

# C++ Pointer and Reference Variables

## More on C++ Functions

### In this Week

- L-values and R-values
- Pointer Variables
- Reference Variables
- Parameter Passing
  - Parameter passing by value
  - Parameter passing by pointer
  - Parameter passing by reference
- Function Return
  - Returning pointers
  - Returning references
- The typedef Specifier

# L-values and R-values

- So far we have seen how to declare variables in C++ and use these variables in our programs
- Consider for example

**int a = 10;**

- Now, we may ask what exactly is **a**? We say it is an integer variable
- Importantly, the variable **a** refers to a memory location
- Moreover, memory locations have identification number assigned to them by the operating system which is known as the memory address of the memory location
- Thus the variable **a** has a memory address
- In C++, any thing that refers to a memory location which can be uniquely identified by its memory address is known as **L-value**
- Therefore the variable **a** is an **L-value**

# L-values and R-values

- A simplified definition of L-Value is as follows
- An ***L-value*** (*locator value*) represents an object that occupies some identifiable location in memory space (i.e. has an address)
- The definition of **R-value** goes
- An *R-value* is any thing that is not L-value
- Based on this definition it is easy to see that an L-value can be placed on the left hand side of assignment operator but not an R-value

# L-values and R-values

- It is easy to see that an **L-value** can be used in the following ways
  - Left hand side of an assignment operator where a value will be assigned to it. For example  
`x = 10;`
  - Right hand side of an assignment operator where its value will be used in the assignment operation. Here the L-value will behave as an R-value. For example:-  
`y = 2*x;`
  - In any expression (arithmetic, boolean, printing, etc) where its value will be used in the expression. Here the L-value will behave as an R-value. For example:-  
`if (3-x > 5%a)`  
`cout << pow(x, 2.7);`
- An **R-value** on the other hand can not be used on the left hand side of an assignment operator
- An **R-value** however can be used in expressions, printing, right hand side of assignment operator, etc

# L-values and R-values

- To better understand **R-value** and **L-value**, we look at the following example code snapshot

```
int main()
{
    int i, j, k;

    i = 7; // Correct usage: the variable i is an L-value.

    7 = i; // Incorrect usage: The left operand must be an L-value
    j * 4 = 7; // Incorrect usage: The left operand must be an L-value

    const int ci = 7;
    ci = 9; // Incorrect usage: the variable is a non-modifiable L-value

    ((i < 3) ? i : j) = 7; // Correct usage: the conditional operator returns an L-value

    k = i+j; // Correct usage. The variable k is L-value. The + operator requires R-values.
            // Therefore i and j are converted to R-values and an R-value

    system("pause");
    return 0;
}
```

# The Address Operator &

- Given a variable which refers to a certain memory location, how do we compute the memory address of the memory location?
- Answer:- We use the address operator (&) as a prefix to the variable name
- For example, given

**int a;**

- The print statement

**cout << &a << endl;**

prints the memory address of the memory location to which the variable **a** refers to

- See the following program for more examples

# The Address Operator &

- The following program demonstrates the usage of the & operator to print the address of several variables

```
int main()
{
    int a = 2;
    float x = 1.2;
    double y = 3.76;
    bool f = true;
    char c = 'y';
    string s = "yonas";

    cout << a << ", " << x << ", " << y << ", " << f << ", " << c << ", " << s << endl;

    cout << &a << endl;
    cout << &x << endl;
    cout << &y << endl;
    cout << &f << endl;
    cout << static_cast<void*>(&c) << endl;
    cout << &s << endl;

    system("Pause");
    return 0;
}
```

# Pointer Variables

- Pointers are C++ variables that store *memory address*
- Syntax: **Pointer Variable Declaration/Initialization**

