22.5HV2

SOFTWARE ENGINEERING II

Revision

The big 4 member functions

- There are 4 member functions that are so important, that the compiler will provide them if the programmer does not:
  - **The default constructor.**
    - Called when an object is created.
  - **The copy constructor.**
    - Called when an object is initialised with another object, passed by value or returned by value from a function.
  - **The destructor.**
    - Called when an object goes out of scope.
  - **The assignment operator.**
    - Called when an object is assigned the value of another object.

A simple string class

- Suppose that we create a class called `String` ...

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

mystring.h

```
#include "mystring.h"
String::String(unsigned int n): len(n) {
  buf = new char[len+1]; // allocate memory
  for (int 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;
}
```

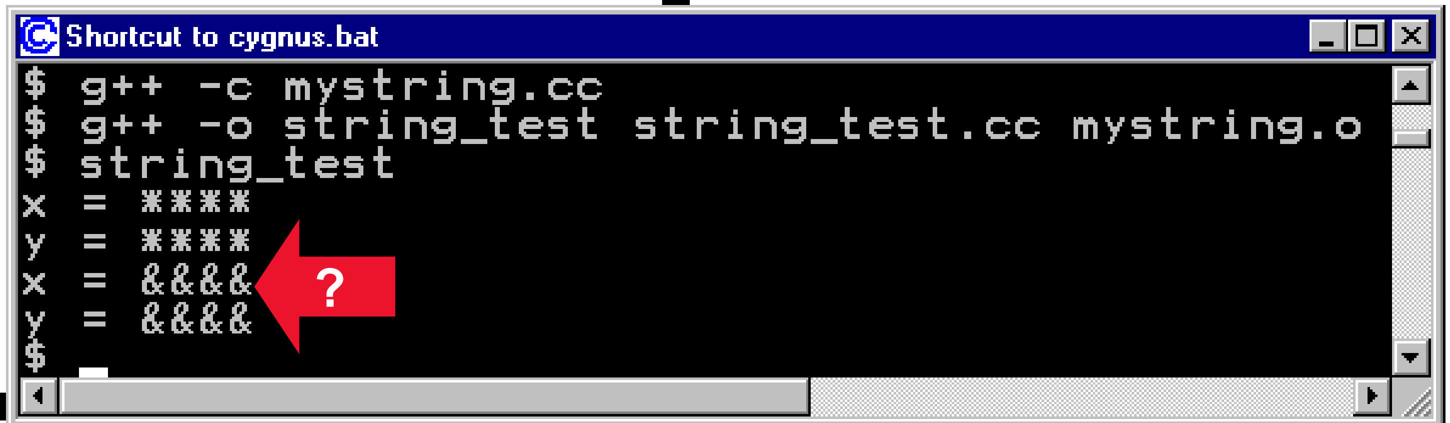
mystring.cc

A simple string class

- And use it as follows:

```
#include "mystring.h"
void main()
{
   int i;
   String x(4);       // calls constructor
   for (i=0;i<4;i++)
      x.set(i,'*'); // fill x with *
   String y(x);       // initialise y with x
   cout << "x = ";
   x.print();         // print x
   cout << "y = ";
   y.print();         // print y
   for (i=0;i<4;i++)
      y.set(i,'&');
   cout << "x = ";
   x.print();
   cout << "y = ";
   y.print();
}
```

string_test.cc



The copy constructor

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

```
#include <iostream.h>
class String {
public:
    String(unsigned int);   // constructor
    String(const String&); // copy constructor
    void set(unsigned int, char);
    void print() { cout << bu
private:
    unsigned int len;
    char *buf;
};
```

mystring.h

```
#include "mystring.h"
String::String(unsigned int n): len(n) {
  buf = new char[len+1]; // allocate memory
  for (int i=0;i<len;i++) buf[i] = ' ';
  buf[len] = '\0'; // ends in NULL character
}
void String::set(unsigned int i, char s)
{ if (i<len) buf[i] = s; }
String::String(const String &s): len(s.len) {
   int i;
   buf = new char[len+1]; // allocate memory
   for (i=0;i<=len;i++)
      buf[i] = s.buf[i];  // copy contents
}
```

