Introduction to C++

(Courtesy of Prof. Nihan Kesim Çiçekli. Used with permission. Please do not copy or distribute without a written permission of Prof. Nihan Kesim Çiçekli.)
Programming in C++

- C++
  - Improves on many of C's features
  - Has object-oriented capabilities
    - Increases software quality and reusability
  - Developed by Bjarne Stroustrup at Bell Labs
    - Called "C with classes"
    - C++ (increment operator) - enhanced version of C
  - Superset of C
    - Can use a C++ compiler to compile C programs
    - Gradually evolve the C programs to C++
Object Oriented Programming

➢ The emphasis is on creating a set of tools which can be used cleanly, with a minimum knowledge about implementation in the user’s driver files. The following concepts are relevant to accomplishing clean interface:

• **Data Abstraction**
  – Define an object by its data and allowable operations: an abstract data type (i.e. interface).

• **Encapsulation**
  – Bundle data and operations into one logical unit (i.e. the implementation)

• **Information hiding**
  – Restrict access to data so that it can be manipulated only in authorized ways. Separate class declarations from implementation (e.g. public, private in C++).
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.
A Basic C++ Program

#include <iostream>
#include <math.h>

using namespace std;

int main()
{
    float x;

    cout << "Enter a real number: " << endl;
    cin >> x;

    cout << "The square root of " << x << " is: " << sqrt(x) << endl;
}

A Basic C++ Program

// second C++ program

#include <iostream>

using namespace std;

int main()
{
    int a=23;
    int b=34;

    cout << "Enter two integers:" << endl;
    cin >> a >> b;
    cout << endl;

    cout << "a + b =" << a+b << endl;
    return 0;
}
A Basic C++ Program

// third C++ program

#include <iostream>
#include <iomanip>
using namespace std;

int main(){
    double a=15.2;
    double b=34.3434343;

    cout << fixed << showpoint;
    cout << setprecision(2);
    cout << setw(6) << a << endl;
    cout << setw(7) << b << endl;

    return 0;
}
A Basic C++ Program

// third C++ program

#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    double a=15.2;
    double b=34.3434343;

    cout << fixed << showpoint;
    cout << setprecision(2);
    cout << setw(6) << a << endl;
    cout << setw(7) << b << endl;

    return 0;
}
Classes and Objects

- **Class**: a type definition that includes both
  - data properties, and
  - operations permitted on that data

- **Object**: a variable that
  - is declared to be of some Class
  - therefore includes both data and operations for that data

- **Appropriate usage:**
  “A variable is an instance of a type.”
  “An object is an instance of a class.”
Basic Class Syntax

• A class in C++ consists of its members.
  – A member can be either data or functions.
• The functions are called member functions (or methods)
• Each instance of a class is an object.
  – Each object contains the data components specified in class.
  – Methods are used to act on an object.
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;
};
Class Members

• **public** member is visible to all routines and may be accessed by any method in any class.

• **private** member is not visible to non-class routines and may be accessed only by methods in its class.

• Typically,
  – Data members are declared private
  – Methods are made public.

• Restricting access is known as *information hiding*. 
Constructors

• A constructor 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,
  – no return type
  – 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.
// A class for simulating an integer memory cell

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

        int read( ) const
            { return storedValue; } 

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

    private:
        int storedValue;
};

Extra Constructor Syntax

Single constructor (instead of two)
Object Declaration

• In C++, an object is declared just like a primitive type.

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

int main()
{
    // correct declarations
    IntCell m1;
    IntCell m2 (12);
    IntCell *m3;