**DATA\_TYPE\* VARIABLE\_NAME [= MEMORY\_ADDRESS];**

- Initialization during declaration is optional
- Thus a C++ pointer variable does not store a value; rather it stores the address of a memory that contains a value
- Syntax (pointer declaration) example

|                         |  |
|-------------------------|--|
| <code>int* a;</code>    | <code>//int pointer variable</code>    |
| <code>float* b;</code>  | <code>//float pointer variable</code>  |
| <code>char* c;</code>   | <code>//char pointer variable</code>   |
| <code>double* d;</code> | <code>//double pointer variable</code> |



# Initializing Pointer Variables

- Since pointer variables store memory addresses, then the address operator (&) can be used to assign the *memory address of a C++ variable to a pointer variable of the same data type*
- Therefore we may do something like

```
int a = 8;  
int* b = &a;           //Declaration and initialization of pointer variable
```

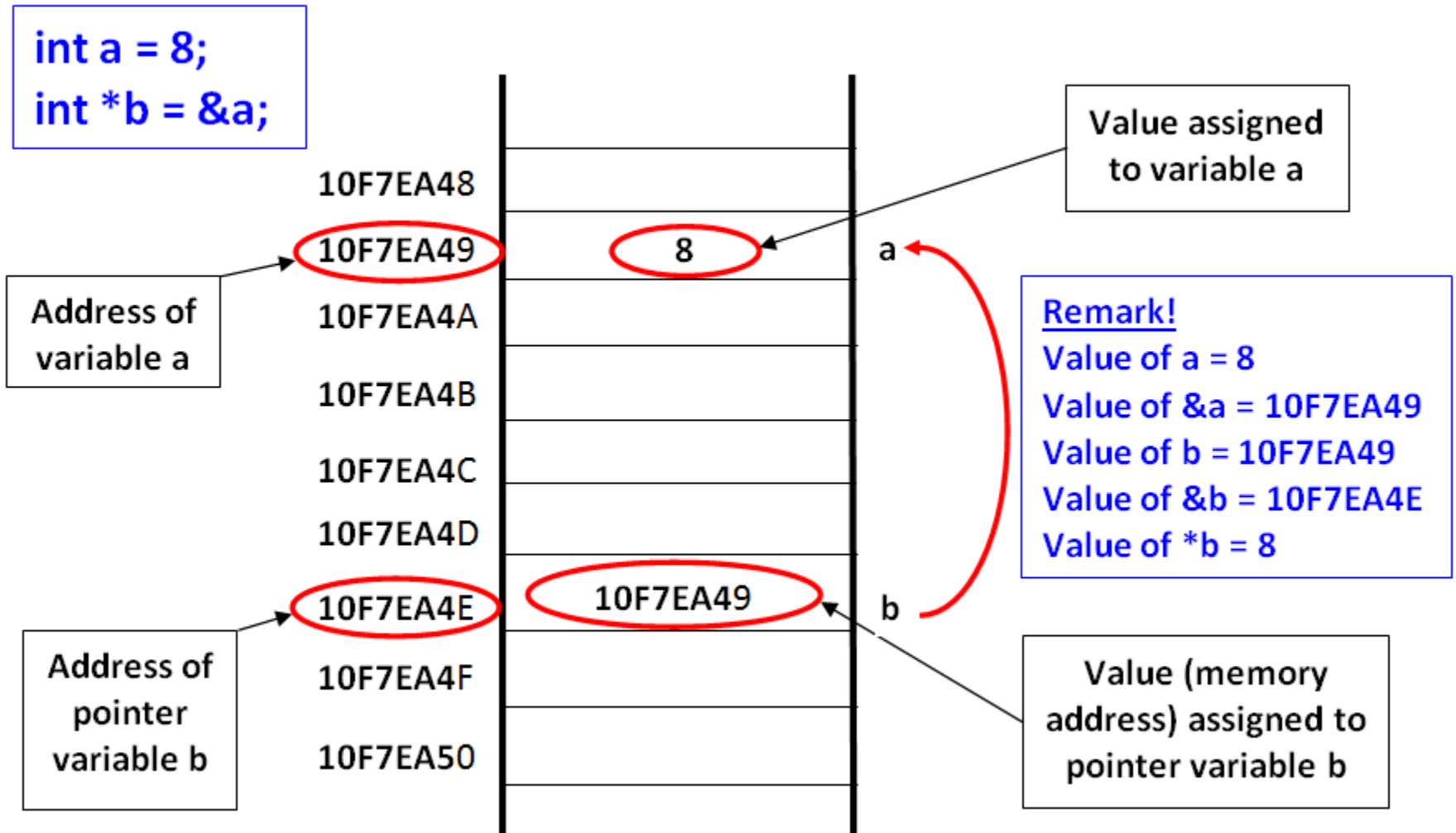
- Alternatively we may separate the declaration and initialization as follows:

```
int a = 8;  
int *b;                //Declaration of pointer variable  
b = &a;                 //Initialization of pointer variable
```

