Escuela de
Ciencia y Tecnología
ECyT_UNSAM

Programación (en C)

Primer Cuatrimestre 2025

programacionbunsam@gmail.com

Variables, funciones, macros y Headers

¿Qué es una variable?

→ Las variables son lugares en la memoria con algún nombre que nos permite guardar algún tipo de dato y recuperarlo cuando sea necesario.

¿Qué es una variable?

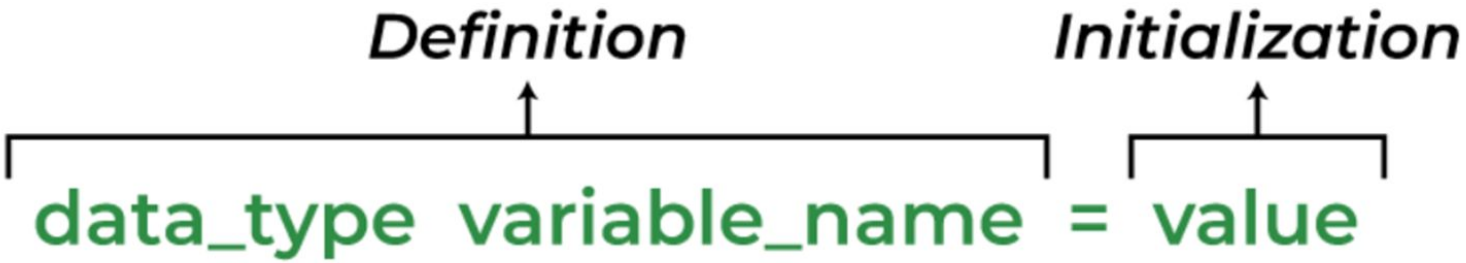
→ Las variables son lugares en la memoria con algún nombre que nos permite guardar algún tipo de dato y recuperarlo cuando sea necesario. Como sabemos, las variables pueden guardar distintos tipos de datos, e incluso cambiar el valor que guardan. No pueden, sin embargo, guardar datos de distinto tipo.

¿Qué es una variable?

→ Las variables son lugares en la memoria con algún nombre que nos permite guardar algún tipo de dato y recuperarlo cuando sea necesario. Como sabemos, las variables pueden guardar distintos tipos de datos, e incluso cambiar el valor que guardan. No pueden, sin embargo, guardar datos de distinto tipo. En C las variables deben ser declaradas, aclarando siempre que tipo de valor guardarán y, opcionalmente, inicializándolas.

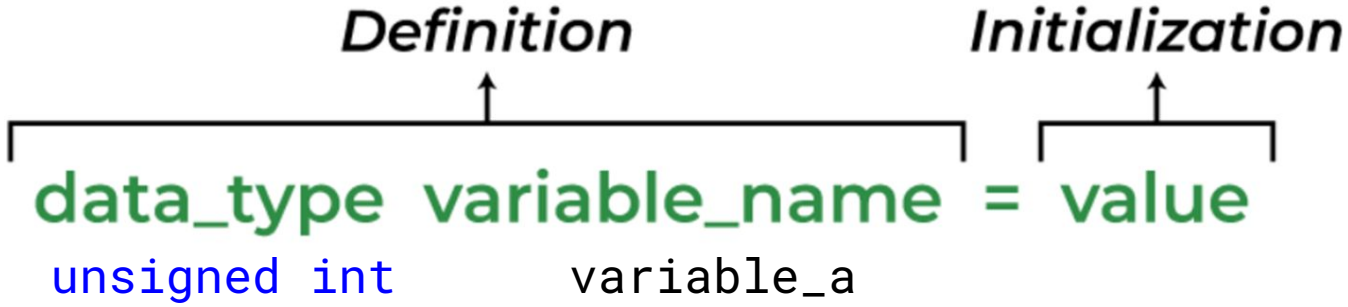
¿Qué es una variable?

→ Las variables son lugares en la memoria con algún nombre que nos permite guardar algún tipo de dato y recuperarlo cuando sea necesario. Como sabemos, las variables pueden guardar distintos tipos de datos, e incluso cambiar el valor que guardan. No pueden, sin embargo, guardar datos de distinto tipo. En C las variables deben ser declaradas, aclarando siempre que tipo de valor guardarán y, opcionalmente, inicializándolas.



