Objects and Classes

Object Orientation and C++

- C++ supports object oriented programming
  - Initial implementation of C++ was "C with Objects"
- Follows the classical model of OO
  - Class defines an aggregate type
- Supports encapsulation
  - May contain data and functions (methods)
  - Visibility controls can be applied to class members
- Objects are instances of class types
  - More options on working with objects than in Java

Defining a Class

- Class will have a name
  - Define members of the class

```
class Job
{
    int  hours;
    int  rate;
    int  jobNum;

    int  GetAmount()  { return rate * hours; }
};
```

Class 'members'. The Job class contains 3 'member variables' (sometimes called 'fields') and a 'member function' (sometimes called a 'method')

In this example the body of the function is contained within the class. Our later examples will implement functions in a separate file

Creating Objects

- C++ allows objects to be created in any storage class
  - Not just on the heap, like Java
- Define object just like any other type instance

```
Job job;

// OR

Job * pJob = new Job;
```

In these example a 'p' prefix will be used on names of variables of pointer type

Visibility of Class Members

- Class definitions contain "sections" that control visibility of the members defined in them
- The visibility controls are somewhat similar to those in Java
  - Though not identical
  - Default visibility is private

```
class Job
{
private:
    int  hours;
    int  rate;
    int  jobNum;
public:
    int  GetAmount()    { return rate * hours; }
};
```

Using the Class

- Private members can only be accessed from within the class definition itself

```
Job job;

job.hours = 2;
job.rate = 30;

int amount = job.GetAmount();
```

Gives compiler error: 'hours' : cannot access private member declared in class 'Job'

Accepted by the compiler as GetAmount is public

Mutator Functions

- Setter functions normally declared as void

```
class Job
{
private:
    int  hours;
    int  rate;
    int  jobNum;
public:
    void SetHours( int hoursValue )        { hours = hoursValue; }
    void SetRate( int rateValue )          { rate = rateValue; }
    void SetJobNum( int jobNumValue )      { jobNum = jobNumValue; }
    int  GetAmount()                       { return rate * hours; }
};
```

Using the Mutator Functions

```
Job job;

job.SetHours( 2 );
job.SetRate( 30 );
job.SetJobNum( 1234 );

int amount = job.GetAmount();
```

Using Pointers with Objects

- Use special -> operator to derefence pointer and access object member

Creating an object using new

```
Job * pJob = new Job;
pJob->SetHours( 4 );
...
delete pJob;
pJob = 0;
```

Passing by address

```
Job job;
job.SetHours( 2 );
job.SetRate( 30 );
job.SetJobNum( 1234 );
OtherFunction( &job );
```

```
void OtherFunction( Job * pJob )
{
    int cost = pJob->GetAmount();
}
```

Implementing Functions outside the Class

- Member functions must be declared inside class
- Definitions may be outside
  - Possibly in a separate source file
  - Recommended for more complex functions
  - Use :: operator in function definition

The class definition (Job.h):

```
class Job
{
private:
    int  hours;
    int  rate;
    int  jobNum;
public:
    int  GetAmount();
    ...
};
```

The class implementation (Job.cpp):

```
int Job::GetAmount()
{
    return rate * hours;
}
```

Initialising Data Members

- Default initial value of object data members will depend on storage class
- C++11 and later allow initialisation in class definition

```
class Job
{
private:
        int  hours = 0;
        int  rate = 50;
        int  jobNum = -1;
public:
        …
};
```

Until C++ 11, data member initialization could only be done in the constructor:

Constructor

- All classes have a constructor function
  - Used for initialisation of data members
  - Compiler generated if none explicitly provided
- Constructor function has same name as class
  - No return type specification

Job.h

```
class Job
{
private:
     int  hours; …
public:
     Job(); …
};
```

Job.cpp

```
Job::Job()
{
     hours = 0;
     …
}
```

Constructor Overloading

- Constructor functions may be overloaded like any other function
  - No-arg constructor known as "default" constructor

Job.h

```
class Job
{ …
public:
      Job();
      Job( int hoursValue, int rateValue, int jobNumValue );
```

This is an example of function overloading

Job.cpp

```
Job::Job( int hoursValue, int rateValue, int jobNumValue )
{
      hours = hoursValue;
      rate = rateValue;
      jobNum = jobNumValue;
}
```

Using the Constructor

- Constructor can be invoked for dynamically or statically created objects

```
Job j( 5, 32, 17 );
```

```
Job::Job( int hoursValue, int rateValue, int jobNumValue )
{
      …
}
```

```
Job * pJ = new Job( 6, 32, 18 );
```

Classes Without Default Constructor

- If a constructor is defined, the default constructor is not generated
  - Cannot create objects without invoking an explicit constructor

```
class Job
{
        …
public:
        Job( int hoursValue, int rateValue, int jobNumValue )
        …
};
```

This version of the class only has one constructor, and this takes 3 arguments

```
Job j1( 5, 32, 17 );
Job j2;
```

Gives compiler error 'Job' : no appropriate default constructor available

Single Argument Constructors

- Single argument constructor can be invoked using direct initialisation syntax
  - Using = operator
  - Cannot be done for dynamic objects

```
Job::Job( int jobNumValue )
{
    jobNum = jobNumValue;
}
```

```
Job j1 = 5;
Job j2( 5 );
```

These two mechanisms are almost always equivalent

Forcing Explicit Constructor Invocation

- The explicit keyword disables the ability to use the initialisation syntax to invoke a single argument constructor
  - Can make code more readable

```
class Job
{
public:
        explicit Job( int jobNumValue );
};
```

```
Job j = 5;
Job k( 5 );
```

- Also prevents other type being transparently converted into class type

```
int SomeFunction( Job j )
{
        ...
}
```

```
SomeFunction( 6 );
```

Without the explicit, this would construct a Job passing 6 to the constructor and pass that object to the function

Destructor Functions

- Destructor is used to perform desired actions when an object's lifetime expires
- Usually performs some form of cleanup
  - Release dynamically allocated objects
  - Close files, network connections
  - Release locks
- Useful in C++ since object lifetime is deterministic

```
int SomeFunction(...)
{
    Job job;
    ...
}
```

Class object declared as local variable. Created on the stack and constructed when function is called

Destructor is automatically called when the function exits as the local variable is no longer in scope

Writing a Destructor

- Destructor function has same name as the class, preceded with the ~ character
  - Cannot pass arguments to the destructor
  - Nothing returned

Definition (.h):

```
class MyClass
{
private:
    int *pArray;
public:
    MyClass( int size );
    ~MyClass();
};
```

Implementation (.cpp):

```
// Constructor
MyClass::MyClass( int size )
{
    pArray = new int[size];
}
// Destructor
MyClass::~MyClass()
{
    delete [] pArray;
}
```