# 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, definition, overloading, etc.
- 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 that are not covered in ENCM 339. But will be covered in the higher level courses:
    - 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 simple of using of cout.

End of line manipulator

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

## Introduction to C++ standard I/O

- In the development of a large software system to avoid name collision among types and identifiers, C++ provides namespaces that keeps related names under one umbrella.
- One of the commonly namespace that belongs to C++ standard I/O library is called `std`.
- To avoid typing the std every time, you can use the following method:

```
#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;
}
```

- The third option which might be even a better way can be:

```
#include <iostream>
using std::cout;
using std::cin;

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

- This way you specifically indicate the object that you need from the name space and minimizes possible name-conflicts.

## Standard I/O:

- Use **cin** and **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 [] = "ENCM 339";
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 ENCM 339
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

# 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 these functions are:

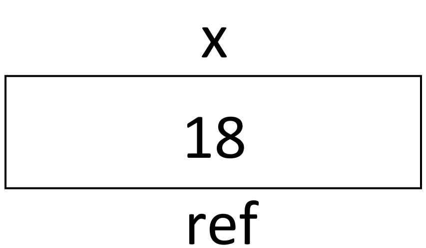
| Function | Mathematical Equivalent | Result (assume x = 2.4, y = -2.0) |
|---|---|---|
| sqrt(x) | $\sqrt{x}$ | 1.54919… |
| pow(x,y) | $x^y$ | 0.17361… |
| fabs(y) | $\lvert y \rvert$ | 2.0 |
| floor(x) | $\lfloor x \rfloor$ (round down) | 2.0 |
| ceil(x) | $\lceil x \rceil$ (round up) | 3.0 |
| exp(x)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

# C++ Reference Type

## C++ Reference type

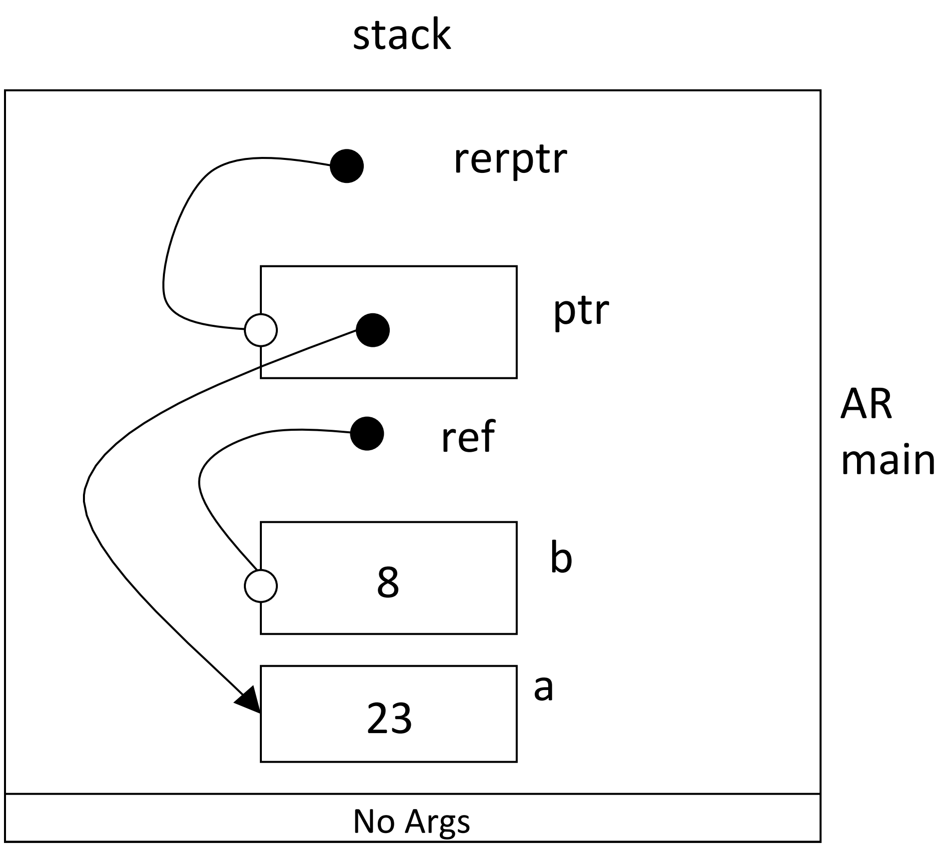
- C++ supports a data type known as a reference-type.
- For the variables of this type it does NOT allocate any memory space.
- Reference type is an alias for a variable name. In the following the example you can use ***ref*** exactly as you can use ***x***:

