CSCI 2270 – Data Structures

Recitation 2, Spring 2021

Objectives

1. Variables
2. Pointers
3. Address-of Operator (&)
4. Dereferencing Operator
5. Structs
6. Pointer to structs.
7. Pass-by-value vs Pass-by-pointer vs Pass-by-reference

1. Variables

A Variable is a location in a computer memory used by the program which can be accessed by their name. This avoids the need for the programmer to remember the physical address of the data in memory and instead use the variable to refer to the data when needed.

For example, in the below example, we see that we are storing 10 in a variable named **a,** and storing 20 in a variable named **b**. The program has no reference to any physical memory address, and instead, all operations on the data 10, and 20 is done using the variables **a,** and **b.**

```
#include<iostream>

using namespace std;

int main() {
    int a = 10;
    int b = 20;
    cout << a + b << endl;
}
```

Note that the code when executed on multiple machines, or at different points of time, allocates different physical memory addresses. The beauty of variables is that YOU as a

programmer do not have to remember the locations of the data. You can focus on the code-logic instead.

2. Pointers

To understand pointers, we investigate how a C++ program operates on a Computer's memory. For a C++ program, a computer memory is like a succession of memory cells, each one byte in size, and each with a unique address (hexa-decimal). These single byte memory cells are ordered such that datatypes with larger memory footprint (integer, float, long etc..) occupy memory cells that have consecutive addresses.

This way, an integer (4 bytes) can be stored in contiguous memory addresses starting from 1330 to 1333. And the unique memory address for this integer is identified as 1330.

For an array of integers of size 10 stored in contiguous memory addresses starting from 1401 to 1440, the array location is easily identified by its unique memory address 1401. The 0^{th} indexed element of the array is stored in the memory block 1401-1404, the 1^{st} indexed element in 1405-1408 and so on. It is here we need Pointers.

A pointer is a variable which stores a memory address of a variable. In the above example of an array, a pointer can be declared to store the address of the array (1401). Using this address, the C++ program can access the contents of the array by calculating the relative position from the starting address (1401).

3. Address-of Operator (&)

For the curious, you can get the physical memory address location of any variable by prefixing the variable with an ampersand (&) operator. This is called an *Address-of or Referencing operator*. The address as shown in the output is in a hexadecimal format.

Filename: address_of.cpp

```cpp
#include<iostream>

using namespace std;

int main() {
    int a = 10;
    cout << a << endl;
    cout << &a << endl;
}
```

Output

```
10
0x7ffffddca5c4
```

3. Dereferencing operator (*)

As we have mentioned in the section “Pointers”, they are just variables which are used to store the address of other variables. Colloquially, the Pointers are said to “point to” the variable whose address they store.

It gets its name as Pointers can be used to directly access the variable whose address it points to. This is done by prefixing the pointer variable with the ***dereferencing operator*** (*).

Pointer variables are declared differently compared to regular variables. We know that we can fetch the address of a variable by prefixing it with an ampersand (&), and we have now learnt that the pointers store the addresses of a variable. The code below shows how the address of **a** is stored in a pointer variable **p,** and **q**.

For the keen eye, one can notice that both the pointers **p** and **q** “point to” the address of the same variable **a**. Thus, it is possible to have multiple pointers “point to” the SAME variable.

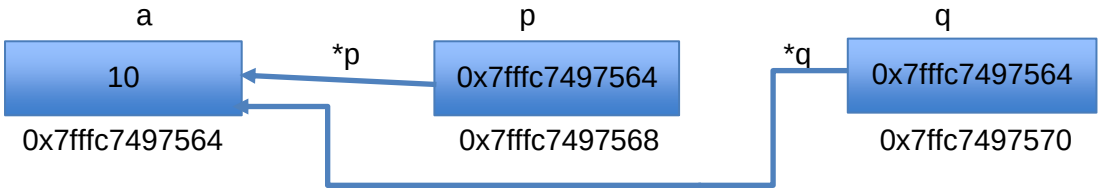
```
#include <iostream>
using namespace std;

int main()
{
    int a = 10;
    // * here denote p is pointer variable
    int *p;
    int *q;
    p = &a;
    q = &a;
    cout<< a <<endl;
    cout<< p <<endl;
    cout << q << endl;
    //* is used to dereference p
    cout << "Address of the pointer p: " << &p << endl;
    cout << "Address stored in pointer p: " << p << endl;
    cout << "The value of the pointer p is pointing to: " << *p << endl;
    cout << "Address of the pointer q: " << &q << endl;
    return 0;
}
```

Output

```
10
0x7fffc7497564
0x7fffc7497564
Address of the pointer p: 0x7fffc7497568
Address stored in pointer p: 0x7fffc7497564
The value of the pointer p is pointing to: 10
Address of the pointer q: 0x7fffc7497570
```

For better understanding, visual representation of the above program is shown below. The data within the box is the data stored in the specific memory location. The variables **a**, **p**, and **q** on accessing provides the data within the memory, and the characters below the boxes represent the physical memory address location.



4. Structs

Sometimes, to represent a real-world object, simple datatypes are not enough. To represent a Student, we need some (if not all) of the following: age, gender, date-of-birth, name, address etc.

