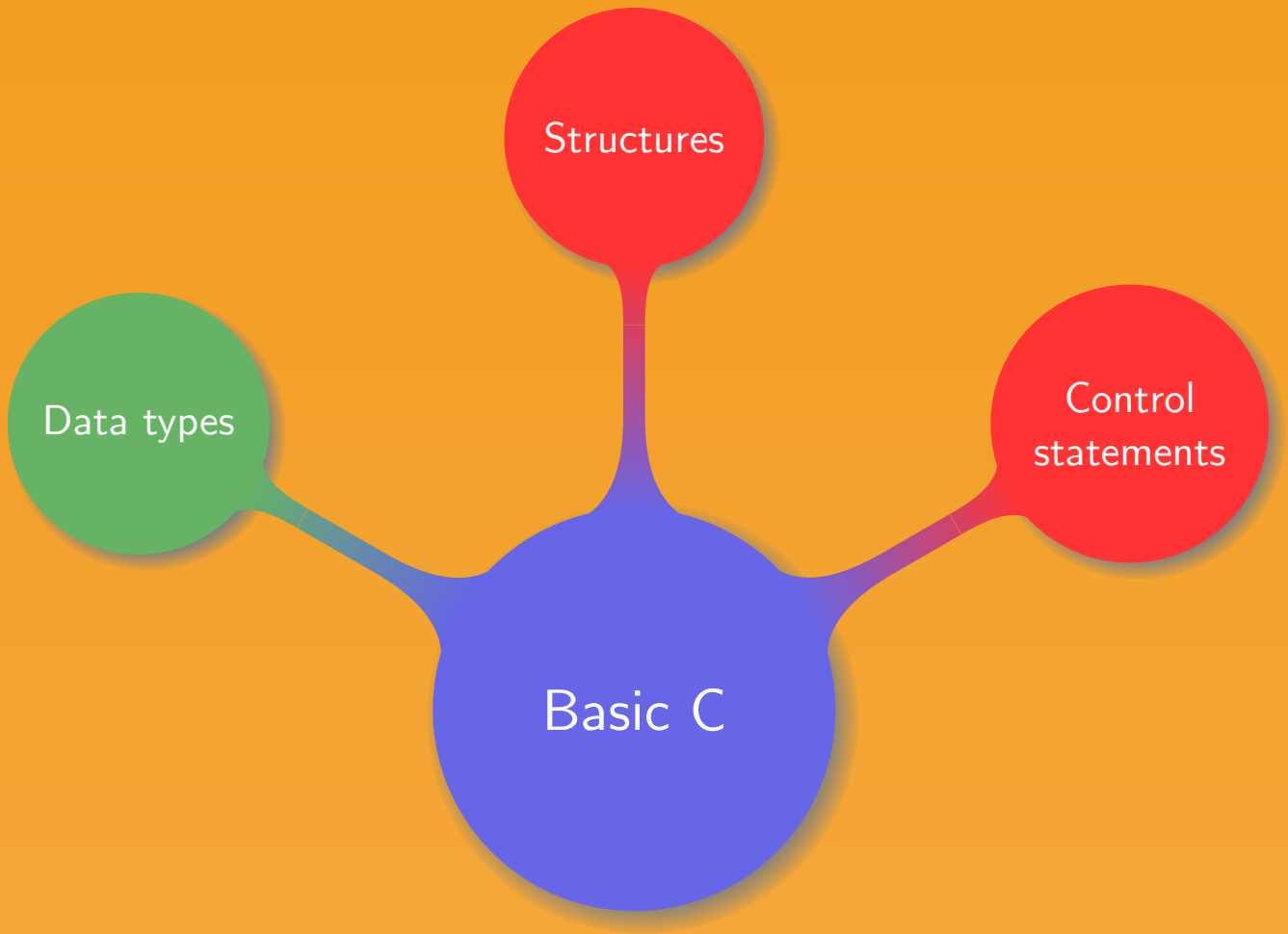
# Introduction to Computer and Programming

