Introduction to C++

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++

Clean Interface

- 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.
- **Information hiding**
  – Restrict access to data so that it can be manipulated only in authorized ways. Separate class declarations from implementation.
- **Encapsulation**
  – Bundle data and operations into one logical unit.

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>

int main()
{
   float x;

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

   std::cout << "The square root of " << x << " is: "
              << sqrt(x) << std::endl;
   return 0;
}
```

Classes and Objects

- **<u>Class:</u>** a type definition that includes both
  - data properties, and
  - operations permitted on that data
- **<u>Object:</u>** 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;
};
```

constructors

Class Members

- <u>`Public`</u> member is visible to all routines and may be accessed by any method in any class.
- <u>`Private`</u> 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 <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,
  - 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.

Extra Constructor Syntax

```
// 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;
};
```

Single constructor (instead of two)

Accessor and Modifier Functions

- A method that examines but does not change the state of its object is an <u>accessor</u>.
  - Accessor function headings end with the word `const`
- A member function that changes the state of an object is a <u>mutator</u>.

Object Declaration

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

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

   // incorrect declaration
   Intcell m4();    // this is a function declaration,
                   // not an object
```

Object Access

```
m1.write(44);
m2.write(m2.read() +1);
std::cout << m1.read() << "   " <<  m2.read()
          << std::endl;
m3 = new IntCell;
std::cout << "m3 = " << m3->read() << std::endl;
```

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

```
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
    . . .
}
```

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

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

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 <u>interface</u> lists the class and its members (data and function prototypes) and describes what can be done to an object.
- The <u>implementation</u> 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

```
#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*

Class Implementation

```
#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 driver program

```
#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*

Another Example: `Complex` Class

```
#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() { }

    Complex operator* ( Complex & rhs );
    float modulus();
    friend ostream & operator<< (ostream &os, Complex & rhs);
};
#endif
```

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

Using the class in a Driver File

```
#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.Imag*c1.Imag );

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

   std::cout << c1 << " " << c2 << std::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;   //someplace 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()
{     // 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
}
```

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

Parameter Passing

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

Example

```
#include <iostream>
using std::cout;
using std::endl;
int squareByValue( int );
void squareByReference( int & );
int squareByConstReference ( const int & );
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.)

```
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 )
{
  return a * a;
}
```

The uses of keyword `const`

1. const reference parameters

   These may not be modified in the body of a function to which they are passed. Idea is to enable pass by reference without the danger of incorrect changes to passed variables.

2. const member functions or operators

   These may not modify any member of the object which calls the function.

3. const objects
   1. These are not supposed to be modified by any function to which they are passed.
   2. May not be initialized by assignment; only by constructors.

Dynamic Memory Allocation with Operators `new` and `delete`

- **`new`** and **`delete`**
  - `new` - automatically creates object of proper size, calls constructor, returns pointer of the correct type
  - `delete` - destroys object and frees space
  - You can use them in a similar way to `malloc` and `free` in C.
- **Example**:
  - `TypeName *typeNamePtr;`
  - `typeNamePtr = new TypeName;`
    - new creates TypeName object, returns pointer (which typeNamePtr is set equal to)
  - `delete typeNamePtr;`
    - Calls destructor for TypeName object and frees memory

More examples

```
// declare a ptr to user-defined data type
Complex *ptr1;
int *ptr2;

// dynamically allocate space for a Complex;
// initialize values; return pointer and assign
// to ptr1
ptr1 = new Complex(1,2);

// similar for int:
ptr2 = new int( 2 );
// free up the memory that ptr1 points to
delete ptr1;
```

```
// dynamically allocate array of 23
// Complex slots
// each will be initialized to 0
ptr1 = new Complex[23];

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

// free up the dynamically allocated array
delete [] ptr1;
```

Default Arguments and Empty Parameter Lists

- If function parameter omitted, gets default value
  - Can be constants, global variables, or function calls
  - If not enough parameters specified, rightmost go to their defaults
- Set defaults in function prototype

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

- Empty parameter lists
  - In C, empty parameter list means function takes any argument
  - In C++ it means function takes no arguments
  - To declare that a function takes no parameters:
    - Write void or nothing in parentheses

Prototypes:
```
void print1( void );
void print2();
```

