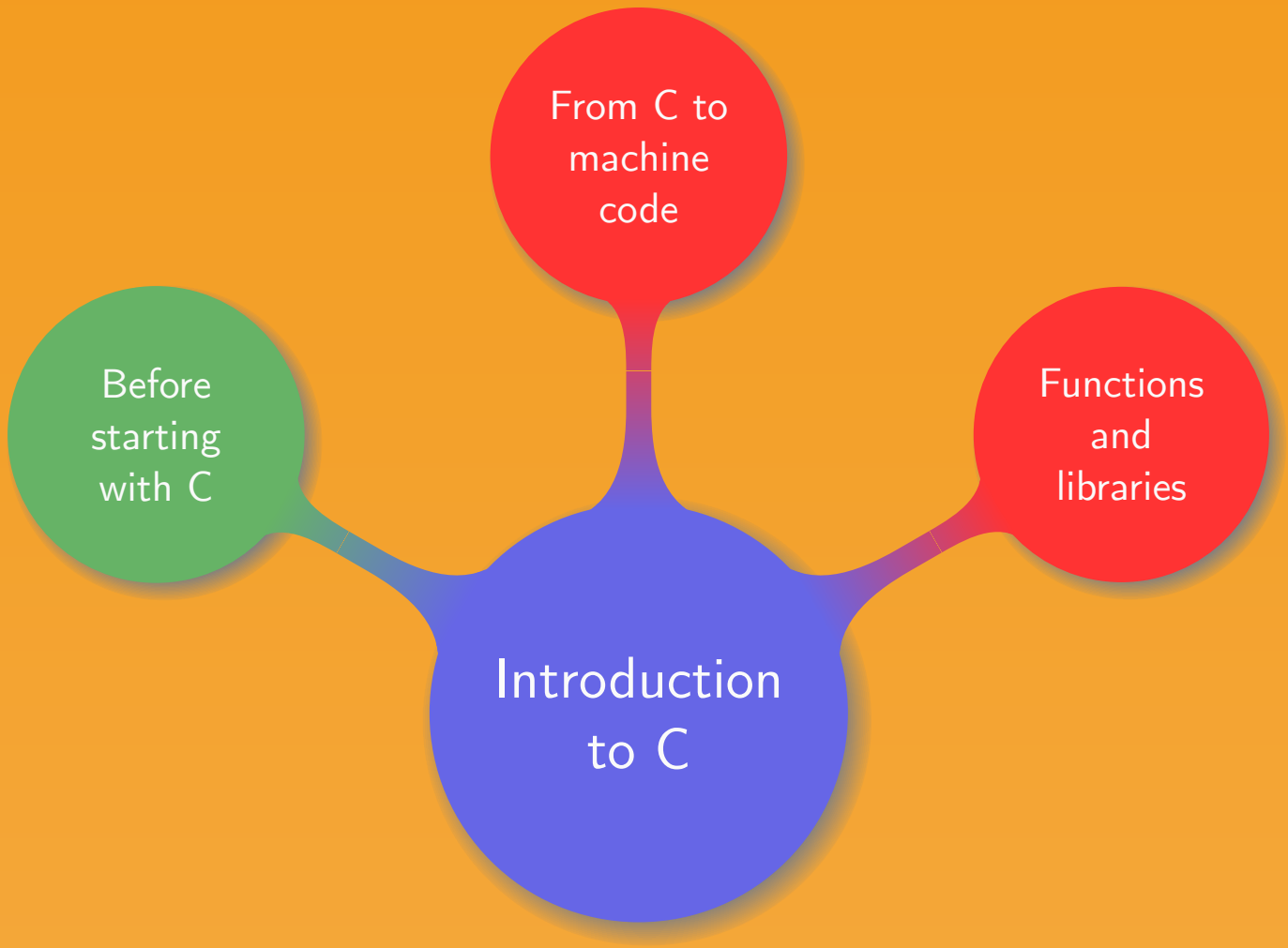
Introduction to Computer and Programming

5. Introduction to C

Manuel – Summer 2019

Chapter organisation



The birth of C

In the old time:

- Unix OS was implemented in assembly
- New hardware implied new possibilities
- New possibilities implied new code
- Much time wasted rewriting the OS for the new hardware

The birth of C

In the old time:

- Unix OS was implemented in assembly
- New hardware implied new possibilities
- New possibilities implied new code
- Much time wasted rewriting the OS for the new hardware

Development of a new language:

- Authors: Ken Thompson & Dennis Ritchie
- Location: AT&T Bell Labs
- Time frame: 1969 – 1973
- Name: C, as derived from B

Why using C?