```
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
   ...
}
```

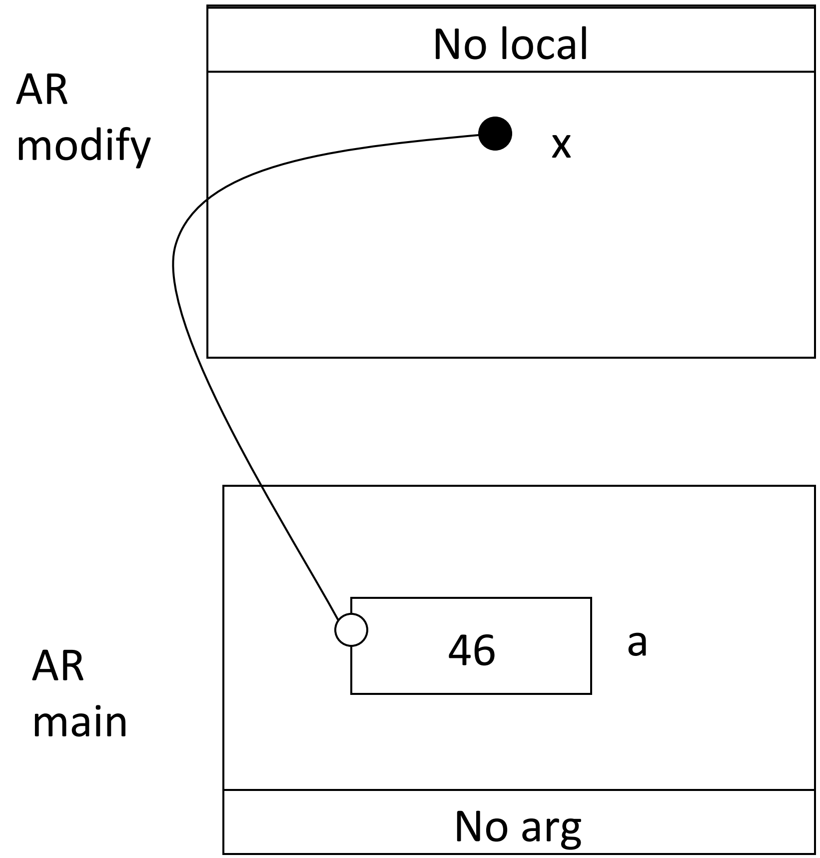


## Reference as a Function Argument

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

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

