CSE225: Data Structure using C++

Outline

- Fundamentals of C++
- Class & inheritance
- Overloading & overriding
- Templates, Error handling,…

The New C++ Headers

- The new-style headers do not specify file-extension.
- They simply specify standard identifiers that might be mapped to files by the compiler.
  - <iostream>
  - <vector>
  - <string>, not related with <string.h>
  - <cmath>, C++ version of <math.h>
  - <cstring>, C++ version of <string.h>
- Programmer defined header files should end in ".h".

Namespaces

- A namespace is a declarative region.
- It localizes the names of identifiers to avoid name collisions.
- The contents of new-style headers are placed in the **std** namespace.
- Example: namespace.cpp

C++ Console I/O (Output)

- cout << “Hello World!”;
  - printf(“Hello World!”);
- cout << iCount; /* int iCount */
  - printf(“%d”, iCount);
- cout << 100.99;
  - printf(“%f”, 100.99);
- cout << “\n”, or cout << ‘\n’, or cout << endl
  - printf(“\n”)
- In general, cout << *expression*;

cout ???

Shift right operator ???

How does a shift right operator produce output to the screen?

polymorphism here

C++ Console I/O (Input)

- cin >> strName; /* char strName[16] */
  - scanf("%s", strName);
- cin >> iCount; /* int iCount */
  - scanf("%d", &iCount);
- cin >> fValue; /* float fValue */
  - scanf("%f", &fValue);
- In general, cin >> *variable*;

namespace

```
// Online C++ compiler to run C++ program online
#include <iostream>

namespace first
{
    int x = 1;
}

namespace second
{
    int x = 2;
}

using namespace first;
int main() {

        int x = 0;
    //std::cout << first::x;
    //std::cout << second::x;
    std::cout << x;
    return 0;
}
```

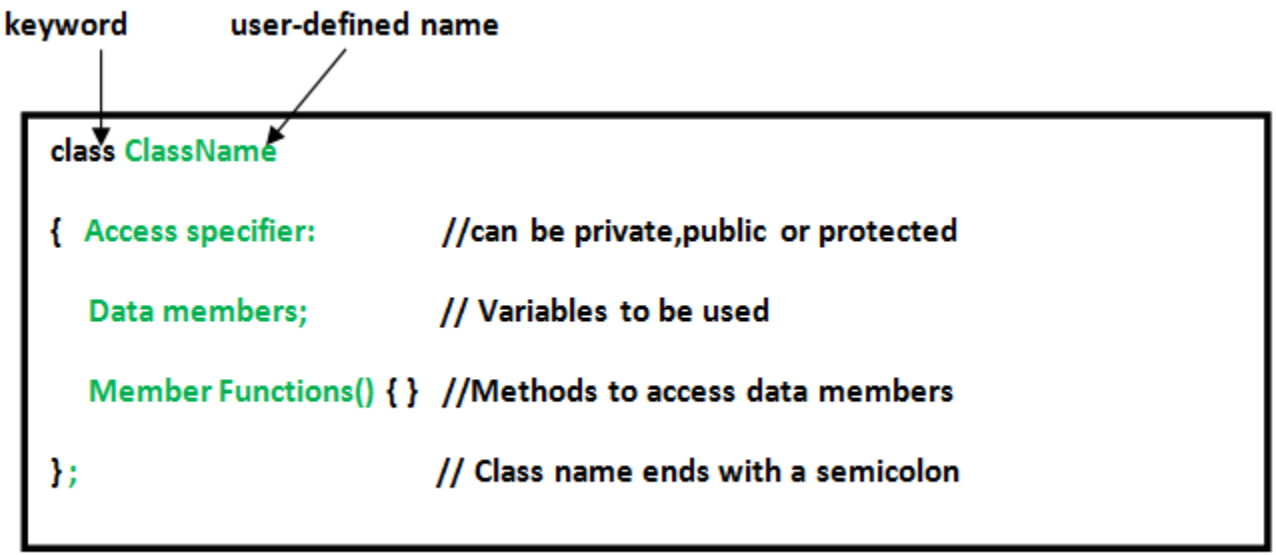
Sample Code

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

int main()
{
        float a;
        cin>>a;
        cout << "Value of a: "<<a;

        return 0;
}
```

Classes: A First Look

- General syntax -

```
class class-name
{
        // private functions and variables
public:

        // public functions and variables
};
```


keyword
user-defined name
class ClassName
{ Access specifier: //can be private,public or protected
Data members; // Variables to be used
Member Functions() { } //Methods to access data members
}; // Class name ends with a semicolon

How to write a class in C++:

A class is an expanded concept of a data structure:
instead of holding only data, it can hold both data and functions.

An object is an instantiation of a class.
In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword class, with the following format:

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
  } object_names;
```

Example-: Crectangle.cpp

A PIE Model of OOP

- Abstraction
- Polymorphism
- Inheritance
- Encapsulation

A PIE Model of OOP

- **Abstraction:** Abstraction of Data or Hiding of Information is called **Abstraction**
- **Polymorphism:** It is the ability to redefine *methods* for *derived classes.* or we can say that object can behave in different forms is call Polymorphism.

- **Inheritance:** I**nheritance** enables new objects to take on the properties of existing objects. There are different ways in which Inheritance can be done.
  - Single Inheritance
  - Multi-level Inheritance
  - Multiple Inheritance
  - Hierarchical Inheritance

- **Encapsulation: Binding of Data and Functions (that manipulate the data) together** and keep both safe from outside interference and misuse is called Encapsulation.

Classes: A First Look (cont.)

- A class declaration is a logical abstraction that defines a new type.
- It determines what an object of that type will look like.
- An object declaration creates a physical entity of that type.
- That is, an object occupies memory space, but a type definition does not.

- **Example:** box.cpp

Classes: A First Look (cont.)

- Each object of a class has its own copy of every variable declared within the class (except static variables which will be introduced later), but they all share the same copy of member functions.

```
class Box
{
   double dLength, dWidth, dHeight;
   double dVolume;
   public:
   double vol(){return dLength * dWidth * dHeight;}

}  b;
```

Public vs. private

- Public functions and variables are accessible from anywhere the object is visible
- Private functions and variable are only accessible from the members of the same class and “friend”
- Protected

```
class Box
{
   double dLength, dWidth, dHeight;
   double dVolume;
   public:
   void setValue(){
       cout<<"Enter value for dLength: ";
       cin>>dLength;
       cout<<"Enter value for dWidth: ";
       cin>>dWidth;
       cout<<"Enter value for dHeight: ";
       cin>>dHeight;
   }
   double vol(){
       return dLength * dWidth * dHeight;
   }

};

int main()
{
    Box b;
            //b.dLength = 10.25;
            b.setValue();
            cout<<"Vol: "<<b.vol();
            return 0;
}
```

Constructors

- A special member function with the same name of the class
- No return type (not void)
- Executed when an instance of the class is the created

- **Destructors:**
- A special member function with no parameters
- Executed when the class is destroyed

- //box1.cpp [for constructor and destructor]

Empty constructor & Copy constructor

- Empty constructor
  - The default constructor with no parameters when an object is created
  - Do nothing: e.g. Examp::Examp(){}
- Copy constructor
  - Copy an object (shallow copy)
  - The default constructor when an object is copied (call by value, return an object, initialized to be the copy of another object)

Constructor

```
int id;

//Default Constructor
Test()
{
    cout << "Default Constructor called" << endl;
    id=-1;
}

//Parameterized Constructor
Test(int x)
{
    cout <<"Parameterized Constructor called "<< endl;
    id=x;
}
};
int main() {

// obj1 will call Default Constructor
Test obj1;
cout <<"Test id is: "<<obj1.id << endl;

// obj2 will call Parameterized Constructor
Test obj2(21);
cout <<"Test id is: " <<obj2.id << endl;
return 0;
```

Destructor

```
using namespace std;

static int Count = 0;

class Test {
public:
    Test()
    {
        Count++;
        cout << "No. of Object created: " << Count << endl;
    }

    ~Test()
    {
        cout << "No. of Object destroyed: " << Count<< endl;
        Count--;
    }
};

int main()
{
    Test t, t1, t2, t3;

    return 0;
}
```

Creating and Using a Copy Constructor

- By default when a assign an object to another object or initialize a new object by an existing object, a bitwise copy is performed.
- This cause problems when the objects contain pointer to dynamically allocated memory and destructors are used to free that memory.
- It causes the same memory to be released multiple times that causes the program to crash.