## 6. Basic C

Manuel – Summer 2019

# Chapter organisation



# Types of variables

Three main categories of variables:

- Constant variables: `#define PI 3.14159`
- Global variables: defined for all functions
- Local variables: defined only in the function

# Types of variables

Three main categories of variables:

- Constant variables: `#define PI 3.14159`
- Global variables: defined for all functions
- Local variables: defined only in the function

Never ever use global variables in VG101

# Initialising variables

Common use:

- Variables for `#define` are UPPERCASE
- Other variables are lowercase, or capitalised
- Variable names cannot exceed 31 characters
- Variable names can start with _ or a character
- Variables starting with _ are “hidden”

# Basic data types

Data types in C:

- Integer: `int`
- Character: `char`
- Valueless type: `void`
- Fractional numbers:
  - Single precision: `float`
  - Double precision: `double`

*The C standard only fixes the size of* `char` *(1 byte)*

# Basic data types

Data types in C:

- Integer: `int`
- Character: `char`
- Valueless type: `void`
- Fractional numbers:
  - Single precision: `float`
  - Double precision: `double`

*The C standard only fixes the size of* `char` *(1 byte)*

Different variations available:

- `char`: `signed char`, `unsigned char`
- `int`: `short int`, `signed short int`, `unsigned short int`, `signed int`, `unsigned int`, `long int`, `signed long int`, `unsigned long int`, `long long int`, `signed long long int`, `unsigned long long int`
- `double`: `long double`

Extra variations: `static`, `register`, `extern`, `volatile`

# Data types

Basic number types:

- `int`: size limitation, from 0 to $2^{32} - 1$
- `float`: 7 digits of precision, from $1 \cdot 10^{-38}$ to $3 \cdot 10^{38}$
- `double`: 13 digits of precision, from $2 \cdot 10^{-308}$ to $1 \cdot 10^{308}$

Example.

```
1 float a=1.0; int b=3; double c;
```

# Data types

Basic number types:

- `int`: size limitation, from 0 to $2^{32} - 1$
- `float`: 7 digits of precision, from $1 \cdot 10^{-38}$ to $3 \cdot 10^{38}$
- `double`: 13 digits of precision, from $2 \cdot 10^{-308}$ to $1 \cdot 10^{308}$

Example.

```
1 float a=1.0; int b=3; double c;
```

Characters:

- Strings are viewed as arrays of characters
- Characters are enclosed in single quotes, e.g. `char a='a';`
- Strings are enclosed in double quotes
- Character are encoded using the American Standard Codes for Information Interchange (ASCII)

# Data type display

What output to expect?

types1.c

```c
#include <stdio.h>
int main() {
  printf("%d %f\n",7/3,7/3);
}
```

# Data type display

What output to expect?

types1.c

```c
#include <stdio.h>
int main() {
  printf("%d %f\n",7/3,7/3);
}
```

types2.c

```c
#include <stdio.h>
int main() {
  printf("%d %f\n",7/3,7.0/3);
}
```

In the previous codes:

- What do `%f`, `%d` and `%c` mean?
- What is the type of `7/3` for the compiler?
- What is displayed for `b`?
- What is this character corresponding to?
- Why is this character displayed?

# Type casting

Changing data type:

- Float to int: `float a = 4.8; int b = (int) a;`
- Int to char: `int a = 42; char b = (char) a;`
- Try double to char, int to float

# Type casting

Changing data type:

- Float to int: `float a = 4.8; int b = (int) a;`
- Int to char: `int a = 42; char b = (char) a;`
- Try double to char, int to float

Always think of the size...

# Type casting

## Example.

types3.c

```c
#include <stdio.h>
int main() {
  float c=4.8; printf("%d\n", (int)c);
  int f=42; printf("%c\n", (char)f);
  double a=487511234.7103;
  char b=(char) a;
  printf("%c, %c\n",b,a);
  int d=311;
  float e=(float) d;
  printf("%d %f\n",d,e);
  printf("%c\n",d);
}
```

# Type casting

Understanding the code:

- Which type castings work well?
- What is the length of a `char`?
- What is the length of an `int`?
- What is printed for `d`?
- What is the issue when displaying `d` as a `char`?

