Systems Programming

Lecture 1: Introduction

Stuart James

stuart.a.james@durham.ac.uk

Structure of Module

Module Coordinator: Dr Stuart James

- Term 1: Systems Programming (C, UNIX command line, Makefiles, C++) -> Dr Stuart James
- Term 2, first half: Functional Programming (Haskell) -> Dr Maximilien Gadouleau
- Term 2, second half: Object-Oriented Programming -> Dr Nelly Bencomo
- For all academic related questions and issues and module content, contact the lecturer of that component.
- For any general organisational questions about the module, contact module leader
    - We're happy to help!

Key topics for this sub-module

- UNIX/Linux shell programming
- Syntax and semantics of the C and C++ programming language
- Memory access and management
- Design of large programs in non-object-oriented language

Organisation

Practicals:

- Start in week 2
- Very important, you will learn most by trying things out yourself.

Module Requirements:

- Some background assumed in programming
- No C/C++ knowledge assumed

Organisation

Formative Assessment:

- Same format as summative
- See what is expected in the summative

Summative Assessment

- Hand-out: Week 4
- 100% of mark for this submodule
- 50% of mark for module

Exam

- Functional (Term 2, first) and Object Oriented Programming (Term 2, second) assessed.
- 50% of mark for module

Feedback is welcome

1. Please do let me and the other lecturers know how we are doing.
   - are we going too fast? too slow?
   - what is going well? badly?
2. Don't hesitate to let us know if there is something you can't see/hear.

Changes from 2023/2024

We always strive to improve. This year, we have changed:

- Coursework: Focus on computational performance as opposed to a single algorithm. More Autograder support to help get basic code running.
- Adding in more systematic C++ teaching through the module
- Thinking about AI or Code Completion tools*

What is "Systems Programming"?

- Involves the development of the individual pieces of software that allow the entire system to function as a single unit.
- Aims to produce software and software platforms which provide services to other software, are performance constrained, or both.
- Examples include Operating systems, computational science applications, game engines, industrial automation, and software as a service applications.
- Requires a great degree of hardware awareness and efficient use of available resources, because
    - software itself is performance critical
    - efficiency improvements lead to savings of time or money

Why C?

- Provides low-level access to memory
- A simple set of keywords
- Efficient and flexible

Many other languages have borrowed syntax/features directly or indirectly from C.

Why C++?

- Provides low-level access to memory
- A simple set of keywords
- Efficient and flexible

However!

- Provides more advanced programming techniques (Classes etc.)

Many other languages have borrowed syntax/features directly or indirectly from C / C++.

Resources and Books

- The good reference text for C programming is
  - The C Programming Language, Kernighan and Ritchie, Second Edition, Prentice Hall, ISBN 0-13-110362-8
  - Exercise answers: https://web.archive.org/web/*/http://www.trunix.org/programlama/c/kandr2/
- Based on the Kernighan and Ritchie book Steve Summit has a good set of free tutorial notes on C programming:
  - http://www.eskimo.com/~scs/cclass/

Resources and Books

- An excellent and comprehensive modern book is:
  - C Programming A Modern Approach, K.N. King, Second Edition, ISBN 978-0-393-97950-3
- See https://stackoverflow.com/questions/562303/the-definitive-c-book-guide-and-list for further book suggestions.
- Try to practice writing code more than you read
  - Site provides very short tasks and shows you other solutions to the problem
  - Code wars: https://www.codewars.com/

Resources and Books

- These slides!

To use:

- Install gcc
- Install g++
- Install jupyter
- Install jupyter-rise (https://rise.readthedocs.io/en/latest/)
- Install the C Kernel https://github.com/brendan-rius/jupyter-c-kernel
- Install the C++ Kernel https://github.com/StTu/jupyter-cpp-kernel

Resources and Books

Why use seperate kernels for C or C++?

