C++ Programming Language

Prepared by: Mohamed Ayman

Algorithm Engineer at Valeo

Deep Learning Researcher and Teaching Assistant

at The American University in Cairo (AUC)

spring 2020

sw.eng.MohamedAyman@gmail.com

facebook.com/cs.MohamedAyman

linkedin.com/in/cs-MohamedAyman

github.com/cs-MohamedAyman

codeforces.com/profile/Mohamed_Ayman

Lecture 7

Pointers & References

Course Roadmap

Part 2: C++ Pointers and Functions

Lecture 7: Pointers & References

Lecture 8: Functions

Lecture 9: Strings

Lecture 10: Structures

Lecture 11: Enumerations

Lecture 12: Unions

Lecture Agenda

We will discuss in this lecture the following topics

1- Introduction to Pointers

2- Pointers vs. Arrays

3- Pointer Arithmetic & Comparisons

4- Array of Pointers

5- Pointer to a pointer

6- Reference Variables


Let's
START UP

Lecture Agenda

Section 1: Introduction to Pointers

Section 2: Pointers vs. Arrays

Section 3: Pointer Arithmetic & Comparisons

Section 4: Array of Pointers

Section 5: Pointer to a pointer

Section 6: Reference Variables

Introduction to Pointers

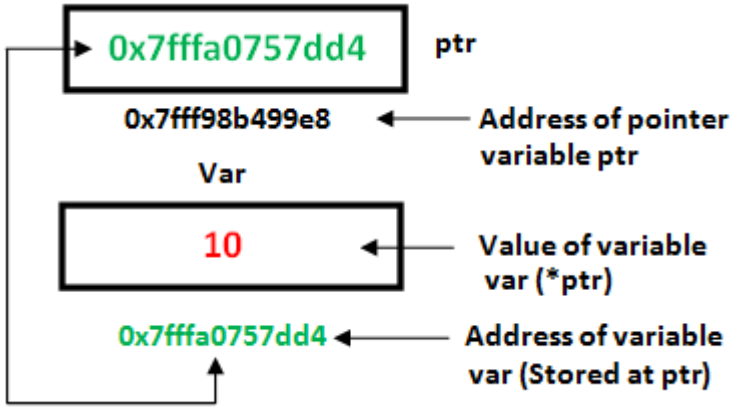
- Pointer is a variable that stores address of another variable. A pointer can also be used to refer to another pointer function. A pointer can be incremented/decremented, i.e., to point to the next/ previous memory location. The purpose of pointer is to save memory space and achieve faster execution time.
- VARIABLE: A value stored in a named storage/memory address
- POINTER: A variable that points to the storage/memory address of another variable
- Declaring a pointer: Like variables, pointers have to be declared before they can be used in your program. Pointers can be named anything you want as long as they obey C's naming rules. A pointer declaration has the following form. [data_type * pointer_variable_name;]
- Initialize a pointer: After declaring a pointer, we initialize it like standard variables with a variable address. If pointers are not uninitialized and used in the program, the results are unpredictable and potentially disastrous. To get the address of a variable, we use the ampersand (&)operator, placed before the name of a variable whose address we need. Pointer initialization is done with the following syntax. [pointer = & variable;]
- Operator *: Serves 2 purpose, Declaration of a pointer and Returns the value of the referenced variable
- Operator &: Serves only 1 purpose, Returns the address of a variable

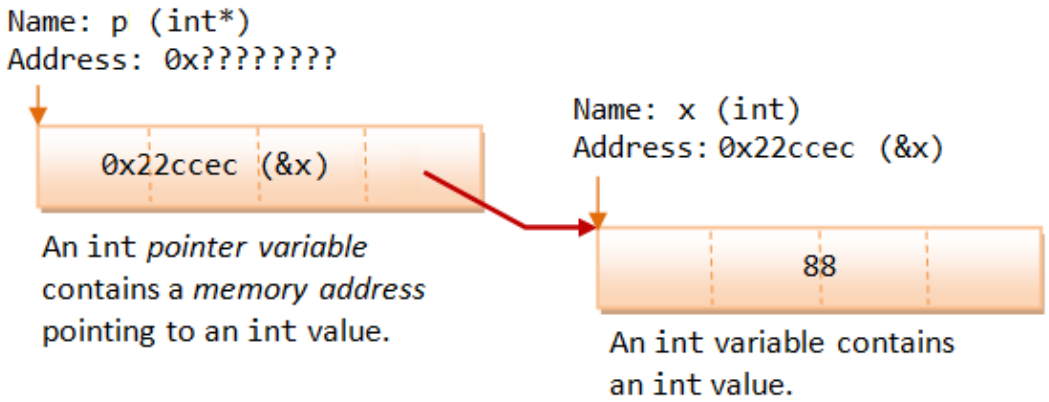
Introduction to Pointers

- There are few important operations, which we will do with the pointers very frequently.

