

Pointers

David Croft

Coventry University

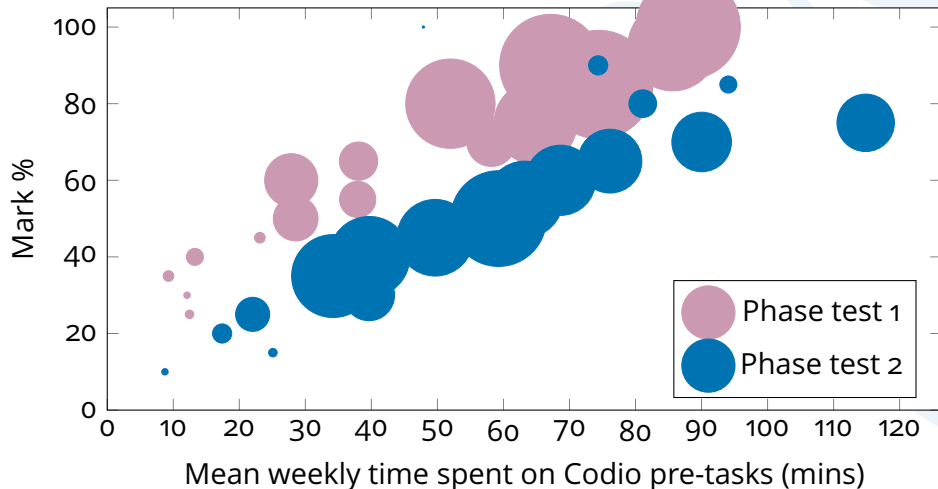
david.croft@coventry.ac.uk

2018

- 1 Introduction
- 2 Memory
 - Variables
 - Arrays
- 3 Pointers
 - Referencing
 - Dereferencing
 - Arithmetic
 - Nulls
- 4 Why do we care?
 - Danger
 - Required for
 - Python/C++ differences
- 5 References
- 6 Dynamic memory
 - Allocation
 - Deallocation
- 7 Smart Pointers
- 8 Recap

You have all attempted the green Codio exercises for this week.

122COM results 2016-17 September starters.



Introduction

Talking about memory this week.

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

Introduction

Talking about memory this week.

- Pointers.
- References.
- Dynamic vs. static memory allocation.
- Memory leaks.
- Very important subject.
 - People can get nervous about them.
 - Not actually difficult.

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++
differences

References

Dynamic memory

Allocation
Deallocation

Smart Pointers

Recap

Talking about memory this week.

- Pointers.
- References.
- Dynamic vs. static memory allocation.
- Memory leaks.
- Very important subject.
 - People can get nervous about them.
 - Not actually difficult.
 - Calm down, have a kitten.



Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++
differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- Variables are pieces of information stored in a computers memory.
- Don't typically 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.
 - Technically Python uses aliases not pointers.
- In C++ pointers are explicitly stated.

- Variables are stored in memory.
 - Can be visualised as series of uniquely addressed boxes.

```
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.
 - Need multiple 'boxes'.
- `myVariable` is our name for memory location 1242.
- In Python can get memory location info using `id(myVariable)` function.

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- 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.

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- 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.

```
array<char,6> myArray = {"Hello"};
```

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

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- 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.

```
array<char,6> myArray = {"Hello"};
```

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

