C++ Overview

C++ Techniques

Relevant techniques include:

1. C++ classes, with *private* and *public* members
2. Function and operator name overloading to give "natural" function calls
3. Templates to allow the same code to be used on a variety of different data types
4. A clean built-in I/O interface, which itself involves overloading the input and output operators

Learning these techniques is much of what C++ is all about.

Outline

1. Classes : constructors, destructors, clean interface
2. Default arguments
3. Function and operator overloading
4. Use of `const`
5. Rule of three: copy constructor, assignment, destructor
6. Templates: function and class templates
7. C++ Error Handling
8. STL: `vector` class

Constructors

- A <u>constructor</u> is a method that executes when an object of a class is declared and sets the initial state of the new object.
- A constructor
  - has the same name with the class,
  - <u>no return type</u>
  - has zero or more parameters (the constructor without an argument is the *default constructor*)
- There may be more than one constructor defined for a class.
- If no constructor is explicitly defined, one that initializes the data members using language defaults is automatically generated.

Class syntax - Example

```
// A class for simulating an integer memory cell

class  IntCell
{
  public:
        IntCell( )
        { storedValue = 0; }

        IntCell(int initialValue )
        { storedValue = initialValue;}

        int read( )
        { return storedValue; }

        void write( int x )
        { storedValue = x;}

  private:
        int storedValue;
};
```

constructors

Extra Constructor Syntax

```
// A class for simulating an integer memory cell

class  IntCell
{
   public:
       IntCell( int initialValue = 0 )
          : storedValue( initialValue) { }

       int read( )
          { return storedValue; }

       void write( int x )
          { storedValue = x; }
   private:
       int storedValue;
};
```

Single constructor (instead of two)

Destructor

- Performs termination housekeeping before the system reclaims the object's memory
- Complement of the constructor
- An **automatic default destructor** is added to your class if no other destructor is defined.
- The only action of the automatic default destructor is to call the default destructor of all member objects.

Automatic Default Destructor

A destructor should never be called directly. Instead, it is automatically called when the object's memory is being reclaimed by the system:

- If the object is on the **stack**, when the function returns
- If the object is on the **heap**, when `delete` is used

Custom Destructor

To add custom behavior to the end-of-life of the function, a custom destructor can be defined:

- A custom destructor is a member function.
- Its name is tilde (~) followed by the class name
  - e.g. `~IntCell( );`
    `~Cube( );`
- Receives no parameters, returns no value
- One destructor per class.

Custom destructor

A custom destructor is essential when an object allocates an external resource that must be closed or freed when the object is destroyed. Examples:

- Heap memory
- Open files

Custom Destructor Example

```
class IntCell{
   public:
     IntCell(int initialValue=0){
        storedValue = new int (initialValue);
     }
     ~IntCell(){
        delete storedValue;
     }
     int read( ){
        return *storedValue;
     }
     void write( int x ){
        *storedValue = x;
     }
     private:
       int *storedValue;
}
```

Separation of Interface and Implementation

- Large-scale projects put the interface and implementation of classes in different files.
  - For small amount of coding it may not matter.

- *Header File*: contains the interface of a class. Usually ends with `.h` (an include file)

- *Source-code file*: contains the implementation of a class. Usually ends with `.cpp`
- .cpp file includes the .h file with the preprocessor command `#include`.
  - Example: `#include "myclass.h"`

C++ header file (.h)

It defines the interface to the class, which includes the declaration of **all** member variables and functions.

```
#ifndef  _IntCell_H_
#define _IntCell_H_

class  IntCell
{
   public:
      IntCell( int initialValue = 0 );
      int read( ) const;
      void write( int x );
   private:
      int storedValue;
};
#endif
```

`IntCell` class Interface in the file *IntCell.h*

C++ Implementation File (.cpp)

It contains the code to implement the class (or other C++ code)

```
#include <iostream>
#include "IntCell.h"
using std::cout;

//Construct the IntCell with initialValue
IntCell::IntCell( int initialValue)
   : storedValue( initialValue) {}

//Return the stored value.
int IntCell::read( ) const
{
    return storedValue;
}
//Store x.
void IntCell::write( int x )
{
    storedValue = x;
}
```

Scope operator ::
ClassName :: member

`IntCell` class implementation in file *IntCell.cpp*

A client program

```
#include <iostream>
#include "IntCell.h"
using namespace std;

int main()
{
      IntCell m;  // or IntCell m(0);

      m.write (5);
      cout << "Cell content : " << m.read() << endl;

      return 0;
}
```