(a) We define a pointer variable.
(b) Assign the address of a variable to a pointer.
(c) Finally access the value at the address available in the pointer variable.
This is done by using unary operator (*) that returns the value of the variable located at the address specified by its operand.

- The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer.





Introduction to Pointers

Example:

```
int x = 5;
cout << "Value   of  x : " << x << '\n';
cout << "Address of  x : " << &x << '\n';

int* y = &x;
cout << "Value   of  y : " << y << '\n';
cout << "Address of  y : " << &y << '\n';
cout << "Value   of *y : " << *y << '\n';

int* z;
z = &x;
cout << "Value   of  z : " << z << '\n';
cout << "Address of  z : " << &z << '\n';
cout << "Value   of *z : " << *z << '\n';
```

Output:

```
Value   of  x : 5
Address of  x : 0x7ffdbb8e87bc

Value   of  y : 0x7ffdbb8e87bc
Address of  y : 0x7ffdbb8e87b0
Value   of *y : 5

Value   of  z : 0x7ffdbb8e87bc
Address of  z : 0x7ffdbb8e87a8
Value   of *z : 5
```

Advantages vs. Disadvantages of Pointers

➢ Importance of pointers:

- some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.
- As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory.

➢ Advantages of Pointers:

- Pointers provide an efficient way for accessing the elements of an array structure.
- Pointers are used for accessing dynamic memory allocation as well as deallocation.
- Pointers are used to form complex data structures such as linked list, graph, tree, etc.

➢ Disadvantages of Pointers:

- Pointers are a little complex to understand.
- If an incorrect value is provided to a pointer, it may cause memory corruption.
- Pointers can lead to various errors such as segmentation faults or can access a memory location which is not required at all.
- Pointers are also responsible for memory leakage, and they are comparatively slower than that of the variables.

Types of a pointer

- **Null pointer:** We can create a null pointer by assigning null value during the pointer declaration. This method is useful when you do not have any address assigned to the pointer. A null pointer always contains value 0.
- **Void Pointer:** a void pointer is also called as a generic pointer. It does not have any standard data type. A void pointer is created by using the keyword void. It can be used to store an address of any variable.
- **Wild pointer:** A pointer is said to be a wild pointer if it is not being initialized to anything. These types of pointers are not efficient because they may point to some unknown memory location which may cause problems in our program and it may lead to crashing of the program. One should always be careful while working with wild pointers.

Example:

```
void* v = NULL;
cout << "Value   of  v : " << v << '\n';
cout << "Address of  v : " << &v << '\n';
```

Output:

```
Value   of  v : 0
Address of  v : 0x7fffff3dff98
```

Types of a pointer

Example:

```
int* p = NULL;
cout << "Value   of  p : " << p << '\n';
cout << "Address of  p : " << &p << '\n';
float* x = NULL;
cout << "Value   of  x : " << x << '\n';
cout << "Address of  x : " << &x << '\n';
long long* y = NULL;
cout << "Value   of  y : " << y << '\n';
cout << "Address of  y : " << &y << '\n';
```

Output:

```
Value   of  p : 0
Address of  p : 0x7fffff3dffb8
Value   of  x : 0
Address of  x : 0x7fffff3dffb0
Value   of  y : 0
Address of  y : 0x7fffff3dffa8
```

(*p) will case STATUS_ACCESS_VIOLATION exception if the pointer is NULL

Types of a pointer

Example:

```
double* z = NULL;
cout << "Value   of  z : " << z << '\n';
cout << "Address of  z : " << &z << '\n';
bool* w = NULL;
cout << "Value   of  w : " << w << '\n';
cout << "Address of  w : " << &w << '\n';
char* c = NULL;
cout << "Value   of  c : " << (void*) c << '\n';
cout << "Address of  c : " << &c << '\n';
```

Output:

```
Value   of  z : 0
Address of  z : 0x7fffff3dffa0
Value   of  w : 0
Address of  w : 0x7fffff3dff90
Value   of  c : 0
Address of  c : 0x7fffff3dff88
```

