Lecture 04 - C Refresher

ENSF461 - Applied Operating Systems

Instructor: Lorenzo De Carli, University of Calgary (*lorenzo.decarli@ucalgary.ca*)
Slides by Lorenzo De Carli

Review of previous lecture

Command-line utilities

- Command-line utilities assist in **automating various tasks**:
  - **Text processing:** filtering and aggregating text data; computing statistics
  - **Operating on the file system:** changing working directory, creating and deleting files and folders
  - Show various machine statistics (e.g., free disk space)

More command line

The `grep` `utility`

- **`grep`** parses a file (or the standard input) and returns **all lines that match (or do not match) a string**
- A few options:
  - **`-i`**: makes matching case insensitive
  - **`-v`**: return all lines that do not match the pattern
  - **`-c`**: print number of matching lines per file
  - **`-l`**: print names of files with matches

Running commands on multiple files

Let’s talk about wildcards

- It is typically to be possible to run a utility on **more than one file** in one pass
- For example, use **`grep`** to find a string in **all files in a directory**
- This is accomplished using **wildcard patterns**:
  - **`$ grep hello *.c`**: find the string “hello” in all files ending in “**`.c`**” in the current directory
  - **`$ grep hello /home/lorenzo/prog*c`**: find the string “hello” in all files starting in “**`prog`**” and ending in “**`c`**” in the directory **`/home/lorenzo`**

Shell programming in Bash

```
#!/bin/sh

VAR=1
while [ $VAR –lt 10 ]
do
  MOD=`expr $VAR % 2`
  if [ $MOD –eq 0 ]
  then
    echo "$VAR is even"
  else
    echo "$VAR is odd"
  fi
  VAR=`expr $VAR + 1`
done
```

The plan for today

Let’s meet an old friend

- Why are we going to use **C** for this course?
- C **basic syntax** + **hello world**
- Simple **data structures**
- Compiling **C programs** from the **command line**
- **Debugging** C programs

A disclaimer

- Learning C is not the topic of this course
- We will provide you a quick refresher in this lecture
- We will also explain how to build and debug programs on Linux VMs
- The rest is up to you

Why C?

Easier to say why not!

- C is very **old**!
  - **Even older than your instructor** 😱
- C is **very unfriendly**
  - It's a lot more work than Python etc.
- C **is dangerous**
  - Easy to make **memory errors**…
  - …which make it **insecure**

> A new report examining the security of programming languages has found that almost 50% of all the vulnerabilities discovered in open source projects since 2009 were coded in C.
>
> The study by WhiteSource revealed that 46.9% of all reported open source vulnerabilities in the past 10 years were developed using C.

https://portswigger.net/daily-swig/c-is-least-secure-programming-language-study-claims

So… WHY C?!

There are some good reasons

- C is as close as it gets to the **operating system interface**
  - Except if we programming in **assembly**, which is a bigger **can of worms**
- C was develop in tandem with **UNIX** operating systems
  - Many UNIX (and Linux) interfaces are **straightforwardly accessible** from C
- C is the language used in the **book**
  - Do you really want to spend all of the course porting the examples from the book to **Rust**?

So yeah, we are using C

A note about C++

- **C++** shared many of the properties discussed before…
- …but is considerably **more complicated** than C
- Also, it hides some of the **low-level details** we care about
- **In class we are mostly going to stick to C**
  - May use some C++ constructs if makes things easier

What is C?

- **Programming language** first proposed by Dennis Ritchie in the early ‘70s
- Why “C”? It was derived from an earlier language called “B”
- Characteristics:
  - **General-purpose**
  - **Compiled (what does it mean?)**
  - **Strongly-typed (what does it mean?)**

C's basic syntax

There are four program sections you need to worry about

```
#include <stdio.h>

void do_stuff();

int main() {
  do_stuff();
}

void do_stuff() {
  printf("Hello, world!");
}
```

- Preprocessor directives
- Functions/globals declarations
- Main function (entry point)
- Function definitions

Let's look at each of them…

Preprocessor directives

What is a "preprocessor"?

