C++ PROGRAMMING

DR. YIHSIANG LIOW (OCTOBER 26, 2025)

Contents

26. Pointers

OBJECTIVES

- Declare pointers
- Assign values to pointers, address-of operator, and NULL
- Comparison of address values
- Use pointers as function parameters modify values
- Use pointers as function parameters speedup function call
- Returning references and pointers
- Dynamic memory manage for non-array values
- Memory deallocation and preventing memory leaks

Now we come to a concept that is extremely useful and powerful in CS.

Drumroll . . . *pointers!!!*

Pointers are variables that can hold physical memory addresses. By pointing a pointer to any memory location in your computer, you can access the data at that point in your computer's memory. Both data and code are stored in your computer memory. Therefore with pointers you can access any data and code. This has several consequences.

- Pointers allow you to write a function that allows pass-by- reference. In fact your references should really be thought of as pointers under the hood.
- In terms of data, pointers allow you to access a part of your program's memory called the free store or memory heap. This allows you to control memory usage and have better control over memory management, requesting memory only when you need it and releasing it back to the system when it's not needed.
- By embedding pointer values into data, we can create very complex relations between data values by linking them together. This allows you to build data structures to model trees, graphs, etc. Graphs are probably one of the most important mathematical structures in CS. A road network is basically a graph. A computer network is basically a graph. Social graph that describes person-to-person relationship in social media is basically a graph. Etc., etc., etc., etc., etc. . . . !!!
- Since code also lives in your computer's memory, pointers can also point to code such as functions. With pointers, you can

actually pass functions (technically speaking the memory location of functions) into functions, i.e., functions can become arguments.

- Nowadays, hardware devices are usually "mapped" to memory locations too. That means that pointers can also be used to access devices to perform I/O.
- Etc!!! ... not to mention bizarre things like code that rewrites itself.

All the reasons have to do with using pointers to achieve some computational goals. Another really important reason why pointers are important is this: Pointers are not just some abstract idea of computations in the theoretical sense. Pointers are fundamental to actual computer architecture in the sense that pointers or memory addresses are understood by your hardware. For many programming languages, you cannot access pointers or memory addresses directly. Those languages will hide pointers away from programmers by providing some language feature(s) to achieve the same goal, bypassing pointers and memory addresses. This is no big deal if all you want to do is to achieve the high level computational goals. However, if you do need to dive into the guts of your computer, you will have to know pointers very well. This also means that people who are trained only in a language that does not provide means to access pointer or memory addresses will have problems doing low-level programming or will have problems fully understanding areas such as assembly language programming and computer architecture. And low level programming does happen in the real world for instance in the case of systems programming and game development.

Because the landscape of pointers is huge let me tell you what's the plan

- First I will talk about general concepts related to pointers: memory addresses, pointer values, pointer operations, etc. Another basic concept you need to know is pointer arithmetic. But I'll delay pointer arithmetic till the second set of notes on pointers so that we can go into some applications of pointers quickly
- For the first application of pointers, we'll see that pointers allow us to directly access and manipulation data from anywhere. In particular, a function can access data that is in another function. You recall that this is exactly what references are used for. In fact your C++ compiler actually converts references to pointers.

So understanding pointers is absolutely crucial to understanding references. And if you know pointers well, you don't even need references. In fact in the original C language – the ancestor of C++ – there is no concept of references.

- The second application is similar to the first: we'll call functions with pointers (we'll also look at passing in reference) and possibly returning pointers (we'll also look at returning reference). In this case, the goal is to speed up function call.
- For the third application, we'll go into an area in your computer called the heap and perform dynamic memory management ourselves. (The heap is also called the free store.) This means that we can ask the heap to give us an *int* value when we need it and give the *int* value back to the heap when we're done with it. This is very different from using an *int* value that belongs to an *int* variable. For an *int* variable, the value exists when you declare the *int* variable. The value is given back to the computer when the variable goes out of scope. You don't have as much control over memory usage for your regular variables. To use the heap ... you must use pointers.

In the next set of notes on pointers, I'll talk about pointer arithmetic, the relationship between pointers and arrays, and dynamic memory management of (dynamic) arrays.