(*p) will case STATUS_ACCESS_VIOLATION exception if the pointer is NULL

Lecture Agenda

- ✓ Section 1: Introduction to Pointers
- **Section 2: Pointers vs. Arrays**
- Section 3: Pointer Arithmetic & Comparisons
- Section 4: Array of Pointers
- Section 5: Pointer to a pointer
- Section 6: Reference Variables

Pointers vs. Arrays

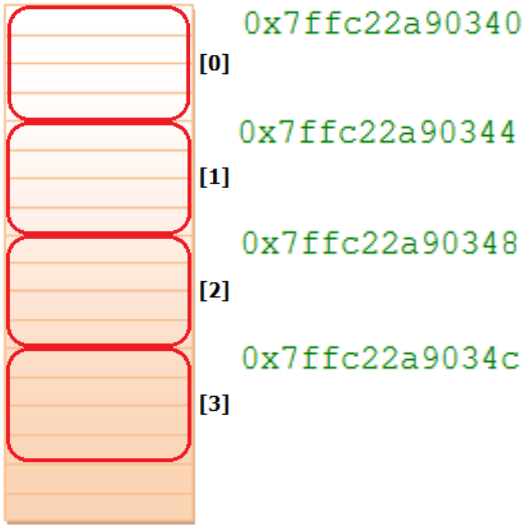
- Pointers and arrays are strongly related. In fact, pointers and arrays are interchangeable in many cases. For example, a pointer that points to the beginning of an array can access that array by using either pointer arithmetic or array-style indexing.
- It is perfectly acceptable to apply the pointer operator * to var but it is illegal to modify var value. The reason for this is that var is a constant that points to the beginning of an array and can not be used as l-value. Because an array name generates a pointer constant, it can still be used in pointer-style expressions, as long as it is not modified.

Computer		Programmers		
Address	Content	Name	Type	Value
90000000	00			
90000001	00			
90000002	00	sum	int (4 bytes)	000000FF(255_{10})
90000003	FF			
90000004	FF			
90000005	FF	age	short (2 bytes)	FFFF(-1_{10})
90000006	1F			
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF	averge	double (8 bytes)	1FFFFFFFFFFFFFF ($4.45015E-308_{10}$)
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90			
9000000F	00			
90000010	00	ptrSum	int* (4 bytes)	90000000
90000011	00			

Pointers vs. Arrays

Example:

```
int x[4] = {11, 23, 41, 17};
cout << "Address of " << x[0] << " or " << *x
     << " is : " << &x[0] << " or " << x << '\n';
cout << "Address of " << x[1] << " or " << *(x+1)
     << " is : " << &x[1] << " or " << x+1 << '\n';
cout << "Address of " << x[2] << " or " << *(x+2)
     << " is : " << &x[2] << " or " << x+2 << '\n';
cout << "Address of " << x[3] << " or " << *(x+3)
     << " is : " << &x[3] << " or " << x+3 << '\n';
```



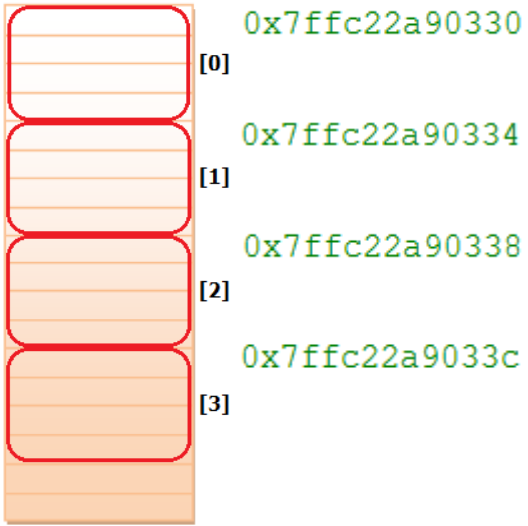
Output:

```
Address of 11 or 11 is : 0x7ffc22a90340 or 0x7ffc22a90340
Address of 23 or 23 is : 0x7ffc22a90344 or 0x7ffc22a90344
Address of 41 or 41 is : 0x7ffc22a90348 or 0x7ffc22a90348
Address of 17 or 17 is : 0x7ffc22a9034c or 0x7ffc22a9034c
```