- The **C preprocessor** expands **macros** (marked with "**#**") before sending the **source code** to the **compiler** (more on this later)
- Think of it as an advanced **search-and-replace**
- For example, the C preprocessor replaces **#include <stdio.h>** with the **entire content** of the header file **stdio.h** (which defines a bunch of utility functions, like **printf**)

More preprocessing

Preprocessor “constants”

- Useful to define **mnemonics** for **numerical constants**. Example:

  ```
  #define TRUE 1
  #define FALSE 0
  ```

- **Not really constants** (intended as program variables with a fixed value)

- The preprocessor simply **replaces** “TRUE” with “1” and “FALSE” with “0” **throughout the program**

Even more preprocessing

Preprocessor macros

- Using to define **mnemonics** for **short snippets of code**

```
def min(a, b) (a < b ? A : b)
```

- **Incidentally, what's happening in the code above?**
- Useful in the olden days as it is **faster than a function call**
- **Not very useful nowadays** (compilers are very good at **inlining** short functions)

Preprocessor: putting it all together

```c
// Preprocessor: example of include
#include <stdio.h>
#include <stdlib.h>

// Preprocessor: example of "constant"
#define TRUE 1
#define FALSE 0

// Preprocessor: example of macro
#define min(a,b) (a < b ? a : b)

int main(int argc, char** argv) {
    int val1 = atoi(argv[1]);
    int val2 = atoi(argv[2]);

    int val3 = min(val1, val2);
    printf("min(%d, %d) is %d\n", val1, val2, val3);

    return 0;
}
```

Back to the parts of a program

```c
#include <stdio.h>

void do_stuff();

int main() {
  do_stuff();
}

void do_stuff() {
  printf("Hello, world!");
}
```

~~Preprocessor directives~~

Functions/globals declarations

Main function (entry point)

Function definitions

Function declaration

Specifies type, name and parameters of a function

- Example #1:

```
void do_stuff();
```

The function takes no parameters

Name is “do_stuff”

Return type is “void” (does not return anything)

- Example #2:

```
int do_other_stuff(float a, float b);
```

What about this declaration?

Function declarations!

What are they good for?

- **A few things:**
  - C compilers do not allow you to **use a function before it has been defined**
  - You can make sure you always define functions before you call them
    - Can you? What about functions that **call each other**?
- Even if you are not in that case, you may want to use a function somewhere but **define it somewhere else** (e.g., another source code file)
  - Function declarations tell the compiler **how the function looks**, so you can use it without having defined it yet

Declaration: a multi-file example

(Do not worry, we'll get to compiling later)

example03_f1.c > main()

```c
#include <stdio.h>

unsigned pow2(unsigned val);

int main() {
    unsigned rez = pow2(3);
    printf("2^3 is %u\n", rez);
    return 0;
}
```

example03_f2.c > pow2(unsigned)

```c
unsigned pow2(unsigned val) {
    int i, rez = 2;
    if ( val ==0 )
        return 1;
    for (i = 1; i < val; i++)
        rez *= 2;
    return rez;
}
```

```
lorenzo@ensf461:~/class/ensf461F23/lecture04$ gcc -o example03 example03_f1.c example03_f2.c
lorenzo@ensf461:~/class/ensf461F23/lecture04$ ./example03
2^3 is 8
lorenzo@ensf461:~/class/ensf461F23/lecture04$ _
```

Let’s take a break and quiz!

Navigate to D2L->Quizzes->Quiz 4

Back to the parts of a program

```c
#include <stdio.h>

void do_stuff();

int main() {
  do_stuff();
}

void do_stuff() {
  printf("Hello, world!");
}
```

~~Preprocessor directives~~

~~Functions/globals declarations~~

Main function (entry point)

Function definitions

Function definition: the `main()` function

What is the difference between declaring and defining?

