

C Programming



Introduction

C is a *general purpose* programming language, born in early '70

The C Programming Language - 1978 - K&R

To this day, the programming language C underwent several changes to fulfill standards imposed by different organizations (**ANSI, ISO**)

- ♦ 1989 :: C89
- ♦ 1999 :: C99
- ♦ 2011 :: C11
- ♦ 2018 :: C18



Applicazioni

Embedded Systems:

- ♦ Domotics
- ♦ PLC
- ♦ Automotive
- ♦ Medical equipment
- ♦ Security Systems
- ♦ Automation Systems

C Language

Imperative !

It allows the user to have complete control of the device through different *instruction*, seen as orders

Procedural

{ Functions }

Composed by block of code, each one identified by a unique name and enclosed by specific delimiters

High Abstraction

Significant abstraction from the operating details of a specific device and from the characteristics of the machine language

```
1 #include <stdlib.h>
2#include <stdio.h>
3
4 /* The main thing that this program does. */
5 int main(void) {
6     // Declarations
7     double A[5] = {
8         [0] = 9.0,
9         [1] = 2.9,
10        [4] = 3.E+25,
11        [3] = .00007,
12    };
13
14    // Doing some work
15    for (size_t i = 0; i < 5; ++i) {
16        printf("element %zu is %g, \tits square is\n",
17            i,
18            A[i],
19            A[i]*A[i]);
20    }
21
22    return EXIT_SUCCESS;
23 }
```

Compilation and Execution

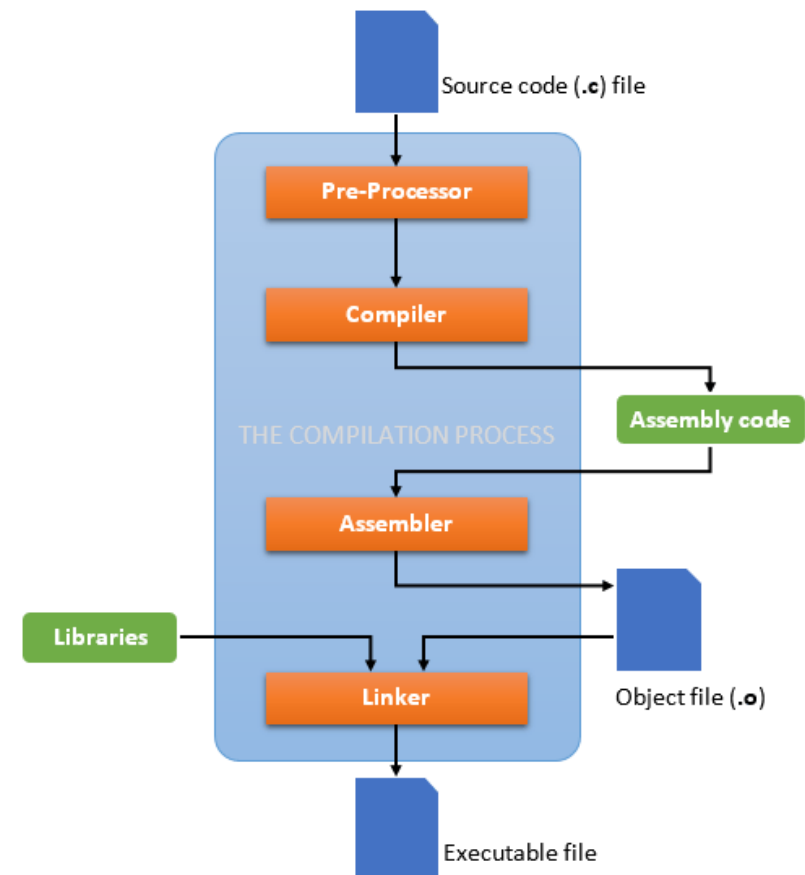
Simple strings, even if very specific, cannot be understood by a computer

C gives a new level of abstraction that can be adapted on different platforms, thanks to **compilers** and **linkers**.

