Moving From C to C++

C Versus C++

Many of the basic constructs of the two languages are almost identical:

- Rules to declare variables and constants
- Simple data types and aggregated data type such as built-in arrays
- Most of the operators
- Control Structures:
  - Selection structures (if … else, switch statement, etc.)
  - Repitiion stuctures (for loop, while loop, do loop)
  - Jump statements (break, continue, goto)
- Function declaration, and definition.
- struct data type are almost the same:
  - Except that in C++, for the declaration of a struct object there is no need for keyword struct. Assume a structure called Point is defined:

    struct Point { double x, y; };
  - The following declaration of opject p, with typedef is valid:

    Point p;
- Both languages need the definition of a global main funcition as an starting point of execution of a program.

C Versus C++

- However, there are many essential and conceptual differences between the two languages:
  - C++ supports reference data type, where C doesn’t
  - C++ supports different style of type casting
  - C++ supports different style of initialization of variables
  - C++ uses different style of standard input/output.
  - C++ uses different style of file I/O
  - C++ is an object-oriented language and supports many features of this type of programming. For example:
    - **class** data type
    - Many pre-defined class libraries. For example class string and class vector.
  - C++ supports more advanced feature :
    - Inheritance
    - Overloading operators
    - Templates
    - Etc.

Introduction to Standard I/O in C++

Introduction to C++ standard I/O

- First you need to Include `iostream` header file to be able to use two standard input/out objects called `cin` and `cout`.
- Here is a sample of using of cout and cin

```
#include <iostream>
using namespace std:
 int main() {
       int a , b ;
       cout << "Enter two integer number:” << endl;
       cin >> a >> b;
       cout << a << " + " << b << " is " << a + b << ".\n";
        return 0;
 }
```

End of line manipulator

Using extraction >> and insertion operators

- Use **cin** **extraction** operator, **>>**, to read one or more data.
- Use **cout** and **insertion** operator **<<**, to display on the screen.

```
int x, y, z;
cout << “Please enter three integer numbers: “;
cin  >> x  >> y  >>  z;
```

- This code prompts the user for reading three integer.
- You could also write:

```
cin >> x;
cin >> y;
cin >> z;
```

Standard I/O

- Displaying a combination of different data types, and string constants:

```
int x = 5;
char ch = ‘B’;
char course [] = “ENSF 619”;
cout << “Your character is “ << ch
     << “\nYour course is: “ << course
     << “\n Your number is: << x << endl;
```

- This code prints:

```
Your character is B
 Your course is ENSF 619
 Your number is 5
```

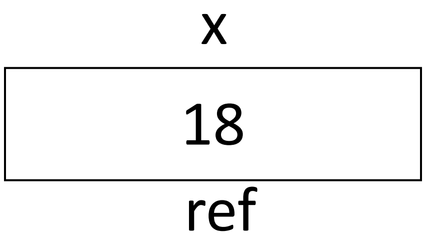
- `cin` assumes a white space as an input terminator. Three characters are considered as white space in C and C++:
  - spacebar
  - tab
  - enter