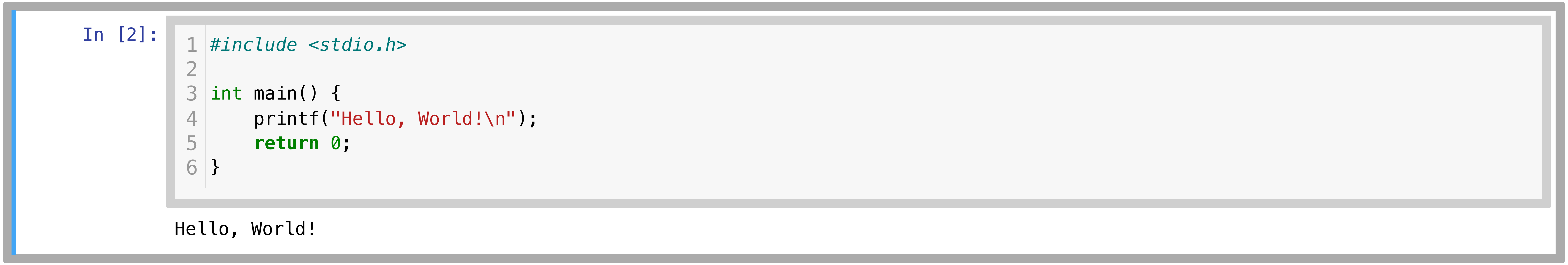
Resources and Books

Why use seperate kernels for C or C++?

- Learn language specifc syntax
- Learn about the limitation of different compilers (C++ is very generous!)

A First Program

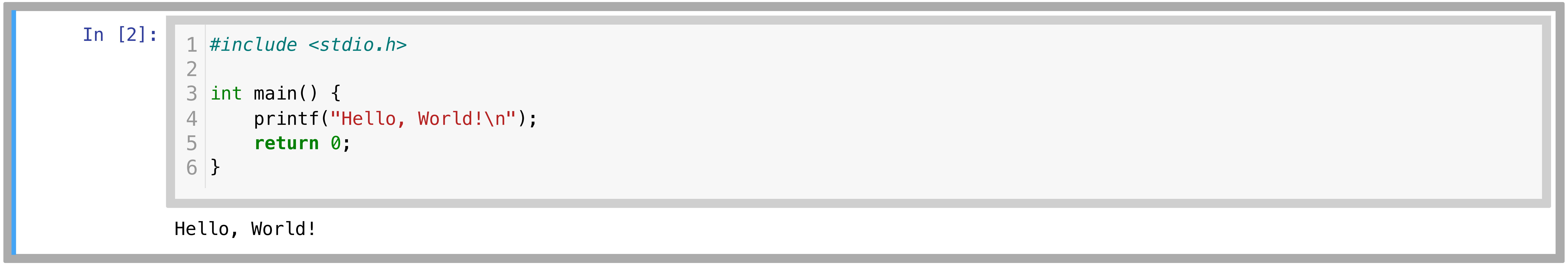
A First Program



```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

```
Hello, World!
```

A First Program



```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

```
Hello, World!
```

Saved in a file with a "`.c`" file extension, for example "`helloworld.c`"

A First Program

In [2]:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

```
Hello, World!
```

Saved in a file with a "`.c`" file extension, for example "`helloworld.c`"

Let's go through this program line by line.

Pre-processor Directives

- Lines that start with a `#` are commands to the C pre-processor

```
#include <stdio.h>
```

- looks for the source code file `stdio.h` and includes it before compilation
- `stdio.h` is a file required to use the standard input and output library

The `main()` Function Declaration

```c
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

- All C programs have an entry function called `main()`. This is called by the runtime system to start your program running.
- You can only have one of these.

The `printf()` Function Call

```c
printf("Hello, World!\n");
```

- Function call to `printf()` which implements formatted text printing to the console window.
- The string argument includes an escape sequence '`\n`'
    - this generates a newline character

Function `return` Statement

