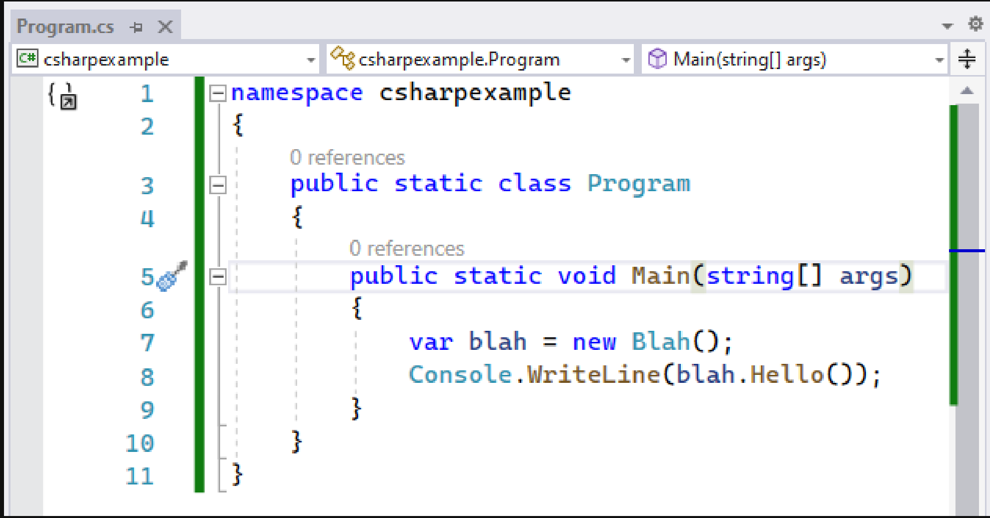
# Translation units & the Preprocessor

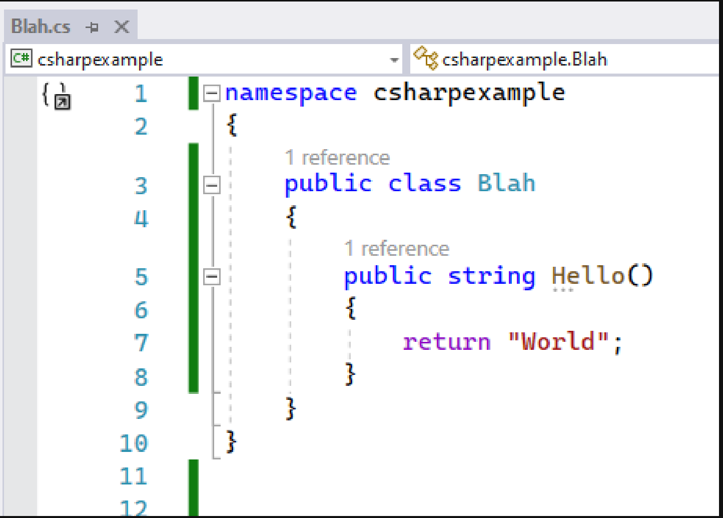
## Welcome!

- 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?



```
namespace csharpexample
{
    0 references
    public static class Program
    {
        0 references
        public static void Main(string[] args)
        {
            var blah = new Blah();
            Console.WriteLine(blah.Hello());
        }
    }
}
```



