## Initializing Class Objects: CONSTRUCTORS

The class designer can guarantee the initialization of every object by providing a special member function called the **constructor**.

The constructor is invoked **automatically** each time an object of that class is created (instantiated).

These functions assign initial values to the data members, allocate memory for members, open files, establish a connection to a remote computer, etc.

The constructor can take parameters as needed, but it cannot have a return value (**even not void**).

The constructor has the **same name** as the class itself.

There are different types of constructors.

For example, a constructor that defaults all its arguments or requires no arguments, i.e., a constructor that can be invoked with no arguments, is called a *default constructor*.

In this section, we will discuss different kinds of constructors.

**Note:** If no initial value is specified for a member variable of a fundamental type (`double`, `int`, `bool` …) or pointer type (`int*`, …), it will contain a random arbitrary junk value.





## Default Constructor:

A constructor that defaults all its arguments or requires no arguments, i.e., a constructor that can be invoked with no arguments.

```
class Point{          // Declaration/Definition of the Point Class
 public:
   Point();           // Declaration of the default constructor
   :
 private:
   int m_x, m_y;             // Attributes are not initialized
};

// Default Constructor
Point::Point()
{
   m_x = 0;          // Assigns zeros to coordinates (just an example)
   m_y = 0;
}
// -------- Main Program -------------
int main()                                          See Example e04_1.cpp
{
 Point point1, point2{};    // Default construct is called 2 times
 Point *pointPtr; // pointPtr is not an object, the constructor is NOT called
 pointPtr = new Point; //Object is created, the default constructor is called
```





## Default Constructor (contd):

If you do not define any constructor for a class, then **the compiler generates** a *default constructor* for you.

It is called a *default default constructor* because it is a default constructor that is generated by default.

The purpose of a default default constructor is to allow an object to be created and all member variables to be set to their initial (default) values.

Remember the examples about the `Point` class from the previous chapter, i.e., `e03_x.cpp`.

We declared the `Point` class without any constructor and created objects from it.

Actually, the compiler generated a default constructor with an empty body, and the variables get the initial values supplied by the *class creator*.

```
class Point{          // Declaration/Definition of the Point Class
 public:
   Point() {};        // Default constructor with an empty body
   :
 private:
   int m_x{}, m_y{};  // Attributes are initialized
};
```





## Constructors with Parameters:

There are two possible sources of initial values for objects.

1. The class creator can provide the initial values in the definition of the class or in the default constructor.
2. Users of a class (client programmers) may (and sometimes must) provide the initial values in a constructor with parameters.

If the class creator defines a constructor with parameters, users of the class (client programmers) must supply the required arguments to create objects.

**Example:**

```
class Point{          // Declaration/Definition Point Class
 public:
   Point(int, int);   // Constructor with two parameters
   :
 private:
   int m_x, m_y;      // Attributes are not initialized
};
```

This declaration shows that the users of the Point class have to supply two integer arguments while defining objects of that class.

For example, `Point point1 {10, 20};` or `Point point1 (10, 20);`

Otherwise, a compiler error is generated: `Point point1; // Error!`





**Example:**

The Point class has a constructor with two parameters to initialize the coordinates.

```
// Constructor with two parameters to initialize x and y coordinates
Point::Point(int firstX, int firstY)
{
  if (firstX >= MIN_x) m_x = firstX;    // Accepts only valid values
  else m_x = MIN_x;
  if (firstY >= MIN_y) m_y = firstY;    // Accepts only valid values
  else m_y = MIN_y;
}                                              See Example e04_2.cpp
```

In our example `e04_2.cpp`, the class creator has already provided initial values for the attributes by the definition `int m_x{MIN_x}, m_y{MIN_y};`

However, now, the client programmer can also provide other initial values under the control of the constructor function.

When the class creator provides a constructor with parameters, the compiler does not provide a default default constructor.

Therefore, the client programmer cannot create objects without providing parameters anymore.

Remember: The class creator sets the rules, and class users must follow them.





## Multiple Constructors

The rules of function overloading are also valid for constructors. So, a class may have more than one constructor with different types of input parameters.

