

C++ Pointers

How, Why and Why NOT

"The Angry Penguin", used under creative commons licence
from Swantje Hess and Jannis Pohlmann.



Introduction and Disclaimer



"C makes it easy to shoot yourself in the foot;
C++ makes it harder, but when you do it blows
your whole leg off"

Bjarne Stroustrup

A decorative blue zigzag shape is located at the bottom right of the slide, extending from the dark blue background into the white background.

What is this course for?

- C++ is a very powerful, and quite complex language with a very long history
- Massive developments in the last ~15 years, many aimed at improving safety of the language
 - Memory safety - avoid leaks by automatic management
 - Data safety - more ways to initialise things at declaration
 - Performance "safety" - more control over how things are copied (or "moved")
- In particular, "bare pointers" - a memory address (almost, see later) do not belong in modern C++
 - Sometimes you might need them anyway, in old codes, codes that interface with other languages, or in Cuda code (which is a C/C++ hybrid a lot of the time)
 - This course will teach you how to use them, but also entreat you to avoid them whenever possible!

What is this course for?

- We will show:
 - What pointers are, and how to use them
 - The dangers you **will** encounter
 - The safer modern alternatives
 - References
 - "Smart" memory managing pointers
 - Not using pointers at all because there is a better idiom - often you can avoid performance costs a better way

Useful Tips

- We show a bunch of code. Some works, some is broken. Code files are marked - see README
- I use scope blocks a lot to control lifetimes
 - { } within your code - variables defined in here no longer exist outside
 - Just like something you define inside a loop
 - Use them here to illustrate variables being destroyed
 - Use sparingly in real code - can be confusing

Useful Tips

- We use a few user-defined classes for illustration
- `'class_name(const class_name & other)'` is used when making copies of `class_name` (copy constructor)
- `'class_name(class_name && other)'` is used for moving - making a new `class_name` but stealing any storage from the old one (move constructor)
- `~class_name()` is used to clean up (destructor)

Useful Tips

- There's two ways to get new-lines in stream IO
 - Print the character '\n'
 - Use the "std::endl" thing
 - This forces a flush of the stream
 - We DO use this with code with undefined behaviour as it increases the chances of seeing the prints if things crash
 - Favour '\n' unless you want this

Pointers - same Data,
another Name

A decorative blue zigzag shape is located at the bottom right of the slide, extending from the dark blue background into the white background.

C's legacy

- Something being a legacy from C will come up quite a bit in this course
- Using pointers to pass a variable to a function by "reference" is classic case
- C/C++ are "pass by value"
 - They make a copy - could be costly!
 - Can't modify what we pass in - this won't affect passed value

Passing by Reference

- C++ has "references" - a variable which has no data, but is an "alias" for another
- Passing to a function by reference doesn't copy - it passes on "where to find the data"
- Here is a "pass by reference function sig and call:

```
// You can think of this & as just meaning "pass this argument by  
reference"
```

```
void a_function(massive_data & data){  
    std::cout<<"I'm a function who got some massive_data by  
reference\n";  
};
```

```
.....
```

```
a_function(arg);
```

Explored in 01-PassByReference.cpp

Passing by Reference

- C++ has "references" - a variable which has no data, but is an "alias" for another
- Passing to a function by reference doesn't copy - it passes on "where to find the data"
- Here is a "pass by reference function sig and call:

```
// You can think of this & as just meaning "pass this argument by  
reference"
```

```
void a_function(massive_data & massive_data) {  
    std::cout<<"I'm a function who got some massive_data by  
reference\n";  
};
```



```
.....
```

```
a_function(arg);
```

Explored in 01-PassByReference.cpp

Passing by Pointer

- Equivalent in C was using a pointer - explicitly taking the "address" of a variable and passing that
- Here's the code:

```
void fn_p(int * a){  
    ++(*a); // Affects whatever we passed  
};
```