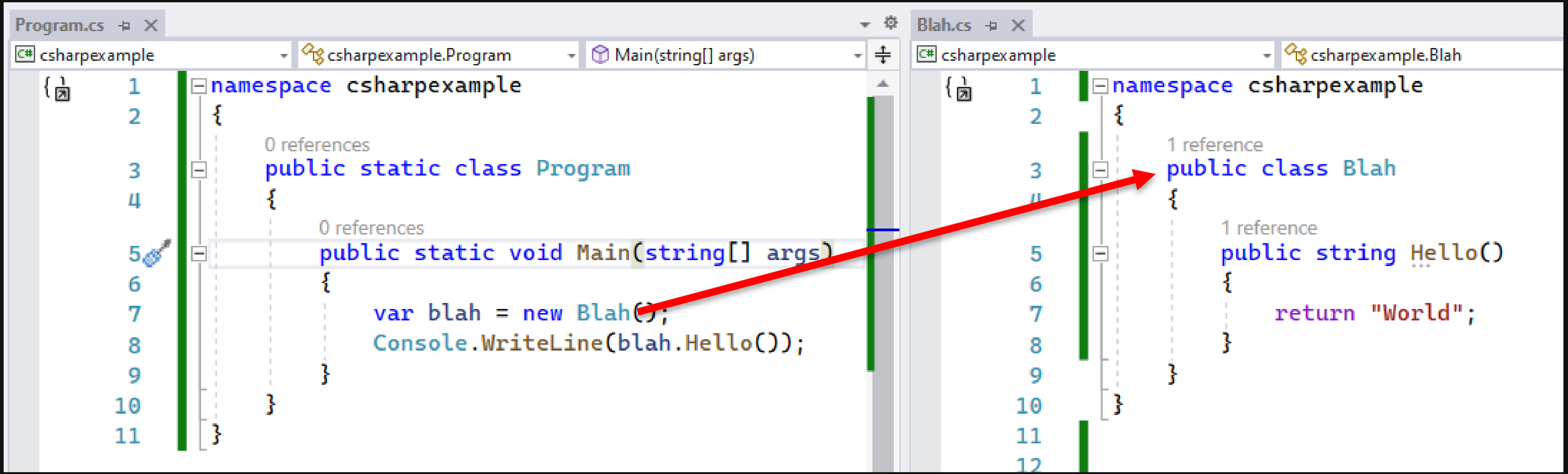
```
namespace csharpexample
{
    1 reference
    public class Blah
    {
        1 reference
        public string Hello()
        {
            return "World";
        }
    }
}
```

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

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

REDPOINT TECHNICALLY GAMES

# How does C# compile?



```csharp
namespace csharpexample
{
    public static class Program
    {
        public static void Main(string[] args)
        {
            var blah = new Blah();
            Console.WriteLine(blah.Hello());
        }
    }
}
```

```csharp
namespace csharpexample
{
    public class Blah
    {
        public string Hello()
        {
            return "World";
        }
    }
}
```

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

C++ <u>does not</u> compile this way

## C++ compiles <u>one file at a time</u>

Each file is a “translation unit”

digital
pdp11/70
PAR ERR
ADRS ERR
RUN
PAUSE
MASTER
USER
SUPER
KERNEL
DATA
ADDRESSING
16
18
22
USER D
USER I
SUPER D
SUPER I
KERNEL D
KERNEL I
CONS PHY
PROG PHY
POWER
LOCK
OFF
PARITY
HIGH
LOW
DATA PATHS
µ ADRS FPP/CPU
BUS REG
DISPLAY REGISTER
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
LOAD ADRS
EXAM
DEP
CONT
ENABLE
HALT
S INST
S BUS CYCLE
START

## 1972

## 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

## 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

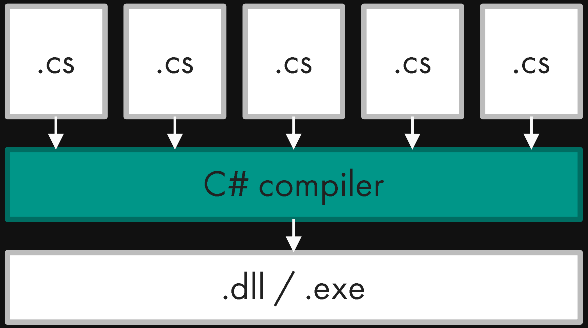
Only need enough memory for compiler to compile each .c file in sequence...

**Linker**
Combine object files to get final program

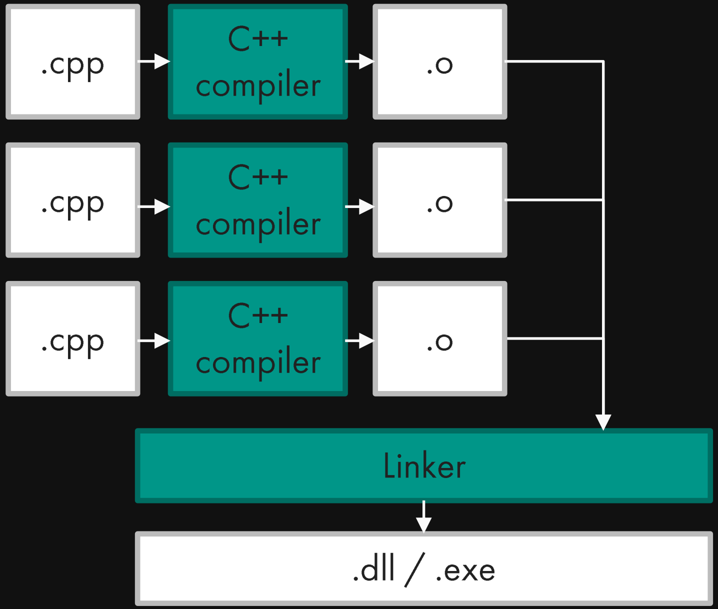
- Read machine code in files
- Stitch machine code together
- Replace placeholder function pointers with actual function locations
- Emit resulting executable