**Example:**

```
class Point{          // Declaration/Definition Point Class
 public:
   Point();           // Default Constructor
   Point(int, int);   // Constructor with two parameters
   :
 private:
   int m_x, m_y;      // Attributes are not initialized
};
```

Now, the client programmer can define objects in different ways:

```
Point point1;                // Default constructor is called
Point point2 { 10, 20 };     // Constructor with parameters is called
```

The following statement causes a compiler error because the class does not include a constructor with only one parameter.

```
Point point3 {30}; //ERROR! There isn't a constructor with one parameter
```

### Defining a default constructor using the `default` keyword

Remember: If the class creator adds a constructor, the compiler no longer implicitly defines a default default constructor.

If you still want your objects to be constructible without providing any parameters, like "`Point point1;`" you should add a default constructor to the class.

If the initial values of the member variables are already provided in the class definition, the body of the default constructor may be empty.

Instead of defining a default constructor with an empty function body to increase the readability of your code,

```
class Point{
 public:
   Point() = default;   // Default Constructor with an empty body
   Point(int, int);     // Constructor with two parameters
    :
 private:
   int m_x{}, m_y{};    // Attributes are already initialized to zero
};
...
Point point1 {10, 20};   // m_x = 10, m_y = 20
Point point2;            // m_x = 0, m_y = 0, (initial values)
```





### Default Arguments for Constructor Parameters

Like other functions, the parameters of constructors may also have default values.

```
class Point{
  public:
   Point (int = 0, int = 0); //Default values must be in the declaration
     :
};

Point::Point(int firstX, int firstY)
{
  if (firstX >= MIN_x) m_x = firstX;      // Accepts only valid values
  else m_x = MIN_x;
  if (firstY >= MIN_y) m_y = firstY;      // Accepts only valid values
  else m_y = MIN_y;
}
```

Now, client of the class can create objects as follows:

```
Point point1 {15, 75};       // m_x = 15, m_y = 75
Point point2 {100};          // m_x = 100, m_y = 0
```

This function also counts as a ***default constructor***.

```
Point point3;               // m_x = 0, m_y = 0
```





### Initializing Arrays of Objects

When an array of objects is created, the default constructor of the class, if any, is invoked for each element (object) of the array once.

```
Point  pointArray[10];      // Default constructor is called 10 times
```

To invoke a constructor with arguments, a **list of initial values** should be used.

```
// Constructor    (can be called with zero, one, ore two arguments)
Point (int = 0,  int  = 0)
```

We do not provide the number of elements — List of initial values

```
Point pointArray[] = { 10 , 20 , {30,40} }; //Array with three objects
```

or to make the program more readable

```
Point array[]= { Point {10}, Point {20}, Point {30,40} };
```

Three objects of type `Point` have been created and the constructor has been invoked three times with different arguments.

| Objects: | Arguments: |
|---|---|
| array[0] | firstX = 10 , firstY = 0 |
| array[1] | firstX = 20 , firstY = 0 |
| array[2] | firstX = 30 , firstY = 40 |





### Initializing Arrays of Objects (contd)

If the class has a default constructor, the programmer may define an array of objects as follows:

```
Point pointArray[5]= { 10 , 20 , {30,40} }; // An array with 5 elements
```

Here, an array with five elements has been defined, but the list of initial values contains only three values.

For the last two elements, the default constructor is called.

To call the default constructor for an object which is not at the end of the array:

```
  Point array[5] = { 10, 20, {}, {30,40} }; //An array with 5 elements
```

or

```
  Point array[5] = { 10, 20, Point{}, {30,40} };
```

or

```
  Point array[5] = { 10, 20, Point(), {30,40} };
```

Here, for objects `array[2]` and `array[4]`, the default constructor is invoked.

The following statement causes a compiler error:

```
  Point array[5]= { 10 , 20 , , {30,40} };   // ERROR! Not readable
```





### Member Initializer List

It is possible to initialize data members of an object using a ***member initializer list*** rather than assignment statements in the constructor body.

The *member initializer list* is the only way to assign initial values to constant members.

**Example:** Point class with constant data members

In our `Point` class, we have two constant data members, i.e.,

```
const int MIN_x{};
const int MIN_y{};
```

Assume that the class creator wants to allow the client programmers to initialize these constant values in a constructor.

