DESTRUCTORS

A destructor of class "MyList" is a function called "~MyList" that does not have a return type (not even "void"). It is **automatically called** when using the keyword **"delete"**.

```
class MyList {
   public:
      MyList(int n);  // A constructor of class X
      ~MyList(); // A destructor of class X
  private:
      int* list;
};

MyList::MyList( int n ) { list = new int[n]; } // Defining the constructor
MyList::~MyList() { delete [] list; } /* Defining the destructor, which clears the memory
                                         of "list" */
int main(){
     MyList*  x = new MyList(100);   // here, the constructor of "MyList" is called
     delete x; // The destructor of "MyList" is called, which clears the memory of x.list
     return EXIT_SUCCESS;
}
```

The destructor is **also automatically called** when an object of type MyList goes **out of scope**, i.e., if:

- The object was defined in a function, and the **function ended**.
- The object was a local variable defined in a block, and the **block ended**.
- The object was defined in a program, and the **program ended**.

ACCESSING NON-PUBLIC MEMBERS

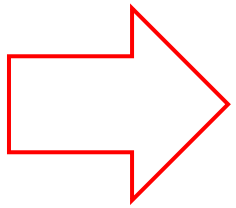
```
class Class_A {
  public:
    Class_A(int n){ y=n;}//constructor


  private:
    int y;
};

void function_1() {
  //define x as an object of Class_A
  Class_A x(100);
  x.y = 50; // ERROR! “y” is private!
}

Class Class_B {
  public:
    void method_1();
}
void Class_B::method_1(){
  //define x as an object of Class_A
  Class_A x(100);
  x.y = 50; // ERROR! “y” is private!
}
```

Solution: declare them as **friends**



```
class Class_A {
  public:
    Class_A(int n) { y=n;}//constructor
    friend void function_1();
    friend class Class_B;
  private:
    int y;
};

void function_1() {
  //define x as an object of Class_A
  Class_A x(100);
  x.y = 50; // No error!
}

Class Class_B {
  public:
    void method_1();
}
void Class_B::method_1(){
  //define x as an object of Class_A
  Class_A x(100);
  x.y = 50; // No error!
}
```

INITIALIZER LIST

An **initialize list** is a different way of initializing members when writing the constructor

```
class Class1 {
  public:
    Class1(int a, double b, float c);
  private:
    int    member1;
    int    member2;
    double member3;
};
// Below, we define the constructor
Class1::Class1( int a, int b, double c){
    member1 = a;
    member2 = b;
    member3 = c;
    // some code goes here ...
}
```

These two do the same thing!

```
class Class1 {
  public:
    Class1(int a, double b, float c);
  private:
    int    member1;
    int    member2;
    double member3;
};
// Below, we define the constructor
Class1::Class1( int a, int b, double c)
: member1( a ), member2( b ), member3( c )
{
    // some code goes here ...
}
```

CHAPTER 2:
OBJECT-ORIENTED DESIGN

INHERITANCE

INHERITANCE

- It is when a class “inherits” attributes and methods from another class.
  - Parent / Superclass / base class - a class being inherited from
  - Child / Subclass/derived class - a class that inherits from another class
- A subclass cannot access the **private** members of its superclass, but it can access members that are **public** or members that are **protected**.

Example:

```
//Parent class
class Employee {
    public:
       // public members go here
       int getSalary() { return salary; }
    protected:
       // protected members go here
       int salary;
};
```

```
// The class “Programmer” is a child class
that inherits the members of “Employee”
class Programmer: public Employee {
  public:
       string programming_language;
       void setSalary(int s) { salary = s; }
       /* see how method can access salary
          because it is a protected member! */
};
```

OVERRIDING INHERITED FUNCTIONS

- You can override an inherited method by redefining it in the child class.

Example:

```
class Animal {// parent class
  public:
    void animalSound() {
        cout << "The animal makes a sound \n";
    }
};

class Dog : public Animal { // child class
  public:
    void animalSound() {
        cout << "The dog says: woof woof \n";
    }
};
```

Why do we need inheritance?

Suppose you want to define two classes:

```
class Faculty {
    private:
      string name;
      string idNum;
      string rank;
    public:
      Faculty( string n, string i, string r);
      void print();
      string getName();
      string getRank();
    }
```

some parts are repeated!!

```
class Student {
    private:
      string name;
      string idNum;
      string major;
    public:
      Student( string n, string i, string r);
      void print();
      string getName();
      void changeMajor(string m);
    }
```

```
Faculty::Faculty( string n, string i, string r) {
 : name(n),  idNum(i); rank(r);  {
}
void Faculty::print( ) {
  cout << name;
  cout << idNum;
  cout << rank;
}
string Faculty::getName( ) { return name; }

void Faculty::getRank( ) { return rank; }
```

```
Student::Student( string n, string i, string m)
   : name(n),  idNum(i); major(m);  {
}
void Student::print( ) {
  cout << name;
  cout << idNum;
  cout << major;
}
string Student::getName( ) { return name; }

void Student::changeMajor( string m ) {
   major = m;
}
```

Benefits of using inheritance

Create a **base** class containing the common members, and two **derived** classes

```
class Person {
  private:
    string name;
    string idNum;
  public:
    Person(string n, string i);
    void print();
    string getName();
  }
```

```
Person::Person( string n, string i)
   :  name(n),  idNum(i) {
}
void Person::print( ) {
   cout << name;
   cout << idNum;
}
string Person::getName( ) { return name; }
```