- A function **declaration** describes the **prototype** of a function
- A function **definition** describes the actual **implementation**
- Each C function must define a function called **`main`** that defines **where execution must start** (and typically end)
- By convention, the function has type **`int`** and its return value "says something" about whether the program **executed correctly** (return **0** if everything good 👍, something **!= 0** if something went bad/unexpected 👎)

main() function: examples

```
int main() {
    return 0;
}

----------

int main(int argc, char** argv) {
  int i;
  printf("Number of arguments: %d\n", argc);
  for (i=0; i<argc; i++)
    printf("Argument #%d: %s\n", i, argv[i]);
  return 0;
}
```

This syntax is used to receive **command line arguments** (let's see an example…)

Back to the parts of a program

```c
#include <stdio.h>

void do_stuff();

int main() {
  do_stuff();
}

void do_stuff() {
  printf("Hello, world!");
}
```

~~Preprocessor directives~~

~~Functions/globals declarations~~

~~Main function (entry point)~~

Function definitions

Function definitions

- Not much more to say
- A function definition is similar to a function declaration, but followed by the actual **function body**
  - Function body is defined within curly braces
  - The return value of the function must match what was defined in its prototype

Function definition example

C example03_f2.c > pow2(unsigned)

```c
unsigned pow2(unsigned val) {
    int i, rez = 2;
    if ( val ==0 )
        return 1;
    for (i = 1; i < val; i++)
        rez *= 2;
    return rez;
}
```