    // incorrect declaration
    Intcell m4();  // this is a function declaration,
                   // not an object
```
Object use in driver program

// program continues

m1.write(44);
m2.write(m2.read() +1);
cout << m1.read() << "   " << m2.read() << endl;
m3 = new IntCell;
cout << "m3 = " << m3->read() << endl;
return 0;
}
Example: Class Time

class Time {
public:
    Time( int = 0, int = 0, int = 0 ); //default constructor
    void setTime( int, int, int ); //set hr, min, sec
    void printMilitary(); // print am/pm format
    void printStandard(); // print standard format

private:
    int hour;
    int minute;
    int second;
};
Declaring Time Objects

// Note that implementation of class Time not given
// here.

int main()
{
    Time t1, // all arguments defaulted
    t2(2), // min. and sec. defaulted
    t3(21, 34), // second defaulted
    t4(12, 25, 42); // all values specified
    ...
}

Data Structures
Class Interface and Implementation

• In C++, separating the class interface from its implementation is common.
  – The interface remains the same for a long time.
  – The implementations can be modified independently.
  – The writers of other classes and modules have to know the interfaces of classes only.

• The **interface** lists the class and its members (data and function prototypes) and describes what can be done to an object.

• The **implementation** is the C++ code for the member functions.
Separation of Interface and Implementation

- It is a good programming practice for large-scale projects to 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` (`.cc` or `.C`)
  - .cpp file includes the .h file with the `preprocessor` command `#include`.
    - Example: `#include "myclass.h"`
Separation of Interface and Implementation

• A big complicated project will have files that contain other files.
  – There is a danger that an include file (.h file) might be read more than once during the compilation process.
    • It should be read only once to let the compiler learn the definition of the classes.

• To prevent a .h file to be read multiple times, we use preprocessor commands #ifndef and #define in the following way.
Class Interface

```cpp
#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
#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;
}
A driver program

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

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
Destructors

- Member function of class
- Performs termination housekeeping before the system reclaims the object’s memory
- Complement of the constructor
- Name is tilde (~) followed by the class name
- E.g. ~IntCell();
  ~Time();
- Receives no parameters, returns no value
- One destructor per class
class IntCell{
    public:
    IntCell(int initialValue=0)
    { storedValue = new int (initialValue); }

    ~IntCell()
    { delete storedValue; }

    int read() const
    { return *storedValue; }

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

    private:
    int *storedValue;
}
When are Constructors and Destructors Called

- **Global scope objects**
  - Constructors called before any other function (including main)
  - Destructors called when main terminates (or exit function called)

- **Automatic local objects**
  - Constructors called when objects defined
  - Destructors called when objects leave scope (when the block in which they are defined is exited)

- **static local objects**
  - Constructors called when execution reaches the point where the objects are defined
  - Destructors called when main terminates or the exit function is called
Accessor and Modifier Functions

- A method that examines but does not change the state of its object is an accessor.
  - Accessor function headings end with the word const.
- A member function that changes the state of an object is a mutator.
Another Example: Complex Class

```c++
#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) const;
    float modulus() const;
    void print() const;
};

#endif
```

Complex class interface in the file Complex.h
#include <iostream>
#include <cmath>
#include "Complex.h"

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;
}

float Complex::modulus() const
{
    return sqrt(re*re + im*im);
}

void Complex::print() const
{
    std::cout << "(" << re <<"," << im << ")" << std::endl;
}
Using the class in a Driver File

```cpp
#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 Imag 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
Function Overloading

• Function overloading:
  – Functions with same name and different parameters
  – Overloaded functions performs similar tasks
    • Function to square ints and function to square floats
      
      ```c
      int square(int x) {return x * x;}
      float square(float x) { return x * x; }
      ```
    – Program chooses function by signature
      • Signature determined by function name and parameter types
      • Type safe linkage - ensures proper overloaded function called
// Using overloaded functions
#include <iostream>
using std::cout;
using std::endl;
int square( int x ) { return x * x; }
double square( double y ) { return y * y; }

int main()
{
    cout << "The square of integer 7 is " << square( 7 )
        << "\nThe square of double 7.5 is " << square( 7.5 )
        << endl;

    return 0;
}
Overloaded Operators

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

\[ 2 + 3 \quad 3.1 + 3.2 \quad \Rightarrow \quad + \text{ is an overloaded operator} \]

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

\[ \Rightarrow \text{ we may overload it} \]
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;
}
Using the class in a Driver File

```cpp
#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)
    c1.print();

    return 0;
}
```

A program that uses Complex in file TestComplex.cpp
Putting the Scalar to the Left

To support multiplying with a scalar on the left, we must define a new function that is outside the class scope.