# Comparison again…

C#



C++

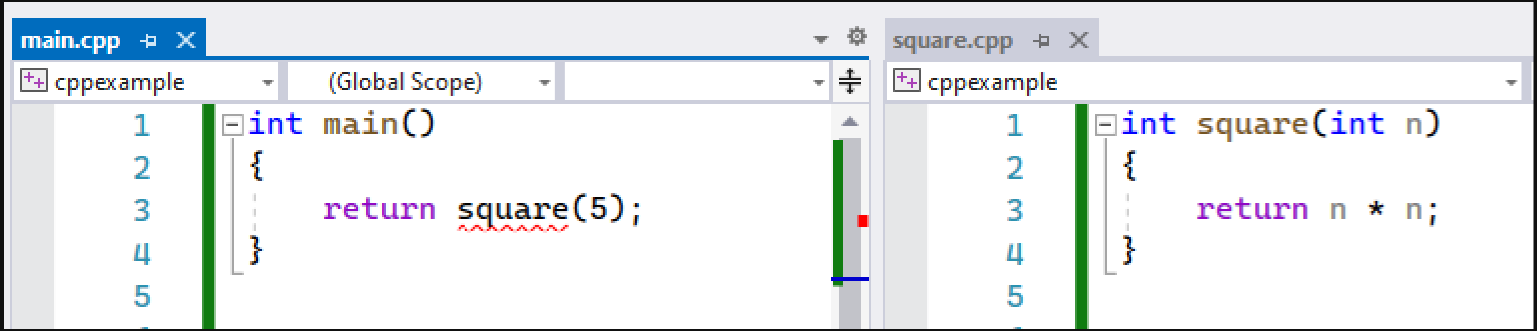


C++ compiler <u>does not know</u> what is in other .cpp files!

# How do we reference across files?

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

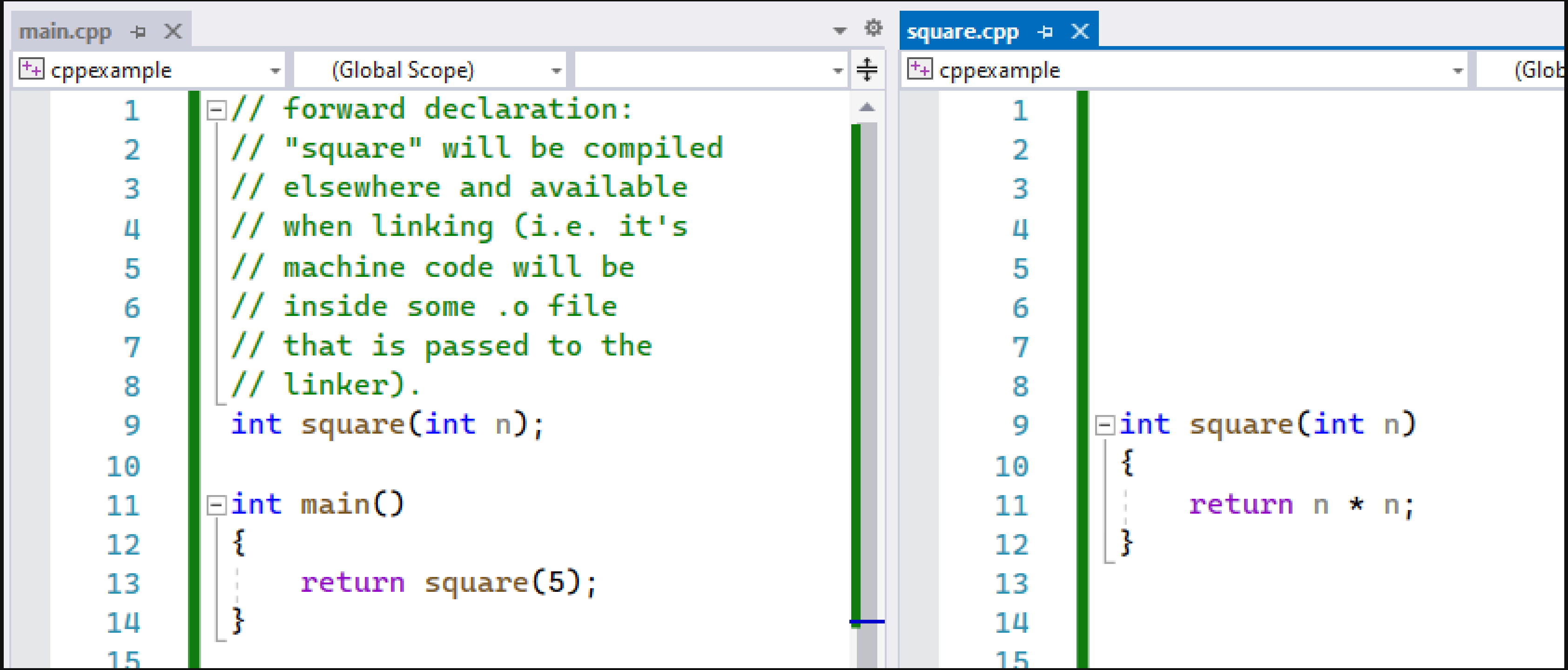
Given these files will be compiled separately, how can we tell the compiler what "square" looks like in main.cpp?



