B39HV2

SOFTWARE ENGINEERING II

UNIT 2: Introduction to Classes

Aims

- In this unit we will consider the following topics:
  - Classes in C++
  - `public` and `private` membership levels
  - member functions
  - creating objects
  - constructor functions
  - the copy constructor
  - destructor functions
  - static data members
  - static functions

Classes

- **In C++, object-oriented programming is implemented using *classes*.**
  - **A class is a *user defined data type* similar to a structure.**
  - **Like a structure, a class can contain a number of members.**
  - **Unlike a structure:**
    - **Classes may contain functions as members.**
    - **Classes contain 3 levels of membership.**

An example class

- Classes are defined in a similar manner to structures.
  - Consider a class called `Rational` that may be used to represent rational numbers:

```
class Rational {
public:
   void assign(int, int);
   double convert();
   void invert();
   void print();
private:
   int num, den;
};
```

  - NOTE: In this treatment of classes, we will follow the examples in Hubbard "Programming with C++", McGraw-Hill.

An example class

1. Definition begins with keyword `class`
2. Followed by class name
3. Class definition contains `public` and `private` sections.
4. `public` section contains member function prototypes.
5. `private` section contains data member declarations.
6. Class definition ends in a semicolon.

```
class Rational {
public:
    void assign(int, int);
    double convert();
    void invert();
    void print();
private:
    int num, den;
};
```

`public` and `private` membership levels

- **One noticeable aspect of this class definition, is the use of the keywords `public` and `private`.**
  - **These membership levels are used to implement the *information hiding* concept behind the principal of *encapsulation*.**
  - **In this example, the data members `num` and `den` are placed in the `private` section of the class.**
    - **This means that these data members can *only be accessed from within* the class.**
  - **The member function declarations are placed inside the `public` section of the class.**
    - **This ensures that these functions can be accessed from *outside* the class.**

`public` and `private` membership levels

- The `public` member functions will be employed by the users of the class.
- The `private` data members cannot be accessed directly, and hence we must use the provided functions to manipulate the data members.
- In general, we will declare member functions to be `public` and data members to be `private`.
- The default class membership level is `private`.
- We will discuss a third membership level, `protected`, when we consider inheritance.

Function definitions

- **The class definition tells the compiler about the existence of the class, its name and its function and data members.**
- **The definition of the member functions provide the *implementation* of the class.**
  - **e.g. The member function `assign()` is *defined* as follows:**

```
void Rational::assign(int n, int d)
{
    num = n;
    den = d;
}
```

The scope resolution operator `::`

- There are a couple of things worth mentioning about this function definition:
  - Firstly we have prefixed the function name with the name of the class and 2 colons (`::`).
  - The 2 colons are known as the *scope resolution operator*.
  - This operator is used to specify the *scope* of the function. The class of which the function is a member is named before the operator.
  - As this function was declared within the `Rational` class declaration, its scope lies within the class `Rational`. The function is not visible outside this class.
  - We need to inform the compiler of this relationship when we define the function.

```
void Rational::assign(int n, int d)
```

Accessing data members

- Also of interest, is the way in which the data members `num` and `den` are accessed from within `assign()`.
  - As the function `assign()` is within the scope of the class `Rational`, we can access the data members directly:

```
void Rational::assign(int n, int d)
{
    num = n;
    den = d;
}
```

  - This is similar to the way in which we can access global variables from within a non-member function. i.e. these variables are not declared within the function and are not passed as parameters.

```
int count=0; // count is global
void increment() {
   count++;  // we can access count from inside function
}
```

Implementing the other functions

❍ **The implementation of the other functions is as follows:**

```
double Rational::convert()
{
    return double(num)/den;
}
```

```
void Rational::invert()
{
    int temp = num;
    num = den;
    den = temp;
}
```

```
void Rational::print()
{
    cout << num << "/" << den;
}
```