A program that uses `IntCell` in file *TestIntCell.cpp*

Another Example: `Complex` Class

```
#ifndef _Complex_H
#define _Complex_H

using namespace std;
class Complex
{
   float re, im; // by default private
   public:
     Complex(float x = 0, float y = 0)
       : re(x), im(y) { }

     Complex operator*(Complex rhs);
     float modulus();
     void print();
};

#endif
```

`Complex` class Interface in the file *Complex.h*

Implementation of `Complex` Class

```
#include <iostream>
#include <cmath>
#include "Complex.h"

Complex Complex::operator*(Complex rhs)
{
   Complex prod;
   prod.re = (re*rhs.re - im*rhs.im);
   prod.im = (re*rhs.im + im*rhs.re);
   return prod;
}
float Complex::modulus()
{
    return sqrt(re*re + im*im);
}
void Complex::print()
{
    std::cout << "(" << re <<"," << im << ")" << std::endl;
}
```

`Complex` class implementation in file *Complex.cpp*

Using the class in a client program

```
#include <iostream>
#include "Complex.h"
int main()
{
   Complex c1, c2(1), c3(1,2);
   float x;
   // overloaded  * operator!!
   c1 = c2 * c3 * c2;

   // mistake! The compiler will stop here, since the
   // re and im parts are private.
   x = sqrt(c1.re*c1.re + c1.im*c1.im);

   // OK. Now we use an authorized public function
   x = c1.modulus();

   c1.print();
    return 0;
}
```

A program that uses Complex in file *TestComplex.cpp*

Outline

1. Classes : constructors, destructors, clean interface
2. **Default arguments**
3. **Function and operator overloading**
4. **Use of `const`**
5. **Rule of three: copy constructor, assignment, destructor**
6. **Templates: function and class templates**
7. **C++ Error Handling**
8. **STL: `vector` class**

Default Arguments

- In C++, functions can have default arguments
- This is specified in the function declaration;

```
int foo(int x = 1, int y = 2, int z = 3);

foo(); // all parameters use the default value
foo(5); // y and z use the default value
foo(5,8); // z uses the default value
foo(5,8,9); // default values are not used
```

Default Arguments

- Note that it is impossible to suppy a user-defined value for z without also supplying a value for x and y. That is the following does not work:

```
foo(,,9); // compile error
```

- For this reason the default parameters must be the rightmost ones:

```
int foo(int x = 1, int y = 2, int z); // WRONG
int foo(int z, int x = 1, int y = 2); // CORRECT
```

Function Overloading

- Functions with same name and different parameters
- Overloaded functions should perform similar tasks (otherwise it would be confusing):
- Function to square `int`s and function to square `float`s

```
int square( int x) {return x * x;}
float square(float x) { return x * x;}
```

- Compiler chooses based on the actual parameter types:

```
square(4); // calls the integer version
square(4.0f); // calls the float version
```

Function Overloading

- Functions that only differ by return type cannot be overloaded:

```
int square(int x);
float square(int x); // Compile error
```

Overloaded Operators

- An operator with more than one meaning is said to be ***overloaded***.

  2 + 3   3.1 + 3.2   ➔ + is an overloaded operator

- To enable a particular operator to operate correctly on instances of a class, we may define a new meaning for the operator.

  ➔ we may overload it

Operator Overloading

- Operator overloading allows us to use existing operators for user-defined classes.
- The following operators can be overloaded:

+	-	*	/	%	^	&	|
~	!	,	=		=		
++	--	<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	& =	|=	*=
<<=	>>=	[ ]	( )	->	->*	new	delete

- Note that the precedence, associativity, and arity of the operators cannot be changed!