Copy Constructor

```
    Test(int x, int y)  //Parameterized Constructor
    {
        cout <<"Parameterized Constructor called "<< endl;
        a = x; b = y;
    }

    Test(Test &obj){
        cout <<"Copy Constructor called "<< endl;
        a = obj.a;  b = obj.b;
    }

    void display(){
        cout<<"a : "<<a<<", and b : "<<b<<"\n";
    }
};
int main() {

    Test obj1;  // Default Constructor is called
    obj1.display();

    Test obj2(10, 15);  // Parameterized Constructor is called
    obj2.display();

    //Test obj3 = obj1;
```

- Copy constructors are used to solve this problem while we perform object initialization with another object of the same class.
  - MyClass ob1;
  - MyClass ob2 = ob1; // uses copy constructor
- Copy constructors do not affect assignment operations.
  - MyClass ob1, ob2;
  - ob2 = ob1; // does not use copy constructor

Creating and Using a Copy Constructor (contd.)

- If we do not write our own copy constructor, then the compiler supplies a copy constructor that simply performs bitwise copy.
- We can write our own copy constructor to dictate precisely how members of two objects should be copied.
- The most common form of copy constructor is
  - classname (const classname &obj) {
  - // body of constructor
  - }

Creating and Using a Copy Constructor (contd.)

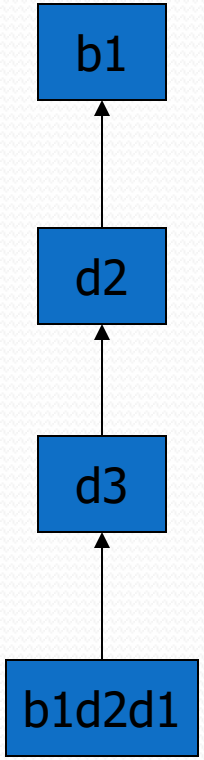
- Object initialization can occur in three ways
  - When an object is used to initialize another in a declaration statement
    - MyClass y;
    - MyClass x = y;
  - When an object is passed as a parameter to a function
    - func1(y); // calls "void func1(MyClass obj)"
  - When a temporary object is created for use as a return value by a function
    - y = func2(); // gets the object returned from "MyClass func2()"
- See the examples from the book and the supplied codes to have a better understanding of the activities and usefulness of copy constructors.
  - Example: copy-cons.cpp

Inheritance

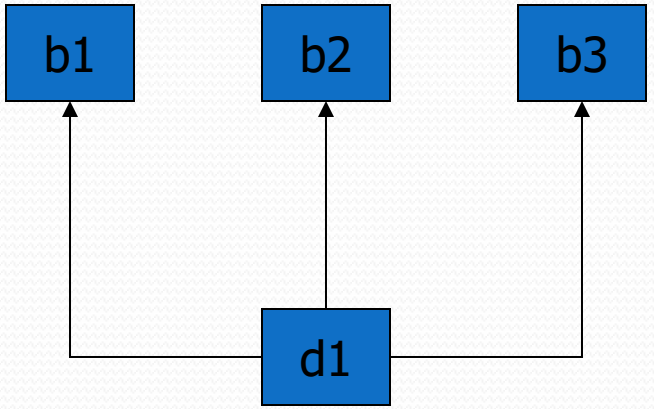
- Base class
- Derived class

- class derived-class-name : ***access*** base-class-name { … };
- Here ***access*** is one of three keywords
  - public
  - private
  - protected
- Use of ***access*** is optional
  - It is private by default if the derived class is a **class**
  - It is public by default if the derived class is a **struct**

Multiple Inheritance (contd.)