Using the class

- We can now use the class `Rational` to solve problems involving rational numbers:

```
void main()
{
   Rational x; // x is an object of class Rational
   x.assign(22,7);
   x.print();
   cout << "=" << x.convert() << endl;
   x.invert();
   x.print();
   cout << "=" << x.convert() << endl;
}
```

Declaring an object of class `Rational`

- **In this example we declared a single variable `x` of type `Rational`.**
  - **`x` is known as an *object* of class `Rational`.**
  - **The object `x` will represent an *instance* of the class `Rational`.**
  - **The object `x` will have its own data members `num` and `den` which allow it to represent a particular Rational number.**
  - **The object `x` will also have the member functions `assign()`, `convert()`, `invert()` and `print()`, which allow us to access and manipulate its data members.**
  - **We cannot access these data members directly (e.g. `x.num = 21`) as they are declared `private`.**

Constructor functions

- **In our example, we provided a function `assign()` to set the data members `num` and `den`.**
- **Ordinary (predefined) data types can be initialised on declaration:**

```
int n = 22;
char name[] = "Jamie";
```

- **It is possible to achieve the same effect with classes using a special member function called a *constructor function*.**

A constructor function is called automatically when an object is declared.

Constructor functions

- A constructor function must have the same name as the class, and is declared without a return type.
  - We may modify the `Rational` class to provide a constructor function in place of the member function `assign()`.
    - (For simplicity, we will drop functions `convert()` and `invert()`).

```
class Rational {
public:
    Rational(int n, int d) { num=n; den=d; }
    void print();
private:
    int num, den;
};
```

Inline member functions

- **In this example we have provided the *definition* for the constructor function inside the class definition.**
  - **This means that the constructor function is *inline*.**
  - **Earlier, we saw that inline functions are replaced by their implementation code when they are called.**
  - **This has the effect of reducing execution time at the expense of increased executable code size.**
  - **Any member function that is *defined* inside the class definition is automatically declared as inline.**
  - **In practice, we will tend to define constructor functions within the class definition and other member functions outside.**

Using the constructor function

- The constructor function provided here, takes two arguments that are used to initialise the data members `num` and `den`.
  - The function is called each time we declare an object of the class Rational:

```
Rational x(22,7);
```

  - After this declaration, `x` will have a numerator of 22 and a denominator of 7.
  - As we have provided a constructor function that takes 2 arguments, we <u>must</u> initialise each object as shown above.

WHY?

The default constructor

- The answer to this, is that the compiler will create a constructor function, if you do not.
  - This constructor function is known as the *default constructor*, and will have no parameters.
  - It was this constructor that was called in our first version of the class `Rational`, when we declared an object thus:

```
Rational x;
```

  - When we provided our own constructor function (requiring 2 parameters) the default constructor was not supplied.
  - Hence, we must supply the necessary parameters.

Declaring more constructors

- As we may not wish to initialise all objects of class `Rational`, it makes sense to make provision for this.
  - This is possible by providing a number of constructor functions. In other words, we will *overload* the constructor:

```
class Rational {
public:
   Rational() { num=0; den=1; }
   Rational(int n) { num=n; den=1; }
   Rational(int n, int d) { num=n; den=d; }
   void print();
private:
   int num, den;
};
```

Calling the overloaded constructors

- If we do not provide any arguments, when we declare an object, the first constructor will be called and the `num` and `den` are initiliased as 0 and 1.

```
Rational x;
```

- If we provide a single parameter, the second constructor is called and `num` is assigned to the specified value and `den` is set to 1.

```
Rational x(3);
```

- If we provide 2 parameters, the third constructor is called as before.

```
Rational x(22,7);
```

Constructors with default parameters

❍ **A better approach is to provide a single constructor, and supply *default* values for the parameters:**

```
class Rational {
public:
   Rational(int n=0, int d=1) { num=n; den=d; }
   void print();
private:
   int num, den;
};
```