# Integer rounding

Exercise.
Write C program featuring a function `apbp1(float a, float b)` which returns the nearest integer to `a+b+1`

# Integer rounding

Exercise.
Write C program featuring a function `apbp1(float a, float b)` which returns the nearest integer to `a+b+1`

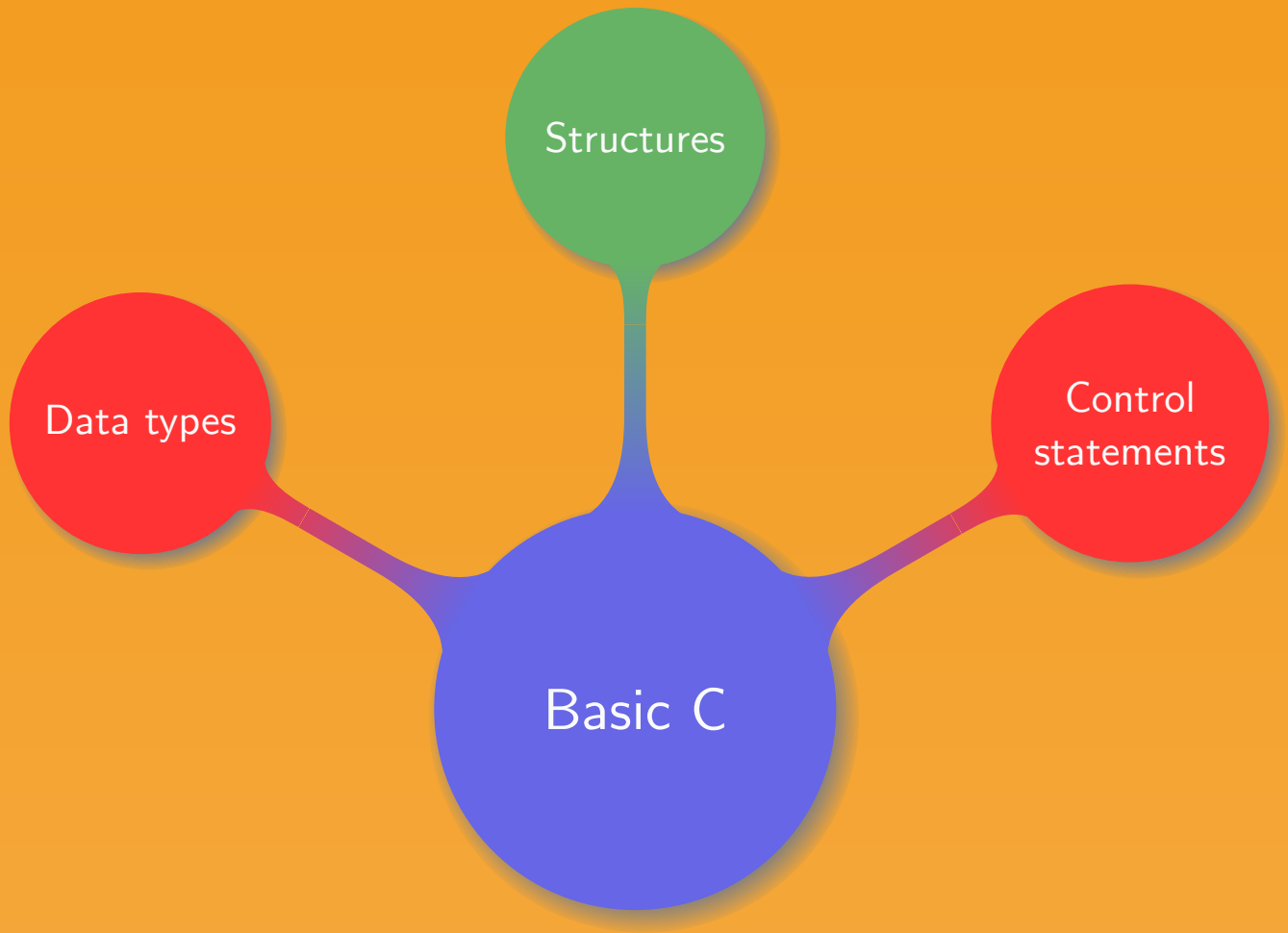
apbp1.c

```c
#include <stdio.h>
int apbp1 (float a, float b);
int main () {
  float a, b;
  scanf("%f %f", &a,&b);
  printf("%d\n", apbp1(a,b));
}
int apbp1 (float a, float b) {
  a++; a+=b;
  return((int) (a+0.5));
}
```