However, you cannot assign a value to a constant in the constructor's body.

```
// Constructor to initialize all members of a Point object
Point::Point(int firstMINX, int firstMINY, int firstX, int firstY)
{
    MIN_x = firstMINX;      // ERROR! MIN_x is not modifiable
    MIN_y = firstMINY;      // ERROR! MIN_y is not modifiable
    :
}
```





### Member Initializer List (contd)

**Example:** Point class with constant data members (contd)

The constructor uses a *member initializer list* to initialize constant data members.

```
// Constructor to initialize all members of a Point object
Point::Point(int firstMINX, int firstMINY, int firstX, int firstY)
                                : MIN_x {firstMINX}, MIN_y {firstMINY}
{
  ...     // Code to initialize x and y coordinates
}
```

After the initialization in the constructor, the constant members cannot be modified later.

```
Point point1 {50, 60, 100, 200};
// MIN_x = 50, MIN_y = 60
// m_x = 100, m_y = 200
Point point2 {-10, 0, -15, 20};
// MIN_x = -10, MIN_y = 0
// m_x = -10, m_y = 20          The given firstX (-15) is not accepted
```

We have two `Point` objects with different constant minimum values.

**Member Initializer List (contd)**

You can also use a member initializer list to initialize non-constant members.

However, you cannot check their values in this way.

```
// Constructor to initialize all members of a Point object
Point::Point(int firstMINX, int firstMINY, int firstX, int firstY)
        : MIN_x{firstMINX}, MIN_y{firstMINY}, m_x{firstX}, m_y{firstY}
{
  ...     // You may check and modify x and y coordinates
}
```

- When you initialize a member variable **using an assignment statement** in the body of the constructor, first, the member variable is created in memory, and then the assignment is carried out as a separate operation.
- When you **use an initializer list**, the initial value is used to initialize the member variable as it is created.

  This can be a more efficient process, particularly if the member variable is an object of a class.

  We will cover these cases in the following chapters.





**DESTRUCTORS**

- The *destructor* is a special method of a class **called automatically**
  1. When each of the objects goes out of scope or
  2. A dynamic object is deleted from memory using the delete operator.
- It is executed to handle any cleanup operations that may be necessary.
- You only need to define a class destructor when something needs to be done when an object is destroyed.

  For example,
  Closing a file or a network connection,
  Releasing the memory if memory is allocated by a constructor using new.
- A destructor is characterized as having the same name as the class but with a tilde '**~**' preceding the class name.
- A destructor has no return type and receives no parameters.
- A class may have only one destructor.
- The destructor for a class is always called automatically when an object is destroyed.

  The circumstances where you need to call a destructor explicitly are so rare that you can ignore the possibility.





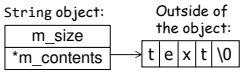
**Example:** A user-defined String class

Actually, the standard library of C++ contains a **std::string** class. Programmers do not need to write their String classes.

We write this class only to show some concepts.

A string is a sequence (array) of characters.

It terminates with a null character '\0'.




```
class String{
public:
   String(const char *);     // Constructor
   void print();             // An ordinary member function
   ~String();                // Destructor
private:
   size_t m_size;            // Length (number of chars) of the string
   char *m_contents;         // Contents of the string
};
```

Since the String class contains a pointer to strings (array characters), the constructor must allocate storage for characters, and the destructor must release memory when the object is destroyed.





**Example:** A user-defined String class (contd)

```
// Constructor
// Allocates memory and copies the input character array to contents
String::String(const char *in_data)
{
  size = std::strlen(in_data);
  m_contents = new char[m_size +1];// Memory allocation, +1 for null character
  if (m_contents)                  // If memory is allocated, copy the contents
     for (unsigned index = 0; index < m_size + 1; index++)
        m_contents[index] = inData[index];   // copy the contents
}

// Destructor
// Memory is released
String::~String()
{
   delete[] m_contents;
}
```

```
int main()          // Test program
{
  String string1{"string 1"}; // Constructor
  String string2{"string 2"}; // Constructor
  string1.print();
  string2.print();
  return 0;     // Destructor is called twice
}
```

See Example e04_3.cpp





**Constant Objects and const Member Functions**

The programmer may use the keyword **const** to specify an object is constant (not modifiable).

Any attempt to modify (to change) the attributes of a const object directly or indirectly (by calling a function) causes a **compiler** error.

Any member variable of a const object is itself a const variable and thus immutable.

For example:

```
const Point fixedPoint {10, 20};
```

The object fixedPoint has the coordinates (20,30), and this point cannot be moved to another location.

**const Member Functions:**

C++ compilers totally disallow any member function calls for const objects.

The programmer may declare some functions as **const**, which do not modify any member data (attributes) of the object.

Only const methods can operate on const objects.