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

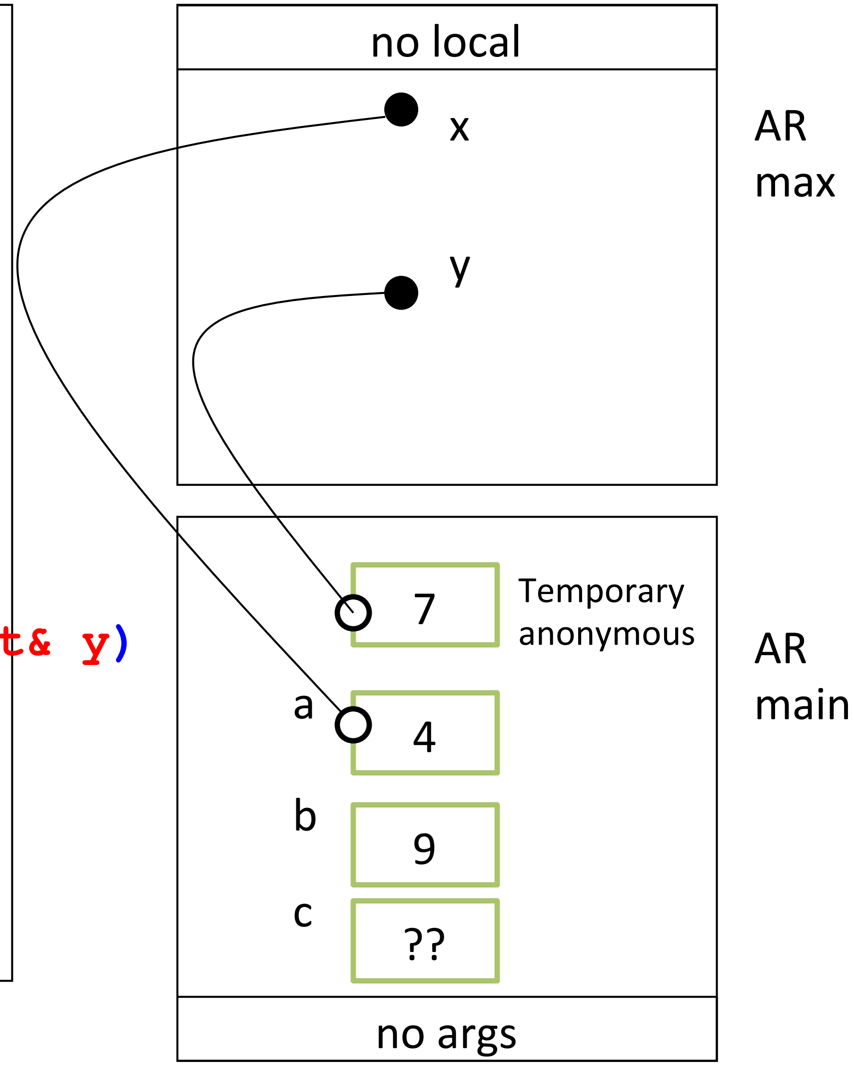



x is a reference and a is called a referent of x

- 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;
}
```