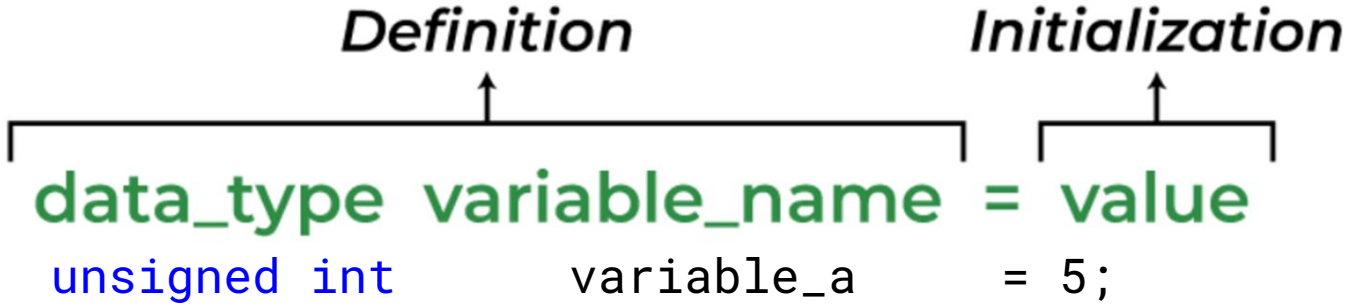
¿Qué es una variable?

→ Las variables son lugares en la memoria con algún nombre que nos permite guardar algún tipo de dato y recuperarlo cuando sea necesario. Como sabemos, las variables pueden guardar distintos tipos de datos, e incluso cambiar el valor que guardan. No pueden, sin embargo, guardar datos de distinto tipo. En C las variables deben ser declaradas, aclarando siempre que tipo de valor guardarán y, opcionalmente, inicializándolas.



¿Qué es una variable?

→ Las variables son lugares en la memoria con algún nombre que nos permite guardar algún tipo de dato y recuperarlo cuando sea necesario. Como sabemos, las variables pueden guardar distintos tipos de datos, e incluso cambiar el valor que guardan. No pueden, sin embargo, guardar datos de distinto tipo. En C las variables deben ser declaradas, aclarando siempre que tipo de valor guardarán y, opcionalmente, inicializándolas.



Etapas en la creación de una variable

1. Declaración: es el proceso mediante el cual C le avisa al compilador de la existencia de una variable, con su respectivo nombre y tipo de dato.

Etapas en la creación de una variable

1. Declaración: es el proceso mediante el cual C le avisa al compilador de la existencia de una variable, con su respectivo nombre y tipo de dato.

2. Definición: en la definición de una variable, el compilador le asigna la memoria a la variable junto con algún valor. Si la variable no es inicializada, se le asignará algún valor random.

Etapas en la creación de una variable

1. Declaración: es el proceso mediante el cual C le avisa al compilador de la existencia de una variable, con su respectivo nombre y tipo de dato.

2. Definición: en la definición de una variable, el compilador le asigna la memoria a la variable junto con algún valor. Si la variable no es inicializada, se le asignará algún valor random. (obs. para nosotros, en este caso, 1 y 2 son indistinguibles)

Etapas en la creación de una variable

1. Declaración: es el proceso mediante el cual C le avisa al compilador de la existencia de una variable, con su respectivo nombre y tipo de dato.

2. Definición: en la definición de una variable, el compilador le asigna la memoria a la variable junto con algún valor. Si la variable no es inicializada, se le asignará algún valor random. (obs. para nosotros, en este caso, 1 y 2 son indistinguibles)

3. Inicialización: es el proceso en el cual el usuario le asigna por primera vez un valor significativo a una variable.

Incialización	Asignación
Ocurre cuando una variable es declarada y, al mismo tiempo, se le asigna un valor inicial. Una variable puede ser inicializada una única vez en un programa.	La asignación de una variable involucra settear o actualizar el valor de una variable que ya fue declarada (más allá de si la misma fue, además, inicializada). Este proceso puede ocurrir muchas veces a lo largo de un archivo.

Tipos de variable

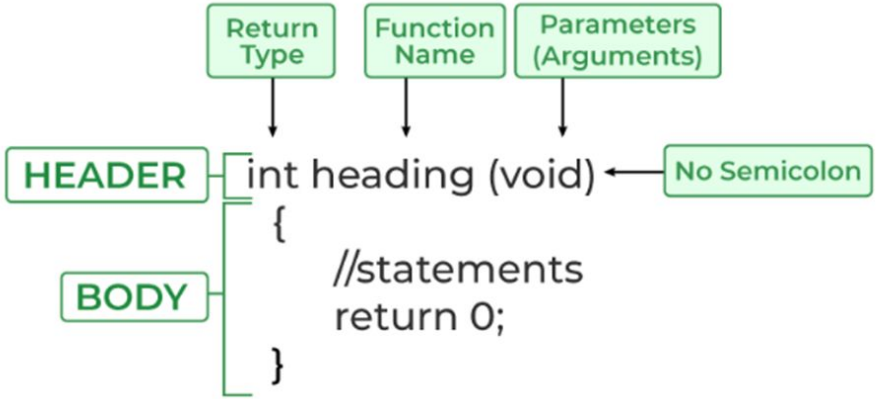
Variable	Declaración	Alcance
Local	Dentro de un bloque de código o una función	Bloque o función en la que fue declarada
Global	Fuera de un bloque de código o una función	Todo el source file
Static	keyword *static*. Solo puede ser definida una vez en el programa, y no es destruida cuando el programa sale del bloque / función	Depende de donde haya sido declarada. Nunca más de un source file
Automatic	keyword *auto*. Es el default de las variables locales, por lo que se lo utiliza muy raramente	Local. Son eliminadas cuando el programa sale del bloque / función
External	keyword *extern*. Se declaran fuera de un bloque de código / función. Se definen en un source file y se declaran en otros con el keyword extern.	Todo el programa, permite acceder a una misma variable desde múltiples source files

