C++. POINTERS AND DYNAMIC ARRAYS

MIKHAIL MARKOV

DEPARTMENT OF COMPUTING AND SOFTWARE
MCMASTER UNIVERSITY

SFWRENG/COMPSCI 2S03 (FALL 2015)

TUTORIAL 10

Contacts

- **Graduate Teaching Assistants**:

Akhil Krishnan: akhil4490@gmail.com

Michael Liut: liutm@mcmaster.ca

Mikhail Markov: markoma@mcmaster.ca

- **Undergraduate Teaching Assistants**:

Jemar Jones: jonesjk@mcmaster.ca

Wenqiang Chen: chenw25@mcmaster.ca

Outline

- Pointer
- Address-of operator (&)
- Address-of operator (&): example
- Dereference operator (*)
- Dereference operator (*): example
- Declaration of the pointers
- Declaration and reassigning the pointers: examples
- Call by Reference vs Call by Pointer
- Pointers and Arrays
- Pointers and Arrays: example
- Pointers arithmetics
- Pointers arithmetics: example
- Const pointers
- Void pointers
- Pointers to pointers
- Dynamic memory

Pointer

- A **pointer** is a programming language object, whose value **refers to** (or "**points to**") another value stored elsewhere in the computer memory using its address*

* https://en.wikipedia.org/wiki/Pointer_(computer_programming)

Address-of operator (&)

x = **&** var1

- means that x takes the value of the **address** of the variable var1

We don't know the address of the variable preliminary as it's assigned by the operating system in the runtime.

Address-of operator (&): example

Memory Address	Value
1000	
1001	1
1002	
1003	hello
1004	^^%$
1005	

var1 = 103;

Memory Address	Value	Variable
1000		
1001		
1002	**103**	**var1**
1003	hello	
1004	^^%$	
1005		

x1 = &var1;

x2 = var1;

What are the values of the x1 and x2?

x1 = 1002

x2 = 103

The ***variable that stores the address of another variable*** (x1) is a **pointer.**

Pointers "**point to**" the variable whose address they store.

Dereference operator (*)

y = * x1

- means that y takes the **value pointed to by** the variable x1

i.e. if the variable x1 contains the address of the variable z (x1 = &z), y will take the value of the z

Dereference operator (*): example

Memory Address	Value
1000	
1001	1
1002	
1003	hello
1004	^^%$
1005	

var1 = 103;

x1 = &var1;

x2 = var1;

Memory Address	Value	Variable
1000		
1001		
1002	103	var1
1003	hello	
1004	^^%$	
1005		

y = * x1;

What are the values of the x1 and y?

x1 = 1002

y = 103

Declaration of the pointers

type * name;

- ***type*** is the **data type pointed to by the pointer**. This type is *not the type of the pointer itself*!

i.e. for

*int * numberInt; double * numberDouble; char * character1;*

the space in the memory occupied by each of these pointers most likely will be the same. But the data to which they point to, occupy different amount of space!

Declaration of the pointers

- NOTE that **asterisk** (*) used when **declaring a pointer** (*int * numberInt;*) and **dereference operator** (*) are different things represented with the same sign!

*When **declaring a pointer** it **only means** that it is a **pointer** (it is part of its type compound specifier but not a dereference).*

Declaration and reassigning the pointers: example

```
1  #include <iostream>
2  using namespace std;
3
4  int main ()
5  {
6    int x, y;
7  // Declaration of the pointer
8    int * point;
9  // Assigning the pointer to x
10   point = &x;
11 // Assigning 1 to the value pointed to by point (x)
12   *point = 1;
13 // Reassigning the pointer to y
14   point = &y;
15 // Assigning 999 to the value pointed to by point (y)
16   *point = 999;
17
18   cout << "point = " << point << endl;
19   cout << "x = " << x << endl;
20   cout << "y = " << y << endl;
21
22   return 0;
23 }
```

Result:

```
point = 0x7fff53b43c74
x = 1
y = 999
```

Declaration and reassigning the pointers: example

```cpp
#include <iostream>
using namespace std;

int main ()
{
  int x = 1, y = 2, z = 3;
// Declaration of the pointers p1, p2, p3 and p4
  int * p1, * p2, * p3, * p4;
// Pointers p1, p2 and p3 points to the x, y and z respectively
  p1 = &x;
  p2 = &y;
  p3 = &z;
// Assigning 99 and 77 to the values pointed to by p1 (x = 99) and p2 (y = 77)
  *p1 = 99;
  *p2 = 77;
// Assigning the value pointed to by p2 (y = 77) to the value pointed to by p3 (z = 77)
  *p3 = *p2;
// Pointer p4 now points to the same value as p2, i.e. to (y)
  p4 = p2;
// Assigning 5 to the value pointed to by p4, i.e. (y = 5)
  *p4 = 5;

  cout << "x = " << x << endl;
  cout << "y = " << y << endl;
  cout << "z = " << z << endl;

  return 0;
}
```