Option - 1



Option - 2

Inheritance

	base	derived
Public inheritance	public	public
	protected	protected
	private	N/A
Private inheritance	public	private
	protected	private
	private	N/A
Protected inheritance	public	protected
	protected	protected
	private	N/A

Inheritance

```
        {
            cout << "Base ID: " << id_p << endl;
        }
};

// Sub class
class Child : public Parent {
public:
    // derived class members
    int id_c;
    void printID_c()
    {
        // id_p = 10;
        cout << "Child ID: " << id_c << endl;
        // cout << "Base ID: " << id_p << endl;
    }
};

// main function
int main()
{
    // creating a child class object
    Child obj1;

    obj1.id_p = 7;
    obj1.printID_p();
```

Static members in class

- Static variables
  - Shared by all objects
- Static functions
  - Have access to static members only
- Static members can be accessed by the class name

Friend functions / Friend Class

- Have access to the private members of a class.
- Must be declared as friend in that class.
- Why friend functions?
  - Efficiency
- A class can be declared as the friend of another class.

Static Members

```
class Test{
  public:
  static int a;
  int b = 25;

  void display(){
      cout<<"(Non-Static Funciton) Value of a: "<<a<<", b:
"<<b<<endl;
  }

  static void display_static(){
      cout<<"(Static Function) Value of a: "<<a<<endl;
    //   cout<<", b: "<<b<<endl;
  }
};

int Test::a=10;

int main()
{
    Test obj1, obj2;
    cout<<obj1.a<<endl;
    //Test::a=12;
    cout<<obj2.a<<endl;

    obj1.display();
```

Friend Function

```
class Distance {
    private:
        int meter;
        // friend function
        friend void addFive(Distance);

    public:
        Distance() { meter=0; }
    protected:
        int wheel;
};

// friend function definition
void addFive(Distance d) {
    //accessing private members from the friend function
    d.meter += 5;
    d.wheel = 10;
    cout<<"Wheel: "<<d.wheel<<endl;
    cout << "Distance: " << d.meter;
}

int main() {
    Distance D;
   //D.wheel=20;
    addFive(D);
```

Friend Function

```
// forward declaration
class ClassB;

class ClassA {
    int numA;

    public:
        // constructor to initialize numA to 12
        ClassA() {numA = 12;}

         // friend function declaration
         friend int add(ClassA, ClassB);
};

class ClassB {
    int numB;

    public:
        // constructor to initialize numB to 1
        ClassB() {numB = 5;}

        // friend function declaration
        friend int add(ClassA, ClassB);
};
```

Friend Class

```
    int numA;

    // friend class declaration
    friend class ClassB;

public:
    // constructor to initialize numA to 12
    ClassA() {numA =12;}
};

class ClassB {
    int numB;

public:
    // constructor to initialize numB to 1
    ClassB() {numB =10;}

    // member function to add numA
    // from ClassA and numB from ClassB
    int add() {
        ClassA objectA;
        //objectA.numA = 45;
        return objectA.numA + numB;
    }
};
```

Function overloading

- Define several functions of the same name, differ by parameters.

void Show()

void Show(char *str)

Void Show(int x)

Function overloading

- Must have different parameters
  - int funcı(int a, int b);
  - double funcı(int a, int b);
  - void func(int value);
  - void func(int &value);
- Static binding
  - The compilers determine which function is called.
- (Often used for the multiple constructors)

Function Overloading

```
}

void add(int a, int b, int c)
{
    cout << endl << "sum of three integers = " << (a + b + c);
}

void add(int a, double b)
{
    cout<<endl<<"sum of one integer ("<<a<<") and one
double ("<<b<<") = "<<(a+b);
}
void  add(double a, int b)
{
    cout<<endl<<"sum of one double ("<<a<<") and one
integer ("<<b<<") = "<<(a+b);
}

int main()
{
    add(10, 2);
    add(5, 6, 4);
    add(13,2.5);
    add(8.5,6);
    return 0;
```

Overloading summary

- Same name
- Different parameters
- Static binding (compile time)
- Anywhere

Important Point on Inheritance

- In C++, only public inheritance supports the perfect IS-A relationship.
- In case of private and protected inheritance, we cannot treat a derived class object in the same way as a base class object
- If we use private or protected inheritance, we cannot assign the address of a derived class object to a base class pointer directly.
- This is one of the reason for which Java only supports public inheritance.