# 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 – this is an advanced topic, which is out the scope of this course (Will be discussed in ENSF 409).
  - Inheritance – this is also an advanced topic, which is out the scope of this course (Will be discussed in ENSF 409).

# 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 in fact a "type"
  - In other words, a class is an abstraction, a way of classifying similar objects.
  - Example of Class Interface (Definition):

```
class Person
{
 private:
  char name[20];
  int age;
 public:
  void showName();

  ...
};
```

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

```
Person x, y, z;
```

## Class and Object Fundamentals

- Once a Class such as ‘Person’ is defined, it can be used if:
  - The definition of the Person is included as a header file or it is defined in the .cpp file before any prototype or function that uses this definition.
- 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.

## Designing Classes in C++

Designing classes includes both the definition and the implementation of the class .

- Class Interface
  - Declare all data members
    - The data members, are usually declared <u>private</u>.
  - Declare prototype of member functions
    - The member functions, are usually declared <u>public</u>.
  - Declaration constructor(s), destructor, etc.
- Class Implementation
  - Definition/Implementation of of class member functions
  - Definition/Implementation of constructor(s), destructor, etc.

In C++, the implementation is normally put into a .cpp file.

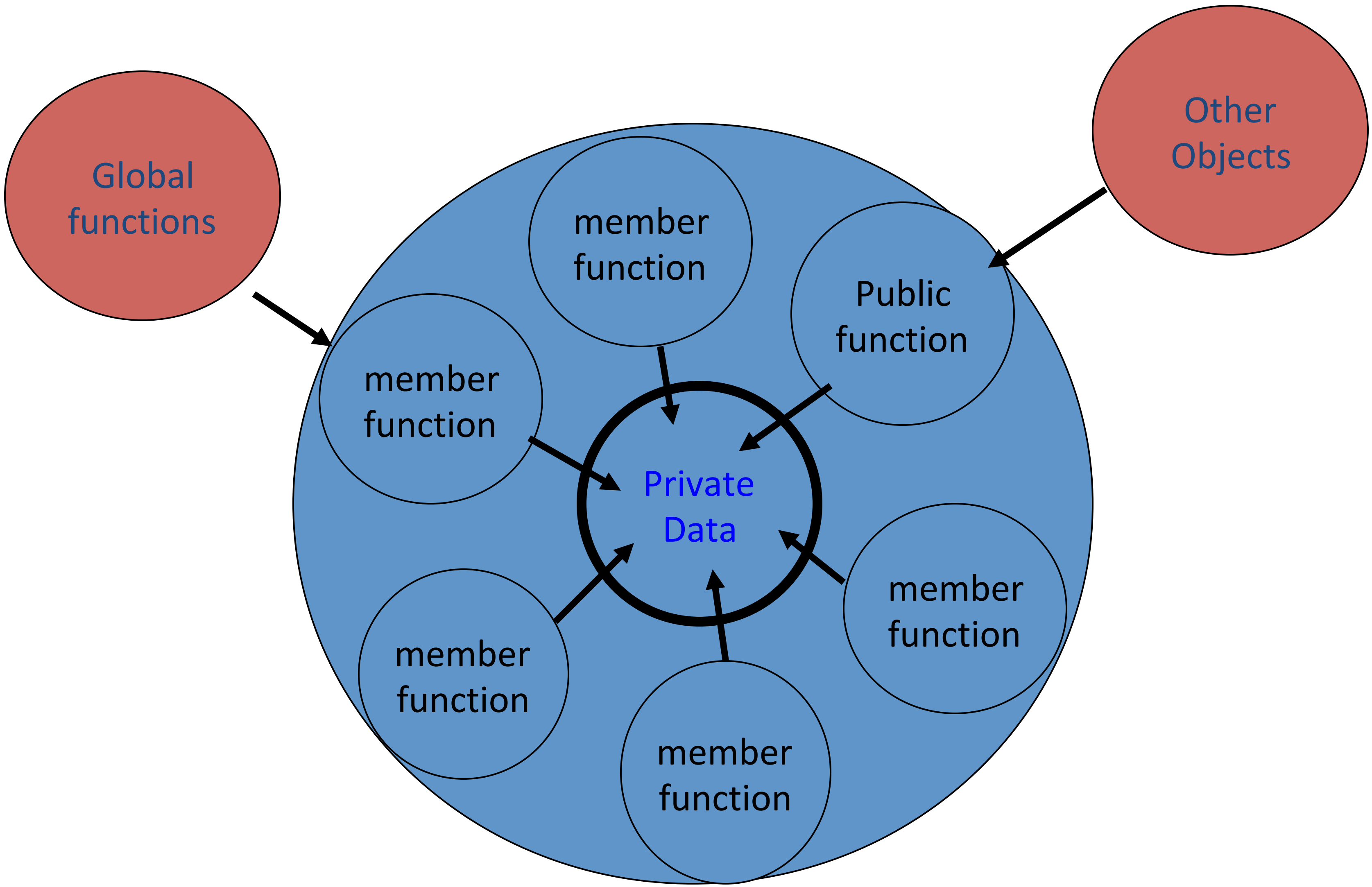
## Class Definition

- The general form for defining a class is as follows:

```
class class_name
{
    public:
        normally member function declaration
    private:
        normally data members/ variable
};
```

- Every class definition must start with the **class** keyword followed by the name of the class. By convention, the first letter of the class name is capitalized.
- The class definition contents are included within a pair of braces and normally includes the prototypes of member functions and the declaration of member variables or so called data members.

