#ifndef***‘ command, which means ‘if not defined’. At the start of our interface file, we include the following lines;

***#ifndef RECTANGLE\_H***

***#define RECTANGLE\_H***

and after the end of the class we add the following line;

***#endif //RECTANGLE\_H***

This means, if the rectangle class hasn’t been defined yet, then define it, otherwise, skip forward until the endif. We can call the rectangle\_H whatever we like, it merely represents the class and when the compiler defines the class it is placed on a list to track which classes have been defined.