```
return 0;
```

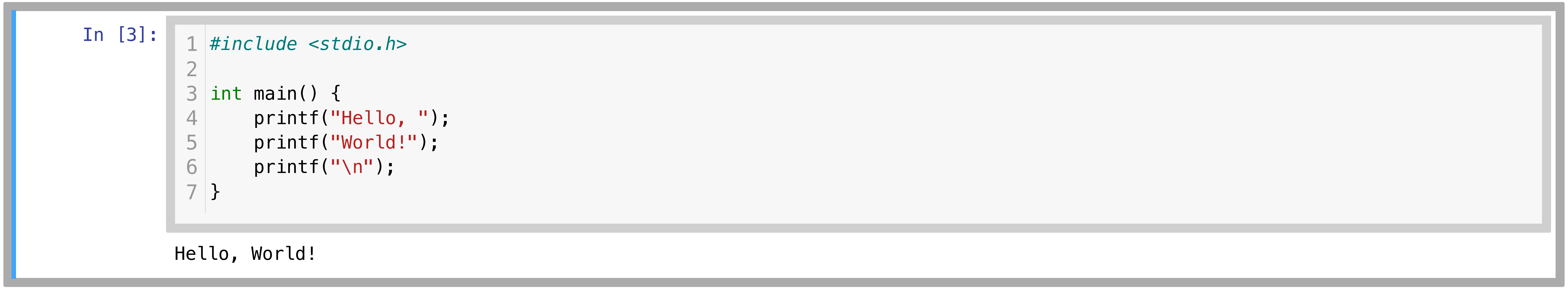
- UNIX programs often return a zero value to indicate they have exited normally.
- If there is no return statement, this will not cause a problem at compile-time.
- If the return value is of the wrong type this may cause a warning at compile-time or a problem at run-time.

Structure of any C project

- Must have a `main()` function.
- Only functions called in the main will be executed
- Cannot have more than one main even in seperate files (will not compile)

Alternative version of the program

Alternative version of the program



In [3]:

```c
#include <stdio.h>

int main() {
    printf("Hello, ");
    printf("World!");
    printf("\n");
}
```

```
Hello, World!
```

Alternative version of the program

In [3]:

```c
#include <stdio.h>

int main() {
    printf("Hello, ");
    printf("World!");
    printf("\n");
}
```

```
Hello, World!
```

- This produces identical output to the first program

A Temperature Converter

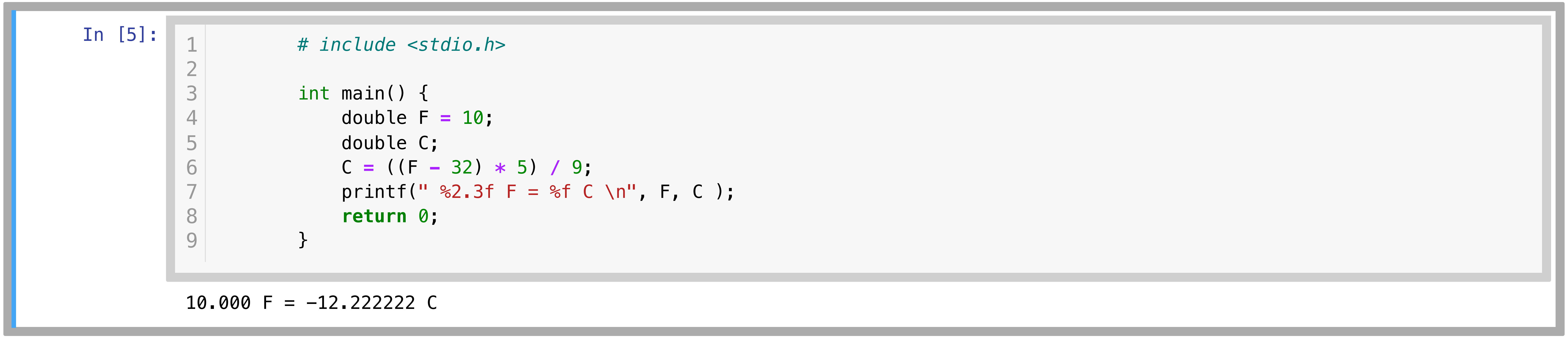
A Temperature Converter

include <stdio.h>

```
int tempConv(int F){ return ((F - 32) * 5) / 9; }

int main() { int F = 10; int C; C = tempConv(F); printf(" %d F = %d C \n", F, C ); }
```

- This code fragment converts a temperature from Fahrenheit to Celsius and prints the result
- We could change the variable `C` to a `double`
    - Store a floating point number
    - We would need to change the output format

- This code fragment converts a temperature from Fahrenheit to Celsius and prints the result
- We could change the variable `C` to a `double`
  - Store a floating point number
  - We would need to change the output format



In [5]:

```c
        # include <stdio.h>

        int main() {
            double F = 10;
            double C;
            C = ((F - 32) * 5) / 9;
            printf(" %2.3f F = %f C \n", F, C );
            return 0;
        }
```