## Access to Private Data



## 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?

## Class Definition

- Lets consider the design and definition of a class for a counter.

**Class Design**

```
Class Name:    Counter

Public members:
   function prototype to initialize value
   function prototype to increment value
   function prototype to decrement value
   function prototype to get current value

Private members:
   variable to store the value
```

**Class Definition**

```
class Counter
{
  public:
    void init_to(int num);
    void increment(int n);
    void decrement(int n);
    int  get_value();

  private:
    int  value;
};
```

- The member functions are declared inside the class body.
- The member functions may or may not be defined inside the class body.

## Class Implementation

- Now that we know how to design and 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.

## Class Implementation

- Consider the following implementation for the class Counter in previous slides:

```
void Counter::init_to(int num)
{
 value = num;
}

void Counter::increment(int n)
{
 if( value < INT_MAX -n )
     value += n;
 else
     cout << "Overflow!\n";
}
```

```
int Counter::get_value()
    {
return value;
}

void Counter::decrement(int n)
{
    if( value > INT_MIN + n )
      value -= n;
    else
      cout << "Underflow!\n";
}
```

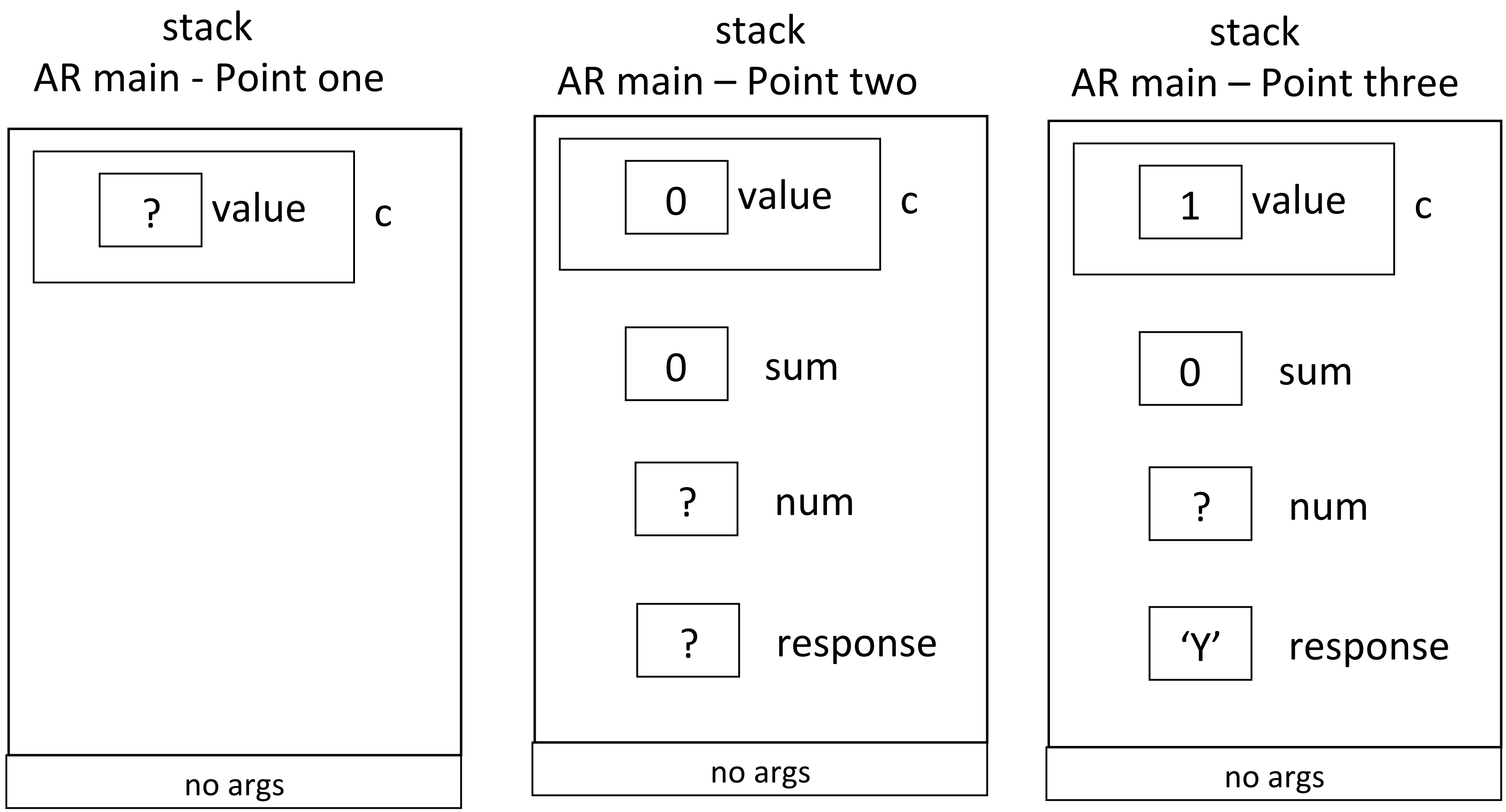
- Note that we did not include a dot operator when accessing the member variable `value` within the member function.
- `INT_MIN` and `INT_MAX` are the minimum and maximum values that an integer can hold and are defined in `limits.h`