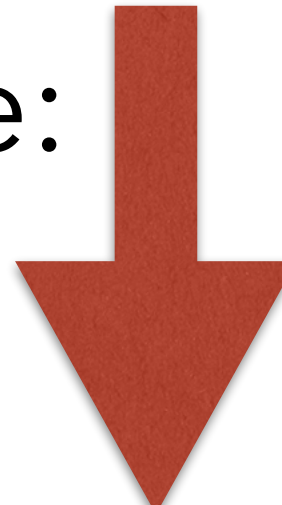
.....

```
int arg = 10;  
fn_p(&arg);  
// arg is now 11
```

Explored in 04-PassByPointer.cpp



Passing by Pointer

- Equivalent in C was using a pointer - explicitly taking the "address" of a variable and passing that
- Here's the code:



```
void fn_p(int * a){  
    ++(*a); // Affects whatever we passed  
};
```

.....



```
int arg = 10;  
fn_p(&arg);  
// arg is now 11
```

Explored in 04-PassByPointer.cpp

Passing by Reference

- Neither simple reference nor pointer lets us pass "literal" values without a copy
- E.g. `f(1)` or `f("Bob")` - no variable to take pointer to, or have reference to
- References are designed for this sort of task so they do, with "const" keyword

```
// Takes a reference – can only take variables
void fn2(int & b){};
// Takes a const reference – can take variables or literals and wont copy
void fn3(const int & c){};

.....
// fn2(10); // This line cannot compile – what if fn2 changes its
argument
fn3(10); // This is OK – fn3 CANNOT change its argument
```

So What ARE Pointers?

- So pointers are sort of memory addresses
- When we have a data item of given data_type we can:
 - Take it's "address" with the "address of" operator '&'
 - Store this into a variable of type 'data_type *'
 - Use this address to get the data with the "de-reference" operator '*'
- 03-Pointers.cpp explores these operations

Aren't those symbols doing a lot of jobs?

- You might have noticed that the same symbols do several jobs
- Sorry! There are only so many available! '*' has it bad, but '>' might be the worst contender in C++
- Think of it like this:
 - When you see '*' it means "this is a pointer" - either defining one, or using one to get back a value (dereference)
 - When you see '&' it means "this is a reference" - defining one, or getting one (address of)

Aren't those symbols doing a lot of jobs?

- By the way, for classes/structs there is a shortcut operator to de-reference and look up a member in one
- Particularly useful because we need brackets otherwise
- Instead of `(* cls).member` we use `cls->member`
- `'>'` symbol really has it tough!

Pointers are Variables

- A pointer is a variable itself
- Mostly we want to associate them with data- no reason that data cannot be another pointer!
- Used for 2-d arrays in C

```
int data = 10;
p = &data; // Take the address of data with '&' (address of) and store this
value
// Pointers like p are variables too – we can have pointers to pointers
int ** q = &p; // q is a pointer to pointer to int. *q is a pointer to int,
equal to p. **q is data
std::cout<<"q: "<<q<<" *q, same as p: "<<*q<<" **q, same as data:
"<<**q<<"\n";
```

Explored in 03-Pointers.cpp

Const-ness

- The const keyword tells the compiler to prevent us modifying a variable
- With pointers have two possibilities - modify the pointer or modify the data
- Const keyword can do both, you have to place it correctly relative to the '*'
- Roughly read right to left, reading '*' as pointer

```
int * const a = &data; //A const pointer to an int
int const * b = &data; //A pointer to a const int
const int * b = &data; //Identical to previous line
const int * const c = &data; // Both – const pointer to const int
```

Heap and Stack

- There are two kinds of memory our programs can use: the stack and the heap
 - Stack is a linear structure - each new variable is adjacent to the last
 - Fast but small, size set at compile time
 - Every function gets its own stack
 - Heap is hierarchical - imagine a tree
 - Potentially as large as entire RAM, access requires lookups
 - One heap for entire program

Heap and Stack

- *Extremely pedantic note: C++ does not know about heap or stack - that is part of the implementation. The following is "the usual" way for simple variables*
- Stack memory is used for "automatic" variables - ones that we don't have to clean up
 - Function local variables, including in main, are these
- Heap memory is used for things we get with new and it's up to us to "give it back"
- Operating System deals with details