Overloading & overriding

- Polymorphism
- Static and dynamic
  - Compile time and running time
- Parameters
- Anywhere / between the base and derived class

Outline

- Fundamentals of C++
- Class & inheritance
- Overloading & overriding
- Templates, Error handling,…

Generic Functions

- A generic function defines a general set of operations that will be applied to various types of data
- Allows to create a function that can automatically overload itself !!!
- Allows to make the data type, on which to work, a parameter to the function

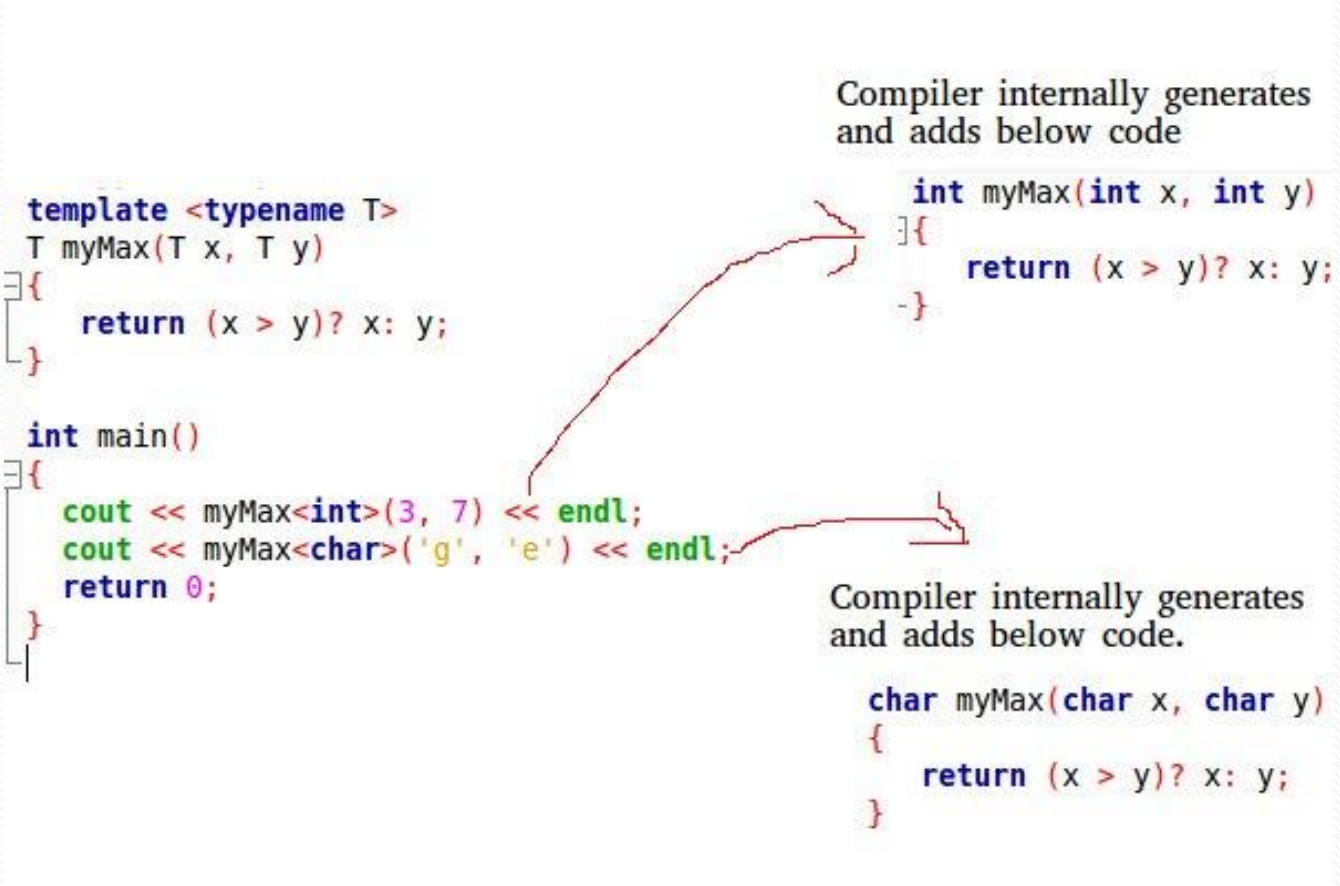
- General form

```
template <class T>
ret-type func-name(param list)
    {
      // body of function
    }
```