## Using Class Object

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

```
void main(){
 Counter counter;        // (1)
 double sum = 0;
 double num;
 char response
 counter.init_to(0);    // (2)

  do
   {
       cout << "Do you wish to enter a number? Press 'Y'< for yes>: ";
       cin >> response;

       if (response== 'Y'){
           c.increment(1); // (3) when reached for the first time
           cout << "\nEnter a number: ";
           cin >> num;
           sum += num;
        }

   }while(response == 'Y');

   cout << "The average of numbers you entered is: "
        << sum/c.get_vlue();
 }
```

# AR Diagrams for points 1, 2, and 3



## Class Implementation

- The function implementation simply defines how the function operates.
  - it is object independent. What does that mean?
  - Consider declaring and using two different Counter objects:

```
Counter object1, object2;
object1.init_to(-1);
object2.init_to(5);
```

  - How does a function know what object to associate with the `value` member variable?
    - In the first case object1.value is is initialized to -1
    - In the second case object2.value is is initialized to -1
  - The answer to this question will be discussed later.

## Class Implementation

- **Every member function of a class has access to all data members of the same class**! They may also call other member functions:
- Any of the following two function are valid implementation of member function plus_two. Both increment the data member value by two.

```
void Counter::plus_two()
{
    increment();
    increment();
}
```

```
void Counter::plus_two()
{
    value++;
    value++;
}
```

