122COM: Pointers

David Croft

December 15, 2015

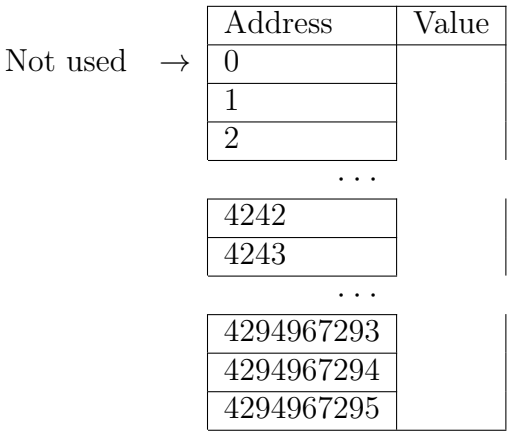
1 Introduction

This week we are going to be looking at variables in memory, pointers, references and memory management. This week will be taught in a combination of Python and C++. Use of both languages is required.

1.1 Memory

A quick description of the computer memory. By memory we mean Random Access Memory (RAM), not disk space (i.e. hard drives). When programs are run they are loaded into memory, as programs run they use memory to store variables etc.

Memory consists of blocks. Each block of memory is the same size and has an address. This memory address describes each block's position in memory and acts as a unique index for each block. We can visualise memory as two columns, one column of addresses and one column of the values that are stored in each of those addresses.



1.2 Variables

Variables are pieces of information stored in a computer's memory. A variables name is really just a label for a specific chunk of memory. We don't usually care exactly where in memory a variable is, just that we we can use it.

When we create a variable the Operating System (OS) picks an unused memory location, with enough space to store the variable. Different types of variables have different sizes and sizes differ depending on OS, compiler, 32 or 64 bit etc. OS allocates the required memory and the name of our variable just acts as an alias for that memory location.

When our code is finished with a variable then the portion of memory that it was occupying is made available for the next set of variables.

```
char myVariable = 'Q';
bool myBool = false;
char anotherVariable = '@';
```

Address	Value
1242	'Q'
1243	false
1244	'@'

1.3 Arrays

How does this apply to arrays? From a programming standpoint arrays are groups of variables where each element in the array has a unique index. From a memory standpoint arrays are groups of variables where each element in the array has a unique address.

When an array is created the memory required is allocated as a single contiguous block. The size of the array in memory is simply the size of an individual element multiplied by the number of elements.

```
char myArray[] = "Hello";
```

Address	Value
4213	'H'
4214	'e'
4215	'l'
4216	'l'
4217	'o'
4218	'\0'

2 Pointers

Pointers are special kind of variable that stores the memory location of another variables. Every variable type has an associated pointer type. Pointers are declared in the same manner as normal variables but with a * after the type name.

```
Typename → int * intPointer;     ← Variable name
           int intVariable;
           char * charPointer;
           char charVariable;
```

2.1 Referencing

Of course if we want to store a memory address in a pointer then we are going to need to find out what the memory addresses of our variables are. This is called referencing, we take our variable and find the location in memory that it is referring to. Once we have the location we can store that address in a pointer.

We get the memory address of a variable using the & operator. Just put it before the variable name that you want the address of. When a pointer has a memory address of another variable it is said to be 'pointing' at that variable. You will need to make sure that you are using the correct types of pointers when referencing. For example, don't try and use a `char *` to store the memory address of an `int`, it won't work. You need to use a `int *` instead.

In the example on the right the value of the pointer `char *myPointer` is the address of the variable

`char myVariable`

.

```
char myVariable = 'Q';
char *myPointer = &myVariable;
```

Name	Address	Value
`char myVariable;`	4213	`'Q'`
	4214	
	4215	
`char *myPointer;`	4216	4213

2.2 Dereferencing

The opposite of referencing is dereferencing. One a pointer is pointing at a variable we can use the pointer to look at that variable indirectly.