Main characteristics:

- One of the most widely used languages
- Available for the majority of computer architectures and OS
- Many languages derived from C

Why using C?

Main characteristics:

- One of the most widely used languages
- Available for the majority of computer architectures and OS
- Many languages derived from C

Advantages of C:

- Performance
- Interface directly with hardware
- Higher level than assembly
- Low level enough
- Zero overhead principle

Development environment

Common software to write C code:

- Text editor + compiler
- Code::Blocks, Geany, Xcode, Clion, Visual studio code
- Microsoft visual C++

Development environment

Common software to write C code:

- Text editor + compiler
- Code::Blocks, Geany, Xcode, Clion, Visual studio code
- Microsoft visual C++

Common C compilers:

- GNU C Compiler: `gcc`
- Clang
- Intel C Compiler: `icc`

Development environment

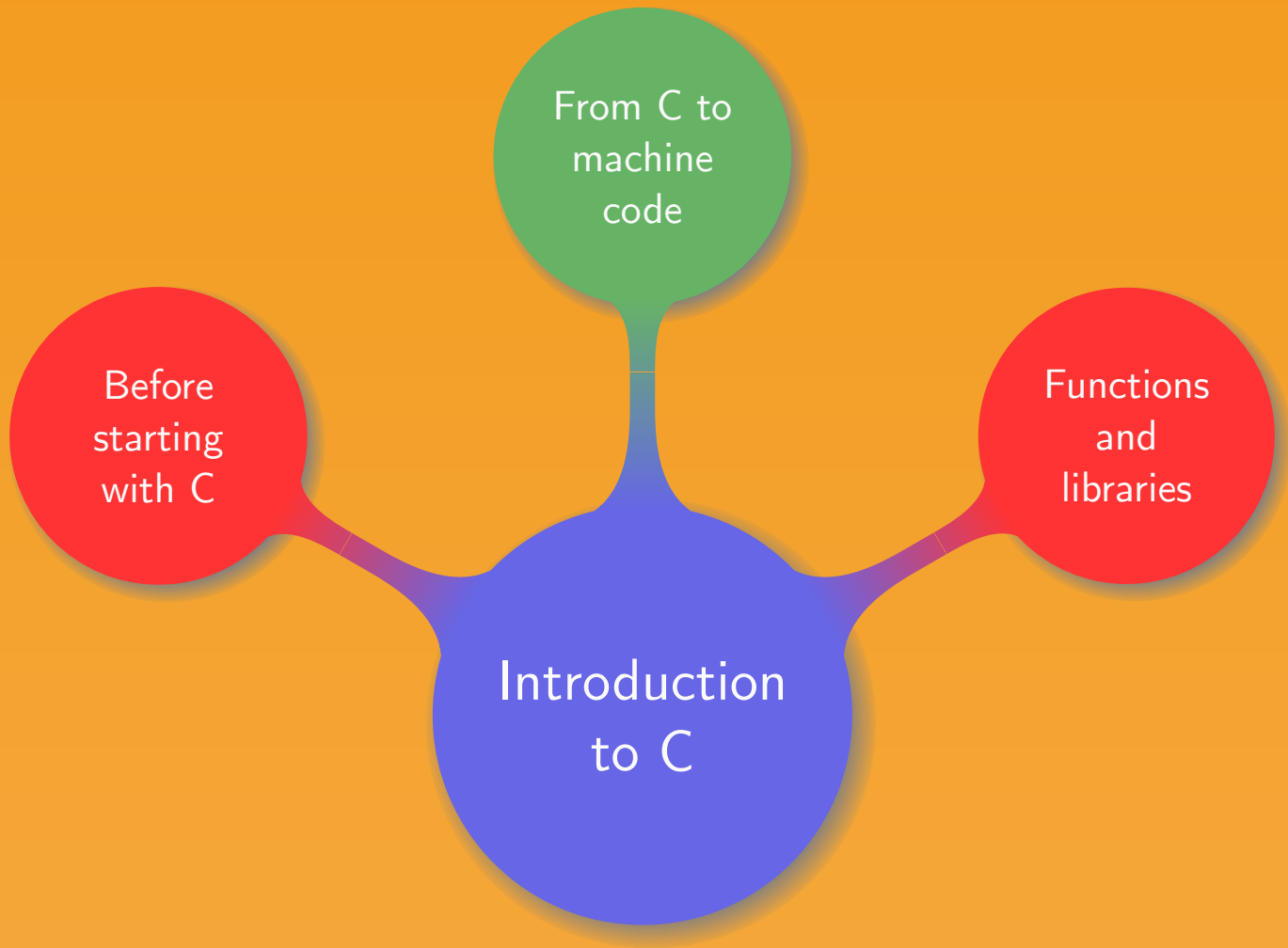
Common software to write C code:

- Text editor + compiler
- Code::Blocks, Geany, Xcode, Clion, Visual studio code
- Microsoft visual C++ $\leftarrow$ **BAD!**

Common C compilers:

- GNU C Compiler: `gcc`
- Clang
- Intel C Compiler: `icc`

Chapter organisation



A first example

gm_base.c

```c
#include <stdio.h>
int main () {
  printf("good morning!\n");
  return 0;
}
```

Program structure:

- A unique `main` function: used only to "dispatch" the work
- Other functions: effectively doing the work

A first example

gm_base.c

```c
#include <stdio.h>
int main () {
  printf("good morning!\n");
  return 0;
}
```

Program structure:

- A unique `main` function: used only to "dispatch" the work
- Other functions: effectively doing the work

Writing a C function

```
OType FName(IType IName,...) {
  function's body
}
```

Compiling a C program

```
sh $ gcc gm_base.c -o gm_base
```

Blocks

Explain the following code:

blocks.c

```c
#include <stdlib.h>
#include <stdio.h>
int main () {
  {
    int a=0; printf("%d ",a);
  }
  {
    double a=1.124; printf("%f ",a);
  }
  {
    char a='a'; printf("%c ",a);
  }
//  printf("%d",a);
}
```

Shorthand operators

Questions.

- How is the code indented?
- Why is line 13 commented out?
- What happens if lines 9 and 10 are deleted?

Shorthand operators

Questions.

- How is the code indented?
- Why is line 13 commented out?
- What happens if lines 9 and 10 are deleted?

Common shortcuts:

- Increment: e.g. `a++`
- Decrement: e.g. `a--`
- Add: e.g. `x+=y`
- Subtract: e.g. `x-=y`
- Multiply: e.g. `x*=y`
- Divide: e.g. `x/=y`

The `#include` instruction

Roles of a header file:

- Define function prototypes
- Define constants, data types...
- A function used in a program must have been defined earlier

Syntax to include `header.h`:

- Known system-wide: `#include<header.h>`
- Unknown to the system: `#include "/path/to/hearder.h"`

The `#include` instruction

Roles of a header file:

- Define function prototypes
- Define constants, data types...
- A function used in a program must have been defined earlier

Syntax to include `header.h`:

- Known system-wide: `#include<header.h>`
- Unknown to the system: `#include "/path/to/hearder.h"`

Result of `#include<stdio.h>`:

```
sh $ gcc -E gm_base.c
```

The `#define` instruction

Goal:

- Set “type-less” read-only variables
- Hard-code values in the program
- Quickly alter hard-coded values over the whole file

The #define instruction

Goal:

- Set “type-less” read-only variables
- Hard-code values in the program
- Quickly alter hard-coded values over the whole file

gm_def.c

```c
#include <stdio.h>
#define COURSE "VG101"
int main () {
  printf("good morning %s!\n",COURSE);
}
```