C++ Reference Type

C++ Reference type

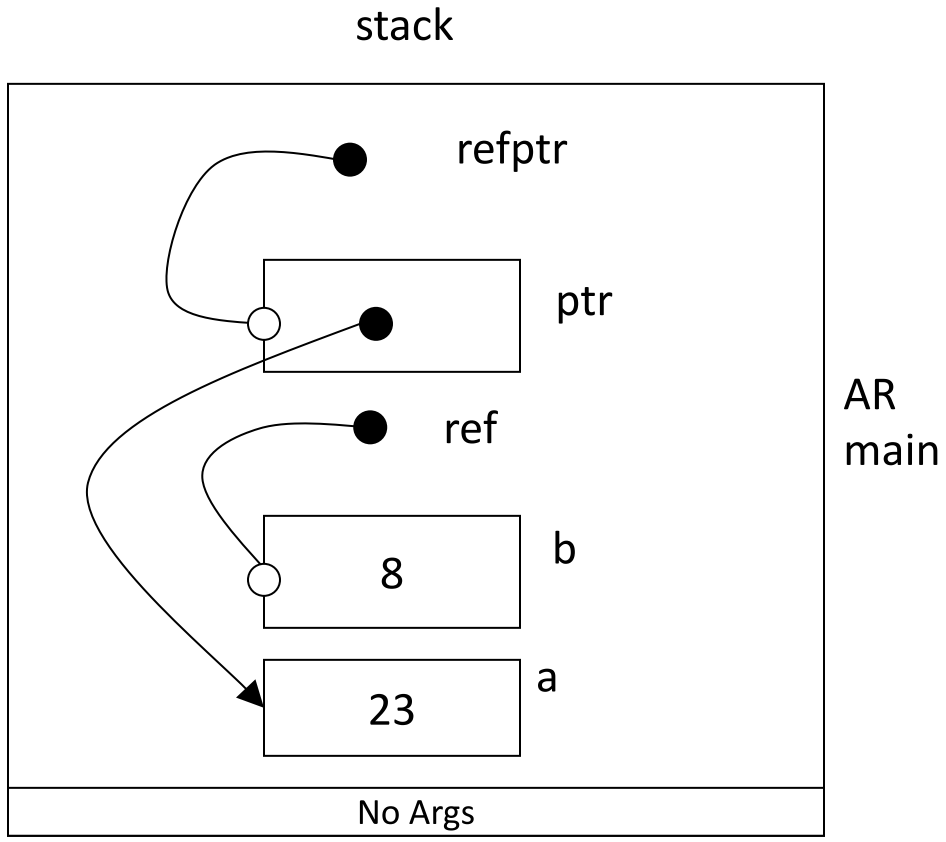
- C++ supports a data type known as a reference-type.
- For the variables of this type it does NOT allocate any memory space.
- Reference type is an alias for a variable name. In the following the example you can use ***ref*** exactly as you can use ***x***:

```
int x = 4;
int& ref = x;
ref = 18;
cout << x;    //  displays 18
```



- In ENCM 339, we use a special notation to show a reference in an AR diagram (a line with two circles at its both ends. One solid circle on the side of declaration of reference and one open-circle on the side that it refers to. Here is an example:

```
 int main()
{
   int a , b;
   int& ref = b;
   int * ptr = &a;
    int* & refptr=ptr;
   *ptr = 4;
     ref = 8;
      *refptr = 23
   // point one
   …
}
```

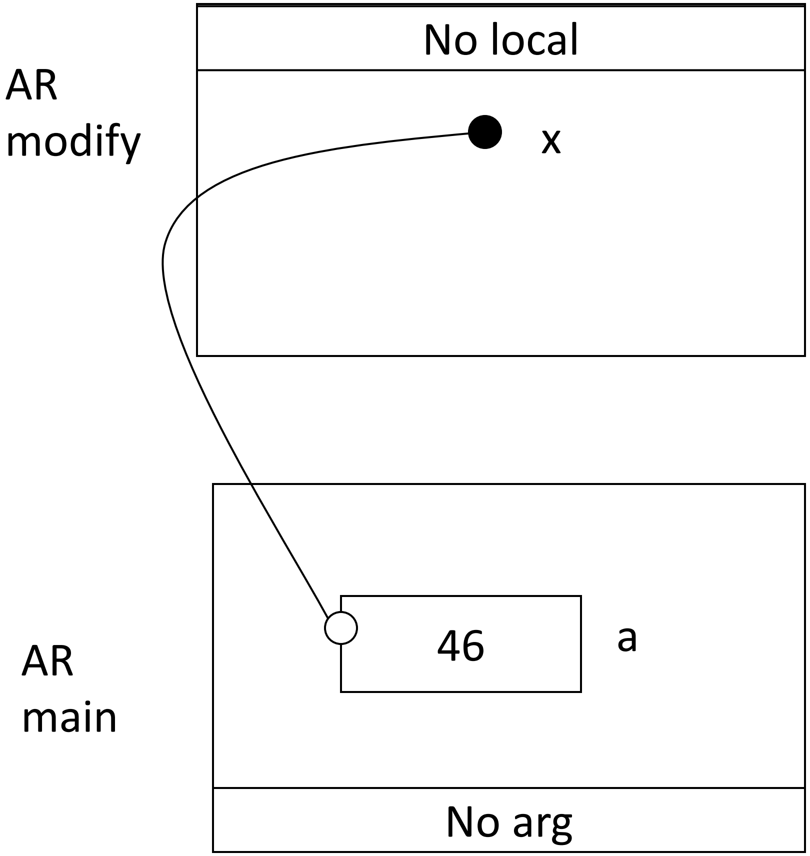


Reference as a Function Argument

- A variable can be passed to a function, by reference:

```
void modify(int & x) {
   x++;
  // point one
}
```

```
int main() {
       int a = 45;
       modify (a);
        …
       }
```