Operator Overloading

- Format
  - Write function definition as normal
  - Function name is keyword **`operator`** followed by the symbol for the operator being overloaded.
  - `operator+` would be used to overload the addition operator (+)
- No new operators can be created
  - Use only existing operators
- Built-in types
  - Cannot overload operators
  - You cannot change how two integers are added

Overloaded Operators -- Example

What if we want to multiply a complex number with a scalar? Define another function with the same name but different parameters.

```
class Complex
{
   ...

   Complex operator*(Complex rhs) const;
   Complex operator*(float k) const;

   ...
};
```

Implementation of `Complex` Class

```
Complex Complex::operator*(Complex rhs) const
{
   Complex prod;
   prod.re = (re*rhs.re - im*rhs.im);
   prod.im = (re*rhs.im + im*rhs.re);
   return prod;
}

Complex Complex::operator*(float k) const
{
   Complex prod;
   prod.re = re * k;
   prod.im = im * k;
   return prod;
}
```

`Complex` class implementation in file *Complex.cpp*

Using the class in a Driver File

```
#include <iostream>
#include "Complex.h"

int main()
{
   Complex c1, c2(1), c3(1,2);

   c1 = c2 * c3 * c2;
   c1.print();

   c1 = c1 * 5; // translated to c1.operator*(5)
   c1.print();

   // How about this?
   c1 = 5 * c1; // CANNOT translate to 5.operator*(c1)

    return 0;
}
```

A program that uses Complex in file *TestComplex.cpp*

Outline

1. Classes : constructors, destructors, clean interface
2. **Default arguments**
3. **Function and operator overloading**
4. **Use of `const`**
5. **Rule of three: copy constructor, assignment, destructor**
6. **Templates: function and class templates**
7. **C++ Error Handling**
8. **STL: `vector` class**

`const` keyword in C++

- Constant is something that doesn't change.
- In C language and C++ we use the keyword `const` to make program elements constant.
- `const` keyword can be used in many contexts in a C++ program. It can be used with:
  - Variables
  - Pointers
  - Function arguments and return types
  - Class Data members
  - Class Member functions
  - Objects

Example uses of keyword `const`

We may encounter `const` in the following cases:

1. Const reference parameter:

   ```
   Complex operator*(const Complex& rhs);
   ```

   In this case it means the parameter cannot be modified in the function.

2. Const member function:

   ```
   Complex operator*(Complex& rhs) const;
   ```

   In this case it means the function cannot modify class members.

3. Const object/variable:

   ```
   const Complex c1(3, 4);
   ```

   In this case it means the object cannot be modified.

Pointers with `const` keyword in C++

- Either we can make the pointer itself a constant or we can apply `const` to what the pointer is pointing to.
- E.g. constant pointer:

```
int * const p = &i; // must be initialized
*p = 6; // it is O.K.
p = &j;     // NOT O.K.
```

Pointer to a `const` variable

- E.g. making what the pointer is pointing to, constant :

```
int i;
const int * p = &i;
*p = 6;  // it is NOT O.K., because i is
      //treated as constant when accessed by p.
```

- However, it can be changed independently:

```
i = 6;  // It is O.K.
```

- It is also possible to declare a `const` pointer to a constant value:

```
const int n = 5;
const int * const p = &n;
```

`const` Reference

- A `const` reference will not let you change the value it references:
- Example:

```
int n = 5;
const int & rn = n;

rn = 6;   // error!!
```

- `const` reference is like a `const` pointer to a `const` object.

Parameter Passing

- **Call by value**
  - Copy of data passed to function
  - Changes to copy do not change original
- **Call by reference**
  - Uses &
  - Avoids a copy and allows changes to the original
- **Call by constant reference**
  - Uses `const&`
  - Avoids a copy and guarantees that actual parameter will not be changed

Example

```
int squareByValue( int ); // pass by value
void squareByReference( int & ); // pass by reference
int squareByConstReference ( const int & ); // const ref.
```

Example (cont.)

```
int squareByValue( int a ){
   return a *= a;   // caller's argument not modified
}

void squareByReference( int &a ){
   a *= a;     // caller's argument modified
}

int squareByConstReference (const int& a ){
  // a *= a;  not allowed (compiler error)
  return a * a;
}
```

Example

```
int squareByValue( int ); // pass by value
void squareByReference( int & ); // pass by reference
int squareByConstReference ( const int & ); // const ref.

int main()
{  int x = 2, z = 4, r1, r2;

   r1 = squareByValue(x);
   squareByReference( z );
   r2 = squareByConstReference(x);

   cout << "x = " << x << " z = " << z << endl;
   cout << "r1 = " << r1 << " r2 = " << r2 << endl;
   return 0;
}
```

Improving the Complex Class

```
#ifndef _Complex_H
#define _Complex_H

using namespace std;
class Complex
{
  float re, im; // by default private
  public:
   Complex(float x = 0, float y = 0)
       : re(x), im(y) { }

   Complex operator*(const Complex& rhs) const;
   float modulus() const;
   void print() const;
};

#endif
```