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

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

# Forward declarations!

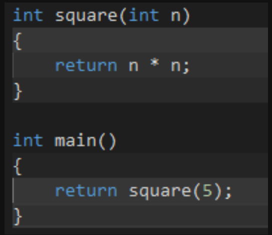


```cpp
// forward declaration:
// "square" will be compiled
// elsewhere and available
// when linking (i.e. it's
// machine code will be
// inside some .o file
// that is passed to the
// linker).
int square(int n);

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

```cpp








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

# C++: compiling to executable vs object file

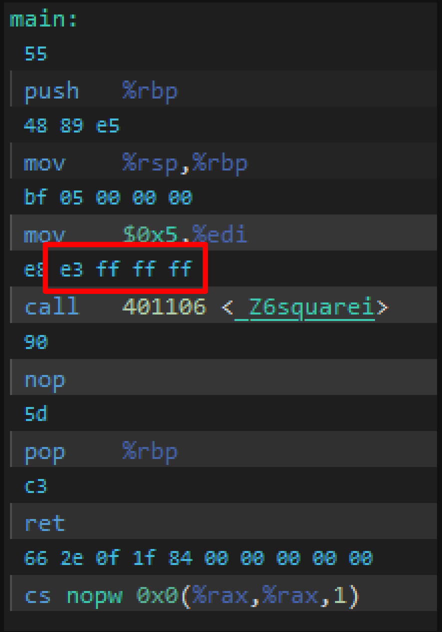
## Input



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

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

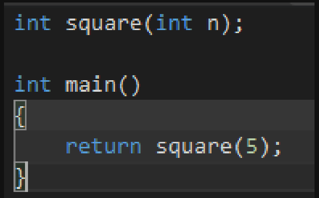
Compiler knows address of "square" in this case.

## 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 00
 cs nopw 0x0(%rax,%rax,1)
```

## Input

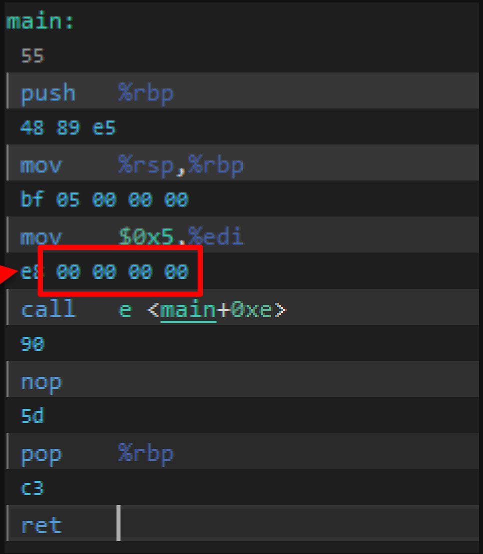


```
int square(int n);