Heap and Stack

- If you actually have to work with pointers it can be helpful to know more
- Simple types like 'int' or 'float' probably obey previous slide, unless optimised out entirely
- Many classes, including most containers like string, vector etc use "out of band" storage - contain a pointer to the actual data
 - "Container" part will obey previous slide
 - "Data" part will very likely be on the heap whatever we do
 - Cleanup happens internally on destruction

Pointer Arithmetic

- If we want heap memory we get it with malloc (very old school, technically wrong but probably works for simple types) or new (C++ approved)
- This gives us a pointer back
- We can ask for multiple items, and we get a pointer to the first in that case
 - To get to the second item we add 1 to the pointer
 - This automatically moves the right number of bytes for the data type we have
- Explored in 05-PointerArithmetic.cpp

Pointer Arithmetic

- Important: until we set the values in this new memory, they are undefined
 - They might seem to have values, but we CANNOT read from them
 - Reading is undefined behaviour
- Important: `*p+1` and `*(p+1)` are not the same
- `[]` operator is used (especially with stack arrays) to access elements
- Not clever - adds LHS to RHS and dereferences the result. See 06-SquareBrackets.cpp for examples

The Rules

- We keep saying pointers are "like" memory addresses - their value probably is one, but there are strict limits on what we can do, so they aren't "just addresses"
- Exactly 4 (plus one for Object Oriented Code) valid things we can do **with** pointers
- There's a couple more things with can do **to** them, such as set to nullptr, compare with another pointer, print, or repoint to a different variable

The Rules - 1

- We can get a pointer by taking the address of a variable, and we can use this pointer to get back to the original data by dereferencing, for as long as the original variable exists
- We can also make copies of the pointer, or pass this pointer to functions, and use these results

The Rules - 2 and 3

- For legacy reasons we can:
 - Cast (convert) a pointer to any type into a pointer to 'char' type and back, and use it before, after and during this process. Used in C-ish code for binary output a lot
 - Cast a structure or class into a pointer to its first member, and use the result (heavily used in C-ish code where first member is a 'tag' representing how to interpret the rest of the data)


The Rules - 4

- For array data, e.g. `int arr[10];` `int * arr = new int[10];` or `malloc(..)` we can:
 - Use the pointer to access the data at all valid indices
 - Increment or add to the pointer to point to any of these data items, and decrement it back to the start
 - Tiny wrinkle - we are allowed to get a pointer to 1-past-the-end and compare against it, but not dereference it
- We CANNOT use the pointer to peek at memory outside our array

The Rules - 5

- In Object Oriented code we can
 - Access and pass a child/derived class using a pointer to its base type
 - Related to "Liskov substitution" which is a big thing in OO design
 - For anything the base class does, it should be meaningful to substitute a child for a base and vice versa
 - For anything extra the child does, it is free to behave how it likes

Pointers - errors you
will know and love

A decorative blue zigzag shape, resembling a stylized 'W' or a series of connected 'V' shapes, is positioned at the bottom right of the slide, extending from the dark blue background into the white background.

Memory Leaks

- If we allocate memory on the heap, the ONLY way we may refer to it is with the pointer we get given
 - Lose the pointer, leak the memory
- This is easy to do, and hard to spot
- Use tools (valgrind)
- 07-MemoryLeak.cpp explores this

Bad Pointers

- Like any other variable a pointer has no specific value at creation
 - No way to tell if value is valid
 - Good practice to always do 'type * p = nullptr'
 - nullptr is null value applicable to any type of pointer
- Trying to use a nullptr, or an uninitialised ptr are both undefined
 - if(ptr) - checks not null, can be invalid

Bad Pointers

- No particular expectation what will happen if we mess up
- Segfault - we tried to access memory that is not allowed (historically was not in an accessible "segment")
- Apparently working code
- Undefined behaviour! Compiler can do anything!
- 08-UninitialisedPointer.cpp explores

Bad Pointers

- The other classic way to get a bad pointer is to let pointed-to data go "out of scope"
- Our pointer becomes invalid
 - No way to know!
 - Even memory checkers probably won't notice!
- Call it "dangling pointer" - 09-DanglingPointer.cpp explores

2D arrays

- We mentioned using pointers to form 2d arrays, and for stack arrays saw this
- For heap arrays, might think it is trivially easy
- Arrays ought to be "contiguous" in memory
- 10-Failed2DHeapArray.cpp explores doing the obvious, wrong thing
- No good answer to arrays - use a library or 1D array/vector and indexer

References - more
than parameters

A decorative blue zigzag shape is located at the bottom right of the slide, extending from the dark blue background area into the white area.