- If we now print the value of variable **b**, we get the address of the variable **a** *while &b will give address of the pointer variable b (see diagram next page)*

```
cout << a << endl;      //prints the value 8  
cout << b << endl;      //prints the memory address of the variable a  
cout << &b << endl;     //prints the memory address of the variable b
```

# Initializing Pointer Variables



# The de-reference (\*) operator

- In the previous example, we have

```
int a = 8;
```

```
int* b = &a;
```

- Could we then access the value stored in the variable **a** using the pointer variable **b**? **YES!**
- C++ allows accessing the value stored in a memory space pointed to by a pointer variable through the pointer variable
- To do so we use the **de-reference (\*) operator**
- Thus, given the above pointer variable definition, the de-reference of the pointer variable denoted as **\*b** refers to the memory space that is pointed to by the pointer variable
- Hence **\*b** is an **L-value**

# The dereference (\*) operator

- Hence

```
cout << *b << endl;
```

prints the value stored in the memory pointed to by the pointer variable **b**; which means it will print the value **8**

- In general, the de-reference operator takes a memory address value and then it will refer to the memory whose address is the given memory address
- Hence the expression **\*(10F7EA49)** refers to the memory whose address is **10F7EA49**
- In the following example, we use **\*b** in an assignment operator in order to modify the value stored in memory pointed to by a pointer

```
int main()
{
    int a = 8;           //assign a meory space named a the value 8
    int* b = &a;         //create a pointer pointing the memory named a
    cout << a << endl;   //prints 8
    cout << *b << endl;  //prints 8
    *b = 27;             //modifies the value of the memory named a to 27
    cout << *b << endl;  //prints 27
    cout << a << endl;   //prints 27 as well
    system("Pause");
    return 0;
}
```

# Pointers, address operator and de-reference operator

## Example Code

```
int main()
{
    int a = 8;
    int *b = &a;
    cout << "Value of a \t a = " << a << endl;
    cout << "Address of a \t &a = " << &a << endl;
    cout << "Value of b \t b = " << b << endl;
    cout << "Address of b \t &b = " << &b << endl;
    cout << "Data in memory pointed by pointer *b = " << *b << endl;

    /*
    //Some wrong usages of pointers and memory addresses
    b = 15;      //Error! A value of int can not be assigned to a pointer
    b = &(a+6); //Error! a+6 is R-value. The & operator requires L-value
    &a = 40;     //Error! Assignment operator requires L-value left operand
    &a = b;      //Error! Assignment operator requires L-value left operand
    int c = b;  //Error! A pointer can not be used to initialize an int variable
    */

    //Some correct usages of pointers and memory addresses
    int d = a + *b; //Correct! Now d has the value 16
    cout << "Value of d \t d = " << d << endl;
    *b = 10;        //Correct! It assigns in the memory pointed by b (i.e. the variable a) the value 10
    cout << "Data in memory pointed to by pointer *b = " << *b << endl;
    cout << "More importantly, a is now a = " << a << endl;

    system("pause");
    return 0;
}
```