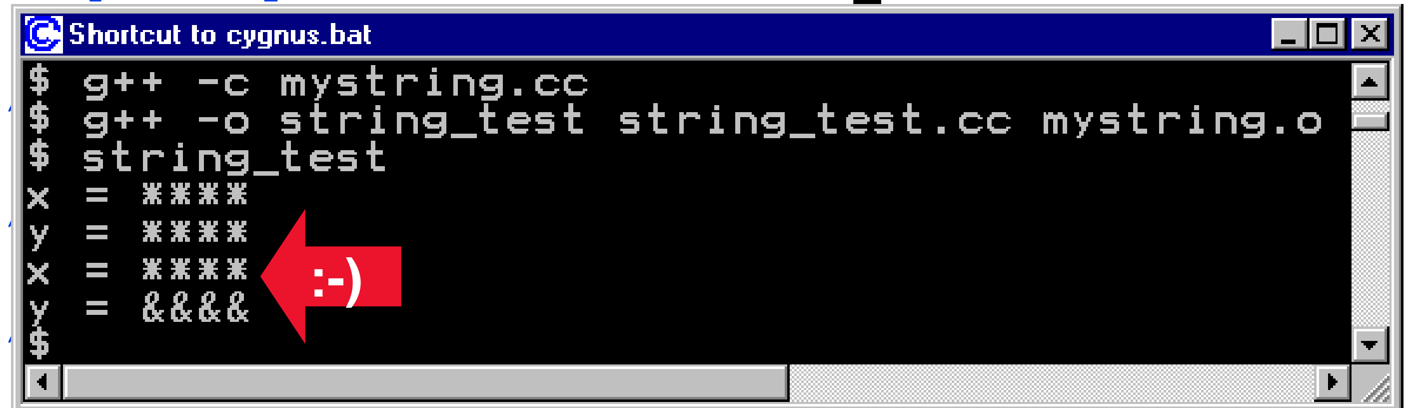
mystring.cc

The copy constructor

- Recompiling and running our test program, we have:

```
#include "mystring.h"
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
    cout << "x = ";
    x.print();          // print x
    cout << "y = ";
    y.print();          // print y
    for (i=0;i<4;i++)
        y.set(i,'&');
    cout << "x = ";
    x.print();
    cout << "y = ";
    y.print();
}
```

string_test.cc

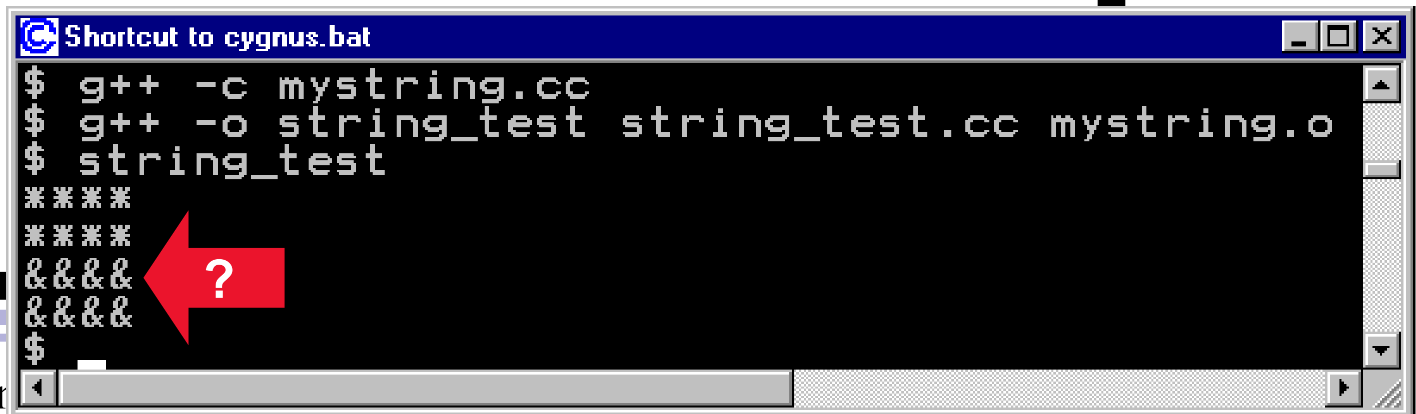


Why not use the default assignment operator?