## Pointers to Objects

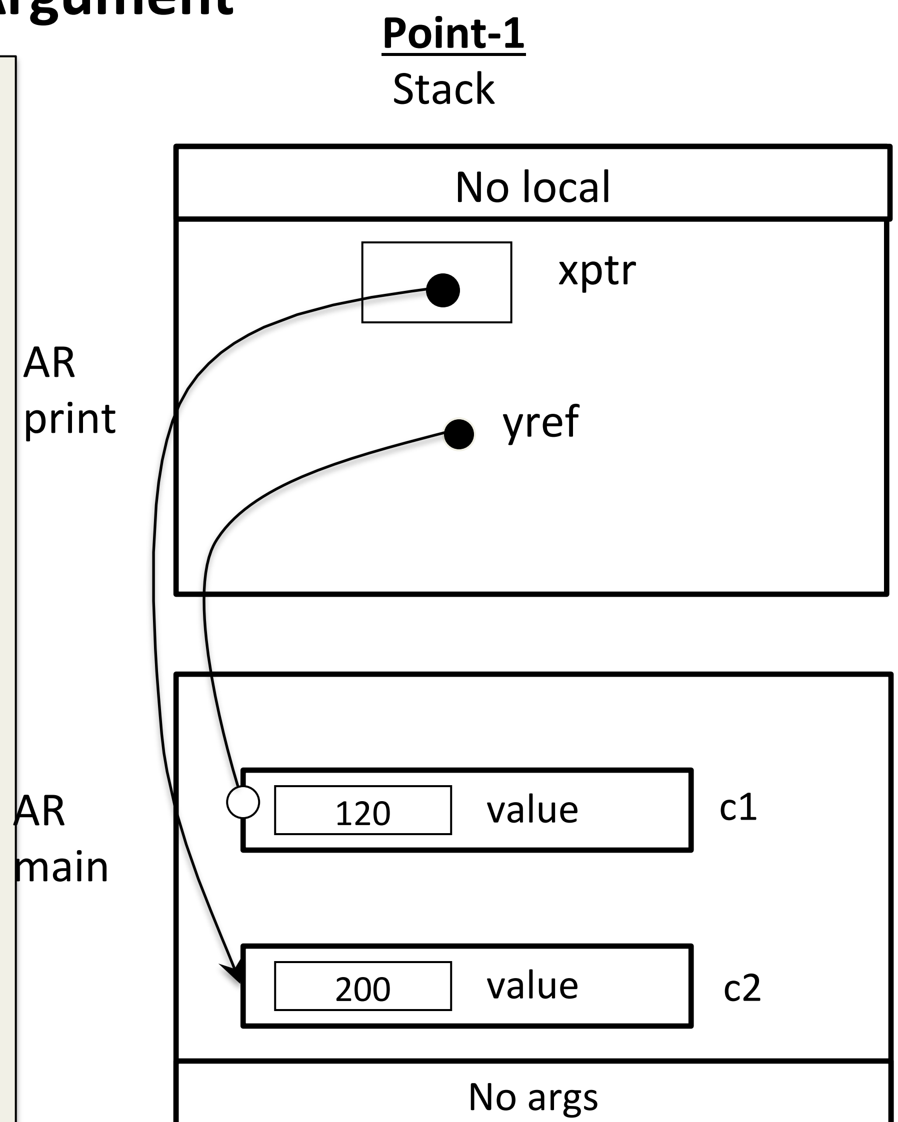
- A pointer in C++ can point to any addressable memory location, including user-defined data types (structures, unions, and classes).
- The principles and notations for pointers are similar for structures, unions and classes.
- Consider the following statements:

```
Counter c;
Counter *ptr;
ptr = &c;
ptr -> init_to(1000);
ptr -> increment(1);
cout  <<  ptr ->get_value();
cout  <<  (*ptr).get_value();
```

- In this example the data member, **value**, is incremented by calling the increment member function through **ptr** pointer.
- 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 preferred). See the following example.

# Object Data Types as a Function Argument

```
void print ( const Counter *xptr , const Counter& yref)
{
   // point one// point one
   cout << xptr -> get_value();

   cout << yref.get_value();
 }



 int main()
 {
     Counter c1;
     c1.init_to(100);
     Counter c2
     c2.init_to(200);
     c1.increment(20);
     print(&c2, c1 );
   }
```



## Object Initialization Using Constructors

- In the previous example we initialized our Objects individually using the member function `init_to()`.