# Incorrect use of pointers

- As mentioned earlier, the memory address of a variable assigned to a pointer variable must have the same data type as the pointer variable
- The following code gives correct and incorrect uses of pointer variables

```
int a = 7;
```

```
float b = 3.5;
```

```
float* c = &b; ← Correct!
```

```
int* d = &b; ← Error! Data type mismatch
```

# The Heap Memory

- In the previous example, we have seen that a pointer can be declared and then make it point to an existing value variable
- It is also possible to declare a pointer variable and not point it to anywhere as follows:

**int\* p;**

- Now **p** is an integer pointer variable pointing to nowhere
- Trying to deference **p** will create a runtime error
- More importantly, we can now point **p** to an existing value variable or **alternatively we can allocate a new memory to be pointed by p**

# The Heap Memory

- In order to allocate a new memory to be pointed by `p`, we use the **new** operator as follows

**`p = new int;`**

- Now `p` points to a memory location
- We can now initialize the memory just allocated using `p` and using **de-reference** operator as follows

**`*p = 5;`**

- Finally we can use (or print) the content of the memory using `p` and using dereference operator as follows

**`cout << *p << endl;`**

- Such a memory location allocated using the **new** operator is known as **dynamic memory**



# The Heap Memory

- When a dynamic memory is allocated with the **new** operator; then the memory location is created in a special memory space known as the **heap memory**
- The heap memory is accessible from any block and any function in a program. Therefore it is common memory to all parts of a program
- This means we can use the new operator inside a main program or inside any function to allocate memory in the heap memory

# The Heap Memory

- We can also define (declare and initialize) a pointer pointing to a dynamic memory as follows

**int\* p = new int(5);**

- Now the dynamic memory location pointed to by **p** contains the value 5
- Also, **int \*p = new int();** is valid and initializes the dynamic memory with a value 0

# The Heap Memory

- Variables created in a program normally cease to exist (and get cleared out from the memory space), once they go out of block
- The **memory spaces in the heap** are however exceptions to this rule
- When our program terminates; while **any pointer variable that points to a heap memory goes out of scope and hence automatically deleted**, **the memory that was allocated in the heap still remains tied to our program even after our program terminates**
- This means if you run such programs repeatedly, then it is possible to eat all the computer's memory and freeze out the computer

# The Heap Memory

- For this reason, it is important we clear the heap memory manually in our program
- We use the **delete** operator to clear any memory that was allocated in the heap
- For example if a pointer **p** is pointing to the heap memory, then in order to delete the heap memory we go as follows:  
**delete p;**
- After deleting the heap memory, the pointer variable that was pointing to the heap is still available and is still pointing to the same memory on the heap
- But it should never be de-referenced because that memory no more belongs to our program and may cause run-time error when we try to access it!
- However the pointer may be re-assigned another valid memory address after which it may be used as we desire (see next example)
- In general, the delete operator takes a memory address value and then deletes the memory spaces whose address is the given address
- Thus

**delete 10F7EA49**

will delete the memory space whose address is **10F7EA49**

# The Heap Memory

```
int main()
{
    //Create a pointer
    int *p;
    //Create a dynamic memory on the heap and point to it by p
    p = new int;
    //Assign an integer value to the heap memory pointed to by p
    *p = 6;
    //Print the value of p. This will print eight hexadecimal digits
    cout << "The value of the pointer variable p is " << p << endl;
    //Print the value in the heap memory pointed to by p. This will print 6
    cout << "The value of the dereferenced pointer variable p is " << *p << endl;

    //Delete the dynamic memory on the heap. This only deletes the heap.
    delete p; //The pointer variable p is NOT deleted. It will still exist

    //Print the value of p. This will still print the same eight hexadecimal digits
    cout << "The value of the pointer variable p is " << p << endl;
    //Print the value in the heap memory pointed to by p.
    //The heap memory was deleted and therefore we will NOT get the integer value 6
    //Instead a garbage value will be printed (We call this Run-Time Error)
    cout << "The value of the dereferenced pointer variable p is " << *p << endl;

    //Because we still have the pointer variable p,
    //let us assign it a different memory address value
    int a = 3;
    p = &a;
    //Print the value of p. This will print a different eight hexadecimal digits
    cout << "The value of the pointer variable p is " << p << endl;
    //Print the value in the memory pointed to by p. This will print 3
    cout << "The value of the dereferenced pointer variable p is " << *p << endl;

    system("Pause");
    return 0;
}
```