`Complex` class Interface in the file *Complex.h*

Improving the Complex Class

```
#include <iostream>
#include <cmath>
#include "Complex.h"

Complex Complex::operator*(const Complex& rhs) const
{
   Complex prod;
   prod.re = (re*rhs.re - im*rhs.im);
   prod.im = (re*rhs.im + im*rhs.re);
   return prod;
}
float Complex::modulus() const
{
    return sqrt(re*re + im*im);
}
void Complex::print() const
{
   std::cout << "(" << re <<"," << im << ")" <<
  std::endl;
}
```


`Complex` class implementation in file *Complex.cpp*

Outline

1. Classes : constructors, destructors, clean interface
2. Default arguments
3. Function and operator overloading
4. Use of const
5. **Rule of three: copy constructor, assignment, destructor**
6. **Templates: function and class templates**
7. **C++ Error Handling**
8. **STL: `vector` class**

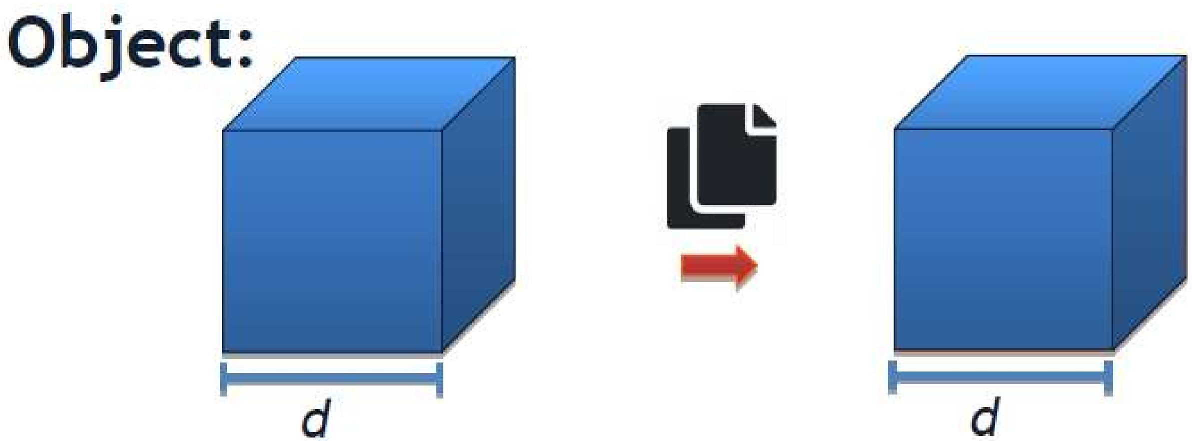
Rule of Three

- Whenever you need to define
  - a copy constructor,
  - assignment operator, or
  - the destructor,

  you must define all three of them
- This is known as the rule of three
- In general, for every class that contains pointer members you must define all three functions

Copy Constructor

- In C++, a **copy constructor** is a special constructor that exists to make a copy of an existing object.



Automatic Copy constructor

- If we do not provide a custom copy constructor, the C++ compiler provides an **automatic default copy constructor** for our class for free!
- The automatic copy constructor will copy the contents of all member variables.
  - Note that compiler provided copy constructor performs *member-wise copying* of the elements of the class (i.e. **Shallow copy**).

Custom Copy Constructor

A custom copy constructor is:

- A class constructor
- Has exactly one argument
  - The argument must be const reference of the same type as the class.

Example:

```
IntCell(const IntCell & obj)
```

- Note that the parameter must be a `const` reference.

Copy Constructor Invocation

Often, copy constructors are invoked automatically:

- Passing an object as a parameter (by value)
- Returning an object from a function (by value)
- Initializing a new object

Example

```
//The following is a copy constructor
//for Complex class. Since it is same
//as the compiler's default copy
//constructor for this class, it is
//actually redundant.

Complex::Complex(const Complex & C )
{
    re = C.re;
    im = C.im;
}
```

Another Example

```
class MyString
{
 public:
     MyString(const char* s = ””);
     MyString(const MyString& s);
     ...
 private:
     char* str;
     int length;
};
```

Example (cont.)