- This is because you cannot initialize a data member when it is defined, for instance:

```
Class Counter{
     private:
     int Value = 0;  // Error! initialization forbidden
};
```

- The easiest way to initialize an Object is to write a Constructor for a Class

- This *automatically* initializes data members *whenever* an Object of the Class is declared

- For example:

## Constructor Concepts

- Consider the slightly modified Counter class definition below:

```
class Counter
{
    public:
        Counter();
        void increment(int n);
        void decrement(int n);
        int get_value();
      private:
          int value;
};
```

- The constructor is implemented as follows:

```
Counter::Counter()
{
    value = 0;
}
```

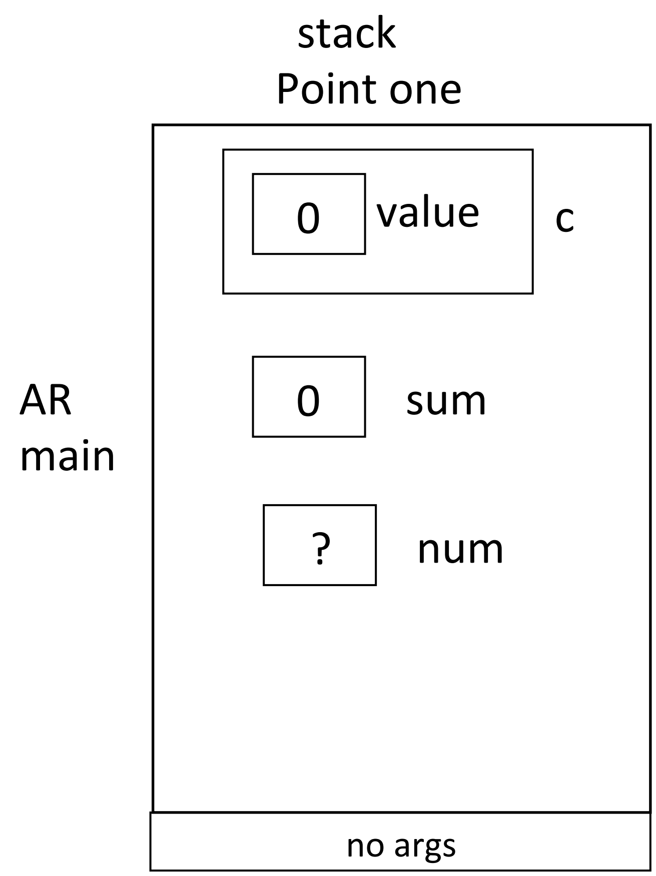
- constructor cannot be called using the dot operator. It will be called automatically when an object is declared.
- Constructor doesn't have a return type.
- Constructor can be overloaded.

## Constructor Concepts

- Now consider the following code segment:

```
void main()
 {
  Counter c;
    double sum = 0;
    double num;

  // point 1

    ...

    return 0;
 }
```

Constructor Called



## Constructor Concepts

- Any constructor that takes no arguments is called a ***default constructor***. In the above case, the initial value of the Counter is not selectable by the user, but at least we know the initial value.

- 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.

## Constructor Concepts

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

```
class Counter
{
  public:
    Counter();              // default constructor
    Counter(int val); // non-default constructor
    void increment(int n);
    void decrement(int n);
    int get_value();
 private:
      int value;
};
```

## Constructor Concepts

- Here are the implementations of the two constructors:

```
Counter::Counter()
{
   value = 0;
}

Counter::Counter(int val)
{
   value = val;
}
```

- To determine which constructor should be called, the compiler looks at the number and type of arguments passed to the function.

## Constructor Concepts

- You can then use any or all of the constructors in your programs:

```
void main()
{
Counter whole_numbers;              //default constructor
Counter positive_numbers(1);  //other deconstructor
Counter bad_idea();                 // ERROR --ILLEGAL!!!

// 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 Counter class as:

```
Counter::Counter(): value(0)
{


}

Counter::Counter(int val): value(val)
{


}
```

## Protecting Data Members

- We have already seen how the `const` keyword can be used to protect formal parameters from being changed inside a function.
- A class member function can be declared as a **"read-only" member functions**.
  - Such a function can access the data member but cannot change it.
  - The `const` keyword will be placed **after the function prototype**. **Make all read-only functions `const`!**
  - Example:

```
class Counter {
  public:
     ...
     int get_value() const;
     ...
 };

// implementation
int Counter::get_value() const
{
   return value;
}
```