# Pointer to a de-referenced Pointer

- A de-referenced pointer variable refers to the memory space that is pointed by the pointer (hence it is an **L-value**)
- For example, if **p** is a pointer to an integer then **\*p** is the integer memory location
- As such we can create a pointer to **\*p** as follows

```
int a = 9;      //An int variable
```

```
int *p = &a;    //An int pointer variable pointing to a
```

```
int *q = &(*p); //An int pointer variable (pointing to *p)
```

- Now we see that **q** is a pointer to **\*p**
- But **\*p** is the same as the integer memory location **a**
- Therefore **q** is a pointer to **a**

# Pointer to a Pointer

- We can also declare a pointer to a pointer
- Such declaration will have **\*\*** in its declaration
- For example

```
int a = 5;
```

```
int *b = &a; //Pointer variable
```

```
int **c = &b; //Pointer to a pointer
```

- Now **c** is a pointer to **b**; where **b** is itself a pointer to **a**
- The value of **b** is the memory address of **a** while the value of **c** is the memory address of **b**
- Moreover the value of **\*c** will be the same as the value of **b** (which is the memory address of **a**) while the value of **\*\*c** will be the value of **a**

# Pointer to de-referenced pointer and Pointer to Pointer Example

```
int main()
{
    int a = 5;
    int* p = &a;
    int* q = &(*p);
    int** r = &p;

    cout << a << endl; //Prints the value of a
    cout << &a << endl; //Prints the memory address of a
    cout << p << endl; //Prints the memory address of a
    cout << q << endl; //Prints the memory address of a

    cout << *p << endl; //Prints the value of a
    cout << *q << endl; //Prints the value of a

    cout << r << endl; //Prints the memory address of p
    cout << *r << endl; //Prints the value of p which is the memory address of a
    cout << **r << endl; //Prints the value of *p which is the value of a

    system("Pause");
    return 0;
}
```



# Reference Variables

## Variable Aliases

- C++ supports yet another type of variables called **reference variables**; in addition to **value variables** and **pointer variables**
- In simple words, reference variable is an **alias** (alternative name) for another variable
- **Syntax**: **Declaration of Reference Variable**  
  
    **DATA\_TYPE& refVariable = existingVariable;**
- Now **refVariable** is simply the same memory location as the **existingVariable**

# Reference Variables

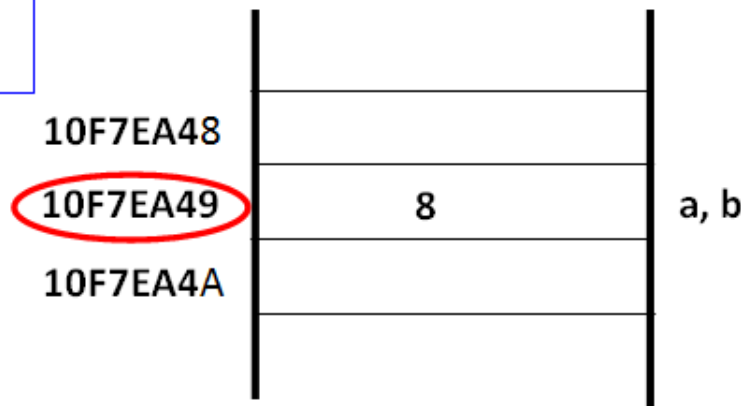
## Variable Aliases

```
int main()
{
    int a = 8;
    int &b = a;
    cout << "a = " << a << ", b = " << b << endl;
    cout << "&a = " << &a << ", &b = " << &b << endl;
    system("pause");
    return 0;
}
```