❒ **This constructor will implement all three types of initialisation.**

Constructor initialisation lists

- **The most common use of constructor functions is to initialise data members.**
  - **Hence, C++ provides a special syntax that simplifies this procedure, known as an *initialisation list*. (Only for constructor functions.)**
  - **We may rewrite our constructor function in this form as follows:**

```
Rational(int n=0, int d=1): num(n), den(d) { }
```

  - **The initialisation list begins with a colon after the parameter list.**
  - **The data members `num` and `den` are assigned to the parameter values `n` and `d` as `num(n)` and `den(n)` (separated by commas).**
  - **Note that the function body is now empty.**

Access functions

- **Although it is common to declare data members as `private` in order to limit access, it is often necessary to provide member functions that provide *read-only access*.**
  - **These functions are known as *access functions*.**
  - **We have provided access functions for the `Rational` class below:**

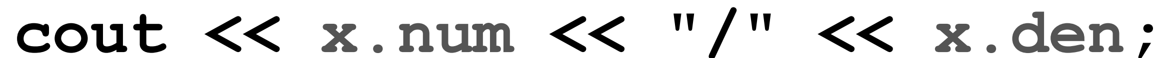
```
class Rational {
public:
   Rational(int n=0, int d=1): num(n), den(d) {}
   int numerator() const { return num; }
   int denominator() const { return den; }
   void print();
private:
   int num, den;
};
```

Using access functions

- We can use these access functions as follows:

```
void main()
{
   Rational x(22,7);
   cout << x.numerator() << "/" << x.denominator();
}
```

- The following code would not be allowed due to the `private` status of the data members `num` and `den`.



X

`const` member functions

- A feature of the accessor functions is the use of the `const` keyword.

```
int numerator() const { return num; }
int denominator() const { return den; }
```

- This has two effects for the function:
  - Firstly, any attempt to change the data members within the function definition will cause a compile error.
  - Secondly, it enables the functions to be accessed from *constant objects* . . .

Constant objects

- It is good practice to declare a "variable" as being a constant if it should not be changed.
  - This ensures that any accidental attempt to change its value in a program will be identified at compile time. Otherwise, it could lead to unexpected results.
  - We do this using the `const` keyword as follows:

```
const int NUM_STUDENTS=88;
const float PI = 3.14159;
```

  - The same holds true for *objects* that should not be changed:

```
const Rational PI(22,7);
```

  - The data members `num` and `den` are now constant.

Constant objects

- A problem occurs when we want to access member functions of such constant objects:

```
PI.print();   // error - not allowed!
```

- The reason for this is that member functions have the ability to <u>change</u> data members.

- To get around this, simply declare all *member functions* that do not change the data members as `const`:

```
void print() const;
```

`private` member functions

- As we have seen, we tend to declare data members as `private` and function members as `public`.
  - This arrangement implements data hiding and encapsulation.
- Sometimes, we may wish to declare a member function as `private`.
  - To declare such a member function, simply place its declaration inside the private section of the class definition.
  - These types of functions cannot be called from objects, they can only be called from within other member functions.
  - They are often used to perform the internal "housekeeping" tasks of the class implementation.

`private` member functions

- We declare two `private` member functions that are used by the constructor function to reduce fractions. These are:
  - `gcd()` - computes the greatest common divisor for `num` and `den`.
  - `reduce()` - uses `gcd()` to reduce the fraction to lowest terms.

```
class Rational {
public:
  Rational(int n=0, int d=1): num(n), den(d) { reduce(); }
  void print();
private:
  int num, den;
  int gcd(int j,int k) { if (k==0) return j;
                         return gcd(k,j%k); }
  void reduce() { int g=gcd(num,den); num/=g; den/=g; }
};
```

  - **NOTE: The function `gcd()` iteratively calls itself!**

`private` member functions

- **We have given definitions for these functions *inside* the class definition. Hence these functions will be inline.**
  - **We have done this since our intention was to have the constructor function inline for purposes of speed of execution, and the constructor function *calls* these functions.**

- **Each time we declare a function, the constructor will call these functions to ensure that the fraction is reduced:**

```
void main()
{
   Rational x(100,360); // calls constructor and
   x.print();           // hence reduce(), gcd()
}
```

output

5/18

The copy constructor

- **We have already met one function that is so important, it is provided by the compiler, namely the *default constructor*.**
  - **There are three more functions that are provided by default.**

- **The first of these is known as the *copy constructor*.**
  - **As its name suggests, it is also a constructor function, and as such will be called when an object is created.**
  - **A good question would be:**

    "why do we need two constructors ?"

