Pointers*

Nabil M. Al-Rousan
nabil@ece.ubc.ca

September 22, 2020

Objectives

In this handout, you will learn

- What pointers are
- How to use pointers
- Why we need of pointers

What is a pointer?

First, remember that a *variable* is defined as a portion of memory that stores a value. A *pointer* is a variable that stores the address of another variable [1]. It points at other variable; hence the name 'Pointer'.

How to use pointers?

An address of a variable can be obtained by preceding the name of a variable with Address-of operator (&).
We can get value pointed to by a pointer *directly* using the dereference operator (*) This is done by preceding the pointer name with (*).

The following code demonstrates how the two operators are used:

```
1  // 1) Since there is * in declaration, ptr
2  // becomes a pointer varaible (a variable
3  // that stores address of another variable)
4  // 2) Since there is int before *, ptr is
5  // pointer to an integer type variable
6  int *ptr; // Pointer Declaration
7
8  int x; // Declare variable x.
9         // Suppose the address of x is A1
10
11
12 // & operator before x is used to get address
13 // of x. The address of x is assigned to ptr.
14 ptr = &x; // ptr now points to x
15            // Suppose the address of ptr is F9
16 x = 10;
17 *ptr = 12; // stores 12 into int location
18           // pointed at by ptr
19
20 out<< *ptr    << x
21     << ptr     << &x
22     << &ptr;
```

Output

```
12 12 A1 A1 F9
```

Edit & Run

*SAMS Teach Yourself C++
[1] http://www.cplusplus.com/doc/tutorial/pointers/

Why Pointers?

0.1 Dynamic Memory: to allocate variable size memory at runtime using the heap.

```
1  int * x;
2  x = new int [5];
3  *(x+4) = 2;
4  cout<< x[4];
5  delete[] pointer;
```

Output

2

Edit & Run

0.2 Call by reference: Pass function parameters by reference[2]

When reference variables are used as formal parameters, this is known as Call/Pass By Reference.
Comparing: Value vs. Reference:

- Pass By Value
  - The local parameters are copies of the original arguments passed in
  - Changes made in the function to these variables do not affect originals
- Pass By Reference
  - The local parameters are references to the storage locations of the original arguments passed in.
  - Changes to these variables in the function will affect the originals
  - No copy is made, so overhead of copying (time, storage) is saved

```
1  void f(int* p) {
2      *p = 10;
3  }
4
5  int main() {
6      int n = 0;
7      f(&n); // passes the address of n
8              // now the value of n is 10
9              cout<<n;
10     return 0;
11 }
```

Output

10

Edit & Run

[2]https://www.cs.fsu.edu/ myers/c++/notes/references.html

0.3 Construct Recursive Data Structures[3]

```
1  struct node
2  {
3    int data;
4    node *next;
5  };
6
7  // This function prints contents of linked list
8  // starting from the given node
9  void printList(node* n)
10 {
11     while (n != NULL) {
12         cout << n->data << " ";
13         n = n->next;
14     }
15 }
16
17 int main ()
18 {
19   node *head = NULL;
20   node *second = NULL;
21   node *third = NULL;
22
23   // allocate 3 nodes in the heap
24   head = new node ();
25   second = new node ();
26   third = new node ();
27
28   head->data = 1; // assign data in first node
29   head->next = second; // Link first node with
30
31   second->data = 2; // assign data to second node
32
33   // Link second node with the third node
34   second->next = third;
35
36   third->data = 3;      // assign data to third node
37   third->next = NULL;
38
39   printList(head);
40
41   return 0;
42 }
```

Output

```
1 2 3
```

Edit & Run

Erroneous Usage of Pointers

Listing 1: Pointer of type `int` should point at variable of type `int`

```
1  int *p = 5; // Error
```

Listing 2: Address can not be assigned to dereferenced pointer or visa versa

```
1  int x, y;
2  int * ptr1, *ptr2;
3  ptr1 = &x;
4  ptr2 = &y;
5  *ptr2 = ptr1; // Error
6  ptr1 = *ptr2; // Error
```

[3]The arrow operator (->) is a dereference operator that is used exclusively with pointers to objects that have members.

Listing 3: Integer can not be assigned to a pointer

```
int x;
int *p;
p = x; // Error
```

Listing 4: Address of a pointer can not be assigned to another pointer

```
int x, y;
int * ptr1, *ptr2;
ptr1 = &x;
ptr2 = &y;
ptr1 = &ptr2; // Error
```

Listing 5: Address can not be assigned dereferenced pointer

```
int x;
int *p;
*p = &x; // Error
```

Listing 6: Pointers must be initialized before they can be used.

```

int *p;
cout <<*p; // Error
```

CheatSheet[4][5]

Table 1: Pointers, Parenthesis, and Math

Pointer Expression	Memory Address	Memory Contents
`*p`	Yep	Nope
`**p`	Nope	Yep
`**p++`	Incremented after value is read	Unchanged
`**(p++)`	Incremented after value is read	Unchanged
`*(*p)++`	Unchanged	Incremented after it's used
`*++p`	Incremented before value is read	Unchanged
`**(++p)`	Incremented before value is read	Unchanged
`*++*p`	Unchanged	Incremented before it's used
`*++(*p)`	Unchanged	Incremented before it's used
`p*++`	Not a pointer	Not a pointer
`p++*`	Not a pointer	Not a pointer

Table 2: Pointers and array brackets

Array Notation	Pointer Equivalent
`array[0]`	`*a`
`array[1]`	`*(a+1)`
`array[2]`	`*(a+2)`
`array[3]`	`*(a+3)`
`array[4]`	`*(a+4)`

Table 3: Pointers and Multidimensional Arrays
Consider pointer notation for the two-dimensional numeric arrays. consider the following declaration
`int nums[2][3] = { {16, 18, 20}, {25, 26, 27} };`

Pointer Notation	Array Notation	Value
`*(*nums)`	`nums[0][0]`	16
`*(*nums + 1)`	`nums[0][1]`	18
`*(*nums + 2)`	`nums[0][2]`	20
`*(*(nums + 1))`	`nums[1][0]`	25
`*(*(nums + 1)+ 1)`	`nums[1][1]`	26
`*(*(nums + 1)+ 2)`	`nums[1][2]`	27

[4]https://c-for-dummies.com/caio/pointer-cheatsheet.php
[5]https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/