These outputs will be identical

These outputs will be identical

```
int a = 8;
int &b = a;
```



### Remark!

Value of a = 8  
Value of &a = 10F7EA49  
Value of b = 8  
Value of &b = 10F7EA49

# Reference Variables

## Variable Aliases

- Modifying the value stored in a reference variable therefore modifies the value stored in the referenced variable as well; after all both of them belong to the same memory location!
- **Example**

```
int a = 8;  
int& b = a;  
cout << "a = " << a << ", and b = " << b << endl;  
b = 25;  
cout << "a = " << a << ", and b = " << b << endl;
```
- In this example, when the value of the reference variable changed, it will change the value stored in the memory location to which both the variables **a** and **b** belong to. Therefore the output is

**a = 8, b = 8**

**a = 25, b = 25**

# Challenge Yourself

- What is the output of the following program?

```
int main()
{
    int a = 5;
    int *b = &a;
    int &c = a;
    cout << "a = " << a << ", c = " << c << ", and *b = " << *b << endl;
    int k = 30;
    b = &k;
    k = 200;

    //At this point, is there a change in value of c?
    cout << "a = " << a << ", c = " << c << ", and *b = " << *b << endl;

    system("pause");
    return 0;
}
```

# Reference to a Pointer

- Similarly, we can create a reference to a pointer variable
- A reference to a pointer variable must itself be a pointer
- For example

```
int a = 5;
```

```
int *c = &a; //A pointer to a value variable
```

```
int* &d = c; //A reference to a pointer variable
```

# Reference to a de-referenced Pointer

- A de-referenced pointer variable refers to the memory space that is pointed to by the pointer
- Hence a de-referenced pointer is an **L-value**
- Thus we can make a reference variable to a de-referenced pointer
- For example, if **p** is a pointer to an integer then **\*p** is an integer memory location
- Therefore we can make an integer reference variable to **\*p** as follows:

```
int a = 9;           //An int variable
int *p = &a;         //An int pointer variable pointing to a
int &b = *p;         //An int variable (reference to *p)
```

- Now **b** is a reference to de-referenced **p** which is the same as saying **b** is a reference to **a**

# Parameter Passing to Functions

- A function declaration in C++ consists of parameter declaration
- Parameter declaration is nothing more than variable declaration that get initialized with function call arguments
- Therefore parameter declaration can be done using **value variables**, **reference variables**, or **pointer variables**
- This brings us to different parameter passing in C++ and their consequences

# Parameter Passing in C++

## Pass by Value

- Consequently the following program which attempts to swap the values of the variables a and b inside a function where pass by value is used **will not work!!!**

```
void swap(int x, int y)
{
    cout << endl << "IN FUNCTION SWAP BEFORE SWAPPING: x = " << x << ", and y = " << y << endl;
    int temp = x;
    x = y;
    y = temp;
    cout << "IN FUNCTION SWAP AFTER SWAPPING: x = " << x << ", and y = " << y << endl << endl;
    return;
}

int main()
{
    int a = 5, b = 8;
    cout << "BEFORE FUNCTION CALL: In the main program a = " << a << ", and b = " << b << endl;
    swap(a, b);
    cout << "AFTER FUNCTION CALL: In the main program a = " << a << ", and b = " << b << endl << endl;

    system("pause");
    return 0;
}
```

**The output after function call will still be  
a = 5, and b = 8**



# Parameter Passing in C++

## Pass by Pointer

- Fortunately, C++ supports a way to pass parameters to functions such that the changes made inside the function are reflected in the main program
- C++ provides two methods for such purposes
- Passing parameters by pointer is one method
- In passing the parameters by pointer, we are effectively passing memory addresses of variables in the main program to the function
- This allows to access the memory space of the main program by dereferencing the pointer parameters of the function; which allows to modify the values of the memory space in the main program from within the function

# Parameter Passing in C++

## Pass by Pointer

- In order to pass parameters by pointer, the function declaration is modified so that the parameters of the function are pointer variables
- **Syntax:** Parameter passing by pointer function declaration  
`void swap(int* x, int* y)`
- The function call is modified as well so that to pass memory addresses as follows
- **Syntax:** Parameter passing by pointer function call  
`swap(&a, &b);`
- The following example demonstrates this...