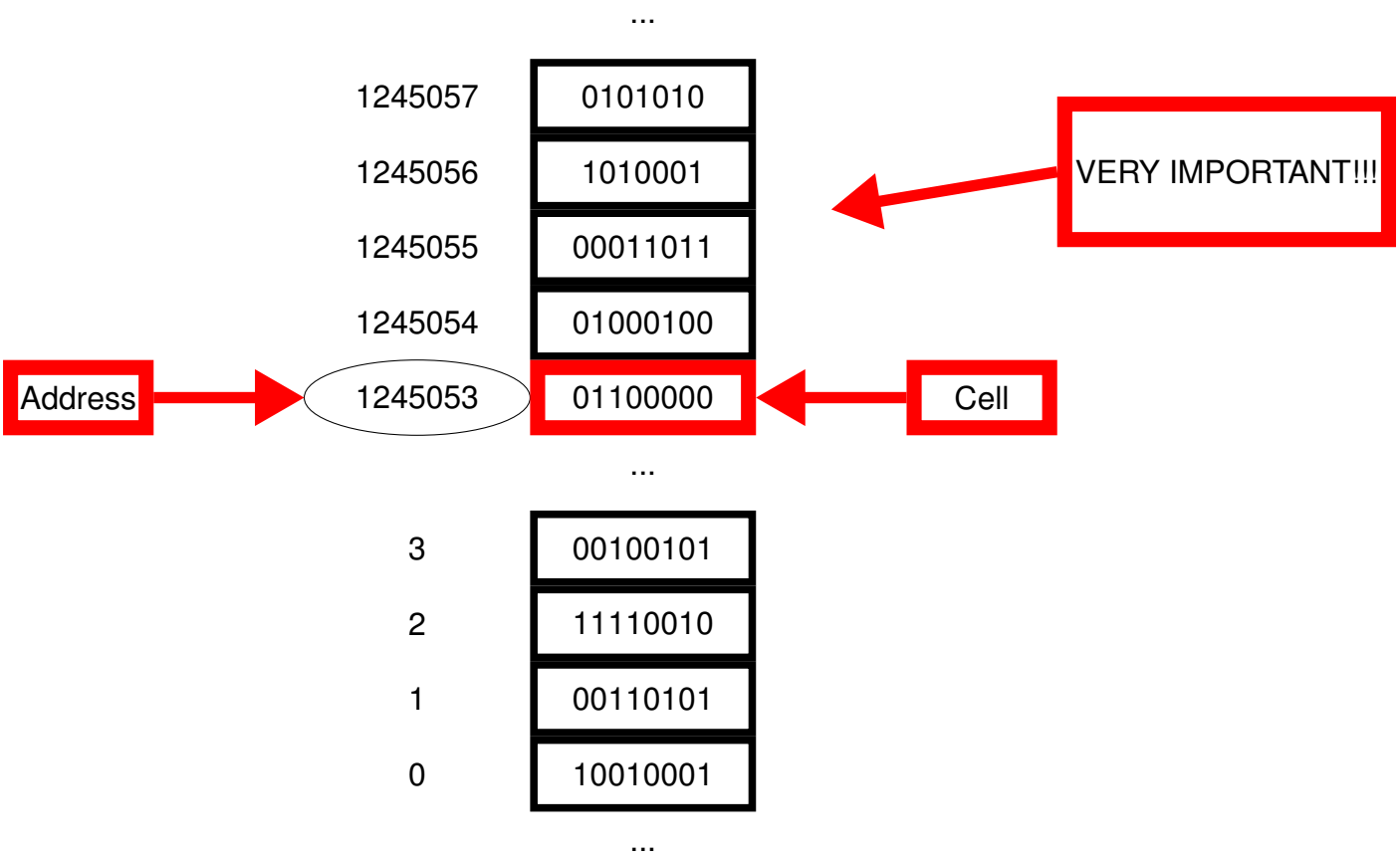
Here we go . . . !!!

Memory

So far the model of the space where variables live does not have any organization. It's just a collection of cells with names and values.

Now for a more accurate model . . .