```c
Complex operator*(float k, Complex c)
{
    Complex prod;
    prod.re = k * re; // Compile Error: cannot access re
    prod.im = k * im; // Compile Error: cannot access im
    return prod;
}
```

Note that this function has access errors: an outside function cannot access the private members of a class! We can solve this in two ways.
class Complex
{
    ...

public:
    // add the following functions to the class

    void setReal(float x) { re = x; }
    void setImag(float x) { im = x; }
    float getReal() const { return re; }
    float getImag() const { return im; }

    ...
};
Solution 1: Setter/Getter Functions

Complex operator*(float k, Complex c)
{
    Complex prod;
    prod.setReal(k * c.getReal());
    prod.setImag(k * c.getImag());
    return prod;
}
Solution 2: Friend Functions

Declare the outside function as the friend of this class. It can then access the private members of the class.

```cpp
class Complex
{
    ...

    friend Complex operator*(float k, Complex rhs);

    ...

};
```
Solution 2: Friend Functions

```cpp
Complex operator*(float k, Complex c) {
    Complex prod;
    prod.re = k * re; // Now it is ok
    prod.im = k * im; // Now it is ok
    return prod;
}
```

Note that the “friend” keyword is not used here. It is only used inside the class (see the previous slide).
Friend Classes

A class may declare another class as a friend as well. In that case all member functions of the “befriended” class can access the private members of its friend class.

```cpp
class A
{
    ...
};

class B
{
    ...
    friend A;
};
```

“A” can access private members of “B” (but not vice versa!!)
References

• References are a type of C++ variable that act as an *alias* to another variable.
• A reference variable acts just like the original variable it is referencing.
• References are declared by using an ampersand (&) between the reference type and the variable name.
Example

```cpp
int n = 5, m = 6;
int &rn = n;

n = 6;
rn = 7,
cout << n << rn << m << endl;
rn = m;
cout << n << rn << m << endl;
```

You cannot declare a reference without giving a value.

Makes n equal to m (doesn't make rn refer to m)
**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.
References vs Pointers

Everything that is accomplished by references can be accomplished by pointers but the syntax of references is simpler:

Example

```c
int n = 5;
int &rn = n;
int *const p = &n;
*p = 6;

rn = 6;  // Same effect
```
Pointers and \texttt{const}

There are two different ways that pointers and \texttt{const} can be intermixed:

1. Constant pointer
2. Pointer to a constant variable
Constant Pointer

• A `const` pointer must be initialized to a value upon declaration, and its value can not be changed.
• However, because the value being pointed to is still non-`const`, it is possible to change the value being pointed to via dereferencing the pointer:

```c
int *const p = &i;
*p = 6;    // it is O.K.
p = &j;    // NOT O.K.
```
Pointer to a `const` variable

- It is also possible to declare a pointer to a constant variable by using the `const` before the data type:
  ```c
  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:
  ```c
  i = 6;  // It is O.K.
  ```

- It is also possible to declare a `const` pointer to a constant value:
  ```c
  const int n = 5;
  const int * const p = &n;
  ```
Parameter Passing

In C, all parameters are passed by value (call by value). But C++ offers three options:

• **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
#include <iostream>
using std::cout;
using std::endl;

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;
}
Example (cont.)

```c
int squareByValue( int a )
{
    return a *= a;  // caller's argument not modified
}
void squareByReference( int &cRef )
{
    cRef *= cRef;  // caller's argument modified
}
int squareByConstReference (const int& a )
{
    // a *= a;  not allowed (compiler error)
    return a * a;
}
```
Improving the Complex Class

```cpp
#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
#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 << ""

Complex class implementation in file Complex.cpp
The uses of keyword `const`

We may encounter `const` in the following cases:

1. Const reference parameter:

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

   In this case it means the parameter cannot be modified.

2. Const member function:

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

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

3. Const object/variable:

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

   In this case it means the object cannot be modified.
Memory Management

In C++ we use `new` and `delete` instead of `malloc` and `free` used in C

- `new` - automatically creates object of proper size, calls constructor, returns pointer of the correct type
- `delete` - destroys object (calls the destructor) and frees space

• Example:

```cpp
int* pi = new int(6);
Complex *pc = new Complex(3, 5);
delete pi;
delete pc;
```
// Allocate an array of complex objects (calls the default constructor for each object).
Complex *ptr1 = new Complex [10];

for (int i = 0; i < 10; ++i)
    ptr[i]->print();

delete[] ptr1; // note the delete[] syntax

// similar for int
ptr2 = new int[12];

// free up the dynamically allocated array
delete [] ptr2;
Default Arguments Revisited

• In C++ functions can have default arguments
• This is specified in the function declaration (not the definition):

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

```cpp
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 Revisited

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

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

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

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

- Functions with same name and different parameters
- Overloaded functions should perform similar tasks (otherwise it would be confusing):
- **Function to square ints and function to square floats**
  ```
  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 Revisited

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

  ```
  int square(int x);
  float square(int x);  // Compile error
  ```
Operator Overloading Revisited

• Remember that we overloaded the * operator for the Complex class.

• Operator overloading allows us to use existing operators for user-defined classes.

• The following operators can be overloaded:

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

• Note that the precedence, associativity, and arity of the operators cannot be changed!
Copy Constructor

➢ The copy constructor for a class is responsible for creating copies of objects of that class type whenever one is needed. This includes:

1. when the user explicitly requests a copy of an object,
2. when an object is passed to function by value, or
3. when a function returns an object by value.
Copy constructor

➢ The copy constructor does the following:
1. takes another object of the same class as an argument, and
2. initialize the data members of the calling object to the same values as those of the passed in parameter.

➢ If you do not define a copy constructor, the compiler will provide one, it is very important to note that compiler provided copy constructor performs member-wise copying of the elements of the class (i.e. Shallow copy).
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.
Syntax for Copy Constructor

\[ A(\text{const } A& \ a2) \ \{ \]

\[ \quad \ldots \]

\[ \quad } \]

- Note that the parameter must be a const reference.
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;
}
Example

class MyString
{
    public:
        MyString(const char* s = "");
        MyString(const MyString& s);
        ...
    private:
        int length;
        char* str;
};
MyString::MyString(const MyString& s)
{
    length = s.length;
    str = new char[length + 1];
    strcpy(str, s.str);
}
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 by Default: Memberwise Copy

• Assignment operator (=)
  – Sets variables equal, i.e., \( x = y \);
  – Can be used to assign an object to another object of the same type
  – Memberwise copy — member by member copy
    \[
    \text{myObject1} = \text{myObject2};
    \]
  – This is shallow copy.
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 (but not friends) 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
        // why?
    {
        this -> Re = rhs.Re; // correct but
        //redundant: means Re = rhs.Re
        this -> Imag = rhs.Imag;
    }
    return *this; // return the calling class object
        // enables cascading
}
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 object

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

```cpp
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

```cpp
MyString::~MyString()
{
    delete[] str;
}
```
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
static Class Members

• Shared by all objects of a class
  – Normally, each object gets its own copy of each variable
• Efficient when a single copy of data is enough
  – Only the static variable has to be updated
• May seem like global variables, but have class scope
  – Only accessible to objects of same class
• Initialized at file scope
• Exist even if no instances (objects) of the class exist
• Can be variables or functions
  • public, private, or protected
Example

In the interface file:

private:
    static int count;
...

public:
    static int getCount();
...

Implementation File

```cpp
int Complex::count = 0; //must be in file scope

int Complex::getCount()
{
    return count;
}
Complex::Complex()
{
    Re = 0;
    Imag = 0;
    count ++;
}
```
Driver Program

cout << Complex :: getCount() << endl;
Complex c1;
cout << c1.getCount();
Templates

• The template allows us to write routines that work for arbitrary types without having to know what these types will be.

• Two types:
  – 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.

```cpp
// swap function template.

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

```cpp
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.

```cpp
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.

