122COM: Pointers

David Croft

February 16, 2016

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 C++ only.

The source code files for this lab and the lecture are available at:

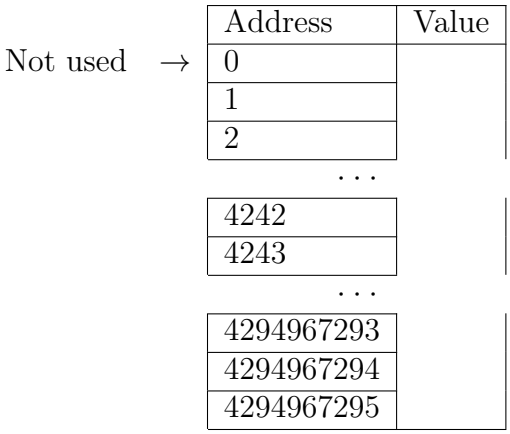
```
https://github.com/covcom/122COM_pointers.git
```

```
git@github.com:covcom/122COM_pointers.git
```

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.

```
array<char,6> 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 vari-

able 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 pointer `char *myPointer` is pointing at

`char myVariable`

.

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

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

Pre-lab work:

1. Complete pre_pointers.cpp.
   - Each of the test functions expects a pointer to a different type of variable, pass each function a pointer to the correct type.
2. Run the code.
   - How much space do the different types need?
   - How much space do the different pointers need?

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 `myPointer` holds the address of the variable `myVarible`. The variable `myOther` has the same value as the variable `myVariable` which it got by dereferencing the pointer `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.

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

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

2.3 Nulls

Pointers do not have to point anywhere, those that don't are called null pointers. Care needs to be taken with null pointers since any attempt to dereference them will cause your program to crash.

Pointers can be set to null in two ways using the old method:

```
int *myPointer = NULL;
```

Or using the new and preferred C++11 method:

```
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.

```
int myVariable = 42;

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

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

This means that references cannot do pointer arithmetic but on the plus side they can't end up in invalid memory either, for this reason it's a good idea to use references instead of pointers whenever possible.

4 Dynamic memory

One place where it is definitely not possible to use references instead of pointers is when dynamically declaring variables.

4.1 Allocation

How do we request additional memory? Use the `new` command to request it. We have to use the old C-style arrays for this, it won't work with C++ STL arrays (i.e. `array<>`)[1] `new` will return

[1] A proposed new type of array, the dynarray, would allow for dynamically declare STL arrays in C++. It is, however, still at the experimental stage as of C++14 so won't make it into the main language until C++17 at the earliest.

a memory address to the newly 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 dynamically declared 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
```

4.2 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 program ends.

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;
```

4.2.1 Leaks

If we dynamically allocate some memory during our program but then forget to deallocate it, we have a memory leak. The memory we allocated hangs around even though nothing is using it.

In order to assist us in detecting preventing memory leaks there are a wide range of tools available (e.g. valgrind and gdb). As most of these are platform or Integrated Development Environment (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 but is also included in this weeks github repository.

Lab work: Dynamic memory

3. Compile the lab_dynamic code.
4. Run the code.
   - Does the program produce any errors?
   - What is the size of `var`?
   - What is the size of the pointer to `var`?
5. Compile the lab_dynamic code again, but THIS time check for memory leaks.
   - You will need to `include "leaker.h"` at the top of your code and use the following compile command.
     `g++ --std=c++11 lab_dynamic.cpp leaker.c -o lab_dynamic`
6. Run the code.
   - What's wrong?
7. Fix lab_dynamic.

Lab work:

8. Complete the lab_adaptive code.
   - The program should ask the user for size, read in that many numbers and then print those numbers in reverse order.
   - Some C++ code has been provided to get your started as well as a complete Python version.

9. Test your program for memory leaks.

5 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.

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 for a comparison.

```
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.

```
#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.

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 so we have to teach you the old way as well. It is also good 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.

Lab work:

10. Convert your lab_adaptive code to use smart_pointers instead of traditional pointers.
    - Make sure to make a backup of your original code first.
11. Test your program for memory leaks.

Extended work:

12. Convert your lab_adaptive code to use vectors instead of arrays.
    - Make sure to make a backup of your original code first.
13. Test your program for memory leaks.
14. Now that we are using leaker to test for memory leaks we're starting to get quite complicated compile statements. You may find it helpful to investigate C++ makefiles, preprocessor statements and compile time flags.