Reference Variables

- References are pretty natural when passing values to functions, but reference is applicable to an ordinary variable too
- Cannot "re-point" a reference, but can create new ones - compiler is bright enough to optimise this away
- Can return references from functions
- Very useful in "container loops" to avoid pointless copying
- Const reference ALWAYS means reference that cannot be used to change the data

Are References Always a Solution?

- 10-ReferenceVariables.cpp shows some of the things we just mentioned
- Hopefully by now you're seeing that references can do "pointer-like" things much more smoothly and with less effort
- BUT there is one big problem still
- "Dangling reference" is the same problem as dangling pointer
 - Easy to cause - for example take reference to vector item, then resize it
- 11-DanglingReference.cpp explores

Lifetime Management

- Bare pointers suffer from two problems:
 - Because of the rules we stated, probably 99% of the code you could write is not valid
 - They are not connected to the lifetime of the data they access
- References solve the first by being much more restrictive
 - Do nothing for the second

Smart Pointers and Lifetime Management

A decorative blue zigzag shape, resembling a stylized 'W' or a series of connected 'V' shapes, is positioned at the bottom center of the slide, extending from the dark blue header area into the white footer area.

Modern C++ Smart Pointers

- "Bare pointers" as a name comes from contrast with a pointer "wrapped away" inside something else
- Modern C++ adds Smart Pointers (C++11, expanded in 14)
- Three kinds:
 - Unique
 - Shared
 - Weak

Modern C++ Smart Pointers

- Unique and Shared pointers are explicitly "lifetime managing"
- They control the data they point to
- It can only be destroyed once they are
- No dangles
- Still have to check for null if something might have nulled it
- Weak pointer does not control data but it knows if it is valid

Unique Pointer

- Unique pointer has sole use of the data inside it
- It can't be copied - can only pass by reference
- You can steal the data, or get a bare pointer to it, but these things take you outside the guarantees
- `std::unique_ptr` (in `<memory>` header)
- Create data by creating pointer. When pointer goes out of scope, data is cleaned up

Unique Pointer

- Best syntax is also the simplest - use `make_unique` and `auto`
- Pass construction parameters to `make_unique` if necessary
- Result IS a pointer, so have to dereference to use. But it is also a class with its own methods

```
// Create a unique pointer.
```

```
auto d1 = std::make_unique<watcher>();
```

Explored in S1 and S1+

```
// Create another watcher, with its own ptr
```

```
// Brackets are parameters to construct the object pointed to
```

```
auto d2 = std::make_unique<watcher>(5);
```

```
std::cout<<"Watcher values: "<<(*d1).ii<<" and "<<d2->ii<<"\n";
```

Shared Pointer

- Shared pointer is for when you need many pointers to the same data
- Can be copied - increments reference count
- Again you can steal the data, or get a bare pointer to it, but these things take you outside the guarantees
- `std::shared_ptr` (in `<memory>` header)
- Again create data by creating pointer. When LAST pointer goes out of scope, data is cleaned up

Shared Pointer

- Syntax almost the same but with 'shared'
- Now allowed to copy - new pointer, SAME data
- New function on pointer itself -use_count - how many pointers exists

```
std::shared_ptr<watcher> d2 = std::make_shared<watcher>(5);
```

Explored in S2 and S2+

```
//Braces open a scope block
{
    // Get another shared pointer to existing data by copying
    auto d3 = d2; // d3 points to same as d2
    std::cout<<"Watcher with 5 has "<<d3.use_count()<<" live references\n";
}
```

Weak Pointer

- Weak pointer is a bit different
 - Does not manage its own data - in fact has to be created from a `shared_ptr` which does the management
 - Is NOT an owner - data will be destroyed when last `shared_ptr` is
 - Will KNOW if `shared_ptr` is still valid
- Very useful in niche circumstances!