- Let us repeat the previous example, with object **y** now being <u>assigned</u> the value of **x**:

```
#include "mystring.h"
void main()
{
   int i;
   String x(4), y(4); // calls constructor
   for (i=0;i<4;i++)
      x.set(i,'*');   // fill x with '*'
   y = x;             // calls default assignment operator
   x.print();         // print x
   y.print();         // print y
   for (i=0;i<4;i++)
      y.set(i,'&');
   x.print();
   y.print();
}
```

string_test.cc



The assignment operator

- We can provide our own version of the assignment operator for the `String` class as follows:

```
#include <iostream.h>
class String {
public:
    String(unsigned int);                  // constructor
    String(const String&);                 // copy constructor
    String& operator=(const String&);  // assignment operator
    void set(unsigned int, char);
    void print() { cout << buf << endl; }
private:
    unsigned int len;
    char *buf;
};
```

mystring.h

The `this` pointer

- We can define this operator as follows:

```
String& String::operator=(const String &s)
{
    int i;
    len = s.len;
    for (i=0;i<=len;i++)
        buf[i] = s.buf[i];  // copy data
    return *this;
}
```

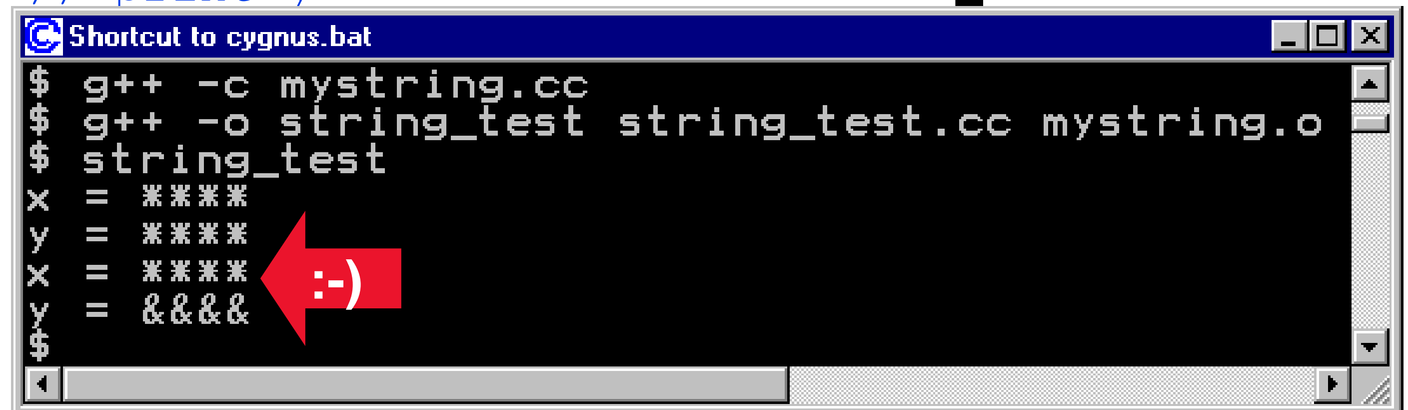
- The keyword `this` is a <u>pointer</u> to the object that we are "inside".
  - Hence when we return `*this`, we are returning "this object".
  - This allows chained assignments: `x = y = z;`

The assignment operator

❍ Now when we recompile and run the previous program:

```
#include "mystring.h"
void main()
{
   int i;
   String x(4), y(4); // calls constructor
   for (i=0;i<4;i++)
      x.set(i,'*');   // fill x with '*'
   y = x;             // calls our assignment operator
   cout << "x = ";
   x.print();         // print x
   cout << "y = ";
   y.print();         // print y
   for (i=0;i<4;i++)
      y.set(i,'&');
   cout << "x = ";
   x.print();
   cout << "x = ";
   y.print();
}
```

string_test.cc



The destructor function

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

```
#include <iostream.h>
class String {
public:
   String(unsigned int);            // constructor
   String(const String&);           // copy constructor
   String& operator=(const String&); // assignment operator
   ~String();                       // destructor
   void set(unsigned int, char);
   void print() { cout << buf << endl; }
private:
   unsigned int len;
   char *buf;
};
```

mystring.h

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

mystring.cc

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

Overloading the * operator

❍ We declare and define the operator as follows:

```
class Rational {
public:
   Rational(int n=0, int d=1): num(n), den(d) { reduce(); }
   Rational operator*(const Rational&);
   // other declarations
private:
   int num, den;
   // other declarations
};
```

```
Rational Rational::operator*(const Rational& r)
{
    Rational z(num*r.num,den*r.den);
    return z;
}
```

Friend functions

❍ We can now write our overloaded `*` operator as a `friend` of class Rational as follows:

```
class Rational {
  friend Rational operator*(const Rational&, const Rational&);
public:
  Rational(int n=0, int d=1): num(n), den(d) { reduce(); }
  // other declarations
private:
  int num, den;
  // other declarations
};
```

```
Rational operator*(const Rational& x, const Rational& y)
{
   Rational z(x.num*y.num,x.den*y.den);
   return z;
}
```