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
```

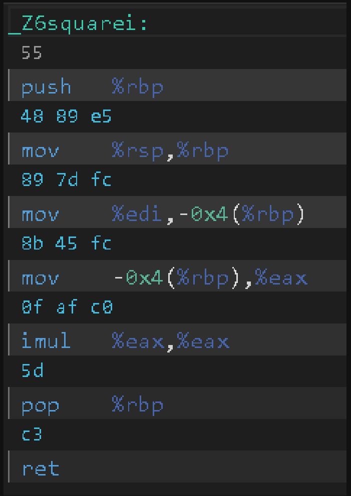
Linker will fill it in later when it has all the object files.

# 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!

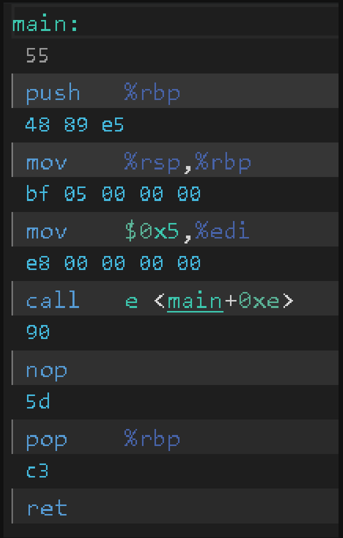
square.o



```
_Z6squarei:
 55
 push    %rbp
 48 89 e5
 mov     %rsp,%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
```

+

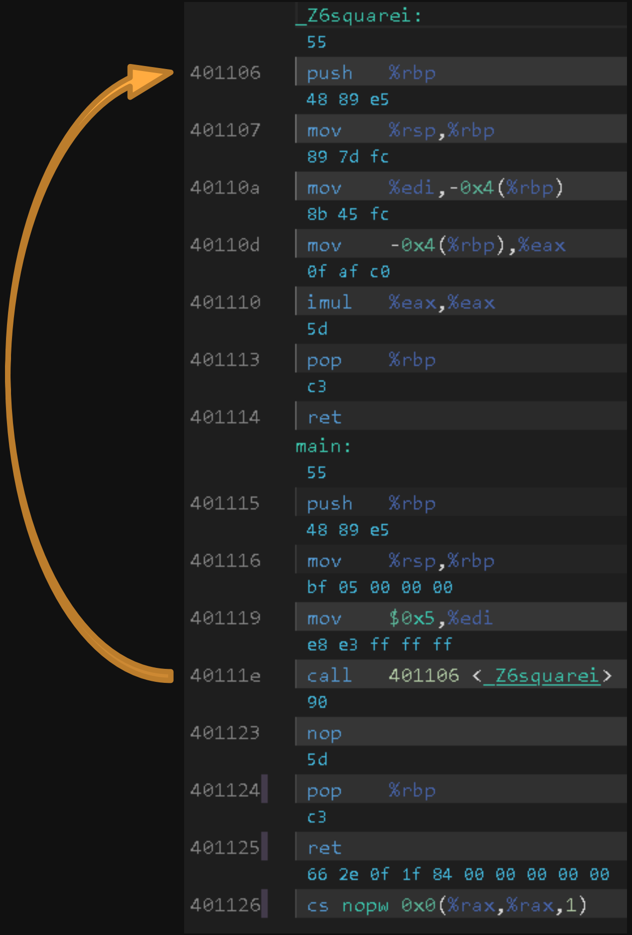
main.o



```
main:
 55
 push    %rbp
 48 89 e5
 mov     %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
```

=

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
401107     mov     %rsp,%rbp
           89 7d fc
40110a     mov     %edi,-0x4(%rbp)
           8b 45 fc
40110d     mov     -0x4(%rbp),%eax
           0f af c0
401110     imul    %eax,%eax
           5d
401113     pop     %rbp
           c3
401114     ret
          main:
           55
401115     push    %rbp
           48 89 e5
401116     mov     %rsp,%rbp
           bf 05 00 00 00
401119     mov     $0x5,%edi
           e8 e3 ff ff ff
40111e     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 nopw 0x0(%rax,%rax,1)
```