The name of the **compiler**, its use and functionalities are specific and related to the platform on which the program runs

```
gcc -Wall -o program_name source_code.c -lm
```

```
arm-linux-gnueabi-gcc -std=c99 -O1 -g3 -c -  
fmessage-length=0 -pthread -MMD -MP program -o  
source.c
```



Generic Program Structure

Syntax

How to correctly write code so it is understandable by a compiler

- **Special Key-Words**

- #include, int, void, double, for, return

- **Punctuation**

- { ... }, (...), [...], /* ... */, < ... >

- **Alpha-numeric values**

- **Specific Identifiers**

- main, printf, size_t, EXIT_SUCCESS, A, i

- **Functions**

- main, printf

- **Operators**

- =, *, <, ++, -

Semantics

How to use programming language to achieve the desired objective

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 /* The main thing that this program does. */
5 int main(void) {
6     // Declarations
7     double A[5] = {
8         [0] = 9.0,
9         [1] = 2.9,
10        [4] = 3.E+25,
11        [3] = .00007,
12    };
13
14    // Doing some work
15    for (size_t i = 0; i < 5; ++i) {
16        printf("element %zu is %g, \ tits square is %g\n",
17            i,
18            A[i],
19            A[i]*A[i]);
20    }
21
22    return EXIT_SUCCESS;
23 }
```

Program Structure

Definitions and Declarations

Before using any particular identifier, you must provide the compiler with a **declaration** specifying what the identifier represents.

```
int main(void);  
double A[5];  
size_t i;
```

- ♦ `main`, followed by round brackets, it is declared as a *function* of *type* `int`
- ♦ `i` it is declared as a numerical *variable* of *type* `size_t`
