# Play with pointers

September 1, 2016

## 1 Memory

The memory layout in c++ is easier to understand than for instance Python. Think of the memory as a large block with bytes. Each memory address stores 8 bits (1 byte):

| Memory address | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... |
|---|---|---|---|---|---|---|---|---|---|
| Memory value | 10010011 | 10011100 | 00000000 | 11100110 | 01010101 | 11010101 | 00010000 | 01101010 | ... |

When we say large, we really mean large. One gigabyte (a typical computer today has several gigabytes) is billions of such bytes. When you create a variable in c++

```
int number = 5;
```

the variable *number* is placed somewhere in this huge block. It starts at a certain *memory address* and uses the next 32 bits. To get this memory address, we can use the *Address-of*-operator *&*

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

int main() {
        int number = 5;
        cout << "Memory address: " << &number << endl;
        return 0;
}
```

which outputs

```
Anderss-MacBook-Pro:Desktop anderhaf$ g++ -o testPointers testPointers.cpp; ./testPointers
Memory address: 0x7fff5aa85f08
```

This memory address is output in hexadecimal [1] which is just a 64 bit number. If we create another variable, setting it equal to the first, we get

```
int number = 5;
int number2 = number;
cout << "Memory address of number: " << &number << endl;
cout << "Memory address of number2: " << &number2 << endl;
```

the output is

```
Anderss-MacBook-Pro:Desktop anderhaf$ g++ -o testPointers testPointers.cpp; ./testPointers
Memory address of number: 0x7fff5206bef8
Memory address of number2: 0x7fff5206bef4
```

[1] Hexadecimal is base 16 numeral system. The digits are 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f. Read more at `http://vlaurie.com/computers2/Articles/hexed.htm`.

We can convert these to base 10 by casting them to long (which is a 64 bits integer)

```
int number = 5;
int number2 = number;
cout << "Memory address of number: " << (long)&number << endl;
cout << "Memory address of number2: " << (long)&number2 << endl;
```

which produces output

```
Anderss-MacBook-Pro:Desktop anderhaf$ g++ -o testPointers testPointers.cpp; ./testPointers
Memory address of number: 140734543548136
Memory address of number2: 140734543548132
```

Notice how the memory difference between these two numbers is 4. $140734543548136 - 140734543548132 = 4$. The datatype *int* is 32 bits (4 bytes), so this makes sense.

This is how the memory layout is. The data just lies in this huge block of bytes and the compiler knows how to use it. We can also use these memory addresses to do cool stuff. When we store memory addresses in a variable, we call these variables pointers.

## 2 Pointers

A pointer is just a variable containing the memory address to another variable. The data type is denoted by an asterix *

```
int number = 5;
int *address = &number; // Create an int pointer to int number
cout << "Memory address of number: " << &number << endl;
cout << "Pointer value: " << address << endl;
```

which produces output

```
Anderss-MacBook-Pro:Desktop anderhaf$ g++ -o testPointers testPointers.cpp; ./testPointers
Memory address of number: 0x7fff5df0def8
Pointer value: 0x7fff5df0def8
```

Notice how the two printed values are identical. This is because our new variable *pointer is* the address to the variable *number*. If we want to get the value from a pointer (what is stored in this address), we can use the *dereference*-operator (*)

```
int number = 5;
int *address = &number; // Store the address here

cout << "The value is: " << *address << endl;
```

which produces output

```
Anderss-MacBook-Pro:Desktop anderhaf$ g++ -o testPointers testPointers.cpp; ./testPointers
The value is: 5
```

The same also works for pointers to other data types, such as doubles

```
double number = 5.3;
double *address = &number; // Store the address here. This time it is double *

cout << "The value is: " << *address << endl;
```

which produces output

```
Anderss-MacBook-Pro:Desktop anderhaf$ g++ -o testPointers testPointers.cpp; ./testPointers
The value is: 5.3
```

If you wonder how many bytes a variable is, you can check with the *sizeof* function

```
int numberInt = 5;
long numberLong = 5;
cout << "sizeof(numberInt): " << sizeof(numberInt) << endl;
cout << "sizeof(numberLong): " << sizeof(numberLong) << endl;
```

which gives output

```
Anderss-MacBook-Pro:Desktop anderhaf$ g++ -o testPointers testPointers.cpp; ./testPointers
sizeof(numberInt): 4
sizeof(numberLong): 8
```

as we should expect since an int is 32 bits and long is 64 bits.

## 3 Arrays

When we create arrays in c++, we ask for enough contiguous memory to store the number of elements we need. What we get is a pointer to the first value. We can use this pointer as we did before (using the *dereference*-operator) to access the value to set it.

```
int *array = new int[10]; // Dynamically create an array with 10 slots
*array = 5; // Dereference and set the first value
```

If we want to set the next element, we can increase the pointer by one (not one byte, but as much we need to move to the next *int*, which would be 4 bytes)

```
int *array = new int[10]; // Dynamically create an array with 10 slots
*array = 5; // Dereference and set the first value
*(array+1) = 10; // Dereference the next address and set the second value
```

This is hard to read, so in c++ we can write code doing the exact same thing (this is usually how we access elements in array)

```
int *array = new int[10]; // Dynamically create an array with 10 slots
array[0] = 5;
array[1] = 10;
```