Pointers

David Croft

Coventry University

david.croft@coventry.ac.uk

2015

Overview

Introduction

Talking about memory this week.

- Pointers.
- References.
- Dynamic vs. static memory allocation.
- Memory leaks.

- Very important subject.
- It's a difficult subject.

Pointers

- Variables are pieces of information stored in a computers memory.
- Don't care where in the memory.
- Just care that we can use the variables.
- Pointers store memory locations.
  - Find where variables are stored.
  - Move through memory.

- In Python almost everything is a pointer.
  - So we don't notice.
- In C++ pointers are explicitly stated.

Variables & Memory

- Variables are stored in memory (RAM).

```
char myVariable = 'Q';
```

Address	Value
1242	’Q’

- OS picks an unused memory location e.g. 1242
  - This location must have enough space to store the variable.
  - Different variable types have different sizes.
  - I.e. sizeof(int) == 4 bytes, sizeof(double) == 8 bytes.
  - But depends on OS, compiler, 32 or 64 bit etc.
- `myVariable` is our name for memory location 1242.
- In Python can get memory location info using `id(myVariable)` function.

Arrays and Memory

- Variables are stored in memory.
- Arrays are groups of variables called elements.
- Array elements stored sequentially in contiguous blocks of memory.
  - Large objects, i.e. arrays, class instances, floats may span multiple blocks.
  - Demonstrating using old C-style arrays.

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

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

Pointer types

- Variables are blocks of memory that hold data.

- Pointers are variables that hold memory addresses.
- Each type of variable has an associated pointer type.
- We declare a pointer using an * after the type name.

```
typename * variableName;
int * i;
char * c;
float * f;
```

- Pointers "point to" other variables.

Referencing

- Referencing is when we store a memory address in a pointer.
- The pointer is now said to be pointing at that memory address.
- Is achieved using the & operator.
- & means the memory address of.

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

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

Dereferencing

- The opposite of referencing is dereferencing.
- A pointer stores a memory address.
- Dereferencing means getting the value that is stored in that memory address.

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

Name	Address	Value
`char myVariable;`	4213	'Q'
	...	
`char *myPointer;`	5617	4213
	...	
`char myOther;`	7584	'Q'

Coventry University

Dereferencing

- Already seen that we can get the value of a variable via a dereferenced pointer.
- Can also set the value of a variable through a pointer.

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

Name	Address	Value
`char myVariable;`	4213	'Z'
	...	
`char *myPointer;`	5617	4213

A

Pointer arithmetic

- Have seen how to change variables pointed to by a pointer.
- Pointers are also variables.
- Can change the values of pointers.
  - Can change where they are pointing.

```
array<int,4> myArray = {{69, 42, 99, 3}};
int *myPointer = myArray->data();


cout << *myPointer << endl; → 69
myPointer += 1;
cout << *myPointer << endl; → 42
myPointer += 2;
cout << *myPointer << endl; → 3
```

Null pointers

- 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;`

References

- C++ also has reference datatypes.
- Safer than pointers.
  - Less powerful.
- Declared like pointers but with & instead of *.

```
int myVariable = 42;

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

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

Differences to pointers.

- Can't be null.
- Can't be changed to point at different locations.
- References automatically redirects to the variable.
  - Automatic dereferencing.
- Have to be initialised on creation.
  - References point at a variable the instant they are created.

Use references instead of pointers whenever possible.

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]
```

Coventry University

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;

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

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.

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.

Dynamic memory

- Can't always know how much memory program will need at compile time.
  - E.g. a program that reads in a file, memory required depends on size of the file.
- Have to allocate it at run time.
  - Dynamic memory allocation.
- Code gives itself more memory, has to remember to give it back when it's finished
  - Deallocation.

Dynamic memory allocation

```
int *myInt;
myInt = new int;
*myInt = 42;
delete myInt
```

Name	Address	Value
int *myInt;	4213	4215
	4214	
	4215	42
	4216	
	4217	
	4218	

Array allocation

How to dynamically allocate arrays?

```
int staticArray[10]; // works
int* dynamicArray = new int[10]; // works
```

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

int staticArray[size]; // won't compile
int* dynamicArray = new int[size]; // works
```

Dynamic memory deallocation

- You **MUST** remember to deallocate your memory.
- Failure to do so causes a memory leak.
  - Memory gradually gets 'lost'.
- No exceptions.
- NO EXCEPTIONS!

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

// do stuff

delete myVariable;
delete [] myArray;
```

Garbage 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)
```

C++ Garbage collection

C++ does not have automatic garbage collection.

- C++11 comes close.
- New features - `shared_ptr` and `unique_ptr`, `weak_ptr`.
- Special new smart pointers.
  - Automatically deallocate memory when nothing pointing at it.
  - Don't need to remember to `delete`.
  - No memory leaks!
- `shared_ptr` is 99.9% the same as 'normal' pointers.
  - `unique_ptr` and `weak_ptr` have extra features.

`shared_ptr<>`

STRONGLY recommend you use `shared_ptr`.

Recap

- Variables stored in memory.
- Different variables need diffe… …nemory.
- Array elements stored in cor… …l blocks of memory.
- Pointers/references store…
- Pointers are dangerous…
- If, at compile time, we c… …emory our program will need use dynamic me…
- Always deallocate m… …exits.

The End