- ♦ `A` it is declared as an *array* with values of *type* `double`

```
int printf(char const format[static 1], ...);  
typedef unsigned long size_t;  
#define EXIT_SUCCESS 0
```

Generally, *declarations* specify the type of object to which an identifier refers, but do not attribute any value to this object: this task is entrusted to the *definition* procedures

```
size_t i = 0;
```

This **initialization** defines the object with a corresponding name; this instruction for the compiler means to allocate an area of memory, which will be referred to by a particular name, and to keep inside it (initially) the value 0

Program Structure

Interactions and Function calls

Perform one or more operations several times

```
for (loop-variable; loop-condition; loop-post-exec)
    operations ...
```

```
while (condition)
    operations ...
```

```
do
    operations ...
while(condition);
```

Delegate the execution of chunks of
block code located elsewhere

A function is called by passing particular
arguments to it, and waiting for it to
perform its functions by **returning values**

- Call by values
- Call by reference (not native but can be achieved through **pointers**)

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

/* The main thing that this program does. */
int main(void) {

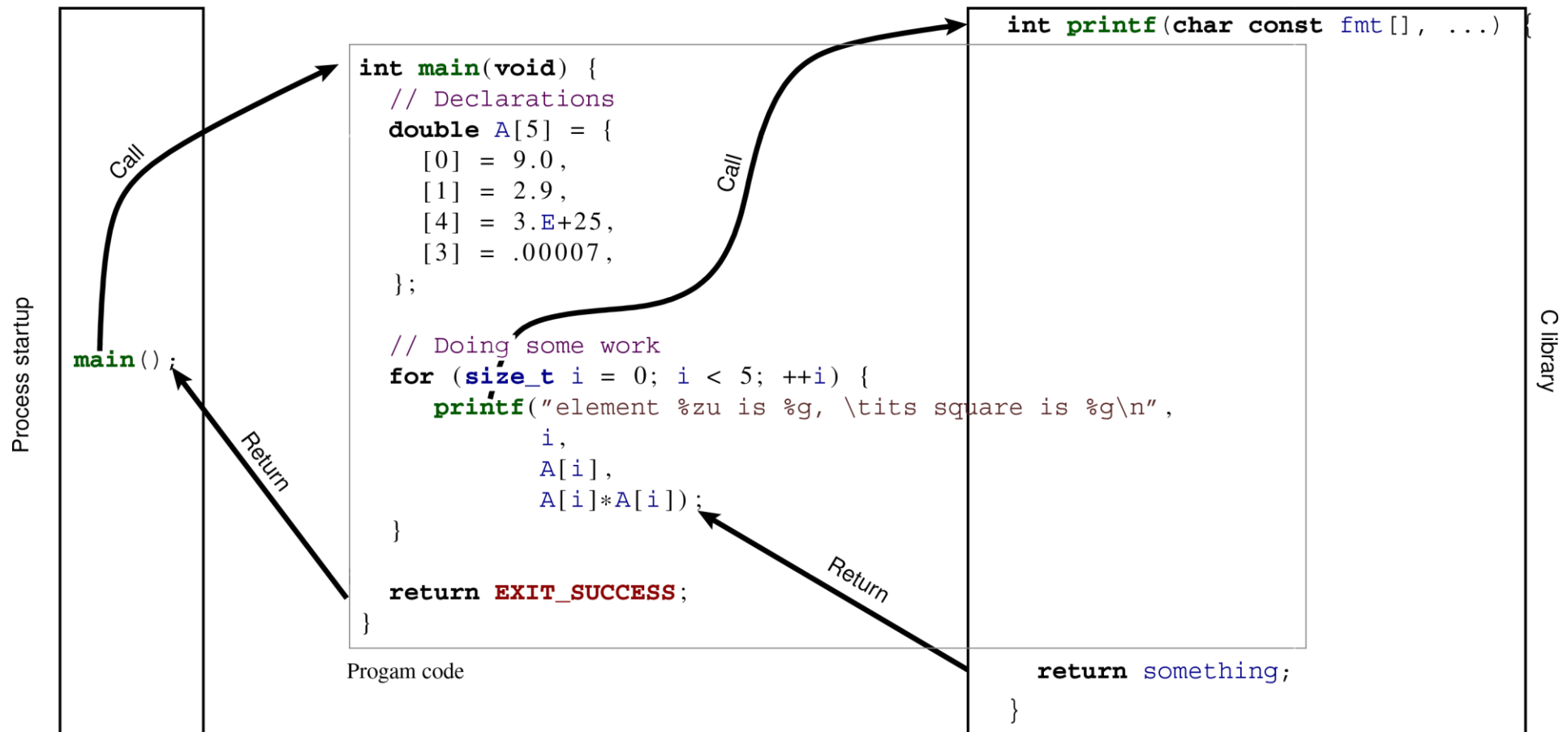
    double A[5] = {
        [0] = 9.0,
        [1] = 2.9,
        [4] = 3.E+25,
        [3] = .00007,
    };

    // Doing some work
    for (size_t i = 0; i < 5; ++i) {
        printf("element %zu is %g, \tits square is %g\n",
            i,
            A[i],
            A[i]*A[i]);
    }

    return EXIT_SUCCESS;
}
```