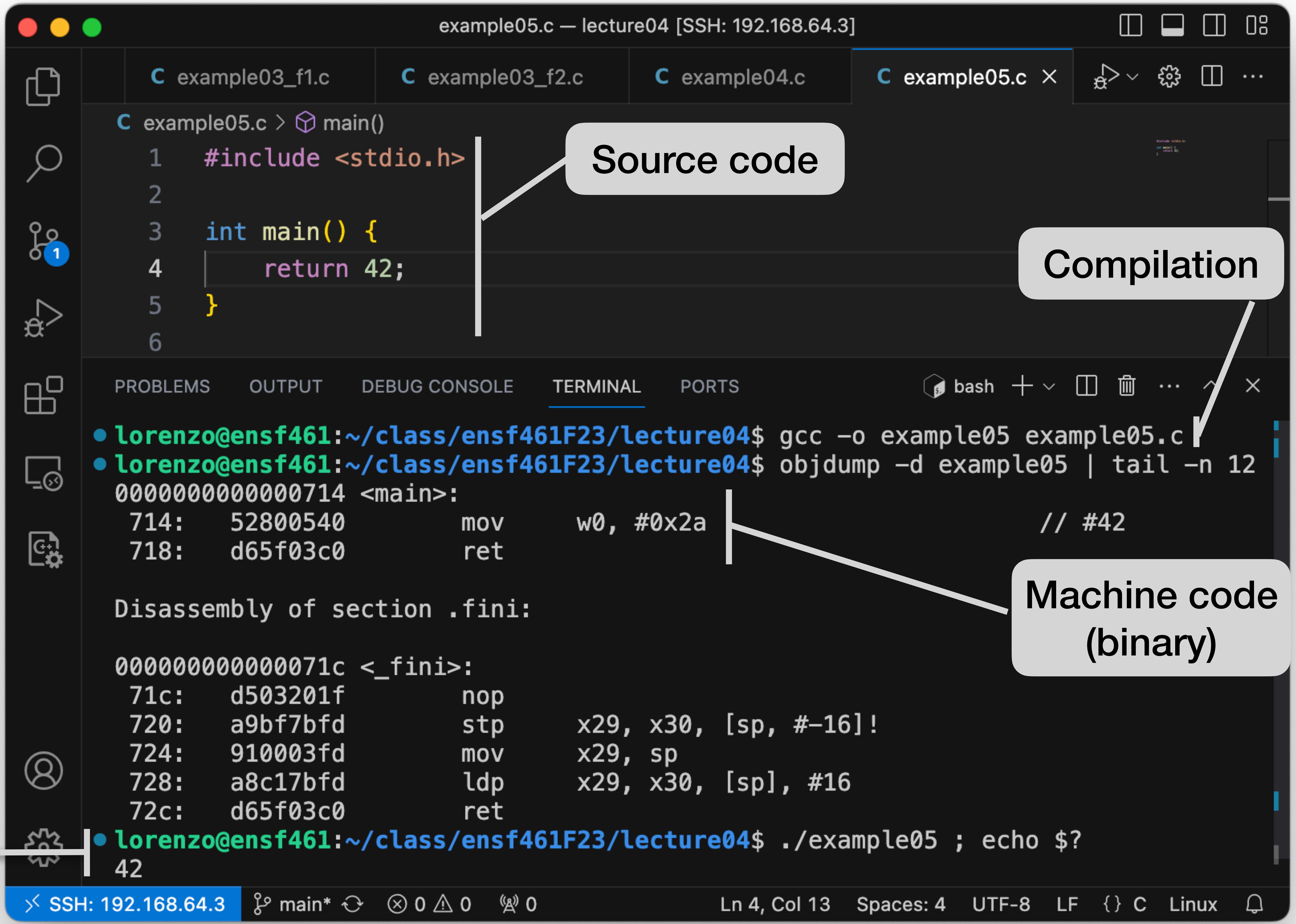
Let’s talk about compilers

C is a **compiled** language

- **What does it mean?**
- Languages like Bash/Python/Node.js are **interpreted**
    - In principle, a program called **interpreter reads your code** and **executes** it
    - Under the hood there is a lot of funky stuff going on to make things go faster, but the above abstraction holds in practice
- **C does nothing of that stuff**

What is a compiled language?

- Source code is not executed by an interpreter
- Instead, is passed to a **compiler** which turns it into **executable machine code**
- The output of the compiler is an executable file which can be executed directly



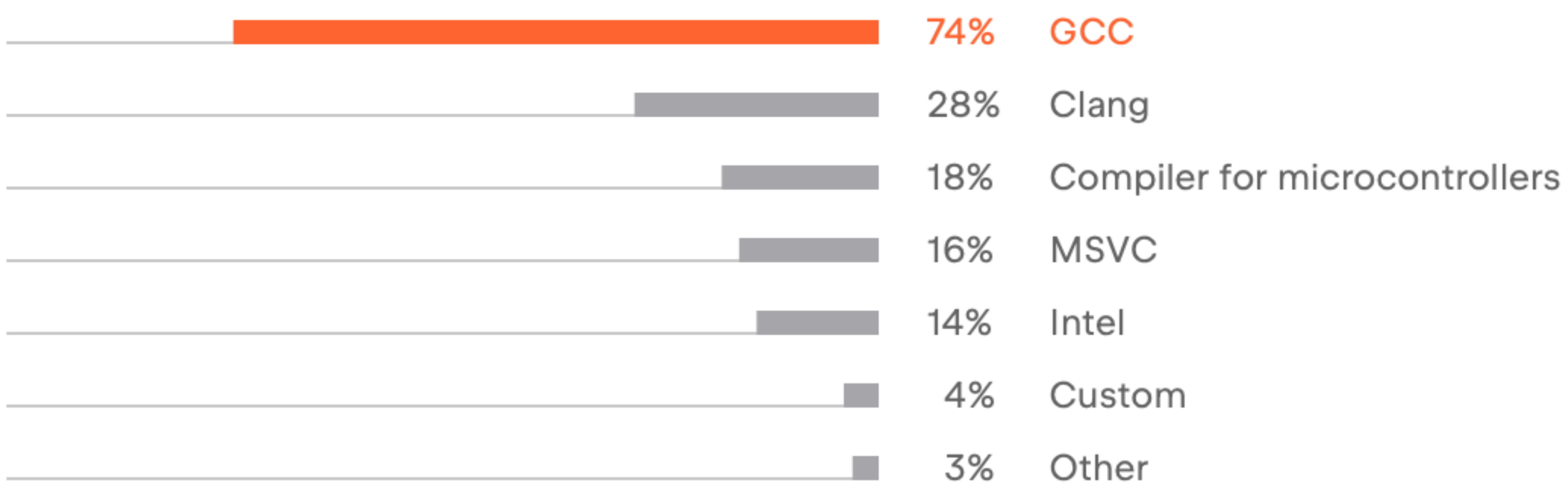
Execution & return value

How do I do this?