In the example on the right the pointer `char *myPointer` holds the address of the variable `char myVarible`. The variable `char myOther` has the same value as the variable `char myVariable`. It got this value by dereferencing the pointer `char *myPointer`.

```
char myVariable = 'Q';
char *myPointer = &myVariable;
char myOther = *myPointer;
```

Name	Address	Value
`char myVariable;`	4213	`'Q'`
`char *myPointer;`	4214	4213
`char myOther;`	4215	`'Q'`

We can also set the value of a variable via a pointer. @@@

2.3 Nulls

- Pointers that don't point to anything are called null pointers.
- Dereferencing a null pointer will cause your program to crash.
- You can set a pointer to point to null.
  - `int *myPointer = NULL;`
  - New in C++11, but the old way works too.
    `int *myPointer = nullptr;`

3 References

So far we've seen how pointers can be used to reference and dereference memory locations. Pointers are an incredibly powerful tool but are also horrifically dangerous. Fortunately C++ has an alternative, references. References are more limited than pointers but also considerably safer. References are defined in the same way as pointers but using a `&` instead of a `*`.

The main differences between the two are:

- References can't be null.
  - Have to be initialised on creation.
- Reference can't be changed.
- References automatically redirects to the variable.

– Automatic dereferencing.

This means that references cannot do pointer arithmetic but on the plus side they can't end up in invalid memory either. Use references instead of pointers whenever possible.

```
int myVariable = 42;

int &refA = myVariable;
int &refB = refA;

int &refC; // will not work
```

4 Why do we care?

Simple Python function that doubles all the values given to it.

```
    def some_function( values ):
        for i in range(len(values)):
            values[i] *= 2

    def main():
        v = range(5)
        print(v)            % [0, 1, 2, 3, 4]

        double(v)
        print{v)            % [0, 2, 4, 6, 8]
```

Same program in C++.

```
void some_function( int values[5] )
{
    for( int i=0; i<5; ++i )
        values[i] *= 2;
}

int main()
{
    int v[5] = {0, 1, 2, 3, 4};

    for( int i=0; i<5; ++i )        // 0, 1, 2, 3, 4
        cout << v[i] << " ";
    cout << endl;

    double(v);
    for( int i=0; i<5; ++i )        // 0, 1, 2, 3, 4
        cout << v[i] << " ";
    cout << endl;
}
```

4.1 Python/C++ differences

The C++ program didn't work, why?

- In Python everything is an 'alias'.
  - Variables are aliases for a memory location.
  - Aliases are similar to pointers/references.
- When Python variable passed to a function, just passing alias to memory location.
- Changing value/s in function changes original variable/s too.

- When C++ variable passed to a function, creates a new variable.
  - New variable stored in a new memory location.
- Changing value/s in function doesn't change original variable/s.

5 Why use pointers/references?

Advantages.

- Pointers/references are small.
  - Instead of copying big data structures around just copy the pointer.
  - E.g. an array storing a picture == millions of bytes.
  - Pointer/reference to an array storing a picture == 4-8 bytes.
- Pointers are required for dynamic memory allocation (C++).

Disadvantages.

- Pointers are dangerous.
  - Buggy pointer code can crash your program/computer.

6 Dynamic memory

When writing code it is not always possible to know how much memory your program will need at compile time. For example, a program that reads in a file the memory required depends on size of the file. Your code is going to need to figure out how much memory it needs and then request (allocate) that memory. When the code has finished with the memory it needs to return (deallocate) that memory.

In order to assist us in detecting preventing memory leaks there are a wide range of tools available, i.e. valgrind and gdb. As most of these are platform of IDE specific we will not bee discussing them directly, however you may want to investigate what options are available on/in your platform/IDE of choice.

We are going to use Leaker, a simple memory leak detector library. It's not as fully featured as some of the alternatives but it is easy and works pretty much everywhere. Leaker is available from `http://left404.com/leaker/` under a GNU General Public License (GPL) v2 license.

Lab work: Dynamic memory

1. Compile and run the lab_dynamic code.
   - You will need to use the following compile command.
     `g++ --std=c++11 lab_dynamic.cpp leaker.c -o lab_dynamic`
2. Run the code.
   - What's wrong?
3. Fix lab_dynamic.

6.1 Allocation

How do we request additional memory? Use the `new` command to request it. `new` will return a memory address to allocated memory so you need to use a pointer to store it. `new` can be used to dynamically allocate variable or arrays of any type.

```
int *myVariable = new int;
char *myArray = new char[10];
```

Of course the real advantage for arrays is that you don't need to know the size at compile time.

```
int size;
cout << "How big an array do you want?" << endl;
cin >> size;

int staticArray[size];                  // won't work
int *dynamicArray = new int[size];      // will work

// do stuff

delete [] dynamicArray;                  // deallocate memory
```

6.2 Deallocation

As we saw in the previous section, if we have dynamically allocated some memory we have to manually deallocate that same memory once we're finished with it. You **MUST** remember to deallocate (free up) the memory. Failure to do so causes a memory leaks, memory gradually gets 'lost' until the computer has to be restarted.