Pointers vs. Arrays

Example:

```
float y[4] = {11.2, 23.3, 41.1, 17.0};
cout << "Address of " << y[0] << " or " << *y
     << " is : " << &y[0] << " or " << y << '\n';
cout << "Address of " << y[1] << " or " << *(y+1)
     << " is : " << &y[1] << " or " << y+1 << '\n';
cout << "Address of " << y[2] << " or " << *(y+2)
     << " is : " << &y[2] << " or " << y+2 << '\n';
cout << "Address of " << y[3] << " or " << *(y+3)
     << " is : " << &y[3] << " or " << y+3 << '\n';
```



Output:

```
Address of 11.2 or 11.2 is : 0x7ffc22a90330 or 0x7ffc22a90330
Address of 23.3 or 23.3 is : 0x7ffc22a90334 or 0x7ffc22a90334
Address of 41.1 or 41.1 is : 0x7ffc22a90338 or 0x7ffc22a90338
Address of 17.0 or 17.0 is : 0x7ffc22a9033c or 0x7ffc22a9033c
```

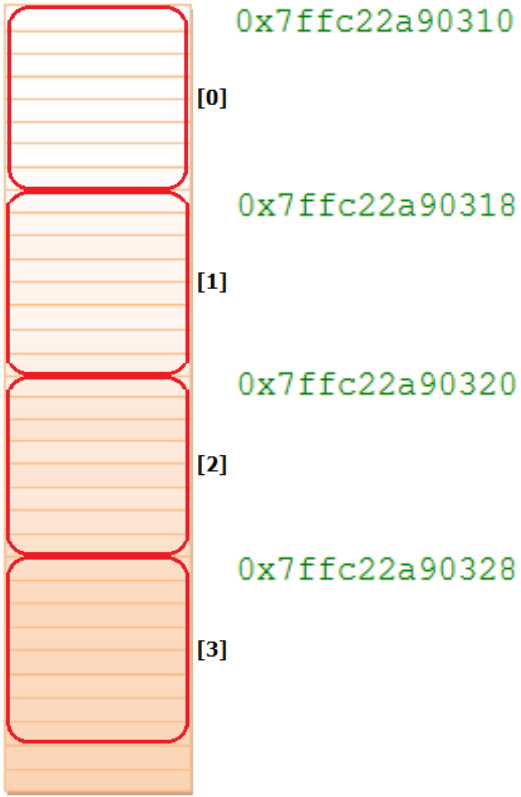
Pointers vs. Arrays

Example:

```
long long z[4] = {11, 23, 41, 17};
cout << "Address of " << z[0] << " or " << *z
     << " is : " << &z[0] << " or " << z << '\n';
cout << "Address of " << z[1] << " or " << *(z+1)
     << " is : " << &z[1] << " or " << z+1 << '\n';
cout << "Address of " << z[2] << " or " << *(z+2)
     << " is : " << &z[2] << " or " << z+2 << '\n';
cout << "Address of " << z[3] << " or " << *(z+3)
     << " is : " << &z[3] << " or " << z+3 << '\n';
```

Output:

```
Address of 11 or 11 is : 0x7ffc22a90310 or 0x7ffc22a90310
Address of 23 or 23 is : 0x7ffc22a90318 or 0x7ffc22a90318
Address of 41 or 41 is : 0x7ffc22a90320 or 0x7ffc22a90320
Address of 17 or 17 is : 0x7ffc22a90328 or 0x7ffc22a90328
```