Program Structure

Function calls



Program Structure

Flow Control

Controlling the flow of a program generally means to manage its behaviour

calling a function is already a flow control mechanism.

Conditional Control

```
if (condition)
    operations ...
else if
    operations ...
else
    operations ...
```

```
switch (arg) {
    case 'arg1':
        operations ...
        break;

    default:
        operations ...
}
```

- 1. if
- 2. for
- 3. do
- 4. while
- 5. switch

Furthermore, in **C** there are other constructs for particular conditional control:

Ternary Operator

```
condition ? A : B
```

Pre-processing conditions

```
#if/#ifdef/#ifndef/#elif/#else/#endif
```

Expressions computation

A crucial part of many programs is the processing of expressions, whether they are **arithmetic**, **boolean** or mixed.

A programming language provides a set of operators to deal with expressions

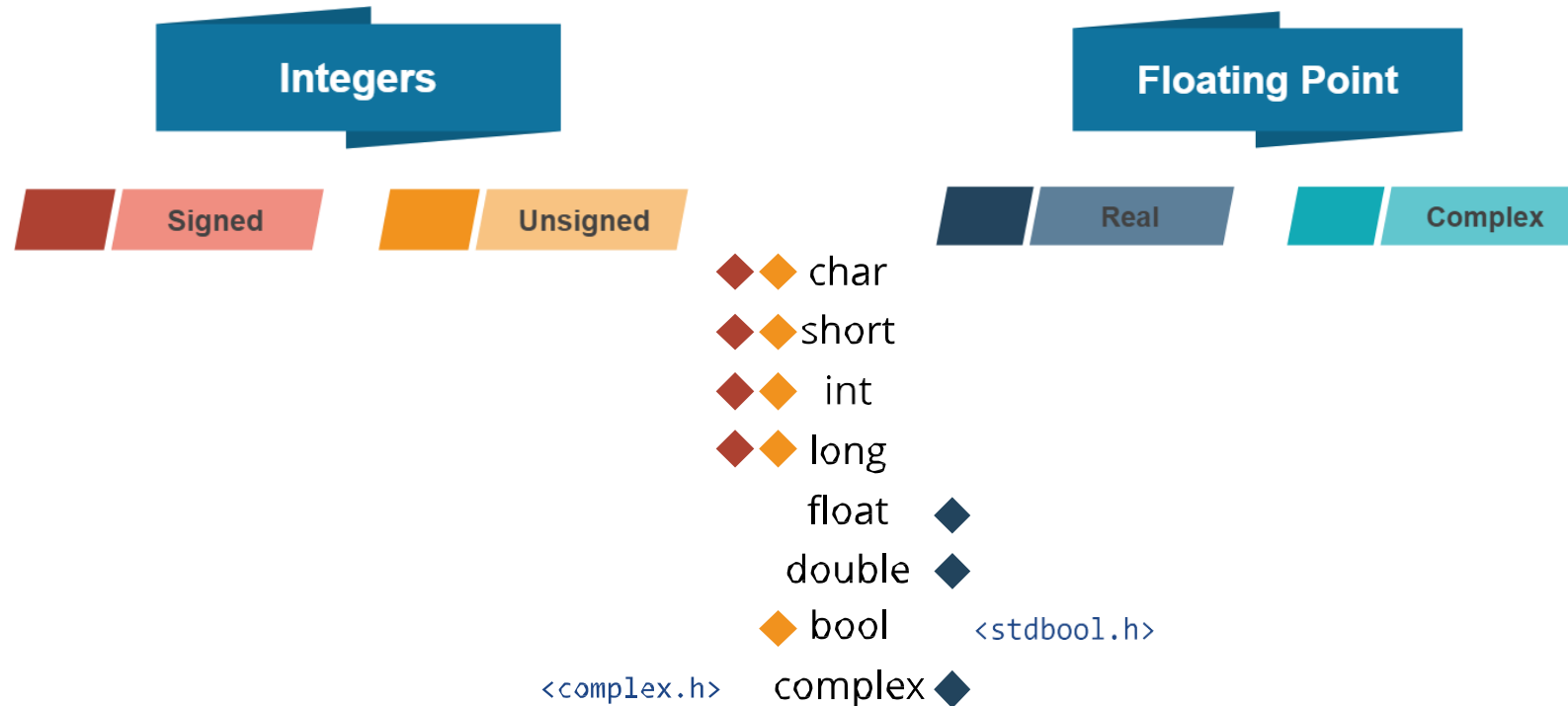
Arithmetic	Comparison	Logic
+	==	&& (<i>and</i>)
-	!=	(<i>or</i>)
*	>	! (<i>not</i>)
/	<	
◇	<=	
	>=	

Binary Arithmetics
~ (<i>complement</i>)
& (<i>bit-and</i>)
(<i>bit-or</i>)
^ (<i>xor</i>)
>> , << (<i>shift</i>)

Types of Data

A programming language like **C** works with **data**, that during the elaboration phases, may assume different **values**.

The **values** that a certain type of **data** can assume depends on its **type**



Types of Data

The amount of memory occupied by a **data** of a certain type depends on the platform for which the code is compiled, although there are directives regarding the possible size a **type** can have

Type	Dimension	Use
◆◆ char	8 bit	<code>char c = 'a'</code>
◆◆ short	16 bit	<code>short x = 3</code>
◆◆ int	16/32 bit	<code>int x = 350</code>
◆◆ long	32/64 bit	<code>long l = 123581321</code>
◆ float	16/32 bit	<code>float f = 3.52</code>
◆ double	32/64 bit	<code>double d = 3.E+25</code>
◆ bool	8 bit	<code>bool b = true</code>

`unsigned char cu = 254`

`unsigned long long ll = 9223372036854775807`

Each data in memory is stored inside a **variable**.