**const Member Functions** (contd):

**Example:**

We specify methods that do not modify an object's attributes as **const**.

```
class Point {
public:
  Point(int, int);          // Constructor to initialize x and y coordinates
  double distanceFromZero() const;   // The distance of a point from (0,0)
  void print() const;       // const method prints coordinates on the screen
  // Getters are constant
  int getX() const { return m_x; }    // Accessor for x coordinate
  int getY() const { return m_y; }    // Accessor for y coordinate
  // Setters are not constant
  void setX(int);
  void setY(int);
  bool move(int, int);             // A non-constant method to move points
};

// Constant method calculates and returns the distance of a point from (0,0)
double Point::distanceFromZero() const {
  return sqrt(m_x * m_x + m_y * m_y);          // distance from (0,0)
}
```

## Constant Objects and const Member Functions (contd)

Example (contd):

```
int main()
{
 const Point fixedPoint {10, 20};              // Constant object
 std::print("Distance from Zero= {}", fixedPoint.distanceFromZero()); //OK
 fixedPoint.print();                    // OK. Print the constant point
 fixedPoint.move(15, 25);   // ERROR! fixedPoint is constant, cannot move
 Point nonFixedPoint{ 30, 40 };       // Non-constant object
 nonFixedPoint.move(100, 200);         // OK, non-constant object can move
}
```

A const method can invoke only other const methods because a const method is not allowed to alter an object's state either directly or indirectly, that is, by invoking some non-const method.

**Specify all member functions that do not change the object's attributes as const to avoid possible errors and to allow users of the class to define constant objects.**





## Constant Objects and const Member Functions (contd)

**The mutable Keyword:**

Sometimes, we want to allow particular class members to be modifiable even for a const object.

We can do this by specifying such attributes as **mutable**.

**Example:**

We want to count how many times a point object is printed.

We will add a mutable variable, m_printCount, to the Point class.

```
class Point {
public:
 Point(int, int);  // Constructor with two parameters to initialize x and y
 bool move(int, int);  // A non-constant function to move points
 void print() const;   // A constant function to print
  :
private:
  :
 int m_x{ MIN_x }, m_y{ MIN_y }; // x and y coordinates are initialized
 mutable unsigned int m_printCount{};  // Mutable data member
};
```





## The mutable Keyword (contd):

**Example** (contd):

```
// This method prints the coordinates on the screen
void Point::print() const
{
  std::printl("X= {} , Y= {}", m_x, m_y);
  std::printl("Print count= {}", ++m_printCount);
}
```

Although the print method is specified as const, it can modify the mutable attribute printCount.

```
int main()
{
  const Point fixedPoint{ 10, 20 };   // Constant object
  fixedPoint.print();                  // m_printCount is incremented
  :
}
```

See Example e04_4.cpp





## The Copy Constructor

- Sometimes, we want to create a new object as a copy (with the same data) of an existing object.
- Copy constructor is a special type of constructor used to copy an object's contents to a new object during the construction of that new object.

**Example:** Creating an object as a copy of another object

```
Point point1 {0, 0, 10, 20};// Define the point1 object using the constructor
Point point2 {point1}; // point2 is a copy of point1. Copy constructor runs
```

Newly created object — Existing object

The input argument is the existing object that will be copied into the new object.

Example: Defining the copy constructor

```
class Point {
public:
  Point(int, int, int, int); // Constructor to initialize limits, x, and y
  Point(const Point&);       // Copy Constructor
   :
```

The input parameter of a copy constructor is a *reference to a const object* of the same type (source object).





## The Copy Constructor (contd)

**Example** (contd):

```
// Copy Constructor copies limits and the coordinates
Point::Point(const Point& originalPoint)
            : MIN_x{originalPoint.MIN_x}, MIN_y{originalPoint.MIN_y},
              m_x{originalPoint.m_x}, m_y{originalPoint.m_y}
{}
```

It does not copy the m_printCount

The copy constructor may delegate to another constructor using the initializer list.

```
// Copy Constructor delegates to another constructor
Point::Point(const Point& originalPoint)
       : Point { originalPoint.MIN_x, originalPoint.MIN_y,
                 originalPoint.m_x, originalPoint.m_y }
{}
```

See Example e04_5.cpp

```
int main(){
  Point point2 {point1};   // Call copy constructor for point2
                           // point2 is created as a copy of point1

// Other (older) notations to create copies of objects
Point point3 = point2;   // Call copy constructor for point3, NOT assignment
Point point4(point1);    // Call copy constructor for point4
```





## The Copy Constructor (contd)

**The compiler-generated default copy constructor:**

Usually, we do not need to write a copy constructor because the compiler already generates one by default.