```
10.000 F = -12.222222 C
```

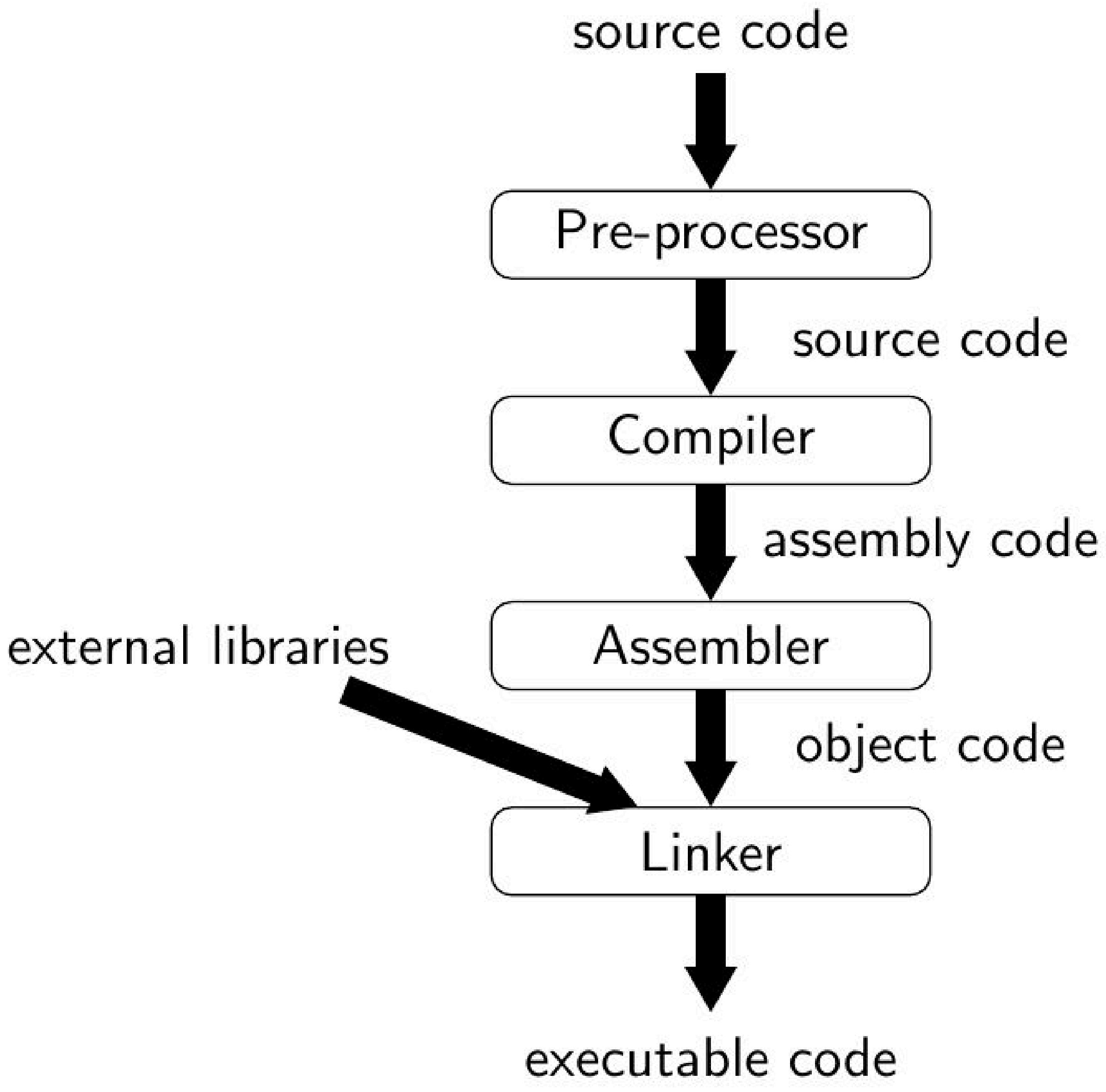
`printf()`

- So popular it was added to Java in 5.0
- Variable number of parameters (also added to Java 5.0)
- We can decide the number of characters to output
- Dot followed by number -- number of decimal places

`printf()`

- First parameter explains how the rest are to be formatted using
  - `%d` signed decimal (`int`)
  - `%u` unsigned decimal
  - `%o`, `%x` octal, hexadecimal
  - `%l` long
  - `%f` floating point so `%1.2f` will give `3.14`
  - `%e` floating point (exponent form)
  - `%c`,`%s` character, string

Compiling


source code
Pre-processor
source code
Compiler
assembly code
external libraries
Assembler
object code
Linker
executable code

Compiling

For now let's look at gcc