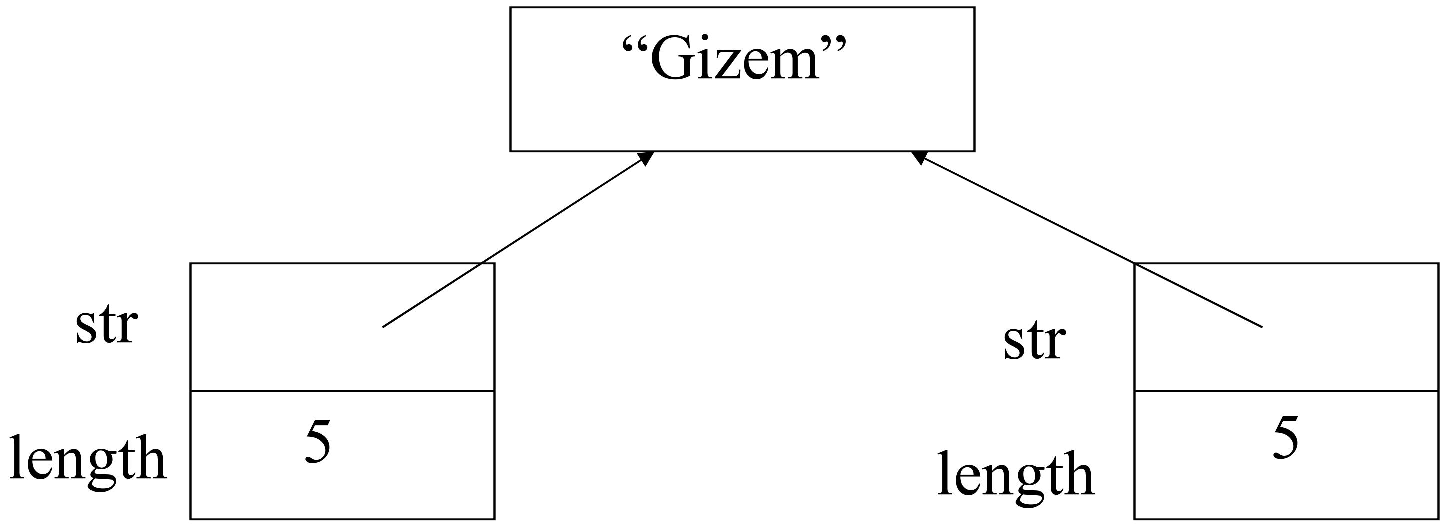
```
MyString::MyString(const MyString& s)
{
 length = s.length;
 str = new char[length + 1];
 strcpy(str, s.str);
}
```

- What is the compiler's default copy constructor?

