

Translation units & the Preprocessor

Welcome!

- This is the fundamental “how does C++ compilation work” session
- C++ compiles very differently to most other modern programming languages
- If you don’t understand how things are compiled, you’re going to get confused very quickly
- We’re going to look at how C++ compiles, how it compares with C# and everything that goes on under the hood
- **Aim:** You understand the path your code takes from editor to the compiled binary

How does C# compile?

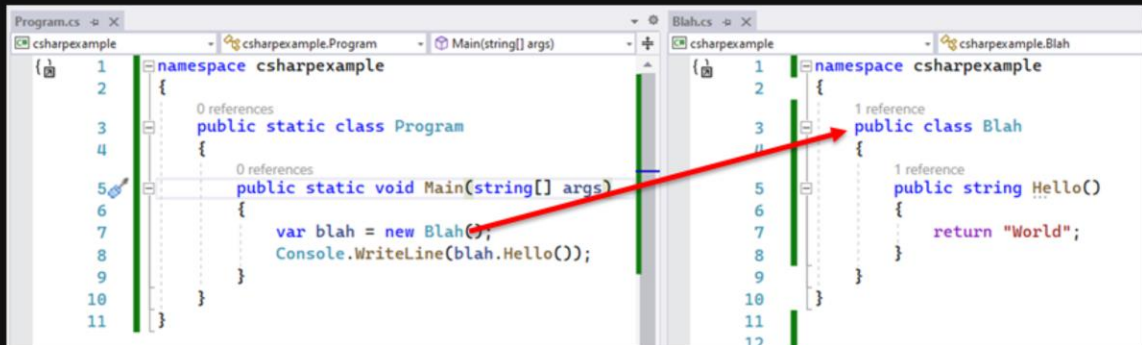
```
Program.cs
1 namespace csharpexample
2 {
3     0 references
4     public static class Program
5     {
6         0 references
7         public static void Main(string[] args)
8         {
9             var blah = new Blah();
10            Console.WriteLine(blah.Hello());
11        }
12    }
13 }
```

```
Blah.cs
1 namespace csharpexample
2 {
3     1 reference
4     public class Blah
5     {
6         1 reference
7         public string Hello()
8         {
9             return "World";
10        }
11    }
12 }
```

This works because the C# compiler reads all files at once:

```
csc.exe /out:obj\Debug\net6.0\csharpexample.dll Blah.cs Program.cs
```

How does C# compile?



```
Program.cs
1 namespace csharpexample
2 {
3     0 references
4     public static class Program
5     {
6         0 references
7         public static void Main(string[] args)
8         {
9             var blah = new Blah();
10             Console.WriteLine(blah.Hello());
11         }
12     }
13 }
```

```
Blah.cs
1 namespace csharpexample
2 {
3     1 reference
4     public class Blah
5     {
6         1 reference
7         public string Hello()
8         {
9             return "World";
10         }
11     }
12 }
```

Because it reads and parses all files at once, it can know what "Blah" refers to in Program.cs!

C++ does not compile this way

C++ compiles one file at a time

Each file is a "translation unit"

digital

pdp11/70

PSR ERR ADDR ERR RUN PAUSE MASTER USER SUPER KERNEL DATA 16 ADDRESSING 18 22

USER D USER I
SUPER D SUPER I
KERNEL D KERNEL I
CONS PHY PROG PHY

DATA PATHS J/L ADDR FFFF/CPU
BUS REG DISPLAY REGISTER

POWER LOCK
PARITY HIGH LOW
OFF

1972

21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
LOAD ADDR EXAM DEP CONT ENABLE S INET
HALT S BUS CYCLE START



Speculative history time

- Turns out it's quite difficult to find rational for decisions made 50 years ago
- But we can make reasonable guesses as to why the design is the way it is
- So this will partly be a *speculative* history explanation
- Stuff that I could find is from "The Development of the C Language"
<https://www.bell-labs.com/usr/dmr/www/chist.html>

- The Development of the C Language was written in 1993
- Even they don't know why they did some of the things they did

Constraints of PDP-7/PDP-11

- C originally written on PDP-7, later ported to PDP-11
- PDP-7 had **8KB memory**. The PDP-11 had **24KB memory**. Total.
- Extremely memory constrained for compilers to operate in.

What does a compiler need to do?

- Read files
 - Lex (tokenise)
 - Parse
 - Abstract syntax tree
 - Type resolution
 - Emit machine code
-
- If you had all the source code in one file, you would run into a problem.
 - While the resulting executable would work within memory limitations, there was not enough memory to compile the whole program at once.