Understanding the code:

- Discuss the use of shorthand operators and type casting
- Why is not all the code in the main function
- How is indentation done?
- Does the code contain any global variable?

# Chapter organisation



# Remarks related to data types

More data types in C:

- Reminder: a bit belongs to $\{0, 1\}$ and a byte is 8 bits
- Operating data at low level, e.g. shift <<, >>
- A `char` does not necessarily contains a character
- Logical operations are of a major importance
- Understanding data representation is important to be efficient
- Structures, enumerate, union

# Structures

struct.c

```c
#include <stdio.h>
typedef struct _person {
  char* name;
  int age;
} person;
int main () {
  person al={"albert",32};
  person gil;
  gil.name="gilbert";
  gil.age=23;
  struct _person so={"sophie",56};
  printf("%s %d\n",al.name, al.age);
  printf("%s %d\n",gil.name, gil.age);
  printf("%s %d\n",so.name, so.age);
}
```

# Structures

Understanding the code:

- How is a structure defined?
- How to define a new type?
- What are two ways to set the value of a field in a structure?
- How to access the values of the different fields in a structure?

# Functions and structures

struct_fct.c

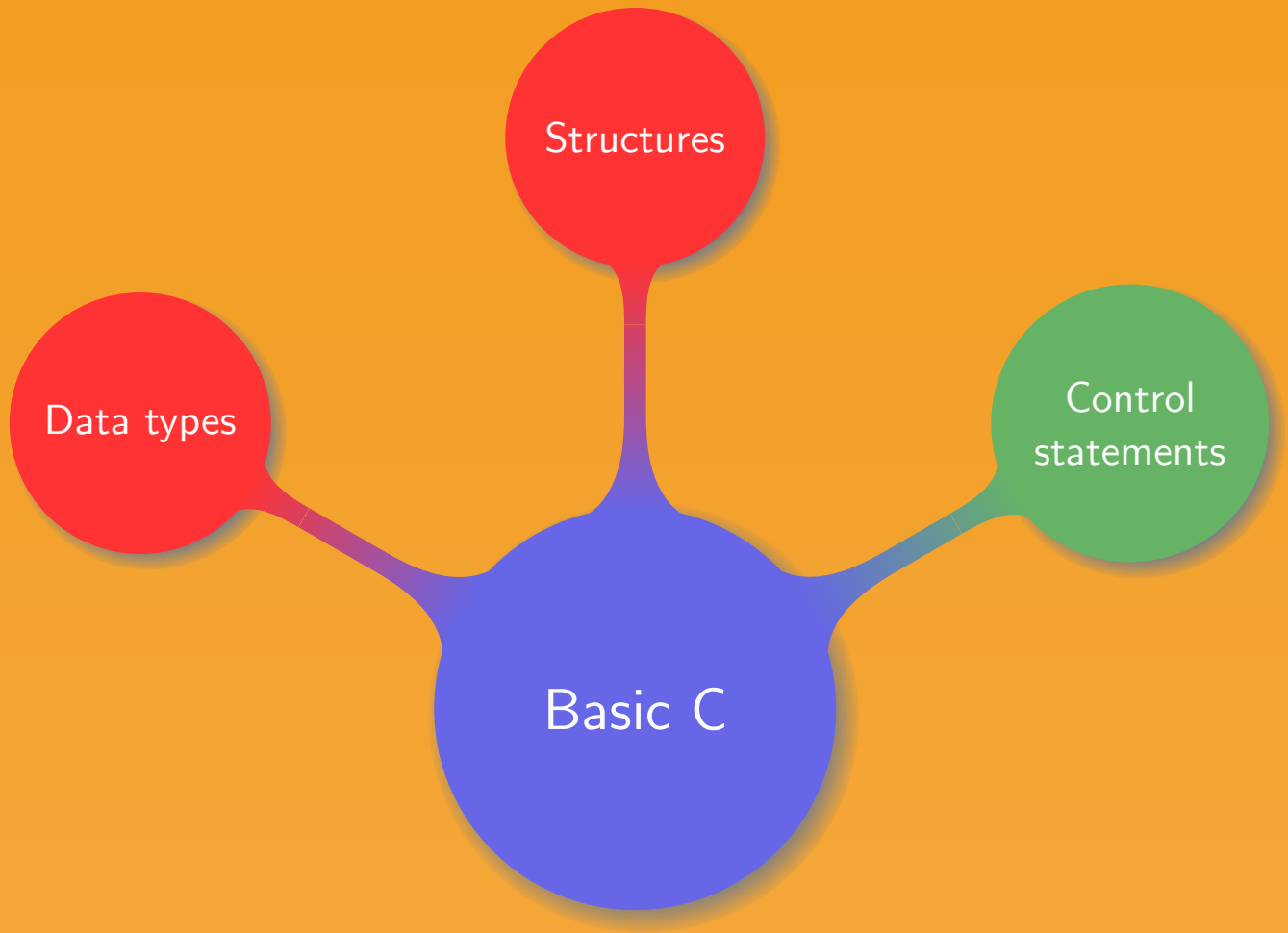
```c
#include <stdio.h>
typedef struct person {
  char* name; int age;
} person_t;
person_t older(person_t p, int a);
int main () {
  person_t al={"albert",32};
  al=older(al,10);
  printf("%s %d\n",al.name,al.age);
}
person_t older(person_t p, int a) {
  printf("%s %d\n",p.name, p.age);
  p.age+=a;
  return p;
}
```