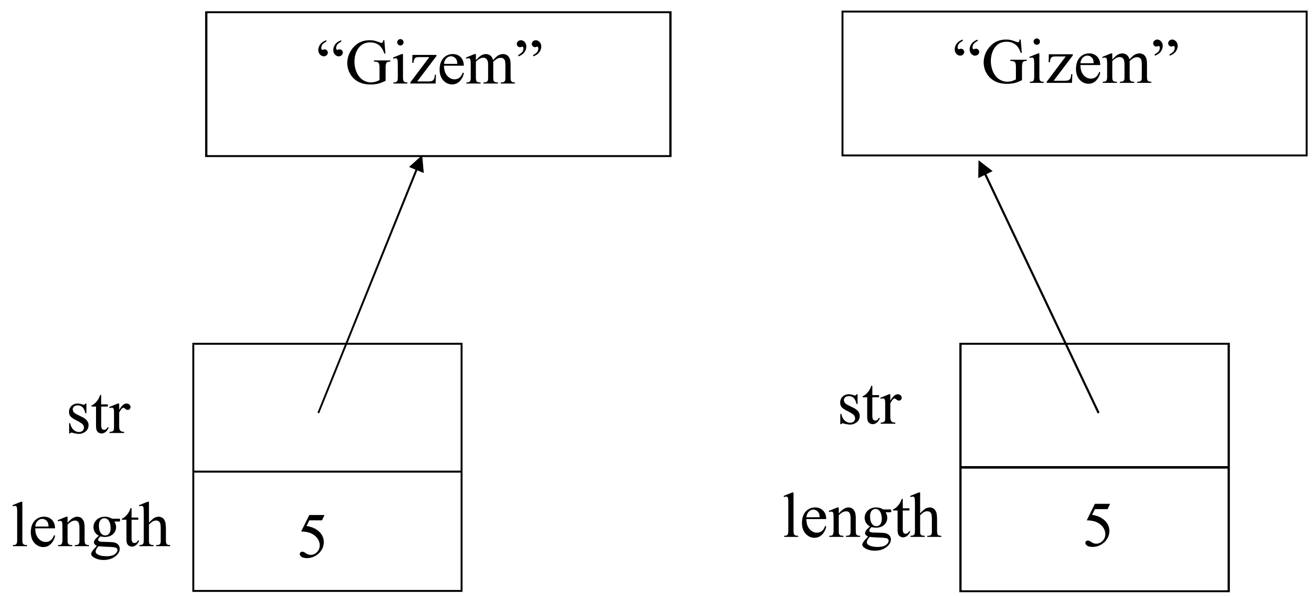
Shallow versus Deep copy

- Shallow copy is a copy of pointers rather than data being pointed at.
- A deep copy is a copy of the data being pointed at rather than the pointers.

Shallow copy: only pointers are copied



Deep copy: the actual data are copied



Deep copy semantics

- How to write the copy constructor in a class that has dynamically allocated memory:
  1. Dynamically allocate memory for data of the calling object.
  2. Copy the data values from the passed-in parameter into corresponding locations in the new memory belonging to the calling object.
  3. A constructor which does these tasks is called a *deep copy constructor*.

Calling the copy constructor

- Automatically called:

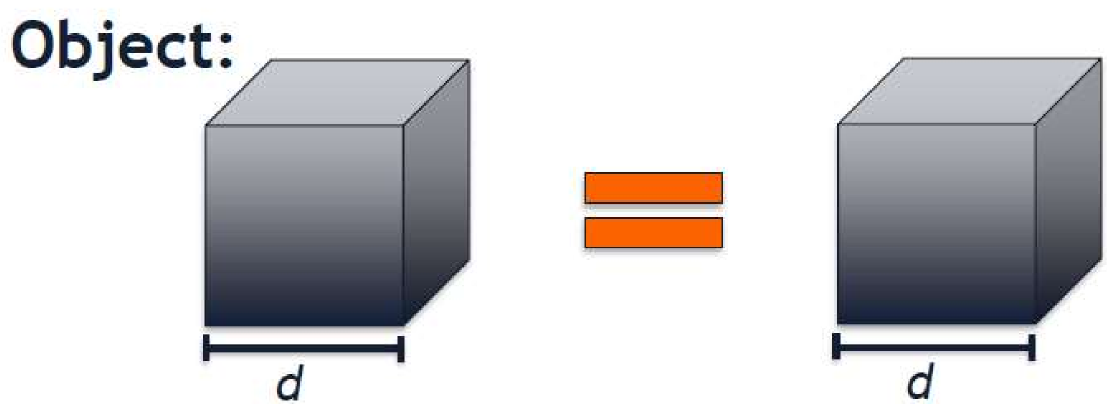
```
A x(y);     // Where y is of type A.
f(x);       // A copy constructor is called
            // for value parameters.
x = g();    // A copy constructor is called
            // for value returns.
```

- More examples:

```
MyObject a;         // default constructor call
MyObject b(a);      // copy constructor call
MyObject bb = a;    // identical to bb(a) : copy
                    // constructor call
MyObject c;         // default constructor call
c = a;              // assignment operator call
```

Assignment operator

- In C++, an **assignment operator** defines the behavior when an object is copied using the assignment operator = .



Copy Constructor vs Assignment

A copy constructor **creates a new object** (constructor).

An assignment operator assigns a **value to an existing object**.

- An assignment operator is always called on an object that has already been constructed.

Automatic Assignment Operator

If an assignment operator is not provided, the C++ compiler provides an automatic assignment operator.

The automatic assignment operator will copy the contents of all member variables.

- **by default - memberwise copy** :
  - Sets variables equal, i.e., x = y;
  - Memberwise copy — member by member copy

myObject1 = myObject2;

- This is *shallow copy*.

Custom Assignment Operator

A custom assignment operator is:

- Is a public member function of the class.
- Has the function name `operator=`.
- Has a return value of a reference of the class' type.
- Has exactly one argument
- The argument must be constreference of the class' type.

Example:

```
IntCell & operator=(const IntCell & obj )
```

Deep vs Shallow Assignment

- Same kind of issues arise in the assignment.
- For shallow assignments, the default assignment operator is OK.
- For deep assignments, you have to write your own *overloaded* assignment operator (`operator=`)
  - The copy constructor is not called when doing an object-to-object assignment.