- A computer's memory is made up of a linear collection of memory cells - each is a byte (i.e. 8 bits)
- Each memory cell has a numeric address, its memory address
- Each memory cell can hold 1 byte of data (1 byte = 8 bits)



Remember: Each memory cell has **two** quantities, the **memory address** and the **content.**

Notice that memory addresses are **integer values**.

When you declare a variable, based on the type, C++ will allocate

the correct number of bytes for that variable. For example, for an *int*, your

C++ (on a 32-bit machine) will allocate 4 bytes.

Also, when you declare two variables in the same scope, the memory used do not overlap – they occupy different memory spaces.

Memory addresses and memory address of a variable

In C++, if *x* is a variable, the **memory address** of *x* is ***&x***.

Try this:

```
int x = 123;

std::cout << "val of x = " << x << '\n'
          << "addr of x = " << \textbf{\&x} << '\n';
```

My output:

```
val of x is 123

addr of x is 1245052
```

(Your output is most probably going to be different from mine.) The **&** in the above is called the **address-of operator**.

Two things you must know right away . . .

First, technically speaking *&x* is the **address of the *value* of *x***. But I will sometimes call *&x* the **address of*x*** since that's the common practice.

Second, in reality, an *int* value takes up more than 1 byte. For a 32-bit machine, the value of *x* is spread out among 4 bytes: Recall that to see how many bytes are used to store an integer value you can do this:

```
std::cout << sizeof(int)
          << std::endl;
```

So in the above example when the program says that the address of *x* is 1245052, it means that the *int* value 123 occupies the memory at addresses 1245052, 1245053, 1245054, 1245055. The **address of *x*** actually refers to the **starting address**, i.e., address of the first byte, i.e. 1245052.

You don't have to worry about how the value 123 is "spread out" among 4 bytes. But briefly, the integer 123 (to human beings) is a bunch of bits (to a 32-bit computer):

00000000000000000000000001111011

(Details will be covered in CISS360.) So here's where you will find the value of our *x* from above:

Notice that 123 is translated to 32 bits. Each byte can hold 8 bits. That's why the bits of an integer requires 4 bytes. At this point, we don't need to worry how the 123 is translated into bits. So I will usually simplify the above picture of your computer's memory by drawing this:

Let me summarize ...

- A computer's memory is made up of bytes.
- Associated to each byte is its address and its contents (value).
- The value of a variable occupies memory, possibly more than 1 byte.
- If *x* is a variable, then in C++
- *x* refers to the contents (value)
- *&x* refers to the beginning address of its value, i.e., the address of the first byte of its value

By the way the address printed when you run your program might contain things like a, c, e, etc. That's because the address is actually printed in **hexadecimals**, i.e., **numbers in base 16**. (You will also learn more about hexadecimals in CISS360.) Certain compilers allow you to typecast hexadecimal values into *int* value. Try the following (don't panic and call 911 if your compiler won't let you):

```
int x = 123;

std::cout << "val of x: " << x
          << '\n' << "addr of x: " << \textbf{int(\&x)}
          << '\n';
```

If it does not work, you can use any scientific calculator to convert from hex to decimals yourself. Most scientific calculators nowadays have the ability to convert between base 10 to base 2, 8, and 16.

In almost all programming languages, a hexadecimal number starts with 0x. So you might see something like this when working with address values:

0x013251a2

The actual base 16 hexadecimal is 13251a2.

Technically speaking the memory address is an **unsigned integer**, i.e., it does not have a negative sign, i.e., it's a positive integer. So you can convert your memory address value to an unsigned integer like this:

```
int x = 123;

std::cout << "val of x: " << x
          << '\n' << "addr of x: "
          << (unsigned int)(&x) << '\n';
```

An ***unsigned int*** can only represent integers up to $2^{32} - 1$. If you have a newer laptop, your address values are very likely larger than 2^32 – 1. If your compiler yells at you, instead of type casting with ***unsigned int***, use ***unsigned long long int***.

An ***unsigned int*** is really a type. So you can declare an ***unsigned int*** variable. By the way, you have actually already seen ***unsigned int*** when using ***srand()***. Likewise ***unsigned long long int*** is also a type.