# Functions and structures

Understanding the code:

- How is the age increased?
- How are the person's information sent to a function?
- How to return the person's information after the function?
- How many output can a C function have?

# Chapter organisation



# Jumping!

jump.c

```c
#include <stdio.h>
int main() {
  int i=0;
  printf("I am at position %d\n",i);
  i++;
  goto end;
  printf("I am at position %d\n",i);
  end:
    i++;
    printf("It all ends here, at position %d\n",i);
  return 0;
  i++;
  printf("Unless it's here at position %d\n",i);
}
```

# Jumping!

Understanding the code:

- What positions are displayed?
- Why are some positions skipped?
- How to use the `goto` statement?
- Why should the `goto` statement (almost) never be used?

# Important operators

Basics on conditional statements:

- No boolean type, 0 means `False`, anything else `True`
- Boolean evaluation: `<`, `<=`, `>`, `>=` , `==`, `!=`
- Not: `!`, short-circuit operators: `&&`, or: `||`
- Bit operations: `&`, `|`, `^`

# Important operators

Basics on conditional statements:

- No boolean type, 0 means `False`, anything else `True`
- Boolean evaluation: `<`, `<=`, `>`, `>=` , `==`, `!=`
- Not: `!`, short-circuit operators: `&&`, or: `||`
- Bit operations: `&`, `|`, `^`

Conditional ternary operator: `?` `:`

```
1 condition ? expression1 : expression2
```

# Important operators

Basics on conditional statements:

- No boolean type, 0 means `False`, anything else `True`
- Boolean evaluation: `<`, `<=`, `>`, `>=` , `==`, `!=`
- Not: `!`, short-circuit operators: `&&`, or: `||`
- Bit operations: `&`, `|`, `^`