x is a reference and a is called a referent of x

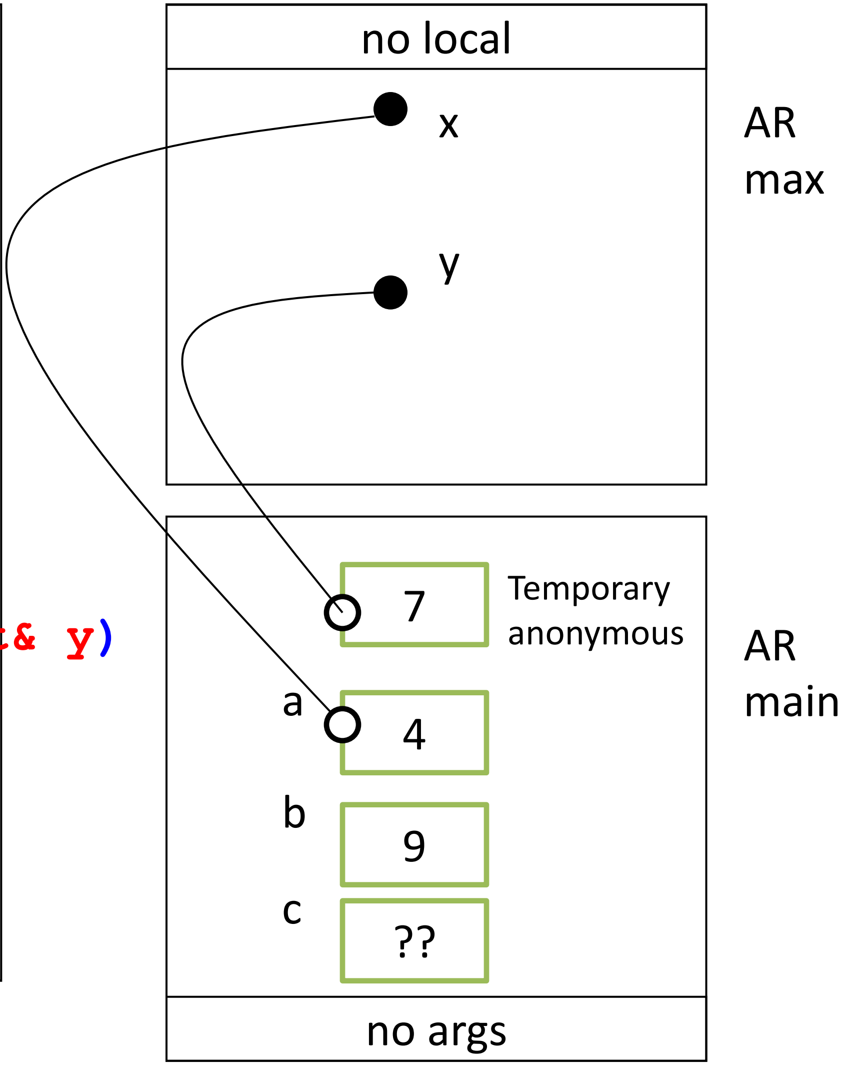
Moving from C to C++

- Like other types of arguments an argument of type reference can be also a const.
- If a numeric constant or expression is passed to a function by reference, a *temporary anonymous* memory space will be created. This space lives long to make the function call work. See the following example

```
int main()
{
  int a = 4;
  int b = 9;
  int c = max (a, b-2);
   …
}


int max(const int& x, const int& y)
{
   return (x>y)? x : y;
}
```

Must be const



Functions that Return a Reference

- Similar to any legal built-in, predefined, or user-defined data type, a function in C++ can also return a reference. For example the following format for the definition of a function is allowed in C++:

```
int&  func (int& x)
{
    …
    …
    return x;
}
```

- However this format of functions are more common for class member functions that serve as a getter or setter. We will discuss this subject in more detail, in future.

Another example

- Let’s revisit the `swap()` function to see how it works with references instead of pointers:

  - Function prototype

```
void swap( int &, int & );
// switches values the two arguments
```

The & means a and b are reference variables.