`this` Pointer

- Each class object has a pointer which automatically points to itself. The pointer is identified by the keyword `this`.
- Another way to think of this is that each member function has an implicit first parameter; that parameter is `this`, the pointer to the object calling that function.

Example: overloading operator=

```
// defining an overloaded assignment operator
Complex & Complex::operator=(const Complex & rhs )
{
   // don't assign to yourself!
   if ( this != &rhs )  // note the "address of" rhs
   {
        this -> Re = rhs.Re; // correct but redundant
                   // it means Re = rhs.Re
        this -> Imag = rhs.Imag;
   }
   return *this;   // return the calling class object
                   // enables cascading
}
```

Another Example

```
MyString& MyString::operator=(const MyString& rhs)
{
  if (this != &rhs) {
      delete[] this->str; // donate back useless memory

      this->length = rhs.length;

      // allocate new memory
      this->str = new char[this->length + 1];

      strcpy(this->str, rhs.str); // copy characters
  }
  return *this;    // return self-reference
}
```

Copy constructor and assignment operator

- Note that the copy constructor is called when a **new** object is being created
- The assignment operator is called when an **existing** object is assigned to a new state.

```
class MyObject {
public:
  MyObject();         // Default constructor
  MyObject(const MyObject &a);  // Copy constructor
  MyObject& operator=(const MyObject& a) // Assignment op.
};
MyObject a;           // constructor called
MyObject b = a;       // copy constructor called
b = a;                // assignment operator called
```

Destructor

- For classes with pointers we also need to define a destructor to avoid memory leaks

```
class MyString {
  public:
      MyString(const char* s = "");
      MyString(const MyString& s);
      ~MyString(); // destructor
      MyString& operator=(const MyString& s);
      ...
  private:
      int length;
      char* str;
};
```

Destructor

- For classes with pointers we also need to define a destructor to avoid memory leaks

```
MyString::~MyString()
{
   delete[] str;
}
```

Outline

1. Classes : constructors, destructors, clean interface
2. Default arguments
3. Function and operator overloading
4. Use of const
5. Rule of three: copy constructor, assignment, destructor
6. **Templates: function and class templates**
7. **C++ Error Handling**
8. **STL: `vector` class**

Templates

- Templates allow us to write routines that work for arbitrary types without having to know what these types will be.

- Two types of templates:
  - Function templates
  - Class templates

Function Templates

- A function template is not an actual function; instead it is a design (or pattern) for a function.
- The compiler creates the actual function based on the actual types used in the program.

```
// swap function template.

template < class T>
void swap( T &lhs, T &rhs )
{
        T tmp = lhs;
        lhs = rhs;
        rhs = tmp;
}
```

The `swap` function template

Using a template

- Instantiation of a template with a particular type, logically creates a new function.
- Only one instantiation is created for each parameter-type combination.

```
int main()
{       int x = 5, y = 7;
        double a = 2, b = 4;
        swap(x,y); //instanties an int version of swap
        swap(x,y); //uses the same instantiation
        swap(a,b); //instantiates a double version of swap

        cout << x << " " << y << endl;
        cout << a << " " << b << endl;

//      swap(x, b); // Illegal: no match
        return 0;
}
```

Class templates

- Class templates are used to define generic classes:
  - e.g. it may be possible to use a class that defines several operations on a collection of integers to manipulate a collection of real numbers.

```
template <class T>
class TemplateTest
{
   // this class can use T as a generic type
   public:
      void f(T a);
      T g();
   ...
   private:
      T x, y, z;
   ...
};
```

Implementation

- Each member function must be declared as a template.
- All member functions must be implemented in the **header file** (so that the compiler can find their definition and replace "T" with the actual parameter type)

```
// Typical member implementation.
template <class T>
void TemplateTest<T>::f(T a)
{
  // Member body
}
```

Object declarations using template classes

Form:

class-name <*type*> *an-object*;

Interpretation:

- *Type* may be any defined data type. *Class-name* is the name of a template class. The object *an-object* is created when the arguments specified between < > replace their corresponding parameters in the template class.