# Parameter Passing in C++

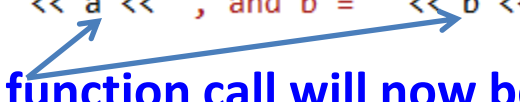
## Pass by Pointer

- This program swaps the values of the variables a and b defined in the main program from within the function **parameter passing by pointer**

```
void swap(int *x, int *y)
{
    cout << endl << "IN FUNCTION SWAP BEFORE SWAPPING: x = " << *x << ", and y = " << *y << endl;
    int temp = *x;
    *x = *y;
    *y = temp;
    cout << "IN FUNCTION SWAP AFTER SWAPPING: x = " << *x << ", and y = " << *y << endl << endl;
    return;
}

int main()
{
    int a = 5, b = 8;
    cout << "BEFORE FUNCTION CALL: In the main program a = " << a << ", and b = " << b << endl;
    swap(&a, &b);
    cout << "AFTER FUNCTION CALL: In the main program a = " << a << ", and b = " << b << endl << endl;

    system("pause");
    return 0;
}
```



The output after function call will now be  
**a = 8, and b = 5**

# Rules for Pass by Pointer

- Since pointer parameter variables expect the address of memory locations, we can not pass literal values to such parameters. Therefore the function call

**swap(4, 5);** ← **Error!!! Memory address expected**

in the above program will be an incorrect C++ statement and is an error!

- Importantly, a function can have some of its parameters as pointers and others non-pointer variables

# Parameter Passing in C++

## Pass by Reference

- C++ supports yet another method of passing parameters to functions such that the changes made to the parameters of the function are reflected in the caller function
- While passing by pointer is very powerful, it is very error prone. In the above program for example the integer values in the pointer variables must be accessed through the dereference operator
- Forgetting the dereference operator, which usually happens, results to syntax errors or worse to semantic errors which are hard to find and therefore correct!!!
- In order to simplify this C++ adds parameter passing by reference

# Parameter Passing in C++

## Pass by Reference

- In passing parameters by reference, the function declaration is modified as follows:
- **Syntax**: Parameter passing by reference function declaration  
`void swap(int& x, int& y)`
- The function call is modified as well as follows
- **Syntax**: Parameter passing by reference function call  
`swap(a, b);`
- Now, in the function the integer values are accessed without dereference operator as follows

# Parameter Passing in C++

## Pass by Reference

- This program swaps the values of the variables a and b defined in the main program from within the function using **parameter passing by reference**

```
void swap(int &x, int &y)
{
    cout << endl << "IN FUNCTION SWAP BEFORE SWAPPING: x = " << x << ", and y = " << y << endl;
    int temp = x;
    x = y;
    y = temp;
    cout << "IN FUNCTION SWAP AFTER SWAPPING: x = " << x << ", and y = " << y << endl << endl;
    return;
}

int main()
{
    int a = 5, b = 8;
    cout << "BEFORE FUNCTION CALL: In the main program a = " << a << ", and b = " << b << endl;
    swap(a, b);
    cout << "AFTER FUNCTION CALL: In the main program a = " << a << ", and b = " << b << endl << endl;

    system("pause");
    return 0;
}
```

The output after function call will now be  
**a = 8, and b = 5**

# Parameter Passing in C++

## Pass by Reference

- Since reference parameter variables expect an actual memory locations, we can not pass literal values to such parameters. Therefore the function call

**swap(4, 5); ← Error!!! L-value expected**

in the above program will be an incorrect C++ statement and is an error!