Overloading the stream insertion operator

❍ Suppose we have a class called `Date`:

```
#include <iostream.h>
class Date {
  friend ostream& operator<<(ostream&, const Date&);
public:
  Date(int d=1, int m=1, int y=2000):
                      day(d), month(m), year(y) {}
private:
  int day, month, year;
};
```

date.h

Overloading the stream insertion operator

```
#include "date.h"
ostream& operator<<(ostream &ostr, const Date &d) {
   ostr << d.day;
   switch (d.day) {
      case 1:
      case 21:
      case 31:
               ostr << "st "; break;
      case 2:
      case 22: ostr << "nd "; break;
      case 3:
      case 23: ostr << "rd "; break;
      default: ostr << "th "; break;
   }
   switch (d.month) {
      case 1:  ostr << "January";  break;
      case 2:  ostr << "February"; break;
      // etc
   }
   return ostr << " " << d.year;
}
```

date.cc

Overloading the stream insertion operator

```
#include "date.h"

void main()
{
   Date today(17,3,2000);

   cout << "Today's date is " << today << endl;
}
```

date_test.cc

Overloading other operators

- We now overload the stream deletion operator (>>) and the less than operator (<).

```
#include <iostream.h>
class Date {
  friend ostream& operator<<(ostream&, const Date&);
  friend istream& operator>>(istream&, Date&);
public:
  Date(int d=1, int m=1, int y=2000):
                      day(d), month(m), year(y) {}
  bool operator<(const Date&);
private:
  int day, month, year;
};
```

date.h

Overloading other operators

```
#include "date.h"
ostream& operator<<(ostream &ostr, const Date &d) {
// etc
istream& operator>>(istream &istr, Date &d) {
   cout << "Enter day: ";
   istr >> d.day;
   cout << "Enter month: ";
   istr >> d.month;
   cout << "Enter year: ";
   istr >> d.year;
   return istr;
}
bool Date::operator<(const Date &d) {
   if ( 12*year+month < 12*d.year+d.month ) return true;
   if ( 12*year+month > 12*d.year+d.month ) return false;
   if ( day < d.day ) return true;
   return false;
}
```

date.cc

Overloading other operators

```
#include "date.h"
void main()
{
   Date start, end;
   cout << "Enter start date: ";
   cin >> start;
   do {
      cout << "Enter end date: ";
      cin >> end;
      if ( end < start )
         cerr << "End date must be after start date ("
              << start << ")" << endl;
   } while ( end < start );
   cout << "Start date: " << start << endl;
   cout << "End date:   " << end << endl;
}
```

date_test.cc

Overloading other operators

```
Cygwin B20
$ g++ -c date.cc
$ g++ -o date_test date_test.cc date.o
$ date_test
Enter start date:
        Enter day: 17
        Enter month: 3
        Enter year: 2000
Enter end date:
        Enter day: 21
        Enter month: 7
        Enter year: 1999
End date must be after start date (17th March 2000)
Enter end date:
        Enter day: 21
        Enter month: 7
        Enter year: 2000
Start date: 17th March 2000
End date:   21st July 2000
$
```

Class relationships

- There are 4 ways in which classes can be related:
  - **Association:**

    Where two or more objects are linked and "pass messages" by calling each others member functions.
  - **Composition:**

    Where the object of a class contains an object of another class as a member and the lifetime of the objects is tied.
  - **Aggregation:**

    Where the object of one class contains a reference to an object of another class and their lifetimes are not tied.
  - **Inheritance:**

    Where one class is created or derived from another class.

Association

```
enum switcher{off,on};
class Light {
public:
   Light(): state(off) {}
   void change_state();
   void inspect_state();
private:
   switcher state;
};
```

light.h

```
#include <iostream.h>
#include "light.h"
void Light::change_state()
{
   if ( state==off ) state = on;
   else state = off;
}
void Light::inspect_state()
{
   if ( state == off )
      cout << "Light is off" <<endl;
   else
      cout << "Light is on" << endl;
}
```

light.cc

Association

```
class Light; // forward declaration
class Button {
public:
   Button(Light *bulb): light_bulb(bulb) {}
   void flick();
private:
   Light *light_bulb;
};
```

button.h

```
#include "light.h"
#include "button.h"

void Button::flick()
{
   light_bulb->change_state();
}
```

button.cc

Association

```
#include <iostream.h>
#include "light.h"
#include "button.h"

void main()
{
    Light spot;
    Button light_switch(&spot);

    spot.inspect_state();
    light_switch.flick();
    spot.inspect_state();
    light_switch.flick();
    spot.inspect_state();
}
```

light_test.cc