Here,

- template is a keyword
- We can use keyword “typename” in place of keyword “class”
- “TtypeN” is the placeholder for data types used by the function

Generic Functions



C++ adds two new keywords to support templates: ***'template'*** and ***'typename'***. The second keyword can always be replaced by the keyword **'class'**.

Template Function

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

template <typename T> T myMax(T x, T y)
{
    return (x > y) ? x : y;
}

int main()
{
    // Call myMax for int
    cout << myMax<int>(3, 7) << endl;
    // call myMax for double
    cout << myMax<double>(4.5, 9.5) << endl;
    // call myMax for char
    cout << myMax<char>('g', 'e') << endl;

    return 0;
}
```

Generic Functions (Example-1)

```
template <class X>
void swapargs(X &a, X &b) {
  X temp;
  temp = a;
  a = b;
  b = temp;
}
template <class X1, class X2>
void print(X1 x, X2 y) {
  cout << x << ", " << y << endl;
}
```

```
void main() {
  int i = 10, j = 20;
  double x = 11.11, y = 22.22;

  print(i, j); // 10, 20
  swapargs<int> (i, j); // (int, int)
  print(i, j); // 20, 10

  print(x, y); // 11.11, 22.22
  swapargs<double> (x, y); (double,
double)
  print(x, y); // 22.22, 11.11

  print(i, y); // 20, 11.11
     // (int, double)
}
```

Generic Functions (contd.)

- The compiler generates as many different versions of a template function as required
- Generic functions are more restricted than overloaded functions
  - Overloaded functions can alter their processing logic
  - But, a generic function has only a single processing logic for all data types
- We can also write an explicit overload of a template function

Template Function (overload)

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

template <typename T> T myMax(T x, T y)
{
    return (x > y) ? x : y;
}

double myMax(double x, double y)
{
   // return (x > y) ? x : y;
   return (x+y);
}

int main()
{

    // Call myMax for int
    cout << myMax(3, 7) << endl;
    // call myMax for double
    cout << myMax(4.5, 9.5) << endl;
    // call myMax for char
    cout << myMax('g', 'e') << endl;
```

Generic Functions (Example-2)

```
template <class X>
void swapargs(X &a, X &b)
    { cout << “template version\n”; }

void swapargs(int &a, int &b)
    { cout << “int version\n”; }

void main()
{
   int i = 10, j = 20;
   double x = 11.11, y = 22.22;
   swapargs(i, j); // “int version”
   swapargs(x, y); // “template version”
}
```

Generic Classes

- Makes a class data-type independent
- Useful when a class contains generalizable logic
  - A generic stack
  - A generic queue
  - A generic linked list etc. etc. etc.
- The actual data type is specified while declaring an object of the class
- General form

```
template <class Ttype1, class Ttype2, ..., class TtypeN>
class class-name
{
   // body of class
};
```