```
// Using default arguments
#include <iostream>
using std::cout;
using std::endl;
int boxVolume(int length = 1,int width = 1,int height = 1);
int main()
{  cout << "The default box volume is: " << boxVolume()
        << "\n\nThe volume of a box with length 10,\n"
        << "width 1 and height 1 is: " << boxVolume( 10 )
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 1 is: " << boxVolume( 10, 5 )
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 2 is: " << boxVolume(10,5,2)
        << endl;
   return 0;
}
// Calculate the volume of a box
int boxVolume( int length, int width, int height )
{    return length * width * height;
}
```

Function Overloading

- Function overloading:
  - Functions with same name and different parameters
  - Overloaded functions performs similar tasks
    - 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; }
```

  - 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   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

- 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

```
class A {
public:
  A(int xval, int yval) { x=xval; y=yval; }
  bool operator==(const A& rhs) const{
  return ((x==rhs.x) && (y==rhs.y));
}

private:
  int x;
  int y;
};
```

Overloaded Operators – Example (cont.)

```
int main() {
  A a1(2,3);
  A a2(2,3);
  A a3(4,5);
  if (a1.operator==(a2)){ cout << "Yes" << endl;}
  else { cout << "No" << endl; }
  if (a1 == a2 ) { cout << "Yes" << endl; }
  else { cout << "No" << endl; }
  if (a1 == a3 ) { cout << "Yes" << endl; }
  else { cout << "No" << endl; }
  return 0;
}
```

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 <u>by value</u>**, or
  3. when a function **returns an object <u>by value</u>**.

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.

Syntax

```
A(const A& a2) {
…
}
```

- 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;
    Imag = C.Imag;
}
```

Example

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

Example (cont.)

```
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:

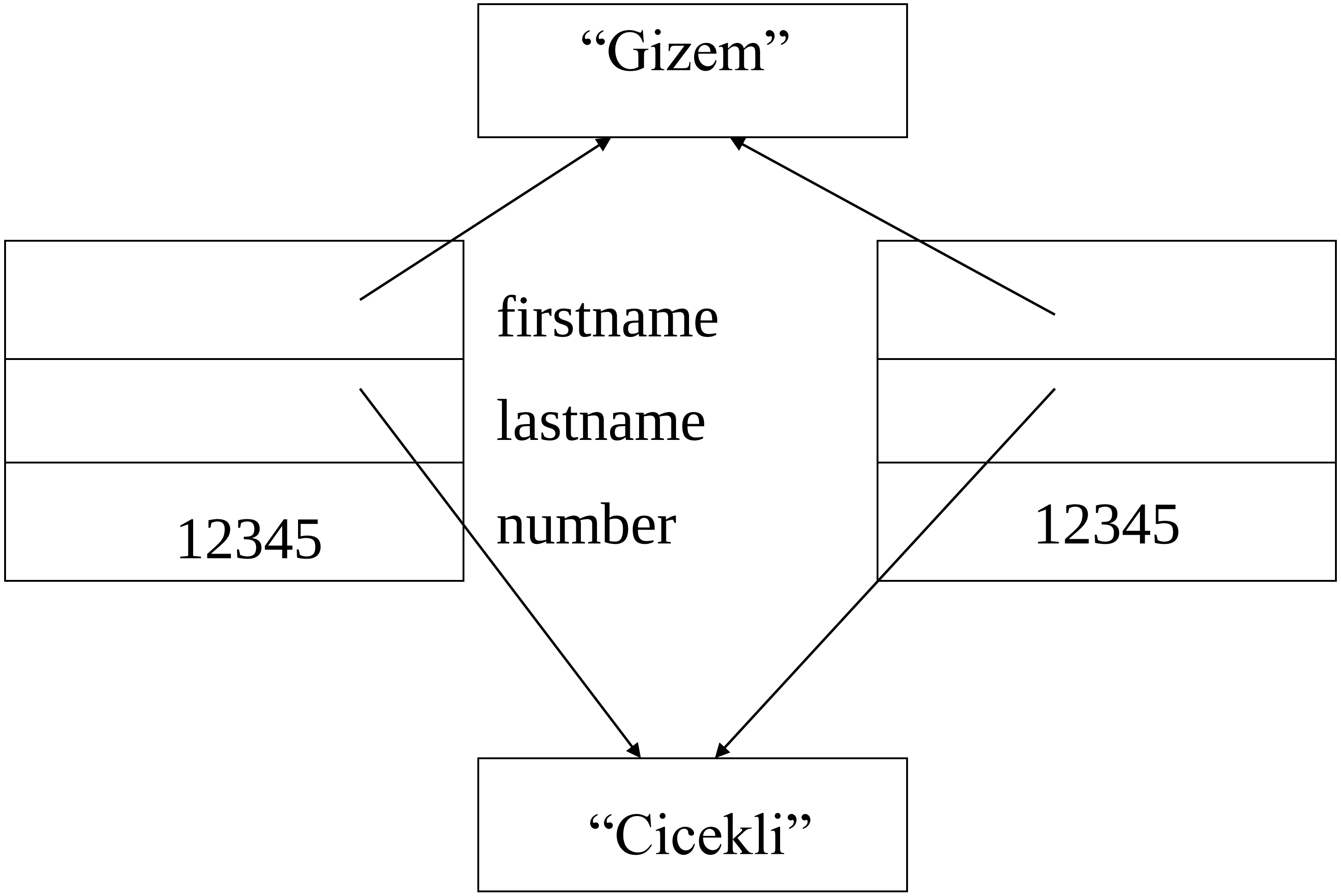
```
MyObject a;         // default constructor call
MyObject b(a);      // copy constructor call
MyObject bb = a;    // identical to bb(a) : copy
                       //constructor call
MyObject c;         // default constructor call
c = a;              // assignment operator call
```

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

myObject1 = myObject2;

  - This is *shallow copy*.



Shallow copy: only pointers are copied

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.

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*.

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

```
// defn of 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: enable cascading
}
```

Example

```
const MyString& operator=(const MyString& rhs)
{
  if (this != &rhs) {
      delete[] this->str; // donate back useless memory
      // allocate new memory
      this->str = new char[strlen(rhs.str) + 1];
      strcpy(this->str, rhs.str); // copy characters
      this->length = rhs.length; // copy length
  }
  return *this;   // return self-reference so cascaded
                  //assignment works
}
```

Copy constructor and assignment operator

- Copying by initialisation corresponds to creating an object and initialising its value through the copy constructor.
- Copying by assignment applies to an existing object and is performed through the assignment operator (=).

```
class MyObject {
public:
  MyObject();        // Default constructor
  MyObject(MyObject const & a);  // Copy constructor
  MyObject & operator = (MyObject const & a)
                         // Assignment operator
}
```

`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

```
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.
  - Similar to `typedef` but more powerful
- 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.
- This design is expanded (like a preprocessor macro) as needed to provide an actual routine.

```
// swap function template.
// Object: must have copy constructor and operator =

template < class Object>
void swap( Object &lhs, Object &rhs )
{
      Object 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);
        swap(x,y); //uses the same instantiation
        swap(a,b);
        cout << x << " " << y << endl;
        cout << a << " " << b << endl;
//      swap(x, b); // Illegal: no match
        return 0;
}
```

Class templates

- Class templates are used to define more complicated data abstractions.
  - 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.

```
// Form of a template interface
template <class T>
class class-name
{
        public:
        // list of public members
        ...
        private:
        // private members
        ...
};
```

Interpretation:

Class *class-name* is a template class with parameter T. T is a placeholder for a built-in or user-defined data type.

Implementation

- Each member function must be declared as a template.