  - **The answer to this lies in the different ways that we can declare objects . . .**

The copy constructor

- We can use our existing constructor function to declare an object `x` as follows:

```
Rational x(22,7);
```

- We can subsequently declare another object `y` as follows:

```
Rational y(x);
```

- This will create object `y` as a *copy* of object `x`.
  - This calls the *copy constructor*.
  - The copy constructor allows an object of a class to be initialised with another object.
  - In this case, the compiler provided default copy constructor is used.

The copy constructor

- **The copy constructor will have the following form:**

```
Rational(const Rational& r): num(r.num),den(r.den){}
```

  - **The copy constructor takes a single parameter, which is a constant reference to an object of class `Rational`.**
  - **The data members of the object parameter are copied to the data members of the calling object, i.e. the object being created.**

- **In addition to being called when an object is initialised with another object, the copy constructor is called when:**
  - **an object is *passed by value* to a function.**
  - **an object is *returned by value* from a function.**

The copy constructor

- To illustrate when the copy constructor is used, we will include one in the class `Rational` as follows:

```
class Rational {
public:
  Rational(int n=0, int d=1): num(n), den(d) { reduce(); }
  Rational(const Rational &r): num(r.num), den(r.den)
       { cout << "copy constructor called" << endl; }
  void print();
private:
  int num, den;
  int gcd(int j,int k) { if (k==0) return j;
                         return gcd(k,j%k); }
  void reduce() { int g=gcd(num,den); num/=g; den/=g; }
};
```

The copy constructor

❍ We now define a non-member function `f()` and use it as follows:

```
Rational f(Rational r)
{
   Rational s = r;  (3)
   return s;  (4)
}
void main()
{
   Rational x(22,7);
   Rational y(x) ;  (1)
   f(y).print();
}  (2)
```

The copy constructor is called 4 times:

output

```
copy constructor   (1)
copy constructor   (2)
copy constructor   (3)
copy constructor   (4)
22/7
```

The copy constructor

- The copy constructor is called for the following reasons:

1. Object `y` is *initialised as a copy* of object `x`.
2. Object `y` is *passed by value* to function `f()`.
3. Object `s` is *initialised as a copy* of parameter `r`. (NB: `s=r` is equivalent to `s(r)`)
4. Object `s` is *returned by value* from `f()`.

```
Rational f(Rational r)
{
   Rational s = r;   // 3
   return s;         // 4
}
void main()
{
   Rational x(22,7);
   Rational y(x) ;   // 1
   f(y).print();     // 2
}
```

The copy constructor

- **The compiler provided default copy constructor will perform the same function as the one that we included in the class declaration (apart from the output operation).**

- **Also, unlike the normal constructor function, we can only use a single parameter of type constant reference.**

> **Q: Why must we use a call-by-reference parameter instead of call-by-value for the copy constructor?**
>
> **A: If we used call-by-value parameter, the copy constructor would call itself infinitely!**

- **Hence, why would we ever need to provide our own copy constructor?**

The copy constructor

```
class String {
public:
   String(unsigned int);  // constructor
   void set(unsigned int, char);
   void print() { cout << buf << endl; }
private:
   unsigned int len;
   char *buf;
};
```