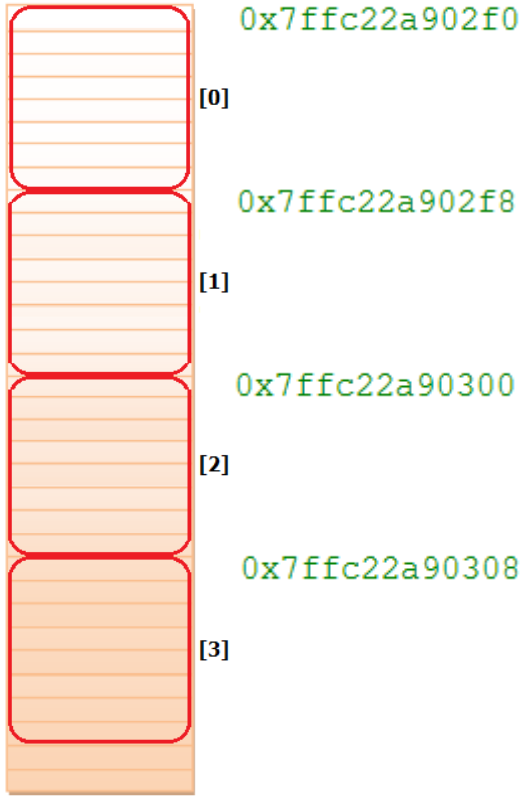
Pointers vs. Arrays

Example:

```
double w[4] = {11.2, 23.3, 41.1, 17.0};
cout << "Address of " << w[0] << " or " << *w
     << " is : " << &w[0] << " or " << w << '\n';
cout << "Address of " << w[1] << " or " << *(w+1)
     << " is : " << &w[1] << " or " << w+1 << '\n';
cout << "Address of " << w[2] << " or " << *(w+2)
     << " is : " << &w[2] << " or " << w+2 << '\n';
cout << "Address of " << w[3] << " or " << *(w+3)
     << " is : " << &w[3] << " or " << w+3 << '\n';
```

Output:

```
Address of 11.2 or 11.2 is : 0x7ffc22a902f0 or 0x7ffc22a902f0
Address of 23.3 or 23.3 is : 0x7ffc22a902f8 or 0x7ffc22a902f8
Address of 41.1 or 41.1 is : 0x7ffc22a90300 or 0x7ffc22a90300
Address of 17.0 or 17.0 is : 0x7ffc22a90308 or 0x7ffc22a90308
```



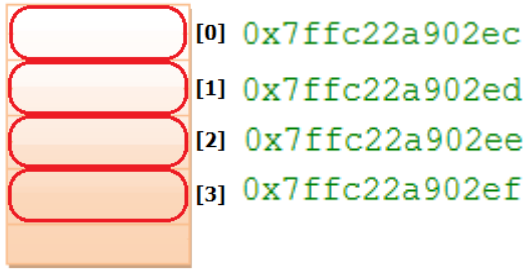
Pointers vs. Arrays

Example:

```
bool b[4] = {false, true, true, false};
cout << "Address of " << b[0] << " or " << *b
     << " is : " << &b[0] << " or " << b << '\n';
cout << "Address of " << b[1] << " or " << *(b+1)
     << " is : " << &b[1] << " or " << b+1 << '\n';
cout << "Address of " << b[2] << " or " << *(b+2)
     << " is : " << &b[2] << " or " << b+2 << '\n';
cout << "Address of " << b[3] << " or " << *(b+3)
     << " is : " << &b[3] << " or " << b+3 << '\n';
```

Output:

```
Address of 0 or 0 is : 0x7ffc22a902ec or 0x7ffc22a902ec
Address of 1 or 1 is : 0x7ffc22a902ed or 0x7ffc22a902ed
Address of 1 or 1 is : 0x7ffc22a902ee or 0x7ffc22a902ee
Address of 0 or 0 is : 0x7ffc22a902ef or 0x7ffc22a902ef
```



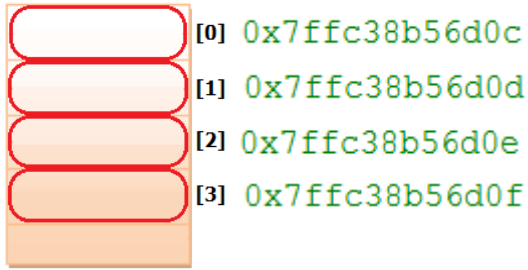
Pointers vs. Arrays

Example:

```
char s[4] = {'a', 'r', 'w', 'h'};
cout << "Address of " << s[0] << " or " << *s
     << " is : " << (void*) &s[0] << " or " << (void*) s << '\n';
cout << "Address of " << s[1] << " or " << *(s+1)
     << " is : " << (void*) &s[1] << " or " << (void*) (s+1) << '\n';
cout << "Address of " << s[2] << " or " << *(s+2)
     << " is : " << (void*) &s[2] << " or " << (void*) (s+2) << '\n';
cout << "Address of " << s[3] << " or " << *(s+3)
     << " is : " << (void*) &s[3] << " or " << (void*) (s+3) << '\n';
```

Output:

```
Address of a or a is : 0x7ffc38b56d0c or 0x7ffc38b56d0c
Address of r or r is : 0x7ffc38b56d0d or 0x7ffc38b56d0d
Address of w or w is : 0x7ffc38b56d0e or 0x7ffc38b56d0e
Address of h or h is : 0x7ffc38b56d0f or 0x7ffc38b56d0f
```



Lecture Agenda

- ✔ Section 1: Introduction to Pointers
- ✔ Section 2: Pointers vs. Arrays
- **Section 3: Pointer Arithmetic & Comparisons**
- Section 4: Array of Pointers
- Section 5: Pointer to a pointer
- Section 6: Reference Variables

Pointer Arithmetic & Comparisons

- We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array. The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type.