Example

```
// Memory cell interface (MemoryCell.h)

template <class T>
class MemoryCell
{
  public:
     MemoryCell(const T& initVal = T());
     const T& read( ) const;
     void write(const T& x);

  private:
     T storedValue;
};
```

Class template implementation

```
// Implementation of class members as template functions

template <class T>
MemoryCell<T>::MemoryCell(const T& initVal) :
  storedValue(initVal){ }

template <class T>
const T& MemoryCell<T>::read() const
{
  return storedValue;
}

template <class T>
void MemoryCell<T>::write(const T& x)
{
  storedValue = x;
}
```

A simple test routine

```
int main()
{
  MemoryCell<int> m; // instantiate int version
  MemoryCell<float> f; // instantiate float version
  MemoryCell<int> m2; // use the previously created class

  m.write(5);
  m2.write(6);
  f.write(3.5);
  cout << "Cell content: " << m.read() << endl;
  cout << "Cell content: " << m2.read() << endl;
  cout << "Cell content: " << f.read() << endl;
  return 0;
}
```

Outline

1. Classes : constructors, destructors, clean interface
2. Default arguments
3. Function and operator overloading
4. Use of const
5. Rule of three: copy constructor, assignment, destructor
6. Templates: function and class templates
7. **C++ Error Handling**
8. **STL: `vector` class**

C++ Error Handling

- In C, errors are reported by returning error codes from functions:

```
int read(const char* filename, char data[])
{
   FILE* fp = fopen(filename, "r");
   if (fp == NULL)
      return -1; // indicate error

   // read file contents into data
   ...
}
```

C++ Error Handling

- In C++, we have a more advanced mechanism called exceptions
- It uses three keywords: **throw**, **catch**, **try**
- The function that encounters an error throws an exception:

```
int read(const char* filename, char data[])
{
   FILE* fp = fopen(filename, “r”);
   if (fp == NULL)
      throw "file open error"; // indicate error

   // otherwise read file contents into data
   ...
}
```

C++ Error Handling

- This exception must be caught, otherwise the program will abnormally terminate:

```
int main()
{
    char data[128];
    try {
        read("test.txt", data);
        ... // possibly some other code
    }
    catch(const char* error) {
        // if read function throws an exception,
        // program will continue executing from here
        cout << "Error message: " << error << endl;
    }
}
```

C++ Error Handling

- Note that we throw an object or a variable, and we catch an object or a variable. These types should match for the exception to be caught
- In the previous example we threw a `const char*` and caught a `const char*`, so it was correct

Another Example

- We can also throw an object of a user defined class:

```
class FileReadError
{
};

int read(const char* filename, char data[])
{
   FILE* fp = fopen(filename, "r");
   if (fp == NULL)
      throw FileReadError(); // indicate error

   // read file contents into data
   ...
}
```

C++ Error Handling

- Then we must update the catch code as well:

```
int main()
{
    char data[128];
    try {
        read("test.txt", data);
    }
    catch(FileReadError error) {
        // if read throws an exception,
        // we will come here
    }
}
```

C++ Error Handling

- There are many details of exception handling
- In this class, you should only know that the destructors of the local objects will be called when an exception is thrown:

```
class A {
public:
    ~A() { cout << "destructor called" << endl; }
};

int read(const char* filename, char data[]) {
   A a;
   FILE* fp = fopen(filename, "r");
   if (fp == NULL)
      throw "file open error"; // a's destructor will be called
   ...
}
```

Standard Template Library

- I/O Facilities: iostream
- Garbage-collected String class
- Containers
  - vector, list, queue, stack, map, set
- Numerical
  - complex
- General algorithms
  - search, sort

Using the `vector`

- Vector: Dynamically growing, shrinking array of elements
- To use it include library header file:

```
#include <vector>
```

- Vectors are declared as

```
vector<int> a(4); //a vector called a,
                  //containing four integers
vector<int> b(4, 3); //a vector of four
                // elements, each initialized to 3.
vector<int> c;   // 0 int objects
```

- The elements of an integer vector behave just like ordinary integer variables

```
a[2] = 45;
```

Manipulating vectors

- **The `size()` member function** returns the number of elements in the vector.

  `a.size()` returns a value of 4.
- **The `operator=`** can be used to assign one vector to another.
- e.g. `v1 = v2`, so long as they are vectors of the same type.
- **The `push_back()` member function** allows you to add elements to the end of a vector.

push_back() and pop_back()

```
vector<int> v;
v.push_back(3);
v.push_back(2);
// v[0] is 3, v[1] is 2, v.size() is 2
v.pop_back();
int t = v[v.size()-1];
v.pop_back();
```