```
float myVariable = 12.34;
```

Address	Value
4213	
4214	12.34
4215	
4216	
4217	
4218	

Variables are named blocks of memory.

- 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 in memory.

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

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

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

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

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic

memory

Allocation

Deallocation

Smart

Pointers

Recap

- Referencing is when we store a memory address in a pointer.
- The pointer is now 'pointing' to 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	
	4216	

- Referencing is when we store a memory address in a pointer.
- The pointer is now 'pointing' to 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	

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for Python/C++ differences

References

Dynamic memory

Allocation
DeallocationSmart
Pointers

Recap

- The opposite of referencing is dereferencing.
- A pointer stores a memory address.
- Dereferencing means getting the value that is stored in that memory address.
- Is achieved using the `*` operator.

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

Name	Address	Value
char myVariable;	4213	'Q'
	...	
	5617	
	...	
	7584	

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- The opposite of referencing is dereferencing.
- A pointer stores a memory address.
- Dereferencing means getting the value that is stored in that memory address.
- Is achieved using the `*` operator.

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

Name	Address	Value
char myVariable;	4213	'Q'
	...	
	5617	
	...	
	7584	

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- The opposite of referencing is dereferencing.
- A pointer stores a memory address.
- Dereferencing means getting the value that is stored in that memory address.
- Is achieved using the `*` operator.

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

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

- The opposite of referencing is dereferencing.
- A pointer stores a memory address.
- Dereferencing means getting the value that is stored in that memory address.
- Is achieved using the `*` operator.

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

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

- The opposite of referencing is dereferencing.
- A pointer stores a memory address.
- Dereferencing means getting the value that is stored in that memory address.
- Is achieved using the `*` operator.

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

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

- 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;
```

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

- 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';
```

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

- 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	

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- 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};
```

Name	Addr	Value
myArray	4213	69
	4214	42
	4215	99
	4216	3
	4217	

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- 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();
```

Name	Addr	Value
myArray	4213	69
	4214	42
	4215	99
	4216	3
myPointer	4217	4213

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- 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
```

Name	Addr	Value
myArray	4213	69
	4214	42
	4215	99
	4216	3
myPointer	4217	4213

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- 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;
```

Name	Addr	Value
myArray	4213	69
	4214	42
	4215	99
	4216	3
myPointer	4217	4214

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- 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.
- Powerful but highly dangerous.

```
array<int,4> myArray {69, 42, 99, 3};
```

```
int *myPointer = myArray.data();
```

```
cout << *myPointer << endl; // 69
```

```
myPointer += 1;
```

```
cout << *myPointer << endl; // 42
```

Name	Addr	Value
myArray	4213	69
	4214	42
	4215	99
	4216	3
myPointer	4217	4214

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- 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.
- Powerful but highly dangerous.

```
array<int,4> myArray {69, 42, 99, 3};
```

```
int *myPointer = myArray.data();
```

```
cout << *myPointer << endl; // 69
```

```
myPointer += 1;
```

```
cout << *myPointer << endl; // 42
```

```
myPointer += 2;
```

Name	Addr	Value
myArray	4213	69
	4214	42
	4215	99
	4216	3
myPointer	4217	4216

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- 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.
- Powerful but highly dangerous.

```
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
```

Name	Addr	Value
myArray	4213	69
	4214	42
	4215	99
	4216	3
myPointer	4217	4216

In modern C++ pointer arithmetic has been mostly replaced by iterators.

- Similar to pointers but safer and with more advanced features.
- Strongly recommend you investigate in your own time.

```
array<int,4> myArray {69, 42, 99, 3};  
  
// stepping through an array with a pointer  
for(int *ptr=myArray.data(); ptr<myArray.data()+myArray.size(); ptr+=1)  
    cout << *ptr << endl;  
  
// stepping through an array with an iterator  
for(array<int,4>::iterator it=begin(myArray); it!=end(myArray);  
    ↪ it=next(it))  
    cout << *it << endl;  
  
// shorter way of writing the iterator code  
for(auto it=begin(myArray); it!=end(myArray); it=next(it))  
    cout << *it << endl;
```

Pointers don't have to point anywhere.

- If they don't point to anything they are called null pointers.
- Dereferencing a null pointer will cause your program to crash.
- You can set any pointer to point to null.

- Old way (still works).

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

- New C++14 way (use this one).

```
int *myPointer = nullptr;
```


Why use pointers/references?

C

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++).
 - Required for some behaviours.

Disadvantages.

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



Pointers let us move around the memory.

- ANYWHERE in memory.
 - Newer systems are getting more secure.
 - Segmentation fault.