- Example

```
int x[4] = {11, 23, 41, 17};
int* p = x;
for (int i = 0 ; i < 4 ; i++) {
    cout << *p << ' ';
    p++;
}
cout << '\n';
```

```
int y[4] = {11, 23, 41, 17};
int* r = y+3;
for (int i = 0 ; i < 4 ; i++) {
    cout << *r << ' ';
    r--;
}
cout << '\n';
```

- Output

```
11 23 41 17
```

```
17 41 23 11
```

Pointer Arithmetic & Comparisons

- Pointers may be compared by using relational operators, such as ==, !=, <, <=, >, >=
  If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

- Example

```
int x[4] = {11, 23, 41, 17};
int* p = &x[0];
while (p <= &x[3]) {
    cout << *p << ' ' << p << '\n';
    p++;
}
cout << '\n';
```

```
int y[4] = {11, 23, 41, 17};
int* r = &y[3];
while (r >= &y[0]) {
    cout << *r << ' ' << r << '\n';
    r--;
}
cout << '\n';
```

- Output

```
11 0x7ffd7e962a70
23 0x7ffd7e962a74
41 0x7ffd7e962a78
17 0x7ffd7e962a7c
```

```
17 0x7ffd7e962a7c
41 0x7ffd7e962a78
23 0x7ffd7e962a74
11 0x7ffd7e962a70
```

Lecture Agenda

- ✔ Section 1: Introduction to Pointers
- ✔ Section 2: Pointers vs. Arrays
- ✔ Section 3: Pointer Arithmetic & Comparisons
- **Section 4: Array of Pointers**
- Section 5: Pointer to a pointer
- Section 6: Reference Variables

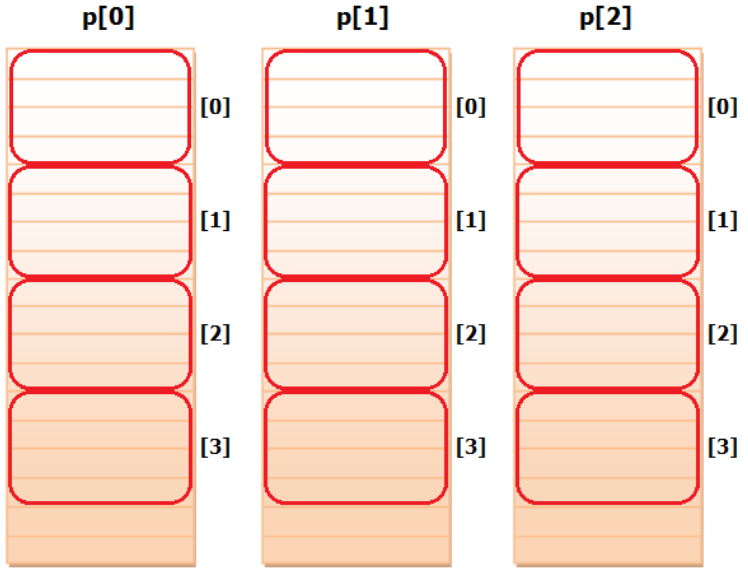
Array of Pointers

- When we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer. This declares ptr as an array of 3 integer pointers. Thus, each element in ptr, now holds a pointer to an int value. Following example makes use of three integers which will be stored in an array of pointers as follows:

Example:

```
int x[4] = {23, 11, 12, 19};
int y[4] = {41, 16, 59, 47};
int z[4] = {32, 27, 15, 13};

int* p[3] = {x, y, z};
```