## 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?

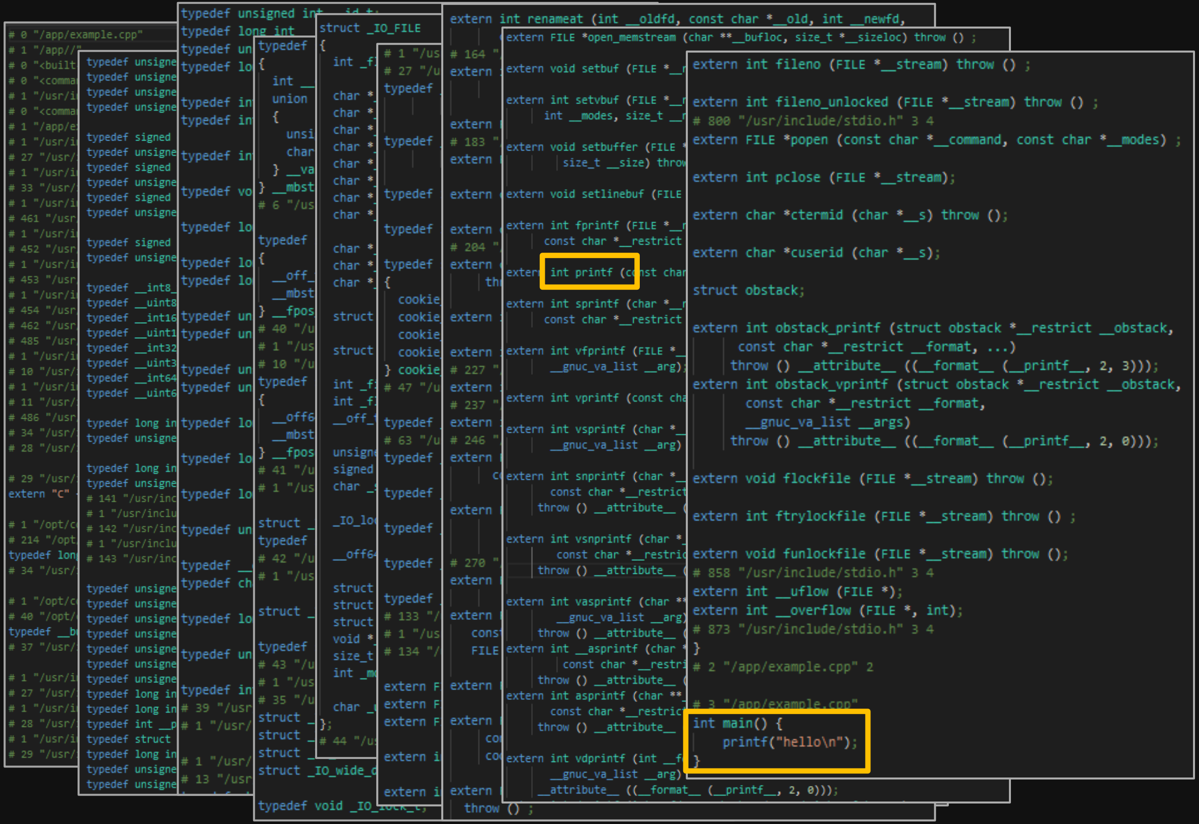
```cpp
#include <stdio.h>

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

→

4 lines of code

**expands to:**

<u>551 lines of code</u> that the compiler actually parses and compiles!



```cpp
extern int fileno (FILE *__stream) throw () ;

extern int fileno_unlocked (FILE *__stream) throw () ;
# 800 "/usr/include/stdio.h" 3 4
extern FILE *popen (const char *__command, const char *__modes) ;

extern int pclose (FILE *__stream);

extern char *ctermid (char *__s) throw ();

extern char *cuserid (char *__s);

struct obstack;

extern int obstack_printf (struct obstack *__restrict __obstack,
      const char *__restrict __format, ...)
     throw () __attribute__ ((__format__ (__printf__, 2, 3)));
extern int obstack_vprintf (struct obstack *__restrict __obstack,
       const char *__restrict __format,
       __gnuc_va_list __args)
     throw () __attribute__ ((__format__ (__printf__, 2, 0)));

extern void flockfile (FILE *__stream) throw ();

extern int ftrylockfile (FILE *__stream) throw () ;

extern void funlockfile (FILE *__stream) throw ();
# 858 "/usr/include/stdio.h" 3 4
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
# 873 "/usr/include/stdio.h" 3 4
}
# 2 "/app/example.cpp" 2

# 3 "/app/example.cpp"
int main() {
    printf("hello\n");
}
```

## *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!