Weak Pointer

- Create from shared pointer
- Does not increment reference count for d1

```
// Create a shared pointer
```

```
std::shared_ptr<watcher> d1 = std::make_shared<watcher>();
```

```
// Now create a weak pointer to this
```

```
std::weak_ptr<watcher> w1(d1);
```

Explored in S3

```
// Weak pointer did not increase reference count
```

```
std::cout<<"Shared pointer knows of "<<d1.use_count()<<" pointers to this data\n";
```

Weak Pointer

- To use, have to 'lock' - prevents data being destroyed "underneath us"
- Actually this just gets a copy of the shared pointer we started with, then it all "just works"

```
{
    auto ptr = w1.lock();
    std::cout<<"Now shared pointer knows of "<<d1.use_count()<<" pointers to this
data\n";
    if(ptr){
        //In here it is safe to use the data!
    }else{
        std::cout<<"Weak pointer now expired\n";
    }
    // When this block ends our ptr is destroyed, reference count goes back down
}
```

Explored in S3

Ownership

- Unique and shared are about managing ownership and lifetime, so passing one as a function argument implies function has part in that
- Unique pointer can't be copied, new one created by "moving" the data from it
 - Can't pass by value, pass by reference if ownership might transfer
- Shared pointer by value creates a new ownership, so favour by reference to avoid needless copy (in the call, and then when you go to store it)
- Many cases just pass the data pointer directly (`ptr.get()`) as you should usually know data remains valid for function duration!

Mistakes and Better Approaches

A decorative blue zigzag shape, resembling a stylized 'W' or a series of connected 'V' shapes, is positioned at the bottom of the slide, extending from the left edge towards the right.

Uses in the Wild

- Lastly, lets look at some occurrence of pointers "in the wild"
 - Reasons
 - Risks?
 - Modern alternative?

Uses in the Wild - 1

- Simple one first :
 - If you ever see 'malloc' applied to a class it is just wrong
 - Use new instead
 - And use delete in place of free
 - For arrays there is new[] and delete[]
 - X1 and X2 illustrate

Uses in the Wild - 2

- Common problem:
 - A class needs to access some data it does not have sole ownership of
 - Call it a "resource"
 - Give it a pointer?
 - IF you are careful, and make sure to clean up, might be OK
 - No help if you ever thread or parallelise this code!

Uses in the Wild - 2

- Better choices:
 - If class is really only meaningful with the resource, reference can be used
 - Especially const reference
 - Have to initialise in constructor
 - Cannot now assign one instance to another

Uses in the Wild - 2

- Better choices:
 - Smart pointer is ideal
 - Shared if class should keep resource alive
 - Weak if class just needs to know validity
- X3 illustrates all the options

Uses in the Wild - 3

- Common problem: Have a large data class and want to put it into a vector
 - If the vector grows, and has to re-lay in memory, objects will be copied
 - Solution 1: Unique pointer
 - Solution 2: implement "move" for your object
 - Allow a new object to be constructed that uses the old data, avoid copying the heavy stuff
- X4 explores this

Uses in the Wild - 4

- By the way, if your problem is really "I want to put this class in a vector but I haven't written copy constructors", just write them! They are not hard!

Uses in the Wild - 5

- Suppose a function creates some large entity and then returns it
- You might suppose it is better to return a pointer to it
- How do you get an entity back from a pointer without a copy?
 - Can use move, but this is complicated! Have to make sure to free things correctly
- Simply returning the value usually "just works" - **no copy is made**

Uses in the Wild - 5

- Secret is something called Return-Value-Optimisation, or RVO
- A kind of "copy-elision", which is one of the few violations of the "as-if" rule - compiler can remove copy even if it has side-effects
- Instead of creating memory for the value to be returned, compiler just constructs it right into the variable you're capturing to
- X5 explores this

Summary

A decorative blue geometric shape, resembling a stylized 'W' or a series of connected triangles, is positioned at the bottom of the slide, extending from the left edge towards the right.

Take Home

- Pointers are powerful, yet dangerous
 - Avoid them whenever you can
 - If possible, understand and use the smart pointers instead
- Almost always, any "clever" trick you try with a pointer is actually undefined behaviour
- Undefined behaviour is extra bad because it will often seem to do just the right thing