```
// Typical member implementation.
template <class T>
ReturnType
class-name<T>::MemberName( Parameter List ) /* const*/
{
  // 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

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

  private:
     Object storedValue;
};
```

Class template implementation

```
// Implementation of class members.
#include “MemoryCell.h”

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

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

A simple test routine

```
int main()
{
  MemoryCell<int> m;

  m. write(5);
  cout << “Cell content: ” << m.read() <<
  endl;
  return 0;
}
```

C++ Exception Handling: `try`, `throw`, `catch`

- A function can **`throw`** an exception object if it detects an error
  - Object typically a character string (error message) or class object
  - If exception handler exists, exception caught and handled
  - Otherwise, program terminates
- Format
  - enclose code that may have an error in **`try`** block
  - follow with one or more **`catch`** blocks
    - each **`catch`** block has an exception handler
  - if exception occurs and matches parameter in **`catch`** block, code in catch block executed
  - if no exception thrown, exception handlers skipped and control resumes after catch blocks
  - **`throw`** point - place where exception occurred
    - control cannot return to **`throw`** point

```
// Fig. 23.1: fig23_01.cpp
// A simple exception handling example.
// Checking for a divide-by-zero exception.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

// Class DivideByZeroException to be used in exception
// handling for throwing an exception on a division by zero.
class DivideByZeroException {
public:
   DivideByZeroException()
      : message( "attempted to divide by zero" ) { }
   const char *what() const { return message; }
private:
   const char *message;
};

// Definition of function quotient. Demonstrates throwing
// an exception when a divide-by-zero exception is encountered.
double quotient( int numerator, int denominator )
{
   if ( denominator == 0 )
      throw DivideByZeroException();

   return static_cast< double > ( numerator ) / denominator;
}
```

EXAMPLE

- Class definition
- Function definition

```cpp

// Driver program
int main()
{
   int number1, number2;
   double result;

   cout << "Enter two integers (end-of-file to end): ";

   while ( cin >> number1 >> number2 ) {

      // the try block wraps the code that may throw an
      // exception and the code that should not execute
      // if an exception occurs
      try {
         result = quotient( number1, number2 );
         cout << "The quotient is: " << result << endl;
      }
      catch ( DivideByZeroException ex ) { // exception handler
         cout << "Exception occurred: " << ex.what() << '\n';
      }

      cout << "\nEnter two integers (end-of-file to end): ";
   }

   cout << endl;
   return 0;      // terminate normally
}
```

- Initialize variables
- Input data
- `try` and `catch` blocks
- Function call
- Output result

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.

  `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();
```

Inheritance

- **Inheritance** is the fundamental object-oriented principle governing the reuse of code among related classes.
- Inheritance models the **IS-A relationship**. In an IS-A relationship, the derived class is a variation of the base class.
  - e.g. Circle IS-A Shape, car IS-A vehicle.
- Using inheritance a programmer creates new classes from an existing class by adding additional data or new functions, or by redefining functions.

Inheritance Hierarchy

- Inheritance allows the derivation of classes from a **base class** without disturbing the implementation of the base class.
- A **derived class** is a completely new class that inherits the properties, public methods, and implementations of the base class.
- The use of inheritance typically generates a **hierarchy** of classes.
- In this hierarchy, the derived class is a **subclass** of the base class and the base class is a **superclass** of the derived class.
- These relationships are **transitive**.

Base and Derived Classes

- Often an object from a derived class (subclass) “is an” object of a base class (superclass)

Base class	**Derived classes**
`Student`	`GraduateStudent` `UndergraduateStudent`

`Shape`	`Circle`
	`Triangle`
	`Rectangle`

`Loan`	`CarLoan`
	`HomeImprovementLoan`
	`MortgageLoan`

`Employee`	`FacultyMember` `StaffMember`
`Account`	`CheckingAccount` `SavingsAccount`

Illustration of Inheritance

```
class mammal        // base class
{
   public:
     // manager functions
     mammal( int age = 0, int wt = 0 ):itsAge(age),
             itsWt( wt ) { }
     ~mammal() { }