```
array<int,4> myArray {69, 42, 99, 3};  
int *myPtr = myArray.data();  
  
for( int i=0; i<=myArray.size(); ++i )  
{  
    cout << *myPtr << endl;  
    myPtr += 1;  
}
```

Address	Value
4213	69
4214	42
4215	99
4216	3
4217	
4218	

← myPtr

Pointers let us move around the memory.

- ANYWHERE in memory.
 - Newer systems are getting more secure.
 - Segmentation fault.

```
array<int,4> myArray {69, 42, 99, 3};  
int *myPtr = myArray.data();  
  
for( int i=0; i<=myArray.size(); ++i )  
{  
    cout << *myPtr << endl;  
    myPtr += 1;  
}
```

Address	Value
4213	69
4214	42
4215	99
4216	3
4217	
4218	

← myPtr

Pointers let us move around the memory.

- ANYWHERE in memory.
 - Newer systems are getting more secure.
 - Segmentation fault.

```
array<int,4> myArray {69, 42, 99, 3};  
int *myPtr = myArray.data();  
  
for( int i=0; i<=myArray.size(); ++i )  
{  
    cout << *myPtr << endl;  
    myPtr += 1;  
}
```

Address	Value
4213	69
4214	42
4215	99
4216	3
4217	
4218	

← myPtr

Pointers let us move around the memory.

- ANYWHERE in memory.
 - Newer systems are getting more secure.
 - Segmentation fault.

```
array<int,4> myArray {69, 42, 99, 3};  
int *myPtr = myArray.data();  
  
for( int i=0; i<=myArray.size(); ++i )  
{  
    cout << *myPtr << endl;  
    myPtr += 1;  
}
```

Address	Value
4213	69
4214	42
4215	99
4216	3
4217	
4218	

← myPtr

Pointers let us move around the memory.

- ANYWHERE in memory.
 - Newer systems are getting more secure.
 - Segmentation fault.
- Reading from invalid memory is bad.
 - Writing to invalid memory can be disastrous.

```
array<int,4> myArray {69, 42, 99, 3};  
int *myPtr = myArray.data();  
  
for( int i=0; i<=myArray.size(); ++i )  
{  
    cout << *myPtr << endl;  
    myPtr += 1;  
}
```

Address	Value
4213	69
4214	42
4215	99
4216	3
4217	?????
4218	?????

← myPtr

Simple function that doubles all the values given to it.

```
import sys

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

def main():
    v = [ i for i in range(5) ]
    print(v)  # [0, 1, 2, 3, 4]

    some_function(v)
    print(v)  # [0, 2, 4, 6, 8]

if __name__ == '__main__':
    sys.exit(main())
```

Same program in C++ doesn't work.

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

int main()
{
    array<int,5> v {0, 1, 2, 3, 4};

    for( int i : v )           // 0,1,2,3,4
        cout << i << ", ";
    cout << endl;

    some_function(v);

    for( int i : v )           // 0,1,2,3,4
        cout << i << ", ";
    cout << endl;
}
```


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



Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart Pointers

Recap

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

- In Python we passed a mutable type to the function.
 - Actually just sends an 'alias' of the original mutable structure.
 - Mutable types, e.g. lists, sets, dicts etc.
 - Changing value/s in function changes original variable/s too.
 - Aliases are similar to pointers/references.

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

- In Python we passed a mutable type to the function.
 - Actually just sends an 'alias' of the original mutable structure.
 - Mutable types, e.g. lists, sets, dicts etc.
 - Changing value/s in function changes original variable/s too.
 - Aliases are similar to pointers/references.
- If we passed an immutable type Python would create actual copy and send that instead.
 - Immutable types, e.g. int, float, string.
 - Original would stay same regardless.

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

- In Python we passed a mutable type to the function.
 - Actually just sends an 'alias' of the original mutable structure.
 - Mutable types, e.g. lists, sets, dicts etc.
 - Changing value/s in function changes original variable/s too.
 - Aliases are similar to pointers/references.
- If we passed an immutable type Python would create actual copy and send that instead.
 - Immutable types, e.g. int, float, string.
 - Original would stay same regardless.
- When C++ variable passed to a function, always creates a new variable.
 - New variable stored in a new memory location.
 - Even for vectors, arrays etc.
- Changing value/s in function doesn't change original variable/s.
- How to fix?