A faculty is a person

A student is a person

```
class Faculty : public Person{
  private:
    string rank;
  public:
    Faculty(string n, string i, string r);
    void print();
    void getRank();
  }
```

```
class Student : public Person{
  private:
    string major;
  public:
    Student(string n, string i, string m);
    void print();
    void changeMajor(string m);
  }
```

We called the constructor of `Person` **in the initializer list** of the constructor of `Student`

```
Student::Student(string n, string i, string m)
   :  Person(n,i),  major(m)  {
}
void Student::print( ) {
   Person::print();
   cout << major;
}
void Student::changeMajor( string m ) {
   major = m ;
}
```

TEMPLATES

FUNCTION TEMPLATE

How do we avoid repeat the same code over and over for different types?

```
void min_1(int x, int y){
    if (x<y) cout << x;
    else cout << y;
}
```

```
void min_2(double x, double y){
     if (x<y) cout << x;
     else cout << y;
}
```

```
void min_3(float x, float y){
     if (x<y) cout << x;
     else cout << y;
}
```

Solution: Instead of writing the above three versions, just write a single template:

```
template <typename T>
void min_fun( T x, T y ){
     if (x<y) cout << x;
     else cout << y;    }
```

```
int i = 5; int j = 7; min_fun(i,j);

double x = 3.67; double y = 9.7881; min_fun(x,y);

float  a = 1.5;  float b = 2.5; min_fun(a,b);
```

You can have more than one typename: template <typename **T**, typename **V**, **…**>

T and **V** can be used as if they were datatypes.

CLASS TEMPLATE

- Similar to a function template declaration, it starts with the keyword template followed by **< >** for the types

Syntax:

```
template <typename T, typename V, ...>
class className
{    private:
       V var; /* You can treat V like
                 any regular type */
     public:
       T functionName(V arg);
       /* You can treat T and V like
          any regular type */
};
```

```
//How to create a class template object?
className< datatype(s) > objectName;
```

Example:

```
className< int, float > objectName;
```

CLASS TEMPLATE

- An example of a class template…

```
template <typename T>
class CalcDisplay {
   private:
      T num1, num2;
   public:
      CalcDisplay (T n1, T n2){
         num1 = n1;
         num2 = n2;
      }
      void displayNumbers(){
         cout << num1 << " and " << num2;
      }
};
```

```
int main() {

   CalcDisplay<int> x(2, 1);
   cout << "The numbers are:";
   x.displayNumbers();

   CalcDisplay<double> y(2.498, 1.2775);
   cout << "The double Numbers:";
   y.displayNumbers();

   return EXIT_SUCCESS;
}
```

EXCEPTIONS

EXCEPTIONS

- When executing C++ code, different **errors** can occur, e.g., **coding errors** made by the programmer, or **user errors** such as providing a wrong input
- To specify what happens when certain errors are made, use "**exceptions**". For this, you need 3 keywords: `try`, `throw`, and `catch`.
- Here is a syntax example:

```
try {
    // some code can go here...

    if( error_happens )
       throw( exception_object );

    // some code can go here...
}
catch ( exception_object ) {
    // write here what you want to happen when error occurs
}
```

- We need to **create an object** that will be "thrown" when there is an error!

EXCEPTIONS

- We can throw an object of `MathException` if we detect a math-related error

```
try {

    // some code can go here...

    if( z == 0 ){
        throw( MathException( “Can’t divide by zero!” ) ); /* Notice how, inside the “throw”
              statement, we created an object by calling the constructor of “MathException” */
    }else
         x = y/z;

    // some code can go here...

}
catch ( MathException& obj ) {
    cout << obj.getError(); /* This line will print the error message stored in the
                               object called “obj” */
}
```

EXCEPTIONS

- First, we need to **define a class** for each type of error. Then, we will be able to throw an object of that class if an error occurs!
- Suppose we define a generic class called `MathException` that we will use whenever there is a math-related error. A possible definition could be:

```
class MathException {

private:
    string errMsg; // The error message that is stored in the object

public:
    /* A constructor. Remember, ": errMsg(err)" is an initializer list
       that sets: errMsg = err; */
    MathException(const string& err) : errMsg(err) { }

    // A method used to retrieve the error message stored in the object
    string getError() { return errMsg; }
};
```

EXCEPTIONS

- **A function may specify the exceptions it throws**. Here is an example:

```
void calculator() throw(ZeroDivide, NegativeRoot) {
    // function body goes here ...
}
```

- You will see many examples in the book where a class has a function (i.e., a method) that specifies the exception it throws, **without worrying about the details of what will happen when that exception is caught**
  - This is mainly meant to emphasize that **you should look out for, and "catch", any errors** that may happen

EXCEPTIONS: EXAMPLE

Below is part of the definition of a class that stores the 10 highest scores in a game:

```
class Scores {
public:
   void remove(int i) throw(IndexOutOfBounds); // this method removes the ith name
   // ...
};
```

Later on, when defining the method "`remove`", we write:

```
void Scores::remove(int i) throw(IndexOutOfBounds) {
    if ((i < 0)||(i >= 10))
       throw (IndexOutOfBounds("Invalid index"));

    // write here the code that removes the ith name
}
```

This denotes the following: When implementing a method to remove the element at index `i`, add code that ensures `i` falls within the boundaries of the list!