     // access functions
     int getAge() const { return itsAge; }
     int getWt() const { return itsWt; }

     // implementation functions
     void speak() const{ cout << "mammal sound!\n";}
     void sleep() const{ cout << zzzzzzzzzzzz!\n"; }

   protected:
     int itsAge, itsWt;
};
```

```
class dog : public mammal
{
   public:
     // manager functions
     dog( int age, int wt, string name ) :
     mammal( age, wt )
                  { itsName = name; }
     dog( int age=0, int wt=0 ) : mammal(age,wt)
                  { itsName = ""; }
     ~dog() { } // nothing to do

     // implementation function
     void speak() const { cout << "ARF ARF\n"; }
     void wagtail() const { cout << "wag wag
  wag\n"; }

   private:
     string itsName;
};
```

```
int main()
{
   dog bowser(3, 25, "Bowser");

   bowser.speak();
   bowser.mammal :: speak();
   bowser.wagtail();
   bowser.sleep();

   cout << "bowser is " << bowser.getAge() << "
years old!" << endl;
   return 0;
}
```

Here is the output of the sample code:

```
ARF ARF
mammal sound!
wag wag wag
zzzzzzzzzzzz!
bowser is 3 years old!
```

Overriding Functions

- If derived class has a member function with the same name, return type and parameter list as in the base class, then the derived class function *overrides* the base class function.
- The base class function is *hidden*.
- The implementation of the base class function has been changed by the derived class.
- Derived class objects invoke the derived version of the function.
- If a derived class object wants to use the base class version, then it can do so by using the scope resolution operator:

```
bowser.speak()   // derived class version is invoked
bowser.mammal::speak() //base class version
```

Private vs protected class members

1. private base class member(s)
   - derived class member functions can not access these objects directly
   - the member still exists in the derived class object
   - because not directly accessible in the derived class, the derived class object must use base class access functions to access them

2. protected base class member(s)
   - directly accessible in the derived class
   - member becomes a protected member of the derived class as well

Constructors and destructors

1. Constructors
   - Constructors are not inherited.
   - Base class constructor is called before the derived class constructor (either explicitly, or if not then the compiler invokes the default constructor).
   - Base class constructor initializes the base class members.
   - The derived class constructor initializes the derived class members that are not in the base class.
   - A derived class constructor can pass parameters to the base class constructor as illustrated in the example.
   - Rules of thumb for constructors under inheritance:
     - Define a default constructor for every class.
     - Derived class constructors should explicitly invoke one of the base class constructors.
2. Destructors
   - Derived class destructor is called before the base class destructor.
   - Derived class destructor does cleanup chores for the derived class members that are not in the base class.
   - Base class destructor does the same chores for the base class members.

Abstract Methods and Classes

- Delete this topic
- An **abstract method** is declared in the base class and always defined in the derived class.
- It does not provide a default implementation, so each derived class must provide its own implementation.
- A class that has at least one abstract method is called an **abstract class**.
- Abstract classes can never be instantiated.

Example

- An abstract class : `Shape`
- Derive specific shapes: `Circle`, `Rectangle`
- Derive `Square` from `Rectangle`
- The `Shape` class can have data members that are common to all classes:e.g. `name`, `positionOf`.
- Abstract methods apply for each particular type of object: e.g. `area`

Abstract base class `Shape`

```
class shape
{
   public:
       Shape(const string & shapeName = “”)
          : name( shapeName ) {}
       virtual ~Shape( ) { }
       virtual double area( ) const = 0;
       bool operator< (const Shape & rhs) const
          { return area () < rhs.area (); }
       virtual void print(ostream & out = cout ) const
          { out << name << ‟ of area ‟ << area();}
  private:
       string name;
}
```

Expanding `Shape` Class

```
const double PI = 3.1415927;

class Circle : public Shape
{
  public :
      Circle( double rad = 0.0 )
         : Shape(“circle”), radius(rad) {}
      double area() const
         { return PI * radius * radius;}
  private:
      double radius;
};
```