Template Class

- A class template starts with the keyword **template** followed by **template parameter(s)** inside <> which is followed by the class declaration.
- In the above declaration, T is the template argument which is a placeholder for the data type used, and **class** is a keyword.
- Inside the class body, a member variable var and a member function functionName() are both of type T.

```
template <class T>
class className {
  private:
    T var;
    ... .. ...
  public:
    T functionName(T arg);
    ... .. ...
};
```

```
        num2 = n2;
    }

    void displayResult() {
        cout << "Numbers: " << num1 << " and " << num2 << "." << endl;
        cout << num1 << " + " << num2 << " = " << add() << endl;
        cout << num1 << " - " << num2 << " = " << subtract() << endl;
        cout << num1 << " * " << num2 << " = " << multiply() << endl;
        cout << num1 << " / " << num2 << " = " << divide() << endl;
    }

    T add() { return num1 + num2; }
    T subtract() { return num1 - num2; }
    T multiply() { return num1 * num2; }
    T divide() { return num1 / num2; }
};

int main() {
    Calculator<int> intCalc(2, 1);
    Calculator<float> floatCalc(2.4, 1.2);

    cout << "Int results:" << endl;
    intCalc.displayResult();

    cout << endl
```

Generic Classes (Example)

```
template <class X>
class stack {
   X stck[10];
   int tos;
public:
   void init( ) { tos = 0; }
   void push(X item);
   X pop( );
};
```

```
template <class X>
void stack<X>::push(X
item) { ... }

template <class X>
X stack<X>::pop( ) { ... }
```

Generic Classes (Example) (contd.)

```
void main( ) {
   stack<char> s1, s2;
   s1.init( );
   s2.init( );
   s1.push('a');
   s1.push('b');
   s2.push('x');
   s2.push('y');
   cout << s1.pop( ); // b
   cout << s2.pop( ); // y
```

```
   stack<double> ds1, ds2;
   ds1.init( );
   ds2.init( );
   ds1.push(1.1);
   ds1.push(2.2);
   ds2.push(3.3);
   ds2.push(4.4);
   cout << ds1.pop( ); // 2.2
   cout << ds2.pop( ); // 4.4
}
```

Function Templates

- Generic function for different types.
  - E.g. get the min value of three variables (int, float, char)
- Define function templates 46.cpp
  - Template<class S, class T…>
    func_name(…)
  - Func_name<type name>(…)
  - something like macro
- More powerful than macro
- Example: templatefun.cpp

Class templates

- Generic classes
- Define class templates

  Template<class T>
  Class {…}

  Template<class T>
  ret_type class_name<type_name> :: func_name (…){
  }

  Example: Templeteclass.cpp

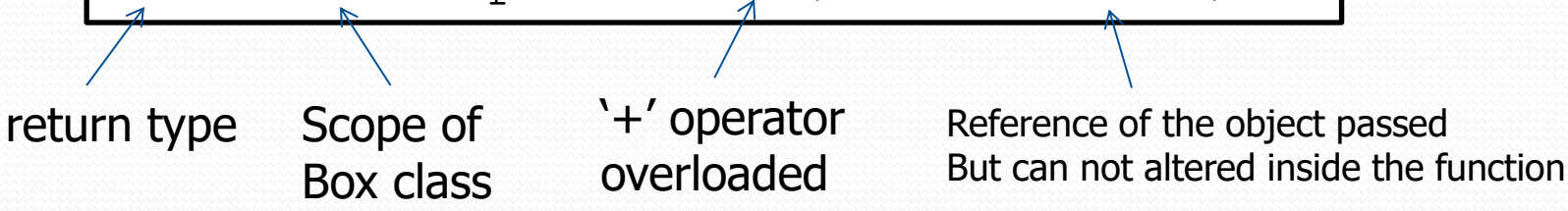
Operator Overloading

- Redefine or overload most of the built-in operators available in C++

- Overloaded operators are functions with special names
  - keyword "operator" followed by the symbol operator
  - an overloaded operator has a return type and a parameter list.

Operator Overloading Syntax

```
Box Box::operator+(const Box&);
```



- declares the addition operator for the Box class that can be used to **add** two Box objects and returns final Box object.