If the compiler generates it, it will simply copy the contents of the original into the new object as a byte by byte copy.

So all members are copied.

In most cases, this copy is sufficient.

**Example:**

What happens if we do not supply a copy constructor for our Point class?

See Example e04_6.cpp

Since the compiler-generated copy constructors copy all members, the print count is also copied. Therefore, the counter does not start from zero for the copies of the original object.

In this case, we must write our own copy constructor.

However, if the compiler-generated copy constructor is sufficient, never write a copy constructor for your classes.

**The compiler-generated copy constructor** (contd):

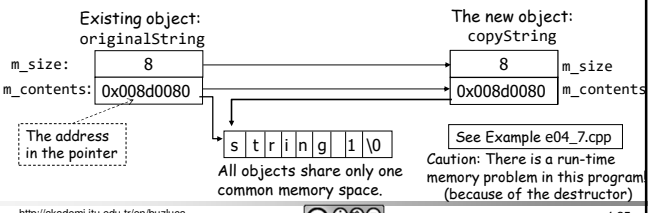
If a class has a member variable of a pointer type, the compiler-generated copy constructor will copy the address in the pointer to the other one.

As a result, the pointers in different objects will be pointing to the same memory location(shared memory).

**Example:**

The copy constructor, generated by the compiler for the user-defined `String` class (e04_3.cpp), will perform the following copy operation:

```
String originalString {"string 1"};
String copyString {originalString}; // Copy constructor
```




See Example e04_7.cpp

Caution: There is a run-time memory problem in this program! (because of the destructor)





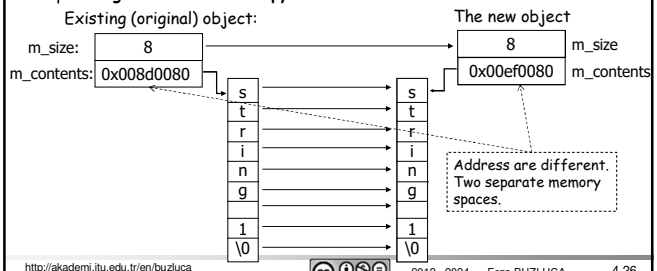
**Example (contd):**

The copy constructor, generated by the compiler, cannot allocate memory or copy the memory locations to which member pointers point.

Since both pointers point to the same memory space, the delete operation in the destructor causes a runtime error.

The programmer must write its copy constructor to allocate memory for the pointer and perform copy operations between two memory spaces.

Example: **Programmer-written copy constructor**








**Example:** The **programmer-written** copy constructor of the String class

```
class String{
public:
  String(const char*);        // Constructor
  String(const String&);      // Copy Constructor
    :
};
// Copy Constructor
// Allocates memory and copies the contents of the existing object to
// the newly constructed object
String::String(const String& originalString)
{
  m_size = originalString.m_size;
  m_contents = new char[m_size + 1];   // memory allocation
  if (m_contents)             // If memory is allocated, copy the contents
     for (std::size_t index = 0; index < m_size + 1; index++)
         m_contents[index] = originalString.m_contents[index];
}
int main() {
  String originalString{"string 1"};
  String copyString{originalString}; // Programmer-defined copy constructor
  String otherString = originalString; // Another notation, NOT assignment
  ...
```

See Example e04_8.cpp





**Deleting the Copy Constructor:**

If the class creator does not want that the objects of this class can be copied, they can prevent the compiler from generating a copy constructor.

They can instruct the compiler not to generate a copy constructor by adding `= delete;` next to the signature of the copy constructor in the class declaration.

Example: **Deleting the copy constructor of the user-defined String class**

```
class String{
public:
String(const char*);              // Constructor
String(const String&) = delete;   // Copy Constructor is deleted
 :
};
```

Another solution is to make the signature of the copy constructor private.

Example: **Private copy constructor**

```
class String{
public:
 String(const char*); // Constructor
private:
 String(const String&);  // Copy Constructor is private
 :
```

```
// Compiler Error!
String copyString{originalString};
```

See Example e04_9.cpp





**Passing Objects to Functions as Arguments**

- Objects should be passed or returned **by reference** unless compelling reasons exist to pass or return them by value.
- Passing or returning by value can be especially inefficient for objects.
- Recall that the object passed or returned by value must be *copied* into the stack.

  The data may be large, thus wasting storage, and the copying itself takes time.
- If the class contains a copy constructor, the compiler uses this function to copy the object into the stack.