Types of Data

Evaluate the range values that the different types can assume

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <limits.h>
4 #include <float.h>
5
6 int main(int argc, char** argv) {
7
8     printf("CHAR_BIT      : %d\n", CHAR_BIT);
9     printf("CHAR_MAX      : %d\n", CHAR_MAX);
10    printf("CHAR_MIN      : %d\n", CHAR_MIN);
11    printf("INT_MAX       : %d\n", INT_MAX);
12    printf("INT_MIN      : %d\n", INT_MIN);
13    printf("LONG_MAX      : %ld\n", (long) LONG_MAX);
14    printf("LONG_MIN     : %ld\n", (long) LONG_MIN);
15    printf("SCHAR_MAX    : %d\n", SCHAR_MAX);
16    printf("SCHAR_MIN    : %d\n", SCHAR_MIN);
17    printf("SHRT_MAX     : %d\n", SHRT_MAX);
18    printf("SHRT_MIN     : %d\n", SHRT_MIN);
19    printf("UCHAR_MAX    : %d\n", UCHAR_MAX);
20    printf("UINT_MAX     : %u\n", (unsigned int) UINT_MAX);
21    printf("ULONG_MAX    : %lu\n", (unsigned long)
    ULONG_MAX);
22    printf("USHRT_MAX    : %d\n", (unsigned short)
    USHRT_MAX);
23
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <limits.h>
4 #include <float.h>
5
6 int main(int argc, char** argv) {
7
8     printf("Storage size for float : %d\n",
9     printf("FLT_MAX      : %g\n", (float) FLT_MAX);
10    printf("FLT_MIN      : %g\n", (float) FLT_MIN);
11    printf("-FLT_MAX     : %g\n", (float) -FLT_MAX);
12    printf("-FLT_MIN     : %g\n", (float) -FLT_MIN);
13    printf("DBL_MAX      : %g\n", (double) DBL_MAX);
14    printf("DBL_MIN      : %g\n", (double) DBL_MIN);
15    printf("-DBL_MAX     : %g\n", (double) -DBL_MAX);
16    printf("Precision value: %d\n", FLT_DIG );
17
18    return 0;
19 }
```

Data and Variable

Qualifiers

Qualifiers map to types in terms of attributes for the variable they are associated with



const

Associated with a specific data, it specifies that you do not have the rights to modify it

```
const int x = 2;  
int x+3; x;
```



volatile

It indicates to the compiler that the value thus qualified could change at any time even by means of code that is not "near" the declaration of the variable



_Atomic

It is associated with a variable whose reading / writing must take place in a single instruction: during the change of value of the variable nothing else must not occur



restrict

Arrays and Structures

There are other types of data (apparently more complex) which however derive from the basic types presented.

Two of these are called **aggregate types**, since they arise from the combination of several objects (even if not of the same type):

- **(Arrays)**
 - It is an aggregation of objects of the same type
- **(Structures)**
 - It is an aggregation of objects of different types

```
double a[4];  
double b[] = { [3] = 42.0, [2] = 37.0, };  
double c[] = { 22.0, 17.0, 1, 0.5, };
```

```
struct microcontroller{  
    char name[50];  
    int bitNo;  
    int ramKb;  
    float price;  
};  
struct microcontroller const my_micro = {  
    .name = "ARM",  
    .bitNo = 32,  
    .ramKb = 128,  
    .price = 2.87  
};
```

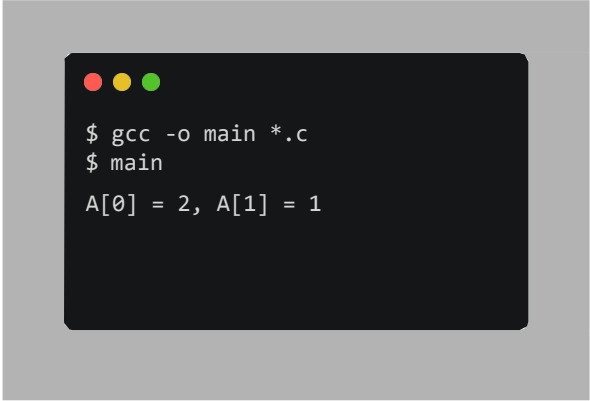
Array and Structures

Examples

```
#include <stdio.h>

void swap_double(double a[2]) {
    double tmp = a[0];
    a[0] = a[1];
    a[1] = tmp;
}

int main(void) {
    double A[2] = { 1.0, 2.0 };
    swap_double(A);
    printf("A[0] = %g, A[1] = %g\n", A[0], A[1]);
}
```



```
$ gcc -o main *.c
$ main
A[0] = 2, A[1] = 1
```


Array and Structures

Examples

```
#include <stdio.h>

float weighted_mean(int x[], int w[], int n) {
    int w_sum = 0, num = 0;

    for (int i = 0; i < n; i++){
        num = num + x[i] * w[i];
        w_sum = w_sum + w[i];
    }

    return (float)(num / w_sum);
}

int main(){

    int x[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int w[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int n = sizeof(x)/sizeof(x[0]);
    int m = sizeof(w)/sizeof(w[0]);

    if (n == m){
        float result = weighted_mean(x, w, n);
        printf("Weigthed mean: %g\n", result);
    }else{
        printf("n!=m\n");
        return -1;
    }
    return 0;
}
```