```
gcc -o outfile file.c
```

- Use `-o` to name the output
    - if not used, the default name is `a.out`
- Use `-E` option to do pre-processing only, or call `cpp`
- Use `-S` option to go as far as compilation only
- Use `-c` option to go as far as assembly only
- Use `nm` tool to investigate object libraries

The C Pre-processor

- Directives such as `#define` and `#include` are handled by the *pre-processor*, a piece of software that edits C programs just prior to compilation.

  ```
  #define PI 3.1415
  ```

- Its reliance on a pre-processor makes C (and C++) unique among major programming languages

The C Pre-processor `#include`

- For system header files use:

```
#include <stdio.h>
```

- Looks for the file `stdio.h` in C's include file directories
  - If `< >` are used, then `/usr/include` is prioritised in UNIX by convention.
  - If `" "` are used, then the current working directory is prioritised.

The C Pre-processor `#include`

- For user header files use:

```
#include "fibonacci.h"
```

This searches in current directory first then in system directories

- We can use the following as one of the `gcc` option:

```
-I path
```

This adds the directory `path` to the search path for include files when using `gcc`

Definitions (Macro)

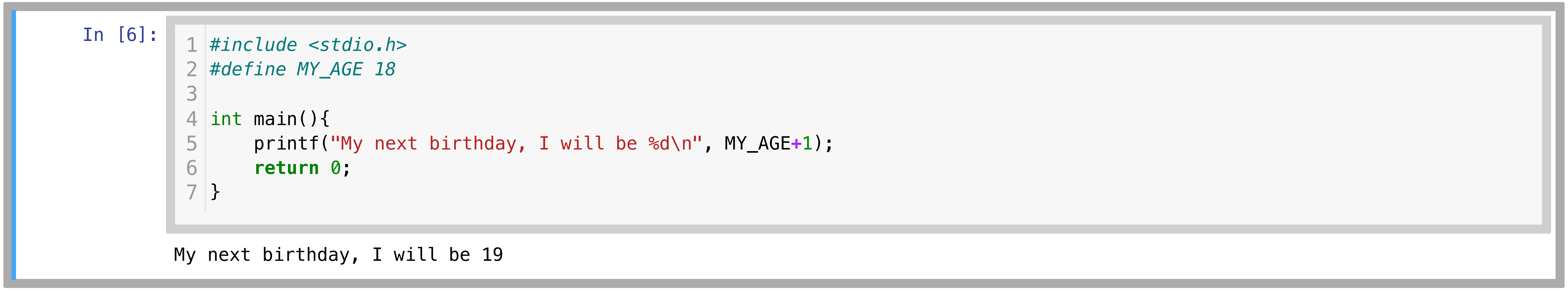
- Used to provide definitions in code:

```
#define MY_AGE 18

...

int nextBirthday = MY_AGE + 1;
```

Definitions (Macro)

- Used to provide definitions in code:

```c
#define MY_AGE 18

...

int nextBirthday = MY_AGE + 1;
```



```c
#include <stdio.h>
#define MY_AGE 18

int main(){
    printf("My next birthday, I will be %d\n", MY_AGE+1);
    return 0;
}
```

```
My next birthday, I will be 19
```

Definitions

- Used to provide definitions in code:

```
#define A_NAME A_VALUE
```

- Can also specify name and value at compile time:

```
gcc -DMY_AGE=18 myProgram.c
```

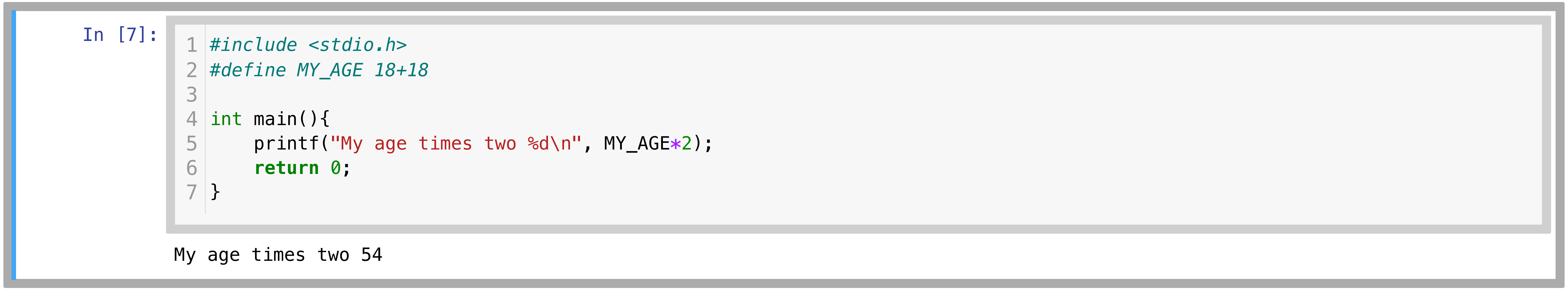
- Pre-processor performs a search and replace of `A_NAME` for `A_VALUE`

Definitions

- Be careful, do not treat these macros like variables!
- What will this do?

Definitions

- Be careful, do not treat these macros like variables!
- What will this do?



In [7]:

```c
#include <stdio.h>
#define MY_AGE 18+18

int main(){
    printf("My age times two %d\n", MY_AGE*2);
    return 0;
}
```

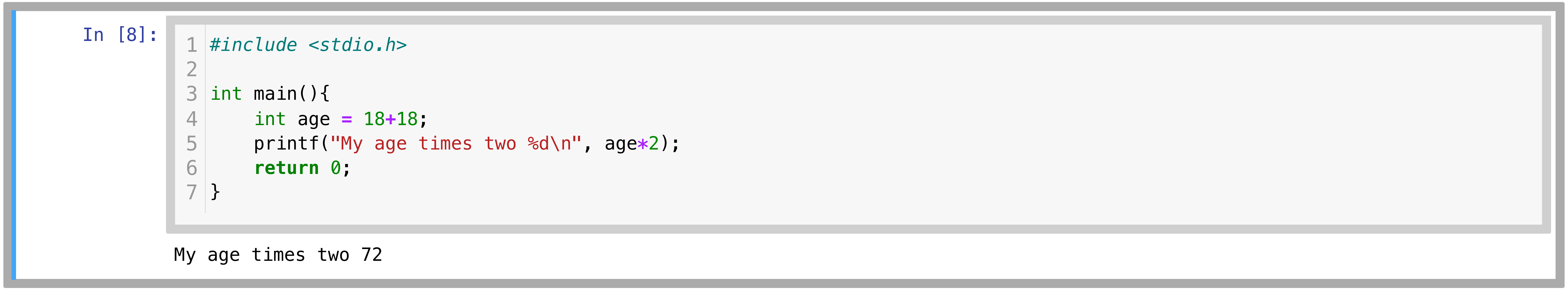
```
My age times two 54
```

Definitions

- What will this do?

Definitions

- What will this do?



```c
#include <stdio.h>

int main(){
    int age = 18+18;
    printf("My age times two %d\n", age*2);
    return 0;
}
```

```
My age times two 72
```

Conditionals

```
#ifdef A_NAME // tests if A_NAME is #defined
  <program text>
#else
  <program text>
#endif
```

Conditionals

- Can also test for the lack of `A_NAME`:

```
#ifndef A_NAME // tests if A_NAME is not #defined
    <program text>
#else
    <program text>
#endif
```

Conditional compilation for debugging

Efficiency can be very important in "systems programming" so actual `if` statements can be expensive.