And that the corresponding arguments are passed by reference)

  - Function definition

```
void swap( int &a, int &b )
{
   int temp = a;                // Line A
   a = b;                 // Line B
   b = temp;                 // Line C
}
```

Explicit Type Conversation in C++

- You can convert any C++ type to another type explicitly, by using the type-cast operator, as illustrated below:

```
int x = 4, y = 7;
double ratio = static_cast <double> x / y;
```

- The above example, first converts x to a double type then stores the result of a real division into variable ratio. Without the type cast operation, the result would have been zero.
- The other possible C++ style for type-casting is:

```
int x = 4, y = 7;
double ratio = double( x ) / y;
```

C++ Style Initialization of Variable

- Function-call-style initialization:

  C++ provides an additional style of initializing variables that looks like a function call:

```
#include <iostream>
int main()
{
    int i(5);
    std::cout << 2 / double(i) << std::endl;
    return 0;
}
```

A Quick Review of C++ Math Library

Quick Look at the Built-In Functions:

- Like C, C++ provides a reach set of library function and library objects.
- To implement some advanced equations, there are a number of mathematical ***functions*** available in the cmath library
  - To use these function type "**`#include <cmath>`**" at the top of your program
- Some of the library math functions are:

Function	Mathematical Equivalent	Result (assume x = 2.4, y = -2.0)
sqrt(x)	$\sqrt{x}$	1.54919…
pow(x,y)	x^y	0.17361…
fabs(y)	$\lvert y \rvert$	2.0
floor(x)	$\lfloor x \rfloor$ (round down)	2.0
ceil(x)	$\lceil x \rceil$ (round up)	3.0
exp(x)	e^x	11.02317…
log(x)	$\ln(x)$	0.87546…
$\log_{10}(x)$	Log10(x)	

Object-Oriented Programming

Principles of Object-Oriented Programming

- The concept of Object-Oriented Programming (OOP) is based on the following principles:
  - Abstraction:
    - Data abstraction is the simplest of principles to understand.
    - It allows us to create a software model of a real-world object.
    - It highlights the common properties (information) and behavior (functionality) of objects in terms of theirs interfaces, instead of their implementation details.
  - Encapsulation
    - Encapsulation is the hiding of data implementation by restricting access to data only by using getter and setter methods.
  - Polymorphism
  - Inheritance

C++ class Type

Class and object definition

- A class is the definition of a set of objects that share a common structure and a common behavior.
  - A class is a "type"
  - In other words, a class is an abstraction, a way of classifying similar objects.
  - Example of Class Interface (Definition):

```
#define SIZE 3
class Point{
private:
    double x;
    double y;
    char label[SIZE];
public:
    void set_x(double value);
    void set_y(double vlaue);
    void set_label(const char* s);
    void display();
};
```

- Every class has the following characteristics:
  - It has a name:
  - It can hold data in the form of variables, arrays, strings or other objects
  - It can provide function to access the data and implement other tasks.

Class Definition – Information Hiding

- The terms private and public define the level of access to the data members and functions
- Private members can only be accessed by other members (i.e functions) of the same Class
  - This means that private members cannot be directly accessed using the dot operator
  - This is known as Data or Information Hiding
- Public members can be accessed from outside the class using the dot operator (the same as for struct data type)
  - Because of this, public members form the public nterface of the class.
  - Public members provide controlled access to the private members
- By default, all class members are private, compared with struct data types where all members are public by default.
- It is always a good idea to **make your data members private and member functions public**. Why?

What is an object?

- An object is an instance of a class, a concrete entity that exists in time and space.
- Example:

```
 #include “Point.h”
int main()
{
     Point x, y;
     ...
     return 0;
}
```

Class Implementation

- Now that we know how to define a class, we need to learn how to implement one. The implementation basically involves writing the definition for the member functions. The general format for the implementation of member functions is:
- SYNTAX:

```
return_type class_name::function_name(parameter_list)
{
    // function implementation
}
```

- The **scope resolution operator** **(::)** it is used to associate a function to its corresponding class.
    - Several classes may have member functions with the same name.

C++ Class Implementation

Consider the following partial implementation for the class Point in previous slides:

```
void Point::set_x(double value) {
    x = value;
}

void Point::set_y(double value) {
    y = value;
}

void Point::set_label(const char* s){
    strcpy(label, s);
}

void Point::display(){
    cout << "Point label is: " << label;
    cout << "x coordinate is: " << x;
    cout << "y coordianate is: " << y << endl;
}
```

Note that we did not include a dot operator when accessing the member variable `x or y` within the member function.

Getter Functions

- In our previous example we had function to set the values of x and y coordinates of point but if we want to retrieve the values of x and y in our main function, we need to have a set of getter functions:

```
#define SIZE 3
class Point{
private:
    double x;
    double y;
    char label[SIZE];
public:
    void set_x(double value);
    void set_y(double value);
    double get_x()const;
    double get_y()const;
    char* get_label()const;
    void display()const;
};
```

- Since getter function don't need to change the values of x and why, we declare that as read-only functions by adding the const keyword to the end of function declaration.