```
$ gcc -o main *.c
$ main
Weigthed mean: 7
```

Array and Structures

Examples

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

bool leapyear(unsigned year) {
    return !(year % 4) && ((year % 100) || !(year % 400));
}

#define DAYS_BEFORE \
    (const int[12]){ \
        [0] = 0, [1] = 31, [2] = 59, [3] = 90, \
        [4] = 120, [5] = 151, [6] = 181, [7] = 212, \
        [8] = 243, [9] = 273, [10] = 304, [11] = 334, \
    }

struct tm time_set_yday(struct tm t) {
    t.tm_yday += DAYS_BEFORE[t.tm_mon] + t.tm_mday - 1;

    if ((t.tm_mon > 1) && leapyear(t.tm_year+1900)){
        t.tm_yday++;
    }
    return t;
}

int main() {
    struct tm today = {
        .tm_year = 2020-1900,
        .tm_mon = 2,
        .tm_mday = 19,
        .tm_hour = 10,
        .tm_min = 0,
        .tm_sec = 47,
    };
    printf("This year is %d, next year will be %d\n",
        today.tm_year+1900, today.tm_year+1900+1);
    today = time_set_yday(today);
    printf("Day of the year is %d\n", today.tm_yday);

    return EXIT_SUCCESS;
}
```

```
struct tm{
int tm_sec; /* secondi (0-59) */
int tm_min; /* minuti (0-59) */
int tm_hour; /* ora (0-23) */
int tm_mday; /* giorno del mese (1-31) */
int tm_mon; /* mese (0-11) */
int tm_year; /* anno dal 1900 */
int tm_wday; /* giorno della settimana (0-6) */
int tm_yday; /* giorno dell'anno (0-365) */
int tm_isdst; /* Ora legale */
}
```

Pointers

A **pointer** is a variable that contains the memory address of another variable

It is therefore a variable that does not directly contain the data we are interested in but points to the data by providing the memory address in which it is stored

Direction operator (dereferencing) : *

Returns the contents of the memory area pointed to by the pointer

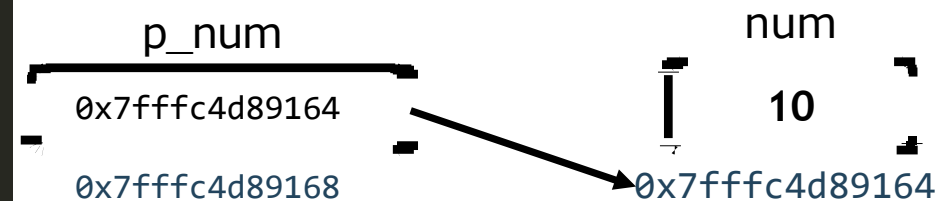
```
int *p1    /* Pointer to an integer */
double *p2 /* Pointer to a double */
char *p3   /* Pointer to a character */
```

Unary referencing operator : &

Returns the address where the pointed variable is located

```
double a = 10;
double *p;
p = &a; /* Get address of 'a' */
```

```
1 #include <stdio.h>
2
3 int main(){
4     int num = 10;
5     int *p_num = &num;
6     printf("Value of variable num is: %d", num);
7     printf("\nAddress of variable num is: %p", p_num);
8
9
10  return 0;
```