**Example:**

- We have a class called `GraphicTools`, which contains tools that can be used to perform operations on `Point` objects.

  For example, the method `maxDistanceFromZero` compares two `Point` objects and returns the object that has the larger distance from zero (`0,0`).
- We will consider two different cases in terms of passing and returning objects.

  Case 1: call by value, return by value

  Case 2: call by reference (to constant), return by reference (to constant)





**Passing Objects to Functions as Arguments** (contd)

**Case 1 (call by value, return by value. Inefficient!):**

In this program, the method `maxDistanceFromZero`

1. Gets two `Point` objects using the call-by-value technique.
2. Finds the object that has the larger distance from zero
3. Returns the object using the call-by-value technique.

See Example e04_10.cpp

Examine the output:

The constructor is called two times for `point1` and `point2`.

The default constructor is called once for `point3`.

These are objects defined by the programmer in the main function.

Moreover, the copy constructor is called three times.

Two times for input parameters and once for the return value.

In total, six `Point` objects have been created.

Three of them are created because of the call-by-value technique.

As expected, the destructor has been called six times.

**Passing Objects to Functions as Arguments** (contd)

**Case 2 (call by reference, return by reference. Efficient!):**

In this program, the method `maxDistanceFromZero`

1. Gets two `Point` objects using the call-by-reference technique.
2. Finds the object that has the larger distance from zero
3. Returns the object using the call-by-reference technique.

See Example e04_11a.cpp

Examine the output:

The constructor is called two times for `point1` and `point2`.

The default constructor is called once for `point3`.

These are objects defined by the programmer in the main function.

In total, three `Point` objects have been created.

**No other constructor is called.**

Additional objects are not created.

As expected, the destructor has been called only three times.

There are other options for receiving the returned object. See Example e04_11b.cpp





**`this` Pointer**

Each object has its own data space in the memory system of a computer. When an object is defined, memory is allocated only for its data members.

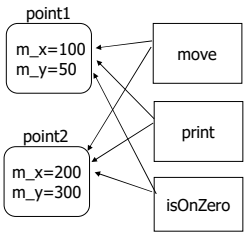
The code of member functions is created only once. Each object of the same class uses **the same function code**.

**Example:**

```
class Point {
 public:
   Point(int, int);
   void move(int, int);
   void print();
   bool isOnZero();
 private:
   int m_x{}, m_y{};
};

int main(){
  Point point1{100, 50};
  Point point2{200, 300};
  :
```

`Point` objects in memory:




How does C++ ensure that the functions reference the proper object?





**`this` Pointer** (contd)

The C++ compiler defines an object pointer with the name **`this`**.

When a member function is called, this hidden pointer contains the address of the object for which the function is invoked.

So member functions can access the data members using the pointer **`this`**.

The compiler compiles our `Point` methods as follows:

```
// A function to move the points
void Point::move(int new_x, int new_y)
{
  this->m_x = new_x;
  this->m_y = new_y;
}

// is the point on the zero point(0,0)
bool Point::isOnZero()
{
  return (this->m_x == 0) && (this->m_y == 0);
}
```

You could write the function explicitly using the pointer `this` if you wanted, but it is not necessary.





**`this` Pointer** (contd)

When you call a method for a particular `Point` object, this pointer will contain the address of that object.

This means that when the member variable `m_x` is accessed in the `move` method during execution, it is actually referring to `this->m_x`, which is the fully specified reference to the object member being used.

For example, when we call the `move` method for `point1`:

```
point1.move(50,100);
point2.move(0,0);
```

The compiler considers this code as follows:

```
this = &point1;     // the address of object point1 is assigned to this
move(50,100);       // and the method move is called.
this = &point2;     // the address of object point2 is assigned to this
move(0,0);          // and the same move method is called.
```

This is not a valid code. It is only given to explain how the compiler uses the `this` pointer to access member data.





**Returning `this`** (as a pointer)

**Example:** We add a new method to the `Point` class: `maxDistanceFromZero` that compares a point object with a second object and returns a pointer to the object with a larger distance from zero (0,0).

For example, the following piece of code calls the method for the `point1` object and compares it with the object `point2` regarding the distance from (0,0).