Split up the work

Compiler

Compile individual .c files to object files

- Read one file
- Lex (tokenise)
- Parse
- Abstract syntax tree
- Type resolution
- Emit machine code as objects

Only need enough memory for compiler
to compile each .c file in sequence...

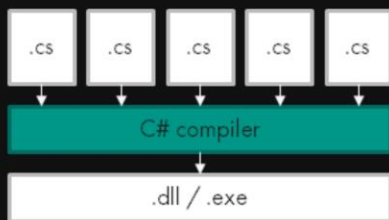
Linker

Combine object files to get final program

- Read machine code in files
- Stitch machine code together
- Replace placeholder function pointers
with actual function locations
- Emit resulting executable

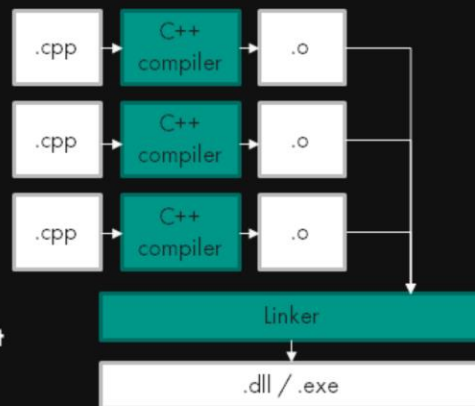
Comparison again...

C#



C++ compiler does not know what is in other .cpp files!

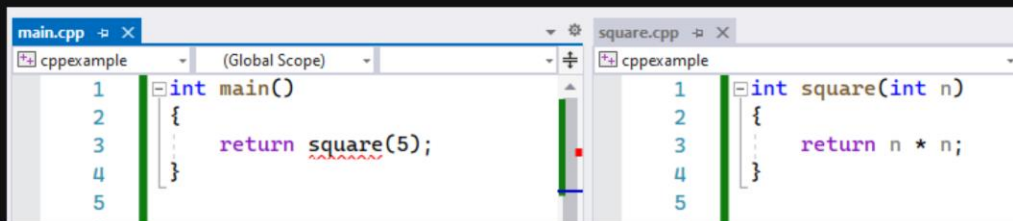
C++



How do we reference across files?

Let's say we want to call `square()` from `main()`.

Given these files will be compiled separately, how can we tell the compiler what "square" looks like in `main.cpp`?



The image shows two side-by-side code editors. The left editor, titled 'main.cpp', contains the following code:

```
1 int main()  
2 {  
3     return square(5);  
4 }  
5
```

The right editor, titled 'square.cpp', contains the following code:

```
1 int square(int n)  
2 {  
3     return n * n;  
4 }  
5
```

Forward declarations!

```
main.cpp  x
cppexample  (Global Scope)
1  // forward declaration:
2  // "square" will be compiled
3  // elsewhere and available
4  // when linking (i.e. it's
5  // machine code will be
6  // inside some .o file
7  // that is passed to the
8  // linker).
9  int square(int n);
10
11 int main()
12 {
13     return square(5);
14 }
15

square.cpp  x
cppexample  (Global Scope)
1
2
3
4
5
6
7
8
9 int square(int n)
10 {
11     return n * n;
12 }
13
14
15
```

C++: compiling to executable vs object file

Input

```
int square(int n)
{
    return n * n;
}

int main()
{
    return square(5);
}
```

Compiler knows address of "square" in this case.

Straight to .exe file

```
main:
55
push %rbp
48 89 e5
mov %rsp,%rbp
bf 05 00 00 00
mov $0x5,%edi
e8 e3 ff ff ff
call 401106 <.Z6squarei>
90
nop
5d
pop %rbp
c3
ret
66 2e 0f 1f 84 00 00 00 00
cs nopw 0x0(%rax,%rax,1)
```

Input

```
int square(int n);

int main()
{
    return square(5);
}
```

Compile does not know where "square" is!

Leaves it as 0x00000000.

To object .o file

```
main:
55
push %rbp
48 89 e5
mov %rsp,%rbp
bf 05 00 00 00
mov $0x5,%edi
e8 00 00 00 00
call e <main+0xe>
90
nop
5d
pop %rbp
c3
ret
```

Linker will fill it in later when it has all the object files.