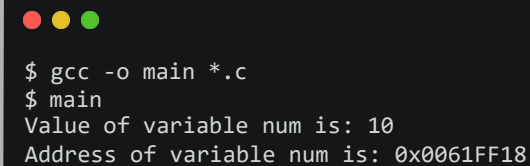


Pointers

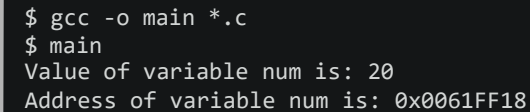
Examples

```
1 #include <stdio.h>
2
3 void pointer_test(int val){
4     val = 20;
5 }
6
7 int main(){
8     int num = 10;
9     int *p_num = &num;
10    pointer_test(num);
11    printf("Value of variable num is: %d", num);
12    printf("\nAddress of variable num is: %p", p_num);
13
14    return 0;
15 }
```

```
1 #include <stdio.h>
2
3 void pointer_test(int *val){
4     *val = 20;
5 }
6
7 int main(){
8     int num = 10;
9     int *p_num = &num;
10    pointer_test(p_num);
11    printf("Value of variable num is: %d", num);
12    printf("\nAddress of variable num is: %p", p_num);
13
14
15    return 0;
16 }
```



```
$ gcc -o main *.c
$ main
Value of variable num is: 10
Address of variable num is: 0x0061FF18
```



```
$ gcc -o main *.c
$ main
Value of variable num is: 20
Address of variable num is: 0x0061FF18
```

Pointers and Array

Pointers and **Arrays** are closely related in C and are often mistakenly interchanged

If you use only the name of the variable associated with an **array**, without any index, you get a **pointer** to its first element

```
int array[2] = { 3, 5 };
```

```
array == &array[0]
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void){
5     char a[7]={'a', 'b', 'c', 'd', 'e', 'f', 'g'};
6     printf("a[3]: %c, *(a+3): %c", a[3], *(a+3));
7     return 0;
8 }
```

```
$ gcc -o main *.c
$ main
a[3]: d, *(a+3): d
```

Pointers Arithmetics

The operations are carried out implicitly taking into account the size of the variables pointed to

Pointers and Arrays

You can **dynamically** define an **array** in memory using **pointers**

This definition leads to the creation of a fixed sized array known at compilation time

```
int array[13];
```

Using pointers, you can define an array of known size at runtime

```
int *array = malloc(x * sizeof(int));
```

- ◆ **malloc** Allocate a specific number of bytes in memory
- ◆ **realloc** Increases or decreases the size of the indicated memory block
- ◆ **calloc** Allocate a specific number of bytes in memory initialized to 0
- ◆ **free** Frees specific number of bytes, leaving them back to the system

Particular type: **void***

Casting operations may be needed

```
int *arr = (int*) malloc(2 * sizeof(int));  
arr[0] = 1;  
arr[1] = 2;  
arr = realloc(arr, 3 * sizeof(int));  
arr[2] = 3;
```

Pointers and Array

Examples

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

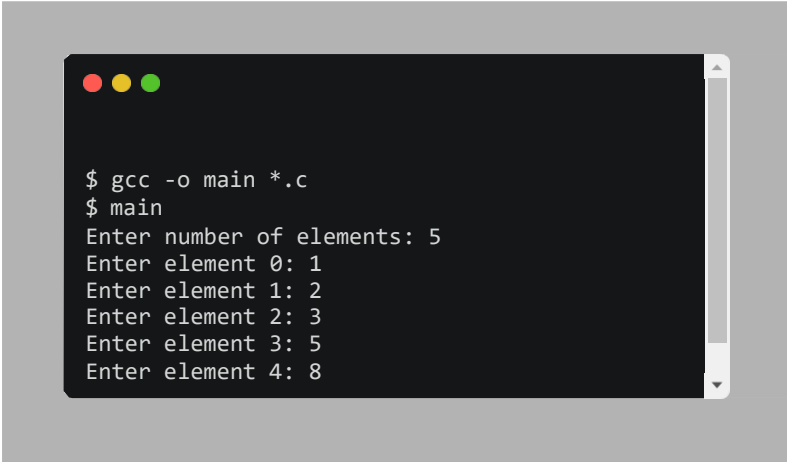
int main(){
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) calloc(n, sizeof(int));

    if(ptr == NULL){
        printf("Error! memory not allocated.");
        exit(0);
    }

    for(i = 0; i < n; ++i){
        printf("Enter element %d: ", i);
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d\n", sum);
    free(ptr);
    return 0;
}
```



A terminal window with a dark background and light-colored text. It shows the compilation and execution of the C program. The user enters 5 for the number of elements, and then enters the values 1, 2, 3, 5, and 8 for each element. The program outputs the sum of these elements, which is 22.