Result of #define:

```
sh $ gcc -E gm_def.c
```

Taking advantage of `#define`

The `#ifdef` and `#ifndef` instructions:

- Test if some "`#define` variable" is (un)set
- Compile different versions of a same program

gm_ifdef.c

```
1 #include <stdio.h>
2 #define POLITE
3 int main () {
4 #ifdef POLITE
5   printf("good morning!\n");
6 #endif
7 }
```

gm_ifndef.c

```
1 #include <stdio.h>
2 int main () {
3 #ifndef RUDE
4   printf("good morning!\n");
5 #endif
6 }
```

Result of `#if(n)def`:

```
$ gcc -E gm_ifdef.c
$ gcc -E gm_ifndef.c
```

More on #define

Writing simple macros:

- Define type-less functions
- Perform fast and simple actions
- To be used only on specific circumstances (e.g. min/max)
- Do not use for regular functions

gm_macro.c

```
1 #include <stdio.h>
2 #define SPEAK(x) printf("good morning %s!\n",x)
3 int main () {
4   SPEAK("VG101");
5   SPEAK("VE475");
6 }
```

Result of macros:

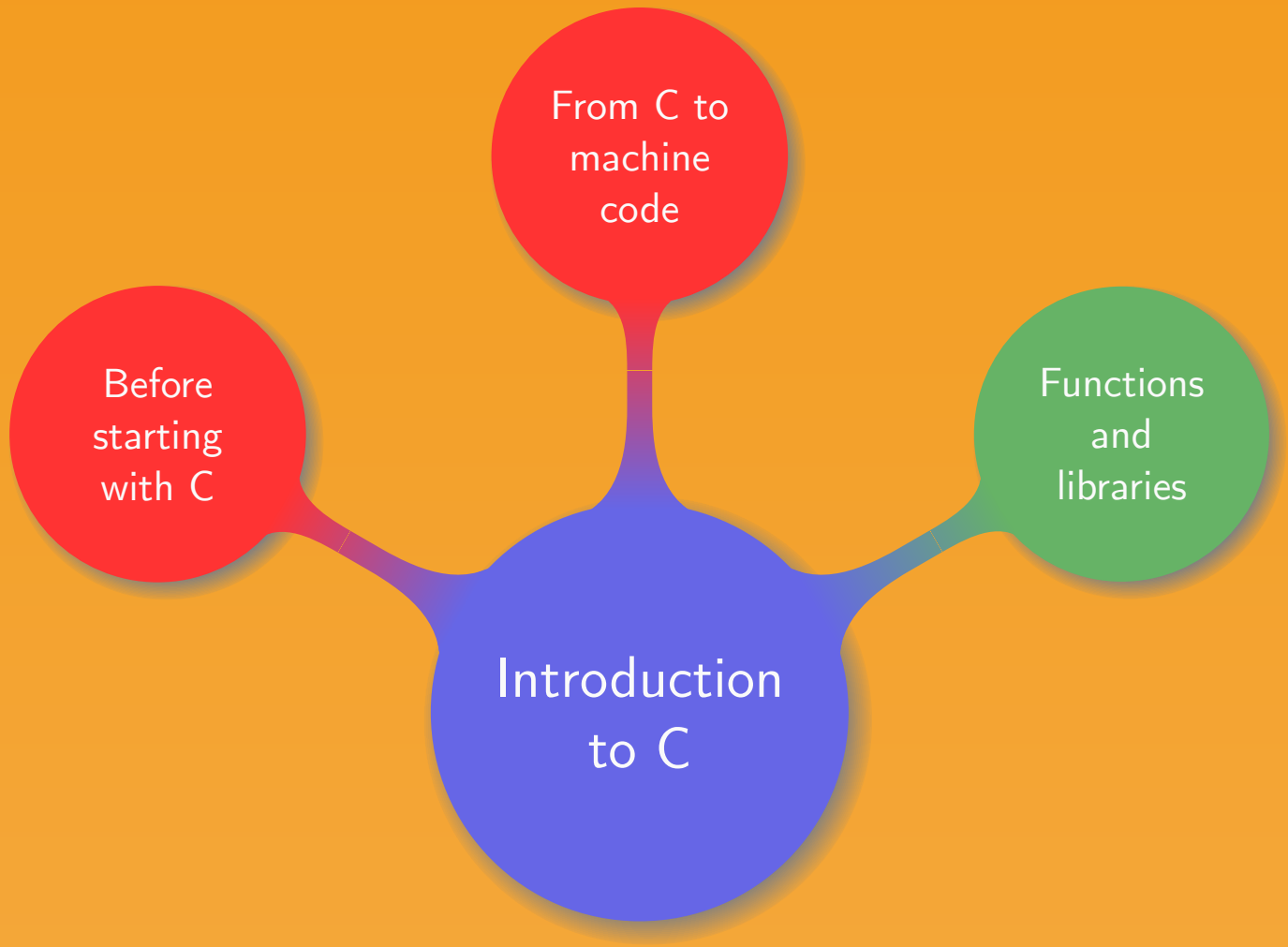
```
$ gcc -E gm_macro.c
```

Common compilation errors

Often the compilation process fails because of:

- Syntax errors
- Incompatible function declarations
- Wrong Input and Output types
- Operations unavailable for a specific data types
- Missing function declarations
- Missing machine codes for some functions

Chapter organisation



More complex programs

The main function:

- Never write a whole program in the main function
- Use the main function to dispatch the work to other functions
- Most of the coding must be done outside of the main function

Reminders:

- Always add comments to the code
  - A single line: start with `//`
  - Multiple lines: anything between `/*` and `*/`
- As much as possible use a function per task or group of tasks
- If the program becomes large split it over several files

A long program

ans_orig.c

```c
#include <stdio.h>
double answer(double d);
int main () {
  double a;
  scanf("%lf",&a);
  printf("%lf\n", answer(a));