```
#define MY_DEBUG // define an identifier

#ifdef MY_DEBUG
     assert( i > 0 );
     printf( "i is  %d  \n",  i );
#endif
```

Conditional compilation for debugging

- This allows the inclusion of your debugging code only when `MY_DEBUG` is defined
- No overhead is generated when it is not defined since no code is included for compilation (compared to a standard `if` statement)
- Can also use `#ifndef` tests if an identifier is not defined

Parameterised macro definitions

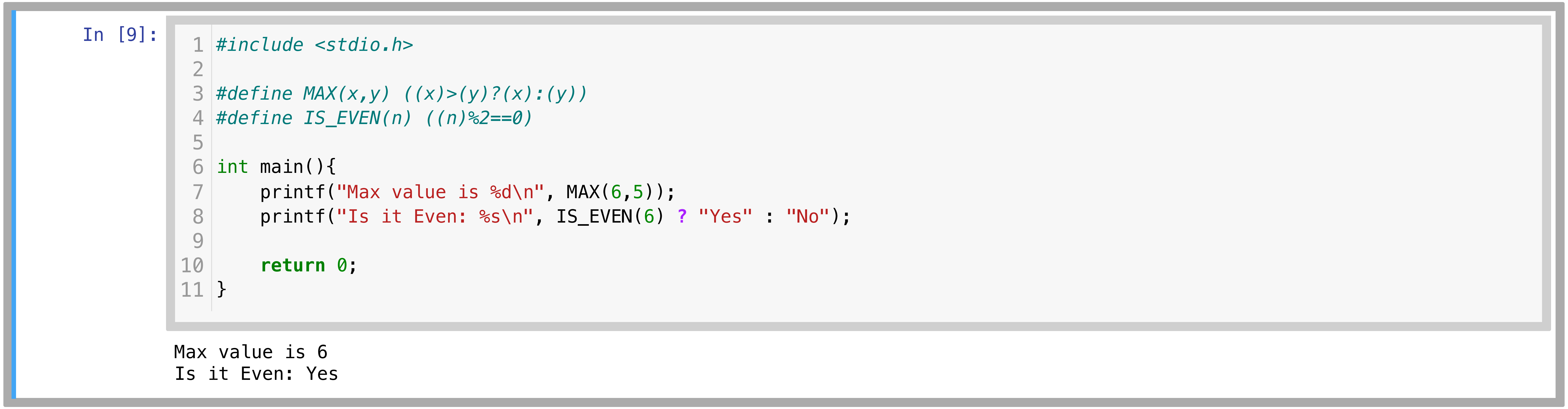
- Definition of a *parameterised macro* (also known as a *function-like macro*):

```
#define identifier replacement-list
```

- e.g. `#define ADD(a,b) a+b`
- The parameters may appear as many times as desired in the replacement list
- N.B. There must be no space between the macro name and the left parenthesis

Example using Macros

Example using Macros



```c
#include <stdio.h>

#define MAX(x,y) ((x)>(y)?(x):(y))
#define IS_EVEN(n) ((n)%2==0)

int main(){
    printf("Max value is %d\n", MAX(6,5));
    printf("Is it Even: %s\n", IS_EVEN(6) ? "Yes" : "No");

    return 0;
}
```

```
Max value is 6
Is it Even: Yes
```

Example using Macros

- Invocations of these macros:

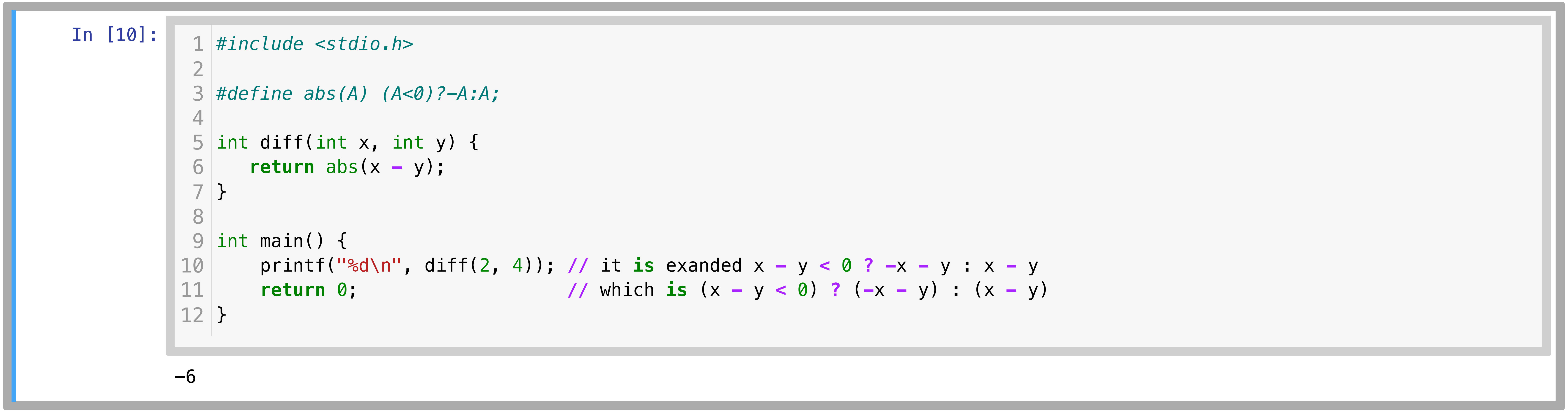
```
int i = MAX(5, 6);
```

- The same lines after macro replacement:

```
int i = ((5)>(6)?(5):(6));
```

Things can go wrong with macros!

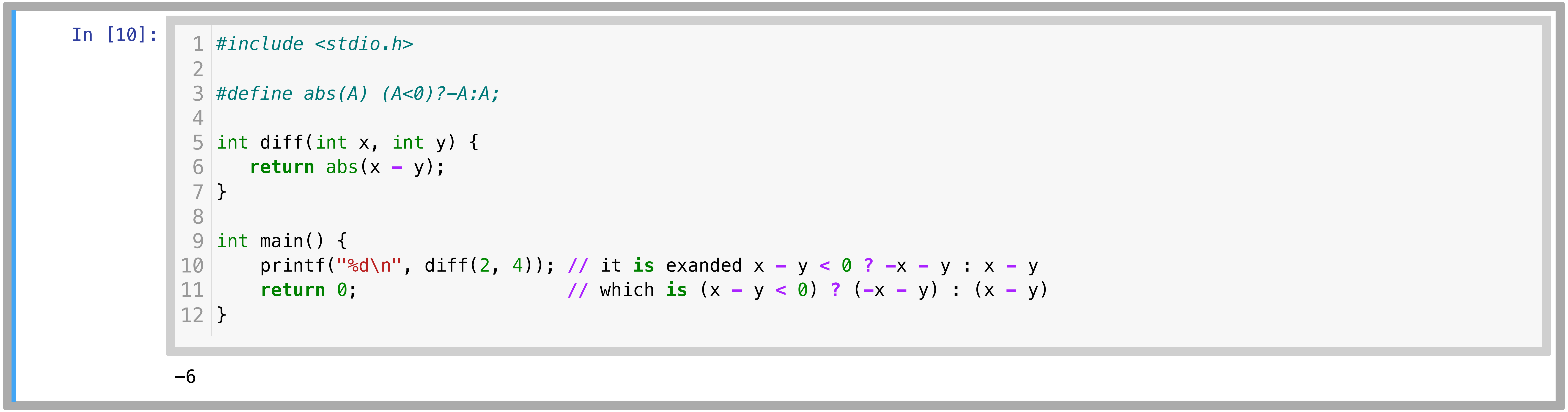
Things can go wrong with macros!



```c
#include <stdio.h>

#define abs(A) (A<0)?-A:A;

int diff(int x, int y) {
   return abs(x - y);
}

int main() {
    printf("%d\n", diff(2, 4)); // it is exanded x - y < 0 ? -x - y : x - y
    return 0;                   // which is (x - y < 0) ? (-x - y) : (x - y)
}
```

```
-6
```

Things can go wrong with macros!



```c
#include <stdio.h>

#define abs(A) (A<0)?-A:A;

int diff(int x, int y) {
   return abs(x - y);
}

int main() {
    printf("%d\n", diff(2, 4)); // it is exanded x - y < 0 ? -x - y : x - y
    return 0;                   // which is (x - y < 0) ? (-x - y) : (x - y)
}
```

```
-6
```

```c
#define abs(A) ((A)<0 ? -(A):(A))
```

would be the correct way of doing it!

Parameterised macro definitions

- Using a parameterised macro instead of an actual function has a couple of advantages:
    - The program may be slightly faster. A function call usually requires some overhead during program execution, but a macro invocation does not.
    - Macros are "generic". A macro can accept arguments of any type, provided that the resulting program is valid.

Parameterised macro definitions

- Potential disadvantages:
    - *Arguments aren't type-checked:* When a C function is called, the compiler checks each argument to see if it has the appropriate type. Macro arguments aren't checked by the pre-processor, nor are they converted.
    - They work as direct substitutions in your code. *Always use brackets to fullest extent possible*!
        - e.g. `#define DOUBLE(x) 2*x` might not do what you expect, as we saw in the previous examples!

Summary

- How to compile a C program
- How to write a very basic program
- pre-processor macros

Next lecture

- Small intro to UNIX