Array of Pointers

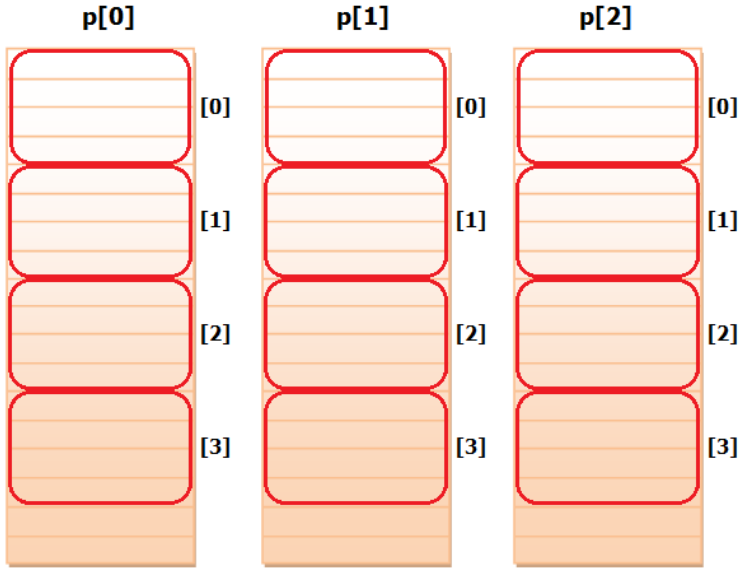
Example:

```
int x[4] = {23, 11, 12, 19};
int y[4] = {41, 16, 59, 47};
int z[4] = {32, 27, 15, 13};

int* p[3] = {x, y, z};
for (int i = 0 ; i < 3 ; i++) {
    int* it = p[i];
    while (it < p[i]+4) {
        cout << *it << ' ';
        it++;
    }
    cout << '\n';
}
```

Output:

```
23 11 12 19
41 16 59 47
32 27 15 13
```



Array of Pointers

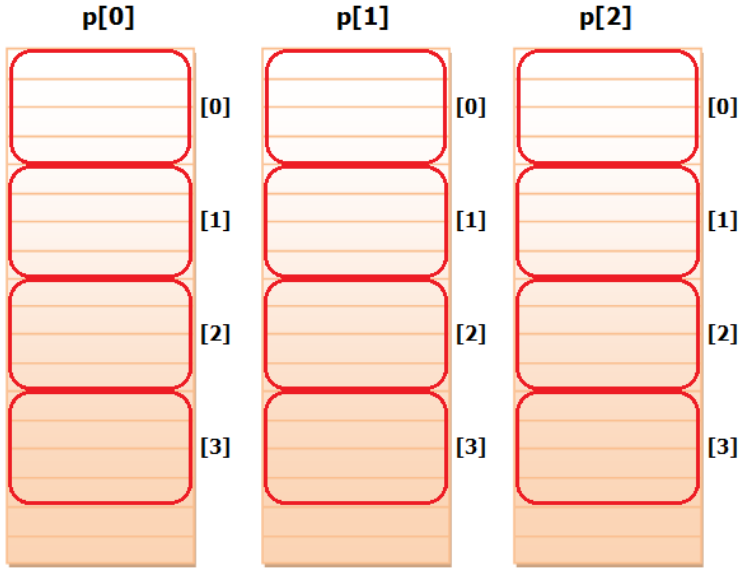
Example:

```
int x[4] = {23, 11, 12, 19};
int y[4] = {41, 16, 59, 47};
int z[4] = {32, 27, 15, 13};

int* p[3] = {x, y, z};
for (int i = 0 ; i < 3 ; i++) {
    int* it = p[i]+3;
    while (it >= p[i]) {
        cout << *it << ' ';
        it--;
    }
    cout << '\n';
}
```

Output:

```
19 12 11 23
47 59 16 41
13 15 27 32
```



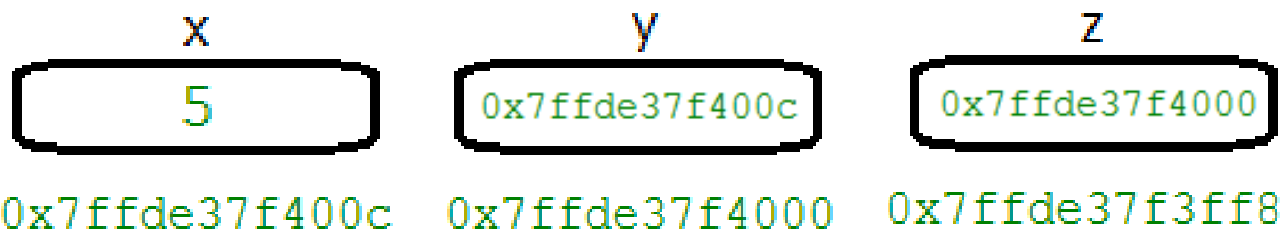
Lecture Agenda

- ✔ Section 1: Introduction to Pointers
- ✔ Section 2: Pointers vs. Arrays
- ✔ Section 3: Pointer Arithmetic & Comparisons
- ✔ Section 4: Array of Pointers
- **Section 5: Pointer to a pointer**
- Section 6: Reference Variables