```
$ gcc -o main *.c
$ main
Enter number of elements: 5
Enter element 0: 1
Enter element 1: 2
Enter element 2: 3
Enter element 3: 5
Enter element 4: 8
Sum = 22
```

Programming

Convolution

In mathematics and, in particular, functional analysis, **convolution** is a mathematical operation on two functions f and g , producing a third function that is typically viewed as a modified version of one of the original functions

(from wikipedia.com)

$$f = \begin{bmatrix} 1 & 4 & 5 & 2 \end{bmatrix} \quad g = \begin{bmatrix} 3 & 4 & 1 \end{bmatrix} \quad h = f * g$$

$$\begin{array}{cccc} & 1 & 4 & 5 & 2 \\ 1 & 4 & 3 & & \end{array}$$

$$h[0] = 1 * 3$$

$$\begin{array}{cccc} & 1 & 4 & 5 & 2 \\ 1 & 4 & 3 & & \end{array}$$

$$h[1] = 1 * 4 + 4 * 3$$

$$\begin{array}{cccc} & 1 & 4 & 5 & 2 \\ 1 & 4 & 3 & & \end{array}$$

$$h[2] = 1 * 1 + 4 * 4 + 5 * 3$$

$$\begin{array}{cccc} & 1 & 4 & 5 & 2 \\ 1 & 4 & 3 & & \end{array}$$

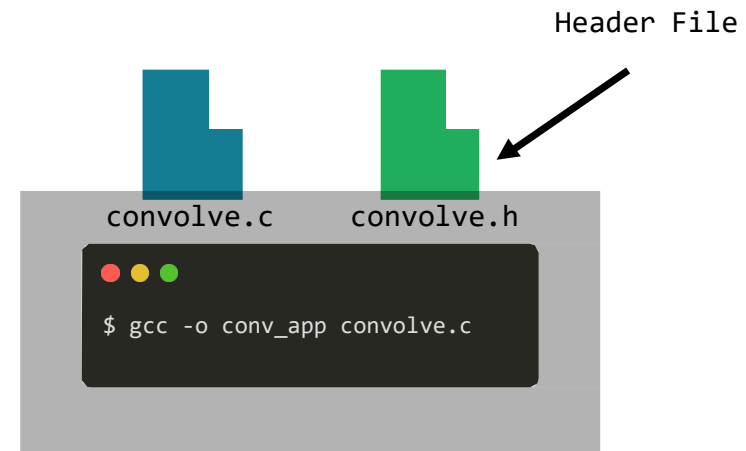
$$h[3] = 4 * 1 + 5 * 4 + 2 * 3$$

$$\begin{array}{cccc} & 1 & 4 & 5 & 2 \\ & 1 & 4 & 3 & \end{array}$$

$$h[4] = 5 * 1 + 2 * 4$$

$$\begin{array}{cccc} & 1 & 4 & 5 & 2 \\ & & 1 & 4 & 3 \end{array}$$

$$h[5] = 2 * 1$$



Programming

Moving Average

A **moving average** is a form of a convolution often used in time series analysis to smooth out noise in data by replacing a data point with the average of neighboring values in a moving window: it operates by taking the arithmetic mean of a number of past input samples in order to produce each output sample.

$$\bar{y}_n = \frac{1}{N} \sum_{i=0}^{N-1} x_{n-i} \longrightarrow y_n = (y_{n-1} + x_n - x_{n-N+1})$$

$$x = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\bar{y}_1 = \frac{1}{5}$$

$$x = \begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\bar{y}_2 = (1 + 2) \frac{1}{5}$$

$$x = \begin{bmatrix} 1 & 2 & 3 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\bar{y}_3 = (1 + 2 + 3) \frac{1}{5}$$

$$x = \begin{bmatrix} 1 & 2 & 3 & 5 & 0 & 0 & 0 \end{bmatrix}$$

$$\bar{y}_4 = (1 + 2 + 3 + 5) \frac{1}{5}$$

$$x = \begin{bmatrix} 1 & 2 & 3 & 5 & 8 & 0 & 0 \end{bmatrix}$$

$$\bar{y}_5 = (1 + 2 + 3 + 5 + 8) \frac{1}{5}$$

$$x = \begin{bmatrix} 1 & 2 & 3 & 5 & 8 & 13 & 0 \end{bmatrix}$$

$$\bar{y}_6 = (2 + 3 + 5 + 8 + 13) \frac{1}{5}$$

$$x = \begin{bmatrix} 1 & 2 & 3 & 5 & 8 & 13 & 21 \end{bmatrix}$$

$$\bar{y}_7 = (3 + 5 + 8 + 13 + 21) \frac{1}{5}$$

```
$ gcc main.c moving_average_filter.c -o maf_app
$ maf_app
```