It returns a pointer to one of these objects depending on the comparison result.

```
const Point* pointPtr;          // pointer to Point objects
pointPtr = point1.maxDistanceFromZero(point2); // method runs for point1
pointPtr->print();           // pointPtr points either to point1 or point2
point1.maxDistanceFromZero(point2)->print();   // Chain of calls

// Definition of the method the returns a pointer to Point objects
const Point* Point::maxDistanceFromZero(const Point& in_point) const
{
  if (distanceFromZero() > in_point.distanceFromZero())
     return this;  // the pointer to the object for which the method is called
  else
     return &in_point;
}
```

See Example e04_12.cpp





**Returning `this`** (as a refference)

Remember: Instead of pointers, passing and returning references increase the readability of the code.

The `maxDistanceFromZero` method could return a reference to the `Point` object as follows:

```
const Point& Point::maxDistanceFromZero(const Point& in_point) const
{
  if (distanceFromZero() > in_point.distanceFromZero())
     return *this;
  else
     return in_point;
}

const Point point3;          // point3 is an object
point3 = point1.maxDistanceFromZero(point2);
point3.print();

// You can chain method calls (do not overuse, understandability!)
double distance = point1.maxDistanceFromZero(point2).distanceFromZero();
point1.maxDistanceFromZero(point2).print();
```

See Example e04_13.cpp

## Static Class Members

**Static data members:**

Each object of a class has its own copy of the ordinary data members.

For example, `point1` and `point2` objects of the `Point` class have different `m_x` and `m_y` variables in memory.

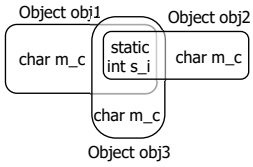
When you declare a member variable of a class as **`static`**, the static member variable is defined only once, regardless of how many class objects have been defined.

Each `static` member variable is accessible in any object of the class and is shared among all existing objects in memory.

The `static` members exist even if no class objects have been created.

```
class StaticExample{
  :
   char m_c;
   static int s_i;
};

int main()
{
  StaticExample obj1, obj2, obj3;
   :
```








**Static data members** (contd):

In certain cases, all class objects should share only one copy of a particular data member.

For example, we can use a static counter to count how many objects of a class exist.

Constructors will increment this counter, and the destructor will decrement it.

**Example:**

```
class Point {
 public:
   :
 private:
   int m_x{}, m_y{};
   static inline unsigned int s_point_count{};  // A static counter
};
```

See Example e04_14.cpp

Inline variables have been supported since C++17.

Before C++17, we should declare the counter as follows:

```
    static unsigned int s_point_count;              // A static counter
```

Then, we should define and initialize the static member outside the class with a definition such as this:

```
    unsigned int Point::s_point_count {};    // It is still valid today
```





**Static constant data members:**

- Constant data members are usually declared `static`.
- If you define **constants as `static` members**, there is only one single instance of that constant that is shared between all objects.
- If you define a **constant as a non-static member** variable, an exact copy of this constant will be made for every single object, which is usually pointless.

**Example:**

In our `Point` class, we have constant data members to represent the limits of the coordinates `MIN_x` and `MIN_y`.

If each object should have its own limits specific to itself, then these constants should not be declared `static`.

However, if the class has limits that are valid for all class objects, then these constants should be declared `static`.





**Static constant data members** (contd):

**Example:**

```
class Point {       // Declaration of the Point Class with low-limits
public:
  // Static constants
  // Lower Limits of x and y coordinates for all objects
  static inline const int MIN_x{};   // Same for all objects of Point
  static inline const int MIN_y{};   // Same for all objects of Point
   :
```

See Example e04_15.cpp

The keywords `static`, `inline`, and `const` may appear in any order you like.

Unlike regular member variables, there is no harm in making constants public because class users can read but cannot modify them.

It is common to define public constants containing boundary values.

Class users can read these values outside of the classes directly using the class name and the scope resolution operator `::`.

**Examples:**

Class name::static variable/constant

```
if (input_x < Point::MIN_x) ...    // makes a decision using the limit

// Define an object using the limits
Point point1 {Point::MIN_x, Point::MIN_y}; // m_x = MIN_x, m_y = MIN_y
```





## Static Class Members (contd)

**Static methods (member functions):**

A `public` **`static`** method can be called even if no class objects have been created.

It can also be invoked from outside the class.

A `static` method can operate on static member variables, regardless of whether any objects of the class have been defined.

For example, a `static` method can be used to initialize `static` data members before any objects have been created.

A `static` method is independent of any individual class object but can be invoked by any class object if necessary.