Pointer to a pointer

- A pointer to a pointer is a form of multiple indirection or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.
- A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int: When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:

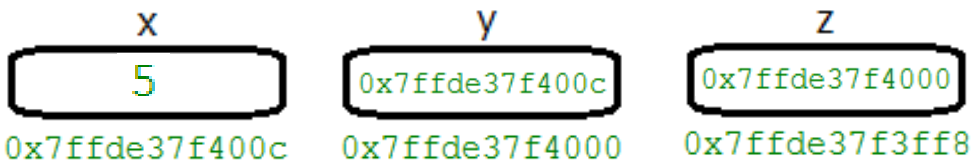
```
int x = 5;
int* y = &x;
int** z = &y;
```



Pointer to a pointer

Example

```
int x = 5;
int* y = &x;
int** z = &y;
```



Output

```
cout << "Value                  of x : " << x << '\n';          : 5
cout << "Address                of x : " << &x << '\n';         : 0x7ffde37f400c

cout << "Value                  of y : " << y << '\n';          : 0x7ffde37f400c
cout << "Address                of y : " << &y << '\n';         : 0x7ffde37f4000
cout << "Pointer                of y : " << *y << '\n';         : 5

cout << "Value                  of z : " << z << '\n';          : 0x7ffde37f4000
cout << "Address                of z : " << &z << '\n';         : 0x7ffde37f3ff8
cout << "Pointer                of z : " << *z << '\n';         : 0x7ffde37f400c
cout << "Address of Pointer of z : " << &*z << '\n';           : 0x7ffde37f4000
cout << "Pointer of Pointer of z : " << **z << '\n';           : 5
```

Lecture Agenda

- ✔ Section 1: Introduction to Pointers
- ✔ Section 2: Pointers vs. Arrays
- ✔ Section 3: Pointer Arithmetic & Comparisons
- ✔ Section 4: Array of Pointers
- ✔ Section 5: Pointer to a pointer

Section 6: Reference Variables

Reference Variables

- A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.
- References are often confused with pointers but three major differences between references and pointers are:

- You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.
- Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
- A reference must be initialized when it is created. Pointers can be initialized at any time.

- Think of a variable name as a label attached to the variable's location in memory. You can then think of a reference as a second label attached to that memory location. Therefore, you can access the contents of the variable through either the original variable name or the reference.
- When a variable is declared as reference, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting '&' in the declaration.

Reference Variables

Example:

```
int x;
int &y = x;
x = 5;
cout << "Value of x : " << x << " , Address of x : " << &x << '\n';
cout << "Value of y : " << y << " , Address of y : " << &y << '\n';
y = 3;
cout << "Value of x : " << x << " , Address of x : " << &x << '\n';
cout << "Value of y : " << y << " , Address of y : " << &y << '\n';
```

Output:

```
Value of x : 5 , Address of x : 0x7ffe34185ffc
Value of y : 5 , Address of y : 0x7ffe34185ffc
Value of x : 3 , Address of x : 0x7ffe34185ffc
Value of y : 3 , Address of y : 0x7ffe34185ffc
```

Reference Variables

➢ References vs. Pointers:

- Both references and pointers can be used to change local variables of one function inside another function. Both of them can also be used to save copying of big objects when passed as arguments to functions or returned from functions, to get efficiency gain.
- Despite above similarities, there are following differences between references and pointers. A pointer can be declared as void but a reference can never be void.

➢ References are less powerful than pointers:

1) Once a reference is created, it cannot be later made to reference another object; it cannot be reseated. This is often done with pointers.

2) References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.

3) A reference must be initialized when declared. There is no such restriction with pointers.

- Due to the above limitations, references in C++ cannot be used for implementing data structures like Linked List, Tree, etc.

Lecture Agenda

- ✓ Section 1: Introduction to Pointers
- ✓ Section 2: Pointers vs. Arrays
- ✓ Section 3: Pointer Arithmetic & Comparisons
- ✓ Section 4: Array of Pointers
- ✓ Section 5: Pointer to a pointer
- ✓ Section 6: Reference Variables


DO
MORE.