Example

```
class Box {
public:
   double getVolume(void);
   void setLength( double );
   void setBreadth( double );
   void setHeight( double );
   Box operator+(const Box& );

private:
   double length;
   double breadth;
   double height;
};
```

```
double Box::getVolume(void) {
    return length * breadth * height;
}
void Box::setLength( double len ) {
    length = len;
}
void Box:: setBreadth( double bre ) {
    breadth = bre;
}
void Box:: setHeight( double hei ) {
    height = hei;
}

Box Box:: operator+(const Box& b) {
    Box box;
    box.length = this->length + b.length;
    box.breadth = this->breadth + b.breadth;
    box.height = this->height + b.height;
    return box;
}
```

Example

```
int main() {

Box Box1;
Box Box2;
Box Box3;

double volume = 0.0;

Box1.setLength(6.0);
Box1.setBreadth(7.0);
Box1.setHeight(5.0);


Box2.setLength(12.0);
Box2.setBreadth(13.0);
Box2.setHeight(10.0);

   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   Box3 = Box1 + Box2;
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;
   return 0;
}
```

Output

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

```
}
void Box:: setBreadth( double bre ) {
   breadth = bre;
}
void Box:: setHeight( double hei ) {
   height = hei;
}

Box Box:: operator+(const Box& b) {
   Box box;
   box.length = length + b.length;
   box.breadth = breadth + b.breadth;
   box.height = height + b.height;
  //cout<<"length = "<<box.length<<endl;
   return box;
}

int main() {

    Box Box1;
    Box Box2;
    Box Box3;

    double volume = 0.0;
```

Another Example

complex.h

```
#ifndef COMPLEX_H_INCLUDED
#define COMPLEX_H_INCLUDED

class Complex {
   public:
          Complex(double, double);
          Complex operator+(Complex);
          void Print();
   private:
          double Real;
          double Imaginary;
};

#endif
```

complex.cpp

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

Complex::Complex() {
   Real = 0; Imaginary = 0;
}

Complex::Complex(double r, double i) {
   Real = r;
   Imaginary = i;
}

Complex Complex::operator+(Complex a) {
   Complex t;
   t.Real = Real + a.Real;
   t.Imaginary = Imaginary + a.Imaginary;
   return t;
}

void Complex::Print() {
   cout << Real << endl;
   cout << Imaginary << endl;
}
```

Exceptions

Exception handling in C++ consist of three

keywords: `try`, `throw` and `catch`:

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.
The `throw` keyword throws an exception when a problem is detected, which lets us create a custom error.
The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.

The `try` and `catch` keywords come in pairs:

```
try
{ // code to be tried ,
   throw exception; }
 catch ( type  exception)
 { // code to be executed in case of exception }
```

All exceptions thrown by components of the C++ Standard library throw exceptions derived from this exception class.
These are:

exception	description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when it fails in a dynamic cast
bad_exception	thrown by certain dynamic exception specifiers
bad_typeid	thrown by typeid
bad_function_call	thrown by empty function objects
bad_weak_ptr	thrown by shared_ptr when passed a bad weak_ptr

```
try {
  int age = 15;
  if (age >= 18) {
    cout << "Access granted - you are old enough.";
  } else {
    throw (age);
  }
}
catch (int myNum) {
  cout << "Access denied - You must be at least 18 years
old.\n";
  cout << "Age is: " << myNum;
}
```

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

int main() {
  try {
    int age = 15;
    if (age >= 18) {
      cout << "Access granted - you are old enough.";
    } else {
      throw (age);
    }
  }
  catch (int myNum) {
    cout << "Access denied - You must be at least 18 years old.\n";
    cout << "Age is: " << myNum;
  }
  return 0;
}
```

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

int main()
{
	try {
		int numerator = 10;
		int denominator = 0;
		int res;

		if (denominator == 0) {
			throw runtime_error(
				"Division by zero not allowed!");
		}

		res = numerator / denominator;
		cout << "Result after division: " << res << endl;
	}

	catch (const exception& e) {
		cout << "Exception " << e.what() << endl;
	}

	return 0;
}
```

(good news)

***Quiz* on 17/02/2025**

***Mid* on 19/03/2025**