For example, we can write a `static` `initPointCounter` method for the `Point` class to initialize the counter.

```
class Point {
 public:
 static void initPointCount(unsigned int);
 static unsigned int getPointCount();
   :
};
```

See Example e04_16.cpp

A simple example:

See Example e04_17.cpp

Class name::static method

```
Point::initPointCount(100);          // Set counter to 100
if (Point::getPointCount > 500){... // makes a decision using the counter
```





## The Unified Modeling Language - UML

UML is a visual language for specifying, constructing, and documenting the artifacts (models) of software.

UML is not a method to design systems; it is used to **visualize** the analysis and the design models.

Benefits:

- It makes it easier to understand and document software systems.
- It supports teamwork. Since UML diagrams are more understandable than the program code, team members (e.g., project leader, software architect, and developers) can discuss the design.
- Some tests and quality measurements can be conducted on UML diagrams, and design flaws can be detected before coding.
- There are tools that can create the code from UML diagrams and draw UML diagrams for a given code.

## The Unified Modeling Language – UML (contd)

UML has evolved from the work of Grady Booch, James Rumbaugh, and Ivar Jacobson (called three amigos) for object-oriented design.

It has been extended as a general-purpose, developmental modeling language to cover a wider variety of software engineering projects.

The Object Management Group (OMG) adopted UML as a standard in 1997 and has managed it ever since.

https://www.uml.org/

In 2005, UML was also published by the International Organization for Standardization (ISO) as an approved ISO standard.

ISO/IEC 19505-1:2012
Information technology —Object Management Group Unified Modeling Language (OMG UML)

The latest version of UML is 2.5.1, published in December 2017.

You can get the specifications for the current version from the website of OMG.

https://www.omg.org/spec/UML/





## The Unified Modeling Language – UML (contd)

There are different kinds of UML diagrams, which are used in various phases of a software development process.

In the latest version of UML, there are 14 diagram types.

There are two main categories: **structure diagrams** and **behavior (interaction)** diagrams.

- **Structure diagrams** show the static structure of the objects in a system.

  In this course, we will draw class diagrams (a type of structure diagram) to present the (compile-time) structure of our programs.

  The **class diagram** displays the attributes and operations of each class and the relationships between them.

- **Behavior diagrams** illustrate the elements of a system that are dependent on time. We can see how the components of the system relate to each other dynamically during its execution (runtime).

  In this course, we will draw **sequence diagrams** and **communication diagrams** to present how objects in our program interact in runtime.

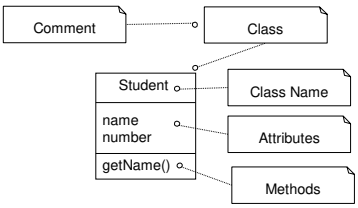
As we cover various concepts in the course, we will see how they are represented using UML diagrams.



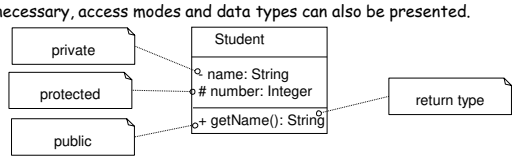


## Class Diagrams

A class diagram shows the structure of the classes and the relationships between them.




If necessary, access modes and data types can also be presented.








## Class Diagrams (contd)

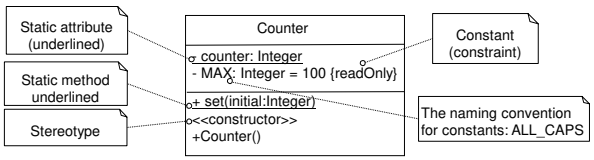
**Comments:** Comments in UML are placed in dog-eared rectangles.

You can use comments to put anything you want in a diagram. You can use comments to add application- and program-specific details.

**Stereotypes:** A stereotype is a way of extending the UML in a uniform manner and remaining within the standard.

You indicate a stereotype using <<stereotype name>>

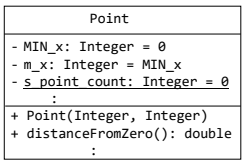
**Constraints:** A constraint in the UML is a text string in curly braces ({usually language-specific}). The UML defines a language (Object Constraint Language – OCL) that you can use to write constraints.








**Example:** The `Point` Class




Since the primary purpose of UML is to demonstrate design, the details of data and methods are not crucial.

Sometimes, we only show attributes without their types and the methods without their parameters.

In the following chapters, we will use UML diagrams to represent static and dynamic relations between classes/objects.