Well, you need a **compiler**

- In this class we are going to use **gcc**
  - The first (and for many years the only) **OSS industry-grade compiler**
  - Ships with pretty much **every Linux distribution**
  - Still the **most popular** C compiler
- You should learn to **run gcc** and **compile programs** from the **command line**

Which compilers do you regularly use?



GCC is the most popular compiler, but compilers for microcontrollers in Embedded development are also particularly popular with a share of 38%, which puts them above Clang and MSVC.

https://www.jetbrains.com/lp/devecosystem-2020/c/

Running gcc: the basics

In a nutshell, this is how you do it

```
gcc  -o  <executable file name>  <source file(s)>
```

Compiler command

Set the name of the generated executable

Source file(s) to be compiled

Source file(s)?

A C program can be split across multiple files

- You can compile such a program with `gcc –o prog file1.c file2.c`
- However, if you define a function in a source file and use it in another one, you need to make the **prototype** of that function known
- You can use **header files** to do this

Header files

example06_f1.c > ...

```c
#include <stdio.h>
#include "example06_f2.h"

int main(int argc, char** argv) {

    int val = multiply_by_42(argc);
    printf("Value: %d\n", val);

    return 0;
}
```

example06_f2.h > ...

```c
int multiply_by_42(int val);

```

example06_f2.c > multiply_by_42(int)

```c
int multiply_by_42(int val) {
    return val*42;
}