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic

memory

Allocation

Deallocation

Smart

Pointers

Recap

C++ also has references.

- Safer than pointers.
 - Less powerful.
- Declared like pointers but with `&` instead of `*`.

```
int myVariable = 42;
```

```
int &refA = myVariable;
```

```
int &refB = refA;
```

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

Looking at the earlier function example.

```
int some_function( array<int,5> &values )
{
    for( int i=0; i<values.size(); ++i )
        values[i] *= 2;
}

int main()
{
    array<int,5> v {0, 1, 2, 3, 4};

    some_function(v);

    for( int i : v )           // 0,2,4,6,8
        cout << i << ", ";
    cout << endl;

    return 0;
}
```

Differences to pointers.



Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- 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.



- 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.

Most important feature of pointers.

- 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.
 - As opposed to Static memory allocation.
- Code gives itself more memory, has to remember to give it back when it's finished
 - Deallocation.

Dynamic memory allocation



Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

```
int *myInt;
```

Name	Address	Value
<code>int *myInt;</code>	4213	
	4214	
	4215	
	4216	
	4217	
	4218	

Dynamic memory allocation

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

```
int *myInt;  
myInt = new int;
```

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

Dynamic memory allocation

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for Python/C++ differences

References

Dynamic memory

Allocation
DeallocationSmart
Pointers

Recap

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

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

Dynamic memory allocation

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for Python/C++ differences

References

Dynamic memory

Allocation Deallocation

Smart Pointers

Recap

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

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

Used to have to dynamically ask for more memory.

- Create a chunk of memory of the size requested.
- Return a pointer to it so know where it is.

E.g. vectors.

- C/C++ arrays can't be resized.
- But vectors are resizeable arrays.
 - How?
 - 1 Dynamically allocate new array.
 - 2 Copy old array contents into new array.
 - 3 Deallocate old array.

How to dynamically allocate arrays?

- Have to use old, C-style arrays.
 - For the moment, talk again after C++17.

```
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
```

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for
Python/C++
differences

References

Dynamic
memoryAllocation
DeallocationSmart
Pointers

Recap

- You **MUST** remember to deallocate any dynamic memory.
- Failure to do so causes a memory leak.
 - Memory gradually gets 'lost'.
- Every `new` needs a matching `delete`.

```
int* myVariable = new int;  
int* myArray = new int[1000];  
  
// do stuff  
  
delete myVariable;  
delete [] myArray;
```


Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- You **MUST** remember to deallocate any dynamic memory.
- Failure to do so causes a memory leak.
 - Memory gradually gets 'lost'.
- Every **new** needs a matching **delete**.
- No exceptions.

```
int* myVariable = new int;  
int* myArray = new int[1000];  
  
// do stuff  
  
delete myVariable;  
delete [] myArray;
```

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart

Pointers

Recap

- You **MUST** remember to deallocate any dynamic memory.
- Failure to do so causes a memory leak.
 - Memory gradually gets 'lost'.
- Every **new** needs a matching **delete**.
- No exceptions.
- NO EXCEPTIONS!

```
int* myVariable = new int;  
int* myArray = new int[1000];  
  
// do stuff  
  
delete myVariable;  
delete [] myArray;
```

- You **MUST** remember to deallocate any dynamic memory.
- Failure to do so causes a memory leak.
 - Memory gradually gets 'lost'.
- Every `new` needs a matching `delete`.
- No exceptions.
- NO EXCEPTIONS

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

```
// do stuff
```

```
delete myVariable;  
delete [] myArray;
```

- 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++ does not have automatic garbage collection for dynamic memory.

- C++11 onwards 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% the same as 'normal' pointers.
 - `unique_ptr` and `weak_ptr` have extra features.



Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

Smart Pointers

Recap

C++ is moving away developer allocated memory.

- Use vectors instead of arrays etc.
 - Handles memory allocation for you.
- Safe, bug free.

C++ is moving away developer allocated memory.

- Use vectors instead of arrays etc.
 - Handles memory allocation for you.
- Safe, bug free.

When you HAVE to dynamically allocate memory...

- C++11 had 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% the same as 'normal' pointers.

STRONGLY
recommend you use
shared_ptr.

- Whenever dynamically allocating memory.
- No memory leaks.

```
int main()
{
    shared_ptr<int> pointerA = make_shared<int>();
    *pointerA = 42;

    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;
}
```

lec_smart_pointers.cpp

Why do I care?

Everyone

- Everyone
 - Need to understand pointers/references to write C++.
 - Important in writing more efficient code.
- Computer Science - Pointers allow direct memory access, allowing greater understanding of computer memory.
- Ethical Hacking - Important in understanding common vulnerabilities, i.e. buffer overflow.
- Games Tech - Important for efficiency, very important for games.

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++
differences

References

Dynamic
memory

Allocation

Deallocation

Smart

Pointers

Recap

- Variables stored in memory.
- Different variables need different amounts of memory.
- Array elements stored in contiguous sequential blocks of memory.
- Pointers/references store memory addresses.
- Pointers are dangerous but necessary.
- If, at compile time, we don't know how much memory our program will need use dynamic memory allocation.
- Always deallocate memory before the program exits.

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++
differences

References

Dynamic
memory

Allocation

Deallocation

Smart
Pointers

Recap

- Variables stored in memory.
- Different variables need different amounts of memory.
- Array elements stored in contiguous sequential blocks of memory.
- Pointers/references store memory addresses.
- Pointers are dangerous but necessary.
- If, at compile time, we don't know how much memory our program will need use dynamic memory allocation.
- Always deallocate memory before the program exits.

Well done! Have another kitten.

Introduction

Memory

Variables

Arrays

Pointers

Referencing

Dereferencing

Arithmetic

Nulls

Why do we care?

Danger

Required for

Python/C++ differences

References

Dynamic memory

Allocation

Deallocation

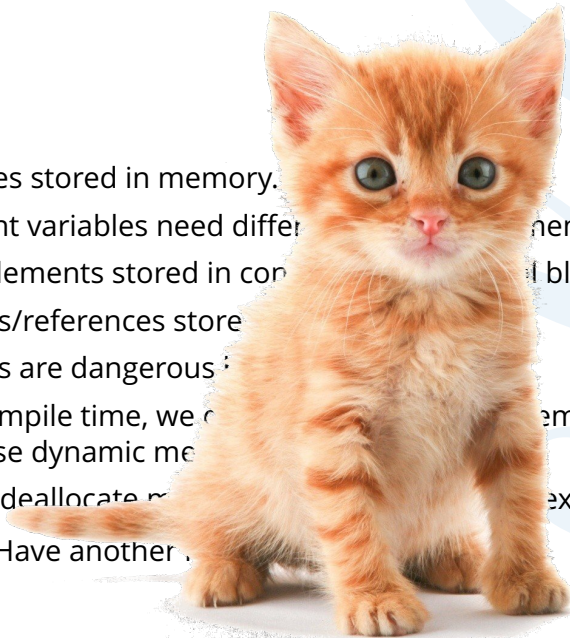
Smart

Pointers

Recap

- Variables stored in memory.
- Different variables need different blocks of memory.
- Array elements stored in contiguous blocks of memory.
- Pointers/references store memory addresses.
- Pointers are dangerous :)
- If, at compile time, we know the size of memory our program will use, we can statically allocate it.
- Always deallocate memory when it exits.

Well done! Have another...



- Complete the yellow Codio exercises for this week.
- Attempt the green Codio exercises for next week.
- If you have spare time attempt the red Codio exercises.

- If you are having issues come to the PSC.
<https://gitlab.com/coventry-university/programming-support-lab/wikis/home>

The End