- Also the statement

**swap(&a, &b); ← Error!!! L-value expected**

will be incorrect because a reference variable expects an L-value (but &a is a memory address value hence an R-value)



# Parameter Passing in C++

## Pass by Reference

- Parameter passing by reference catches the simplicity of pass by value and the powerful feature of pass by pointer without the extra dereference operators
- As such parameter passing by reference in C++ is extensively used by developers
- The function parameter declaration **int& x** makes the variable x a **reference variable**
- **A function can have some of its parameters declared as value parameters, some of them as pointers and others as references**

# Function Return

## Returning a Pointer

- A C++ function may return a pointer in which case the function will effectively return a memory address value
- Thus the calling function (such as the main program) may store the returned value in a pointer variable and then de-reference the pointer to access the memory location pointed by the pointer

# Function Return

## Returning a Pointer

```
int* magic()
{
    int *p = new int(7); //assume the heap memory has address 4F
    cout << p << endl;
    cout << *p << endl;

    return p;
}
int main()
{
    int *a;
    a = magic();
    cout << a << endl;
    cout << *a << endl;
    *a = 6;
    cout << *a << endl;
    return 0; delete a;
}
```

# Function Return

## Returning a Pointer

- It should be noted that when returning a pointer, a function should ensure that the memory location whose address is returned will still have a scope outside the function; for otherwise it will cause a runtime error when we try to access the memory afterwards

```
int* foo()
{
    int x = 2;
    return &x;
}

int main()
{
    int *p = foo();
    cout << *p << endl;
    system("Pause");
    return 0;
}
```

# Function Return

## Returning a Reference

- A function may also return by reference
- When returning by reference, the function will be returning an **L-value**
- That is the function will not return the value inside a memory location
- Rather it will return the memory location itself
- See example below

# Function Return

## Returning a Reference

```
int& foo(int *p)
{
    *p = 7;
    return *p;
}

int main()
{
    int x = 5;
    int y = foo(&x);
    cout << x << ", " << y << endl;
    x = 12;
    cout << x << ", " << y << endl;
    y = 15;
    cout << x << ", " << y << endl;

    system("Pause");
    return 0;
}
```

# Function Return

## Returning a Reference

- It should be noted a function should never return a local memory location by reference because the memory space will be cleared once control goes out of the function and thus the memory can not be referenced any longer. See example below.

```
int& foo(int *p)
{
    *p = 7;
    int q = *p;
    return q;
}
int main()
{
    int x = 5;
    int y = foo(&x);
    cout << x << ", " << y << endl;
    system("Pause");
    return 0;
}
```

# C++ **typedef** Specifier

- Consider the following declaration

**int a, b, c;**

- What is the data type of **a**, what is the data type of **b**, and what is the data type of **c**?

**Answer:-** All **a**, **b**, and **c** are **int** type

- Now consider the following declaration

**int\* a, b, c;**

- What is the data type of **a**, what is the data type of **b**, and what is the data type of **c**?

**Answer:-** The variable **a** is an **integer pointer** data type. But both **b** and **c** are **integer** data type!!!



# C++ **typedef** Specifier

- Why so?
- Because in C++ the pointer \* designation applies only to the first variable in the declaration; all the other variables in the same declaration will not get the pointer designation!
- So how can we declare two or more variables in the same declaration statement and yet we would like all of them to be pointers?
- **Answer:-** We have two ways:
  - One way is to have a pointer designation for each of the variables as follows  

```
int *a, *b, *c;
```
  - Another way is to **define a new named data type** of our interest **using typedef** specifier as follows

# C++ **typedef** Specifier

- In order to define a new integer pointer data type we proceed as follows

**typedef int\* cmpt130IntegerPointer;**

- Now **cmpt130IntegerPointer** is a data type which is **int\***
- Moreover, as we can see it here we can choose any valid identifier name for the new data type definition
- Then we can declare integer pointer variables as follows  
**cmpt130IntegerPointer a, b, c;**
- Here all the variables **a, b and c are now pointer to integer** data type!
- Normally, we put **typedef** statement **at the top of our C++ programs just below the include statements!!!**