You could write code as follows:

```
std::string name;
std::string email;
int birthday;
std::string address;
```

This becomes increasingly complex if we would like to store multiple Student representations. One way to solve this is by using aggregated(grouped) user-defined datatype called **Struct**. This datatype can hold different datatypes grouped under a common variable of type Struct.

```
struct Student {
    std::string name;
    std::string email;
    int birthday;
    std::string address;
};
```

5. Pointer to struct.

Pointers apply not only to simple datatypes, but aggregated datatypes as well.
We can have addresses (pointers) to any type of variable, even for structures. The following code shows how pointers can point to an aggregated datatype, as well as ways to assign values to struct and accessing them directly, as well as through a pointer.

Notice the **cout** statements. To access members of the struct variable we use **dot (.)** operator, and to access the members via a pointer variable, we use **-> (dash followed by a greater than sign)**.

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

struct Distance
{
    int feet;
    int inch;
};

int main()
{
    Distance d;
    // declare a pointer to Distance variable
    Distance* ptr;
    d.feet=8;
    d.inch=6;

    //store the address of d in p
    ptr = &d;
    cout << "Variable access" << endl;
    cout << "Distance= "<< d.feet << "ft "<< d.inch << "inches" << endl;
    cout << "Pointer access" << endl;
    cout << "Distance= " << ptr->feet << "ft " << ptr->inch << "inches" << endl;
    return 0;
}
```

Output

```
Variable access: Distance= 8ft 6inches
Pointer access: Distance= 8ft 6inches
```

6. Pass-by-value vs Pass-by-pointer vs Pass-by-reference

When we pass a variable into a function, what we are passing is called **actual parameters** and where it is received (i.e., the function), they are called **formal parameters**. They are also called ***actual*** and ***formal arguments***. When a function is called, a chunk of memory, called **activation record,** is allocated. It is the place, where the formal arguments and local variables defined in the memory are held.

In ***pass by value***, a **copy** of the **actual arguments** is stored in the **activation record** as formal arguments. Thus, any manipulation of formal arguments in the function will not be reflected after the function returns/exits. The following code shows this behaviour.

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

void add2 (int num)
{
    num = num + 2;
    cout << num << endl;
}

int main ()
{
    int a = 10;
    add2(a);
    cout << a << endl;
}
```

Output

```
12
10
```

As you can see, the above function demonstrates pass-by-value. The manipulation of the variable **num** which is a *formal argument* is not reflected in the *actual argument* **a**. Note that this is the default behaviour.

What if we would like to modify the actual argument in a function or a client program?

Here, we use **pass-by-pointers** and **pass-by-reference**.

Pass-by-pointers

Consider the code below. Like **pass-by-value,** in **pass-by-reference** a copy of actual arguments is made in the **activation record** and stored as formal arguments. However, as the copy made is an address, and as we know that we can use pointers to access and manipulate the values in the datatype's address it is pointing to, the changes hold once the function exits/returns.

Notice the code, as we are passing the address of the variable **a**, the argument in the function **add2** must be a pointer.c

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

void add2 (int * num)
{
    *num = *num + 2;
}

int main ()
{
    int a = 10;
    add2(&a);
    cout << a << endl;
}
```

Output

```
12
```

Pass-by-reference

In pass-by-reference, a copy of the address of the actual parameter is stored in the formal parameter. This is achieved by passing the variable as is into the function but prefixing the function argument with an ampersand. This lets the compiler know that the function call is pass-by-reference.

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

void add2 (int &num)
{
    num = num + 2;
}

int main ()
{
    int a = 10 ;
    add2(a);
    cout << a << endl;
}
```

Output

```
12
```

Now examine the following code and try to find out why it is persisting the change?

```
#include<iostream>

using namespace std;

void add2 (int a[], int len)
{
    for (int i=0; i<len; i++)
    a[i]+= 2;
}

int main ()
{
    int a[] = { 1 , 2 , 3 };
    add2( a, 3 );
    for ( int i=0;i< 3;i++)
      cout << a[i] << endl ;
}
```

Is the previous one similar/different to the one given below?

```
#include<iostream>

using namespace std;

void add2(int *a, int len)
{
    for (int i=0; i<len; i++)
    a[i]+= 2;
}

int main ()
{
    int a[] = { 1 , 2 , 3 };
    add2(&a[0], 3);
    for (int i=0; i<3; i++)
      cout << a[i] << endl;
}
```

Can you tell, which one is pass-by-reference and which one is pass-by-pointer?

Quiz

1. Consider an array **int a[] = {1, 2, 3}.** What is the output for the following?
    a. cout << a+2;
    b. cout << *(a+2);
    c. cout << *a;
    d. cout << *a[0];
2. How come we can pass an array name as an argument to a function and still be able to persist the change?

Exercise

Your zipped folder for this Lab will have a main.cpp, swap.cpp and swap.h files. Follow the TODOs in these files to complete your Recitation exercise!

References

- https://courses.washington.edu/css342/zander/css332/passby.html
- https://www.cplusplus.com/doc/tutorial/pointers/