Result:

```
x = 99
y = 5
z = 77
```

Call by Reference vs Call by Pointer

	Call by Reference	**Call by Pointer**
Point / refer to nowhere (to NULL)	NO (should always refer to the valid variable)	YES (can be empty - NULL or can point to the invalid memory location)
Reassigning	NO (can be bind only one time at initialization)	YES (can be reassigned any number of times)
Access to the memory location	DIRECTLY (value = ref1)	DEREFERENCE (*) (value = *point1)

Pointers and Arrays

- an **array can be implicitly converted to the pointer of the proper type**.

```
int array1 [10];

int * p1;

p1 = array1;
```

- Pointers and arrays support the **same set of operations**. But pointers can be assigned new addresses, while arrays cannot. Arrays always represent the same block of elements of the concrete type.

Pointers and Arrays: example

```
1  #include <iostream>
2  using namespace std;
3
4  int main ()
5  {
6  // Declarations of the array and the pointer
7    int array1[6];
8    int * p1;
9  // Now pointer p1 points to the first element of the array1
10 // i.e. it's equal to the string p1 = &array1[0];
11   p1 = array1;
12 // Assigning 1 to the value pointed to by p1, i.e. array1[0] = 1
13   *p1 = 1;
14 // Go to the next memory cell and assign 2, i.e. array1[1] = 2
15   p1++; *p1 = 2;
16 // Now pointer points to the 3rd element of the array.
17   p1 = &array1[2];
18 // Assigning 3 to the value pointed to by p1, i.e. array1[2] = 1
19   *p1 = 3;
20 // Next string is equal to p1 = &array1[0+3]. Assigning 4.
21   p1 = array1 + 3;
22   *p1 = 4;
23 // Pointer p1 again points to the 1st element of the array
24   p1 = array1;
25 // Assigning 5 to the value pointed by the p1 shifted by 4 memory cells
26 // i.e. it's equal to the string p1 = &array1[4];
27   *(p1+4) = 5;
28
29   for (int i=0; i<5; i++) cout << array1[i] << "; ";
30   return 0;
31 }
```

Result:

```
1; 2; 3; 4; 5;
```

Pointers arithmetics

- only **addition** and **subtraction** operations are allowed

- The important moment with the pointers arithmetic - remember the size of the data types. For example, int is 4 bytes, char is 1 byte, double is 8 bytes, etc *.

* https://msdn.microsoft.com/en-CA/library/s3f49ktz.aspx

Pointers arithmetics

- ***p++**

same as *(p++): increment pointer, and dereference unincremented address

- ***++p**

same as *(++p): increment pointer, and dereference incremented address

- **++*p**

same as ++(*p): dereference pointer, and increment the value it points to

- **(*p)++**

dereference pointer, and post-increment the value it points to

Pointers arithmetics: example

Memory Address	
1000	<-
1001	
1002	
1003	
1004	
1005	
1006	
1007	
1008	
1009	
1010	
1011	
1012	
1013	
1014	

```
char * p1;

int * p2;

double * p2;
```

Let's assume that all of these pointers points to the same address 1000 (it's from 3 sequential runs of the program).

Where these pointers will point after?

```
++p1;

++p2;

++p3;
```

p1 will point to the value at the address 1001 (plus 1 byte); p2 to 1004 (plus 4 bytes); p3 to 1008 (plus 8 bytes)

Const pointers

- To declare pointers that can **access the pointed value to read it, but not to modify it**.
- For this, it is enough with qualifying the type pointed to by the pointer as const.

```
int x =0;

int y = 1;

const int * p1 = &y;

x = *p1;          // works

*p1 = x;          // doesn't work
```

Void pointers

- **void pointers** are pointers that **point to** a **value** that **has no type** (undetermined length and dereferencing properties).

- void pointers can **point to any data type**, BUT the **data pointed to by them cannot be directly dereferenced** (since we have no type to dereference to). Any address in a void pointer needs to be transformed into some other pointer type that points to a concrete data type before being dereferenced.

Pointers to pointers

- To declare pointers to pointers use an asterisk (*) for each level of indirection in the declaration of the pointer, for example:

```
char p1;

char * p2;

char ** p3;

p1 = 'q';

p2 = &p1;

p3 = &p2;
```

Memory Address	Value	Variable
1000		
1001	q	p1
1002		
1003	1001	p2
1004		
1005	1003	p3

Dynamic memory

- Dynamic memory is allocated using operator **new**. It returns a pointer to the beginning of the new block of memory allocated.

pointer = **new** type

pointer = **new** type [number_of_elements]

- If the memory is not needed anymore we can free it using the operator **delete**.

delete pointer [number_of_elements]

References

https://en.wikipedia.org/wiki/Pointer_(computer_programming)
http://www.cplusplus.com/doc/tutorial/pointers/
https://msdn.microsoft.com/en-CA/library/s3f49ktz.aspx