```

Now for the fun part of C

Pointers & memory management

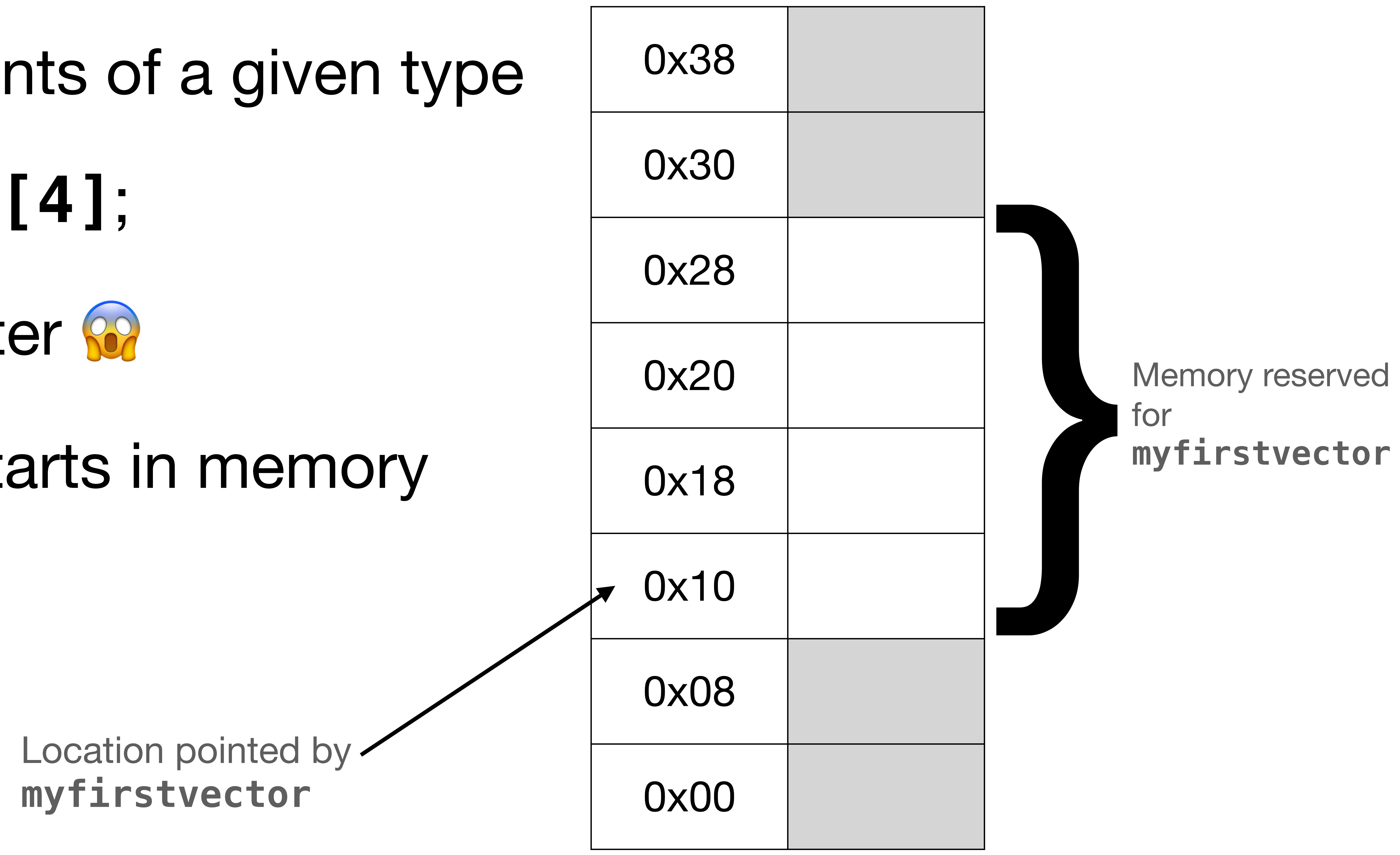
C memory management & you

- C, unlike many other languages, allows to **directly access memory locations** where **variables** are stored
- A few caveats:
  - Those are not **physical** memory locations
  - Those are not **absolute** memory locations
  - Those locations may **not remain the same** across executions

Vectors & vector indexing

Let’s start with an easy one

- In C a vector is an array of elements of a given type
- Example: **`int myfirstvector[4]`**;
- **`myfirstvector`** is really a pointer 😱
    - The location where the array starts in memory



Vectors/2

- When code accesses an **element** of a vector:

```
myfirstvector[2] = 3;
```

- The compiler computes the following **memory location**:

```
myfirstvector + (3 * sizeof(int))
```

What is this?

- …and stores the value “3” in it!

Pointers

- A variable storing a **memory location** is a **pointer**

```
int myvar;     // Variable (stores an integer value)

int *myvar:    // Pointer (stores the value of an
               // integer-sized memory location)
```

How do I get a memory location anyway?

The “&” operator

- A pointer can store a memory location, but how do you make it point to something useful?

- One way to get the address of a variable is to use the “&” operator:

```
int val = 3;
int* myptr = &val;
```

- To write at the location pointed at by myptr, use the “*” operator:

```
*myptr = 42;
printf(“%d\n”, val);
```

[ ] and * are related

- `int vec[3]` and `int* ptr` are both pointers, but…
- The first guarantees that there is a three-integer memory region allocated starting at the memory location pointed by var
- ptr is just a pointer! You can store a location in it, but there is no functional guarantee that it is going to point at a useful or reasonable place in memory
- **What happens if you try this?**

```
int* ptr = 0;
*ptr = 42;
```

Let’s look at an example

```c
#include <stdio.h>

void value_or_reference(int val, int* ref) {
    val = 42;
    *ref = 42;
}

int main() {
    char myvec[4] = {'a', 'b', 'c', 'd'};

    printf("Position 0 of myvec is %c\n", myvec[0]);
    printf("Position 1 of myvec is %c\n", *(myvec+sizeof(char)));

    int byvalue = 2;
    int byref = 2;
    value_or_reference(byvalue, &byref);

    printf("Variable passed by value: %d\n", byvalue);
    printf("Variable passed by references: %d\n", byref);

    return 0;
}
```

Finally, saving you some time

DON’T DO THIS!

```
gcc –o myprogram.c myprogram.c
```

- **What’s wrong with the line above?**
- It passes the **input file** (myprogram.c) as **output file**!

That’s all!