C++ Class Implementation

```
// File: point.cpp
#include <iostream>
#include <cstring>
using namespace std;
#include “point.h”

void Point::set_x(double value) {
    x = value;
}


void Point::set_y(double value) {
    y = value;
}

void Point::set_label(const char* s)
{
    strcpy(label, s);
}
```

```
double Point::get_x()const
{
    return x;
}

double Point::get_y()const{
    return y;
}

const char* Point::get_lable()const{
    return label;
}

void Point::display()const
{
  cout << "Point label is:" << label;
  cout << "x coordinate is: " << x;
  cout << "y coordianate is: "
       << y << endl;
}
```

Using Class Object

- Objects or instances of a class can be declared similar to objects of struct or other built-in data types:

```
int main()
{
    Point a;
    a.set_x(20);
    a.set_y(30);
    a.set_label("A");
    a.display();
    cout << "label: " << a.get_label() << endl;
    cout << "x coordinate: " << a.get_x() << endl;
    cout << "y coordinate: " << a.get_y() << endl;
    return 0;
}
```

Pointers to Objects

- A pointer in C++ can point to any addressable memory location, including user-defined data types (structures, unions, and classes).
- The principles and notations for pointers are similar for structures, unions and classes.
- Consider the following statements:

```
Point c;
 Point *ptr;
 ptr = &c;
cout  <<  ptr ->get_x();
cout  <<  (*ptr).get_x();
```

- Same as other data types an object can also be passed to a function by value, by address or by reference (by address or by reference is normally preferred). See the following example.

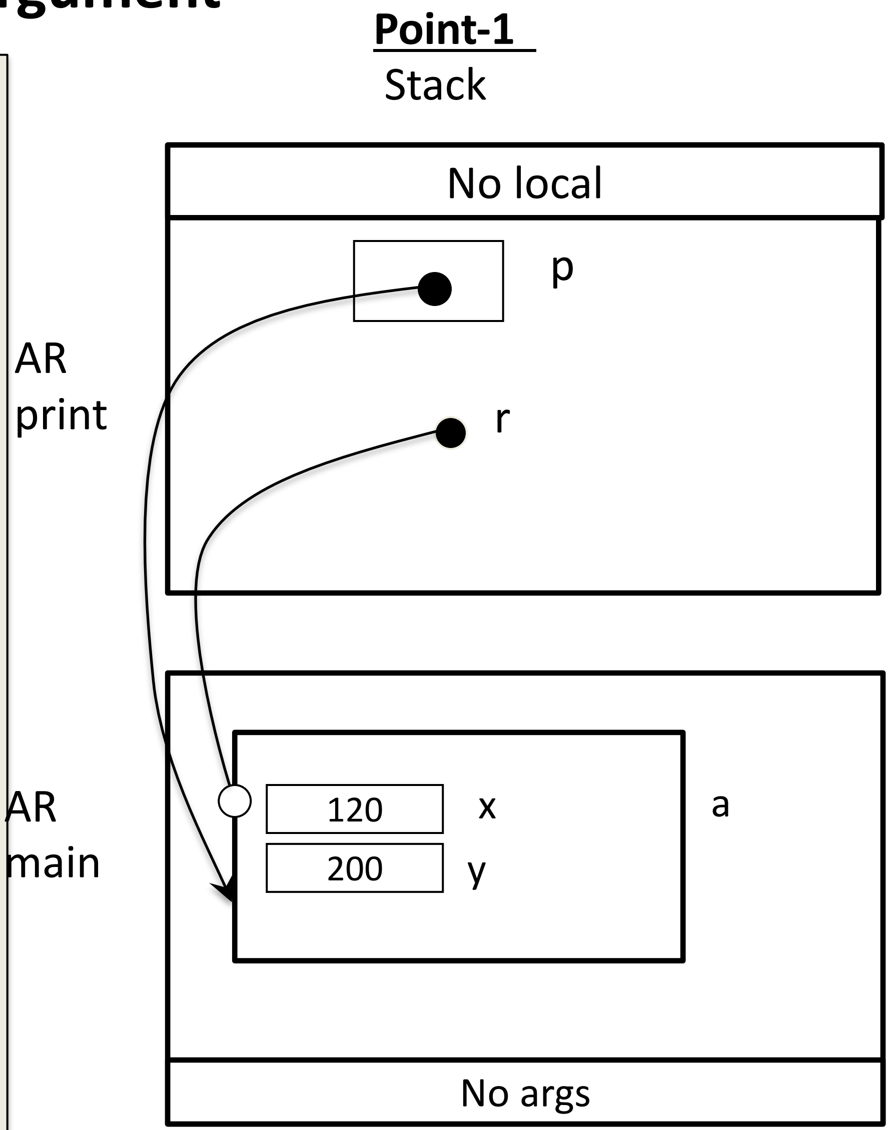
Object Data Types as a Function Argument

```
void print ( const Point *p, const Point & r)
{
   // point one// point one
   cout << p-> get_x();

   cout << r.get_x();
 }



 int main()
 {
      Point a;
      a.set_x(120);
      a.set_y(200);
      print(&a, a );
   }
```