```cpp
template <class T>
void TemplateTest<T>::f(T a)
{
    // Member body
}
```
Object declarations using template classes

Form:

\[ \text{class-name } \text{}\text{<type>} \text{}\text{an-object;} \]

Interpretation:

- \textit{Type} may be any defined data type. \textit{Class-name} is the name of a template class. The object \textit{an-object} is created when the arguments specified between \texttt{< >} 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

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

```cpp
int main()
{
    MemoryCell<int> m; // instantiate int version
    MemoryCell<float> f; // instantiate float ver.
    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;
}
```
Friend functions - revisited

• A friend function of a class is a nonmember function of the class, but has access to all members of the class.

• Reserved word friend appears only in the function prototype(not in definition of the friend function)
#include <iostream>
#ifndef _Complex_H
#define _Complex_H
using namespace std;
class Complex
{ private: // default
    float Re, Imag;

public:
    Complex( float x = 0, float y = 0 )
    {
        Re = x;
        Imag = y;
    }

    ~Complex() { }

    float modulus();
    void print() const;
    friend void dummy(Complex One);
};
#endif
Example: Friend functions of a Class (cont’d)

```cpp
void dummy(Complex One) {
    One.Re = 3;
    One.Imag = 5;
    cout << One.Re << One.Imag << endl;
}

...

int main() {
    Complex MyComplexNo(1,1);
    dummy(MyComplexNo);
    return 0;
}
```
Example 2: Complex Class

```cpp
#include <iostream>
#ifndef _Complex_H
#define _Complex_H
using namespace std;
class Complex
{
 private:
  float Re, Imag;
 public:
  Complex( float x = 0, float y = 0 )
  {  Re = x; Imag = y; }
  ~Complex() { }
  Complex operator*( Complex & rhs ) const;
  float getReal() const;
  float getImag() const;
  float modulus() const;
  friend ostream & operator<< (ostream &os, Complex & rhs);
};
#endif
```

Complex class interface in the file Complex.h
Using the class in a Driver File

```cpp
#include <iostream>
#include "Complex.h"
using namespace std;
int main()
{
    Complex c1, c2(1), c3(1,2);
    float x;

    c1 = c2 * c3 * c2;

    x = c1.modulus();

    cout << c1 << " " << c2 << endl;
    return 0;
}
```

A program that uses Complex in file TestComplex.cpp
Implementation of Complex Class

// File complex.cpp
#include <iostream>
#include "Complex.h"

Complex Complex:: operator*( Complex & rhs )
{
    Complex prod;  //some place to store the results...
    prod.Re = (Re*rhs.Re - Imag*rhs.Imag);
    prod.Imag = (Imag*rhs.Re + Re*rhs.Imag);
    return prod;
}

float Complex:: modulus() const
{
    // this is not the real def of complex modulus
    return Re / Imag;
}

ostream & operator<< (ostream & out, Complex & rhs)
{
    out << "(" << rhs.Re <<"," << rhs.Imag << ")";
    return out;  // allow for concat of << operators
}

float Complex::getReal() const { return Re; }
float Complex::getImag() const { return Imag; }

Complex class implementation in file Complex.cpp
C++ Error Handling

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

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

```cpp
int read(const char* filename, char data[]) {
    FILE* fp = fopen(filename, "r");
    if (fp == NULL)
        throw "file open error"; // indicate error
    // read file contents into data
    ...
}
```
C++ Error Handling

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

```cpp
int main()
{
    char data[128];
    try {
        read("test.txt", data);
        ... // some other code
    }
    catch(const char* error) {
        // if read 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:

```cpp
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:

```cpp
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:

```cpp
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
}
```
Example of a try-catch Statement

```
try
{
    // Statements that process personnel data and may throw
    // exceptions of type int, string, and SalaryError
}
catch ( int )
{
    // Statements to handle an int exception
}
catch ( string s )
{
    cout << s << endl; // Prints "Invalid customer age"
    // More statements to handle an age error
}
catch ( SalaryError )
{
    // Statements to handle a salary error
}
```
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.
  \[ \text{a.size()} \text{ returns a value of 4.} \]

• The `=` operator can be used to assign one vector to another.

• e.g. \[ \text{v1 = v2, so long as they are vectors of the same type.} \]

• The `push_back()` member function allows you to add elements to the end of a vector.
**push_back() and pop_back()**

```cpp
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();
```