We create a class called `String` ...

```
String::String(unsigned int n): len(n)
{
  int i;
  buf = new char[len+1]; // allocate memory
  for (i=0;i<len;i++)
    buf[i] = ' ';  // fill string with spaces
  buf[len] = '\0'; // ends in NULL character
}
```

```
void String::set(unsigned int i, char s)
{
   if (i<len) buf[i] = s;
}
```

The copy constructor

. . . and use it as follows:

```
void main()
{
    int i;
    String x(4);        // calls constructor
    for (i=0;i<4;i++)
       x.set(i,'*'); // fill x with *
    String y(x);        // calls copy constructor
    x.print();          // print x
    y.print();          // print y
    for (i=0;i<4;i++)
       y.set(i,'&'); // fill y with &
    x.print();          // print x
    y.print();          // print y
}
```

output

```
****
****
&&&&
&&&&
```

The copy constructor

- What is going on here?
  - We define an object `x` to represent a string of 4 characters.
  - The constructor function sets the value of `len` to 4 and allocates sufficient space to store a string of 4 characters (plus the String terminator). The start address is stored in the data member `buf`.
  - We fill `x` with 4 '`*`' characters, and then create an object `y` as a copy of `x`. *This calls the default copy constructor*.
  - When we call the `print()` member function for objects `x` and `y`, we see that each represents `"****"`.
    - So far, everything is going as expected. However . . .
  - When we fill `y` with 4 ampersands, and print out both objects, `x` has been changed as well.

WHY?

The copy constructor

- **The answer is, that the compiler provided copy constructor simply copies the data members from the object in the parameter list to the calling object:**

```
String(const String& s): len(s.len),buf(s.buf){}
```

  - **Hence, when we create object `y`, we have not allocated any memory to store a string, we have merely copied the address of where `x`'s String data is stored in memory, to the data member `buf`.**
  - **Hence, when we change `y`, we are actually changing the string data of object `x`.**

The copy constructor

- We can correct this, by providing our own copy constructor as follows:

```
class String {
public:
   String(unsigned int);  // constructor
   String(const String&); // copy constructor
   void set(unsigned int, char);
   void print() { cout << buf << endl; }
private:
   unsigned int len;
   char *buf;
};
```

```
String::String(const String &s): len(s.len)
{
   int i;
   buf = new char[len+1];
   for (i=0;i<=len;i++)
      buf[i] = s.buf[i];
}
```

The copy constructor

The new output will be:

```
void main()
{
   int i;
   String x(4);      // calls constructor
   for (i=0;i<4;i++)
      x.set(i,'*'); // fill x with *
   String y(x);      // calls our copy constructor
   x.print();        // print x
   y.print();        // print y
   for (i=0;i<4;i++)
      y.set(i,'&'); // fill y with &
   x.print();        // print x
   y.print();        // print y
}
```

output

```
****
****
****
&&&&
```

The destructor function

- **Wait a minute . . . what happens to the memory that we *allocated* in the constructor functions for objects `x` and `y`?**

- **Remember, we must deallocate dynamically allocated memory to avoid a "memory leak".**

  - **The question is:**

  Where in the program should be deallocate this memory?

  - **The obvious answer is:**

  When we have finished with the object that requires it.

- **To do this, we meet the third of the member functions that are so important the compiler will provide them by default if we do not, namely the *destructor function*.**

The destructor function

- The destructor function is called automatically whenever an object *goes out of scope*.
  - An object goes out of scope when the program execution reaches the end of the block within which the object was declared.

- Destructor functions are analagous to constructor functions, with the difference that they are called at the end of an object's life instead of the beginning.

- To reflect this relationship, destructor functions share a similar naming convention to constructors:
  - They are given no return type and have the same name as the class prefixed by a tilde (~).

The destructor function

- We have added a destructor function to our `String` class:
  - We use it to deallocate the memory for `buf`.

```
class String {
public:
   String(unsigned int);  // constructor
   String(const String&); // copy constructor
   ~String();             // destructor
   void set(unsigned int, char);
   void print() { cout << buf << endl; }
private:
   unsigned int len;
   char *buf;
};
```

```
String::~String()
{
   delete buf[]; // deallocate memory
}
```

- Deallocating memory, allocated by the constructor function, is the most important use of destructor functions.