Constructor Concepts

- Consider the slightly modified Point class definition below, that defines also a member called “constructor”. Constructor initializes and object.

```
class Point{
private:
    double x;
    double y;
    char label[SIZE];
public:
    Point(double a, double b);   //
    void set_x(double value);
    void set_y(double vlaue);
    void set_label(const char* s);
    double get_x()const;
    double get_y()const;
    const char* get_label()const;
    void display();
};
```

- The constructor is implemented as follows:

```
Point::Point(double a, double b){
     x = a;
     y = b;
}
```

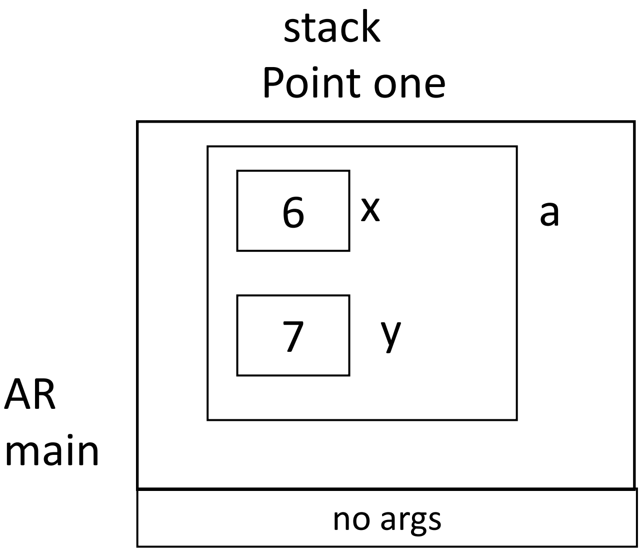
- constructor cannot be called using the dot operator. It will be called automatically when an object is declared.
- Constructor doesn’t have a return type.
- Constructor can be overloaded.

Constructor Concepts

- Now consider the following code segment:

```
int main()
{
 Point a(6, 7);
// point one
a.get_x() << endl;
a.get_y() << endl;
return 0;
}
```

Constructor Called



Constructor Concepts

- Any constructor that takes no arguments is called a ***default constructor***.

- If you do not declare **any** constructor, the compiler will generate one for you of the form:

```
class_name::class_name()
{
     /* Some code: Normally initialization construction  */
 }
```

- Default constructors are used when you declare an object without any arguments:

```
class_name object_name;
```

- If you have defined at least one non-default constructor, the compiler will not generate a default constructor for you. Therefore, you must write a default constructor yourself, if needed.

Constructor Concepts

- Like any other function, constructors can be overloaded. To illustrate this, consider a different version of the Point class:

```
class Point
{
 private:
    double x, y;
  public:
      Point();                    // default constructor
      Point(double a, double b);  // non-default

};
```

```
Point::Point()
{
    x = 0;
    y = 0;
}
```

```
Point::Point(double a, double b){
     x = a;
     y = b;
}
```

```
void main()
{
    Point a;           //default constructor
    Point b(6, 7);    //other constructor

    // use other member functions as necessary
}
```

Constructor Concepts

- When initializing member variables, there are two possible approaches. The first is as follows:

```
class_name::class_name(value_1, value_2)
 {
    member1 = value_1;
    member2 = value_2;
 }
```

- The initialization values can either be hard-coded or passed as arguments to the constructor. The second method of initializing values is to use the following syntax:

```
class_name::class_name(value_1, value_2): member1(value_1),
                                          member2(value_2)
{

}
```

Constructor Concepts

- The latter approach is generally preferred.
- We could therefore have implemented the two constructors of the last Point class as:

```
Point::Point(): x(0), y(0)
{


}

Point::Point(double a, double b):x(a), y(b)
{


}
```