}
double answer(double d) {return d+1337;}
```

A long program

ans_orig.c

```c
#include <stdio.h>
double answer(double d);
int main () {
  double a;
  scanf("%lf",&a);
  printf("%lf\n", answer(a));
}
double answer(double d) {return d+1337;}
```

Functions and operators used:

- Display the integer contained in *a*: `printf("%d",a)`
- Read and store an integer in *a*: `scanf("%d",&a)`
- Both functions can take a variable number of parameters
- Arithmetic operators: `+`, `-`, `/`, `%`

Organising a long program

Splitting the code over several files:

ans_main.c

```c
#include <stdio.h>
#include "ans.h"
int main () {
  double a; scanf("%lf",&a); printf("%lf\n", answer(a));
}
```

ans.c

```c
#include "ans.h"
double answer(double d) {
  return d+1337;
}
```

ans.h

```c
#ifndef ANS_H
#define ANS_H
double answer(double d);
#endif
```

Compilation:

```
$ gcc ans_main.c ans.c -o ans
```

Libraries

A *library* is a collection of functions, macros, data types, and constants

Example.
The C mathematics library:

- Mathematical functions, e.g. log, exp, trigonometric, floor, etc.
- Add the header: `#include <math.h>`
- Add the corresponding compiler flag: `-lm`

Libraries

A *library* is a collection of functions, macros, data types, and constants

Example.
The C mathematics library:

- Mathematical functions, e.g. log, exp, trigonometric, floor, etc.
- Add the header: `#include <math.h>`
- Add the corresponding compiler flag: `-lm`

math.c

```c
#include<stdio.h>
#include<math.h>
int main() {
  printf("%g\n", \
    gamma(sqrt(cosh(M_PI/2))));
}
```

```
$ gcc -lm math.c
```

Key points

- Is C a compiled or interpreted language?
- How to transform a C program into machine code?
- Why are data types of a major importance?
- What are header guards, and why using them?

Thank you!

References I

5.3 https://upload.wikimedia.org/wikipedia/commons/1/1b/Ken_Thompson_and_Dennis_Ritchie--1973.jpg