Conditional ternary operator: `?` `:`

```
condition ? expression1 : expression2
```

Example.
A macro returning the max of two numbers:

```
#define MAX(a,b) a>=b ? a : b
```

# The `if` and `switch` statements

```
if (condition) {
  statements;
}
else {
  statements;
}
```

```
switch(variable) {
  case value1:
    statements;
    break;
  case value2:
    statements;
    break;
  default:
    statements;
    break;
}
```

# Conditional statements

## Example.

cards.c

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define ACE 14
#define KING 13
#define QUEEN 12
#define JACK 11
int main () {
  int c; srand(time(NULL)); c=rand()%13+2;
  switch (c) {
    case ACE: printf("Ace\n"); break;
    case KING: printf("King\n"); break;
    case QUEEN: printf("Queen\n"); break;
    case JACK: printf("Jack\n"); break;
    default: printf("%d\n",c);  break;
  }
}
```

# Conditional statements

Understanding the code:

- Write this code using the `if` statement
- Adapt the code such as to display the complete card name, e.g. “Ace of spades”
- What happens if a `break` is removed?
- Explain why and compare to the behavior in MATLAB

# The `while` and `do... while` statements

Structure of the two types of `while` loops:

```
while (conditions) {
  statements;
}
```

```
do {
  statements;
} while (conditions);
```

# The `while` and `do... while` statements

Structure of the two types of `while` loops:

```c
while (conditions) {
  statements;
}
```

```c
do {
  statements;
} while (conditions);
```

Example.

```c
int i=0;
while(i++<3) {
  printf("%d",i);
}
```

```c
int i=0;
do {
  printf("%d",i);
} while(i++<3);
```

Understanding the code:

- What is the difference between the two outputs?
- What happens if `i++` is changed for `++i`?

# The `for` statement

Structure of a `for` loop:

```
1 for(init;test;step) { statements; }
```

- `init`: executed at the beginning of the loop
- `test`: tested at the beginning of each iteration
- `step`: executed at the end of each iteration

# The `for` statement

Structure of a `for` loop:

```
for(init;test;step) { statements; }
```

- `init`: executed at the beginning of the loop
- `test`: tested at the beginning of each iteration
- `step`: executed at the end of each iteration

Example.

```
for(i=0; i<n; i++)
  printf("%d ", i);
i=0; for(;i<n;i++)
  printf("%d ", i);
for(i=0; i<n;)
  {printf("%d\n",i); i++;}
for(i=0;i<n;)
  printf("%d ",i++);
```

```
fct=1;
for(i=1;i<=n;i++) fct*=i;
printf("%d ", fct);
for(i=1,fct=1;i<=n;fct*=i,i++);
printf("%d ", fct);
for(i=1,fct=1;i<=n;fct*=i++);
printf("%d\n", fct);
```

# The `break` and `continue` statements

Understanding the code:

- What are the loops on the right doing?
- How is the code indented?
- Which `for` loop is the clearest and best used?

# The `break` and `continue` statements

Understanding the code:

- What are the loops on the right doing?
- How is the code indented?
- Which `for` loop is the clearest and best used?

Acting from within a loop:

- Early exit of a loop: `break`
- Skip to the next loop iteration: `continue`

# The `break` and `continue` statements

Understanding the code:

- What are the loops on the right doing?
- How is the code indented?
- Which `for` loop is the clearest and best used?

Acting from within a loop:

- Early exit of a loop: `break`
- Skip to the next loop iteration: `continue`

Example.

```
1 for(i=0;i<10;i++) {
2   scanf("%d",&n);
3   if(n==0) break;
4   else if(n>=10) continue;
5   printf("%d\n", n);
6 }
```

# Key points

- What are the main data types in C?
- How to perform type casting?
- How to define and use structures on C?
- How to use conditional statements in C?
- How to write loops in C?

Thank you!