Deallocating memory is done using the `delete` operator. `delete` is used to deallocate individual variables and `delete []` is used to delete arrays.

```
int *myVariable = new int;
int *myArray = new int[1000];
```

```
    // do stuff

    delete myVariable;
    delete [] myArray;
```

7 collection

- Python does memory allocation and deallocation for you automatically.
  - Automatically allocates memory as you create variables.
  - Automatically deallocates memory that isn't in use.
  - Garbage collection.
- Can still manually deallocate Python objects.

```
variable = 42

// do stuff

del(variable)
```

7.1 The stack and heap

The description of memory allocation that you have been given so far is not quite right. It it a simplification of what's really happening.

In fact there are two types of memory that variables can exist in, on the stack or in the heap. These exist in the same physical memory but differ in how they are organised.

When a program is run a block of memory will be allocated to it, this block is called the stack. Each program has its own stack. If multiple copies of the same program are run then each instance will have its own stack. As you create variables and use functions, these variables and functions will be added/removed to/from the stack.

The heap is a block of memory shared by all the currently running processes. When `new` is called the memory that it allocates is taken from the heap. The size of the heap is only really limited by the amount of physical memory.

- Fast - processors normally have special instructions for dealing with stacks quickly.
- Contiguous - everything is kept in one continuous block, makes it easier to know where to put the next variable/function.
- Small - the stack has a limited size. Trying to allocate too many variables will fill the stack and case a "stack overflow" when the space runs out.
- Huge - relative to the stack. Big classes, arrays and variables should be put in the stack.

- Dangerous - must remember to deallocate your heap memory otherwise memory leaks happen.

While you have direct control over where variables are stored in C++, Python does everything for you automagically.

Lab slides:

8 Smart pointers

As of C++11 there is a feature built into C++, smart pointers. Smart pointers are a wrapper around the traditional pointers that we've already discussed. They are a step towards automatic garbage collection in C++.

We've talked previously about the issues that can occur if allocated memory is not deallocated. When using traditional pointers it is absolutely vital that any memory which is allocated by your code is also deallocated. Smart pointers handle this automatically.

There are a couple of different types of smart pointer but the only one that we are going to discuss here are shared pointers. Shared pointers work in the same way as normal pointers but with one addition, they have an internal counter which is used to keep track of the number of shared pointers pointing at that memory location.

This means that when a piece of allocated memory is no longer being pointed at by any pointer, it is automatically deleted.

When using a shared pointer the memory allocation syntax changes slightly, there are a couple of different ways to allocate memory to a shared pointer but the recommended method is to use `make_shared<>()`.

Once a shared pointer has been defined it can be used in the same way as a traditional pointer. See listings 1 and 2 on the following page for a comparison.

```cpp
int main()
{
    int *pointer = new int(42);
    cout << *pointer << endl; // 42

    *pointer = 69;
    cout << *pointer << endl; // 69

    delete pointer;

    return 0;
}
```

Listing 1: Traditional C++ pointer example. Notice the `delete` call at the end of the program.

```cpp
#include <memory>

int main()
{
    shared_ptr<int> pointer =
        make_shared<int>(42);
    cout << *pointer << endl; // 42

    *pointer = 69;
    cout << *pointer << endl; // 69

    return 0;
}
```

Listing 2: C++ smart pointer example. Notice that no `delete` call is required.

```cpp
int main()
{
    shared_ptr<int[]> pointerA = make_shared<int[24]>();
    cout << pointerA.use_count() << endl; // 1

    shared_ptr<int> pointerB = pointerA;
    cout << pointerA.use_count() << endl; // 2

    pointerB = nullptr;
    cout << pointerA.use_count() << endl; // 1

    return 0;
}
```

Listing 3: `shared_ptr` example showing pointer use counter.

Going forwards C++ will be making more and more use of smart pointers. However, for the moment traditional pointers are a major part of the language and are likely to remain so for the foreseeable future. It is also helpful to understand memory allocation and deallocation even when it is being handled for you. When writing your own code it may be helpful to use smart pointers, when dealing with older C++ code or working with 3rd party libraries you may have to work with traditional pointers.