- Left hand side: compiler knows the address of square so it can just put it straight in there
- Right hand side: compiler does not know where square is! It leaves it blank for the linker to fill in later.

What does the linker do?

Stitches together object .o files to create the executable.

Side note: Also this is where library references are linked in!

square.o

```
_Z6squarei:
55
push    %rbp
48 89 e5    mov     %rbp,%rbp
89 7d fc    mov     %edi,-0x4(%rbp)
8b 45 fc    mov     -0x4(%rbp),%eax
0f af c0    imul    %eax,%eax
5d         pop     %rbp
c3         ret
```

+

main.o

```
main:
55
push    %rbp
48 89 e5    mov     %rbp,%rbp
bf 05 00 00 00 mov     $0x5,%edi
e8 00 00 00 00 mov     %eax,%edi
call    e <main+0xe>
90         nop
5d         pop     %rbp
c3         ret
```

=

main.o will contain extra data (not shown) that tells the linker "please fill in call at XYZ with _Z6squarei function pointer".

```
_Z6squarei:
55
401106 push    %rbp
48 89 e5    mov     %rbp,%rbp
401107 mov     %edi,-0x4(%rbp)
8b 45 fc    mov     -0x4(%rbp),%eax
40110d imul    %eax,%eax
0f af c0
401110 pop     %rbp
5d
401113 ret
c3
401114 main:
55
401115 push    %rbp
48 89 e5    mov     %rbp,%rbp
bf 05 00 00 00 mov     $0x5,%edi
401119 mov     %eax,%edi
e8 e3 ff ff ff call    401106 <_Z6squarei>
90
401123 nop
5d
401124 pop     %rbp
c3
401125 ret
66 2e 0f 1f 84 00 00 00 00 00
401126 cs nopl 0x0(%rax,%rax,1)
```



- Note how the linker replaced the call location with the actual location of the square() function.
- This is why if you forward declare something but then *don't actually implement it* you get linker errors.
- As far as the C++ compiler is concerned, the function you declared will be available but then the linker can't find it.
- The linker only has machine code and the raw symbol names though: **that's why linker errors can be super cryptic!**

What to do when lots of .cpp files need to forward declare the same function?

- Could forward declare in every file, but tedious.
- Pre-processor created so you can `#include` common code into multiple translation units
- This is a LITERAL TEXT INCLUDE – nothing fancy. You can even run the pre-processor on non-C++ code because it doesn't care about C/C++ code at all.
- So what does the compiler actually see when you compile some normal piece of code?

What does the pre-processor pass to the compiler?

```
#include <stdio.h>

int main() {
    printf("hello\n");
}
```

4 lines of code

expands to:

551 lines of code
that the compiler
actually parses and
compiles!

[illegible]

- Ran out of space on this slide for all the emitted code...!
- This is for a single, very core include of "stdio.h".
- Imagine how many lines of code the Unreal Engine headers expand out to.
- This is why syntax errors in headers can cause problems elsewhere!
 - If you're missing a brace or semicolon in the header, when it all gets expanded into a single file that can cause syntax problems in the files further down.

Side note: Seems like a lot of redundant effort...

- Imagine lots of files that are all including the same headers.
- Each compiler invocation has to parse all of that header code every time.
- Slow and redundant!
- Introducing PCHs: Pre-compiled headers.
- Special compiler mode to compile a set of headers down into a special binary format. Probably a serialized AST (not machine code!)
- Each compiler then loads PCH instead of re-parsing all common headers – speeds up compilation.

Other things the preprocessor does

- Pre-processor also lets you define macros with `#define FUNC(...) ...`
- Again this is just text processing, so you can emit anything with the pre-processor including things that aren't syntactically valid.
- Macros are very very important to the `UCLASS()` / `UObject` system which we will talk about later.
- But for now important take-away points are...

THE SUPER IMPORTANT SUMMARY SLIDE

- Each .cpp file is compiled on it's own!
- We use forward declarations so .cpp files can know about things that will be compiled in another translation unit and available to the linker.
- We put forward declarations in headers because we don't want to copy-paste them to every .cpp file manually.
- Compiler actually compiles the fully expanded output from the pre-processor, not just your file!
- All headers and macros are fully expanded before the compiler does any parsing.
- Think about how the #include and macro use is expanding in your file if you're running nearby syntax errors!



- This is the super important summary slide.
- If you pay attention to only one thing in this session, make sure it is this slide!
- Sure seems like C++ is hacks upon hacks upon hacks..... That's because it is ;P