The destructor function

- To illustrate how destructor functions are called, consider the following example:

```
class Thingy {
public:
  Thingy(char n): name(n) { cout<<"hello from "<<name<<endl; }
  ~Thingy() { cout << "bye bye from " << name << endl; }
private:
  char name;
}
```

```
void main()
{   // start of main
    Thingy x('x');
    {   // start of block
        Thingy y('y');
    }   // end of block
}   // end of main
```

output

```
hello from x
hello from y
bye bye from y
bye bye from x
```

Static data members

- **In certain applications, <u>all</u> objects of a class will *share a common piece of information*. This could be addressed by:**
- **Declaring a data member to represent this piece of information. We would have to ensure that all objects have the same value assigned to their instance of this member.**
  - **This is not satisfactory as if the value of the shared information, changes, we would have to access <u>all</u> objects to change it.**

OR

- **Simply declare the information as being global. This would allow all objects of the class (and everything else in the program) to access the value.**
  - **This is not satisfactory as is violates the principal of data hiding.**

Static data members

- A better approach is to use a *static data member* to represent this piece of information:
  - A static data member is *shared by all objects* of a class.
  - The `static` keyword is used to inform the compiler of this fact.
  - Let us first consider a declaration of a `static` member in the `public` section:

A `public` static member

- In the following class, we have a static member `VAT_rate`.
  - This data member is static, as all items are taxed at the same rate.

```
class Item {
public:
   Item(float c): cost(c) {}
   float get_price() { return cost*(100+VAT_rate)/100.0; }
   static float VAT_rate;     // declared within class def'n
private:
   float cost;
};
float Item::VAT_rate = 17.5;  // defined outside class def'n
```

  - The data member is *declared* within the class definition, and must be *defined* outside the class definition.
  - This is necessary, as we only want to create a *single variable* to be shared by all objects of class `Item`.

A `public` static member

- We have initialised `VAT_rate` to 17.5 (the current rate).
  - Each object of class `Item` can access this value.
  - We can change this value by accessing `VAT_rate` as a member of *any* object of class `Item` (remember `VAT_rate` is public).

```
void main()
{
   Item shirt(20.00), trousers(30.00);
   cout << setiosflags(ios::showpoint) << setprecision(2);
   cout << "Shirt = £" << shirt.get_price() << endl;
   cout << "Trousers = £" << trousers.get_price() << endl;
   shirt.VAT_rate = 15.0;  // government reduces VAT rate!
   cout << "Shirt = £" << shirt.get_price() << endl;
   cout << "Trousers = £" << trousers.get_price() << endl;
}
```

A `private` static member

- In line with the principal of encapsulation, we would prefer to declare `VAT_rate` as `private`:

```
class Item {
public:
   Item(float c): cost(c) {}
   float get_price() { return cost*(100+VAT_rate)/100.0; }
   void set_VAT(float vr) { VAT_rate=vr; }
private:
   static float VAT_rate;      // VAT_rate is private
   float cost;
};
float Item::VAT_rate = 17.5;  // defined outside class def'n
```

  - As `VAT_rate` is now `private`, we must provide a member function (`set_VAT()`) to change its value.

A `private` static member

- We have initialised `VAT_rate` to 17.5 (the current rate).
  - As `VAT_rate` is now `private`, we can no longer access it directly. We must use the member function `set_VAT()` to change its value.

```
void main()
{
   Item shirt(20.00), trousers(30.00);
   cout << setiosflags(ios::showpoint) << setprecision(2);
   cout << "Shirt = £" << shirt.get_price() << endl;
   cout << "Trousers = £" << trousers.get_price() << endl;
   shirt.set_VAT(15.0);  // government reduces VAT rate!
   cout << "Shirt = £" << shirt.get_price() << endl;
   cout << "Trousers = £" << trousers.get_price() << endl;
}
```

`static` function members

- One problem remains :
  - It is awkward to use one object to access the function `set_VAT()` when this function is a member of *all* objects of class `Item`.
  - We can improve this by declaring `set_VAT()` to be a *static function member*:

```
class item {
public:
   Item(float c): cost(c) {}
   float get_price() { return cost*(100+VAT_rate)/100.0; }
   static void set_VAT(float vr) { VAT_rate=vr; }
private:
   static float VAT_rate;      // VAT_rate is private
   float cost;
};
float Item::VAT_rate = 17.5;  // defined outside class def'n
```

`static` function members

- As `set_VAT()` is a `static` function, we can access it by using the notation:

```
Item::set_VAT(15.0);
```

  - In other words, we do not need to use an object of a class to access a `static` member function of that class.

```
void main()
{
   Item shirt(20.00), trousers(30.00);
   cout << setiosflags(ios::showpoint) << setprecision(2);
   cout << "Shirt = £" << shirt.get_price() << endl;
   cout << "Trousers = £" << trousers.get_price() << endl;
   Item::set_VAT(15.0);  // government reduces VAT rate!
   cout << "Shirt = £" << shirt.get_price() << endl;
   cout << "Trousers = £" << trousers.get_price() << endl;
}
```

An example class

- Consider a class called `Polygon`, that can be used to represent different shapes.

```
struct point { float  x, y; }; // struct used by
class Polygon {                // class Polygon
public:
   Polygon(char*);             // constructor
   Polygon(const Polygon&);    // copy constructor
   ~Polygon();                 // destructor
   void translate(float,float);
   void scale(float,float);
   void rotate(float);
   void draw();
private:
   point *vertices, centre;
   int   num_vertices;
};
```

polygon.h

An example class

```
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include <graphics.h> // includes Borland BGI graphics library
#include "polygon.h"
Polygon::Polygon(char *name) {
   int i;
   ifstream ins(name);
   if (ins.fail()) {
     cerr << "Error opening " << name << endl; num_vertices=0;
   }
   else {
      ins >> num_vertices;
      vertices = new point[num_vertices];
       for (i=0;i<num_vertices;i++)
          ins >> vertices[i].x >> vertices[i].y;
      ins >> centre.x >> centre.y;
      ins.close();
   }
}
```

polygon.cc

cont...

An example class

```
Polygon::Polygon(const Polygon& p):
                     num_vertices(p.num_vertices) {
   int i;
   vertices = new point[num_vertices];
   for (i=0;i<num_vertices;i++)
      vertices[i] = p.vertices[i];
   centre = p.centre;
}

Polygon::~Polygon() {
   delete [] vertices;
}
```

polygon.cc

cont...

An example class

```
void Polygon::translate(float dx, float dy) {
   int i;
   for (i=0;i<num_vertices;i++) {
      vertices[i].x += dx; vertices[i].y += dy;
   }
   centre.x += dx; centre.y += dy;
}
void Polygon::scale(float sx, float sy) {
   point  temp=centre;
   int   i;
   translate(-temp.x,-temp.y);
   for (i=0;i<num_vertices;i++) {
      vertices[i].x *= sx;  vertices[i].y *= sy;
   }
   translate(temp.x,temp.y);
}
```

polygon.cc

cont...

An example class

```
void Polygon::rotate(float angle) {
   float  theta =  angle*M_PI/180.0;
   point  temp=centre, p;
   int    i;
   translate(-temp.x,-temp.y);
   for (i=0;i<num_vertices;i++) {
      p.x = vertices[i].x*cos(theta)+vertices[i].y*sin(theta);
      p.y = -vertices[i].x*sin(theta)+vertices[i].y*cos(theta);
      vertices[i] = p;
   }
   translate(temp.x,temp.y);
}
void Polygon::draw() {
   int i;
   moveto(vertices[num_vertices-1].x,
                         vertices[num_vertices-1].y);
   for (i=0;i<num_vertices;i++)
      lineto(vertices[i].x,vertices[i].y);
}
```

polygon.cc

Using the example class

- **In this example, we are using the BGI graphics library supplied with the Borland C++ development environment.**
  - **We have used the functions `moveto()` and `lineto()` that are defined in the BGI graphics library header `graphics.h`.**
  - **An alternative graphics library (e.g. X/Motif, OPEN-GL) could be implemented by changing the code in the member function `draw()`.**
- **To illustrate the use of our class `Polygon`, consider the following program `clock.cc` that displays a clock face depicting a specified time.**

Using the example class

```
#include <iostream.h>
#include <stdlib.h>
#include <graphics.h>
#include "polygon.h"
void setgraph();    // function to set up graphics mode
void main()
{
   int      MAX_X, MAX_Y;  // used to represent size of display
   int      hour, minute;
   Polygon big("arrow.dat");       // calls constructor
   Polygon little=big;             // calls copy constructor

   little.scale(1.5,0.7);    // make little shorter and fatter

   cout << "Enter time (h m): ";
   cin >> hour >> minute;

   setgraph();  // set up graphics
   MAX_X = getmaxx();  MAX_Y = getmaxy();
```

clock.cc

cont...

Using the example class

```
// set size of hands relative to screen height
   big.scale(MAX_Y/2.0,MAX_Y/2.0);
   little.scale(MAX_Y/2.0,MAX_Y/2.0);

// move to centre of screen
   big.translate(MAX_X/2.0,MAX_Y/2.0);
   little.translate(MAX_X/2.0,MAX_Y/2.0);

// now set hands to point to time
   big.rotate(-6*minute);
   little.rotate(-(30*hour+minute/2));

// draw circle and hands
   circle(MAX_X/2,MAX_Y/2,MAX_Y/2);
   big.draw();
   little.draw();

   getc(); closegraph();  // waits for key and closes graphics
}
```

clock.cc

Using the example class

❍ The function `setgraph()` is used to set up the graphics mode:

```
void setgraph() // sets up BGI graphics interface
{
   int gdriver = DETECT, gmode, errorcode;
   initgraph(&gdriver,&gmode,"c:\\bc5\\bgi");
   errorcode = graphresult();
   if (errorcode != grOk) {
      cerr << "Error: " << grapherrormsg(errorcode);
      cout << "Press any key to halt: ";
      getc();
      exit(1);  // exit program in error
   }
}
```

clock.cc

Using the example class

- The display for a time of 3:23 is as follows:

```
7
 0.05  0.05
-0.05  0.05
-0.05 -0.8
-0.1  -0.8
 0.0  -0.95
 0.1  -0.8
 0.05 -0.8
 0.0   0.0
```

arrow.dat

D:\docs\228C1B~1.5HV\bcppexe\clock.exe - FROZEN

Summary

- In C++ object-oriented programming is implemented using classes.
- Classes are user defined data types similar to structures. Unlike structures:
  - Classes can have function members as well as data members.
  - Classes have different levels of membership.
- Member functions should be placed in the `public` section, whilst data members should be placed in the `private` section of a class definition.
  - This arrangement ensures the idea of data hiding which lies behind the principal of encapsulation.

Summary

- A special member function called a constructor function is called each time an object is created.
  - Constructor functions are used to initialised data members and allocate dynamic memory.
- The copy constructor is used when an object is initialised with another object - hence they are called for passing function arguments by value and returning by value.
- The destructor function is called when an object goes out of scope. Its main use is for deallocating memory allocated by the constructor.

Summary

- If you do not provide a constructor, copy constructor or destructor function for your class, the compiler will provide one by default.
- Static data members are shared by all objects of a class.
- Static data members must be defined outside a class declaration.
- Static function members may be used to manipulate static data members, by prefixing the function with the class name and the scope resolution operator.