Funciones

Las funciones son los bloques constructivos básicos de C, proveyendo al código tanto modularidad como reusabilidad. Son un set de instrucciones que, al ser llamadas, realizan una tarea determinada. Este set de instrucciones estará siempre encerrado entre llaves {}.

Funciones

Las funciones son los bloques constructivos básicos de C, proveyendo al código tanto modularidad como reusabilidad. Son un set de instrucciones que, al ser llamadas, realizan una tarea determinada. Este set de instrucciones estará siempre encerrado entre llaves {}.

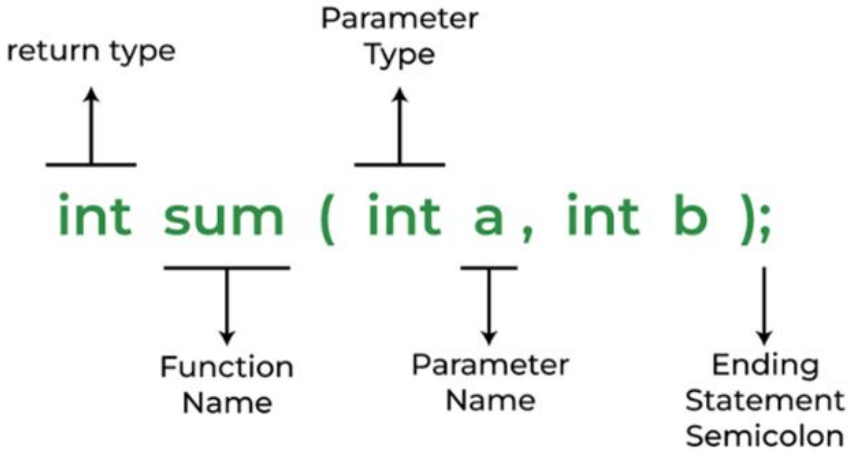


Etapas en la creación de una función

1. Declaración: en esta etapa, debemos proveer el nombre de la función, su tipo de retorno, y el número y tipo de argumentos que la misma toma. La declaración le dice al compilador que hay una función definida con este nombre en algún lugar del programa

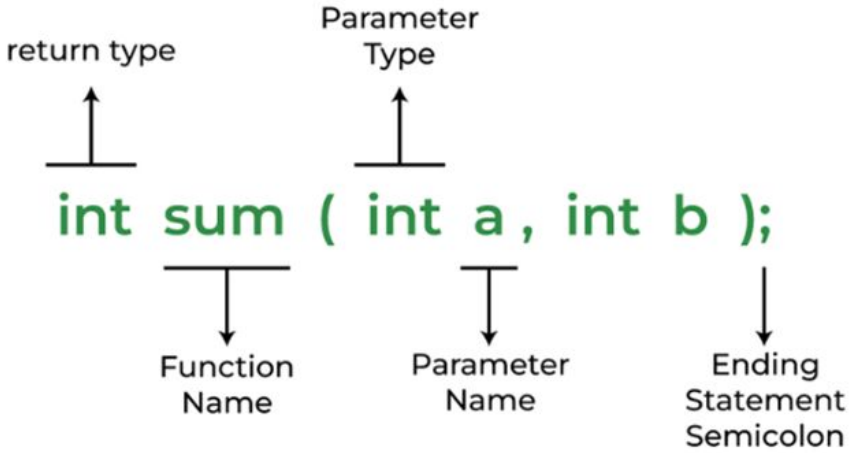
Etapas en la creación de una función

1. Declaración: en esta etapa, debemos proveer el nombre de la función, su tipo de retorno, y el número y tipo de argumentos que la misma toma. La declaración le dice al compilador que hay una función definida con este nombre en algún lugar del programa.



Etapas en la creación de una función

1. Declaración: en esta etapa, debemos proveer el nombre de la función, su tipo de retorno, y el número y tipo de argumentos que la misma toma. La declaración le dice al compilador que hay una función definida con este nombre en algún lugar del programa.



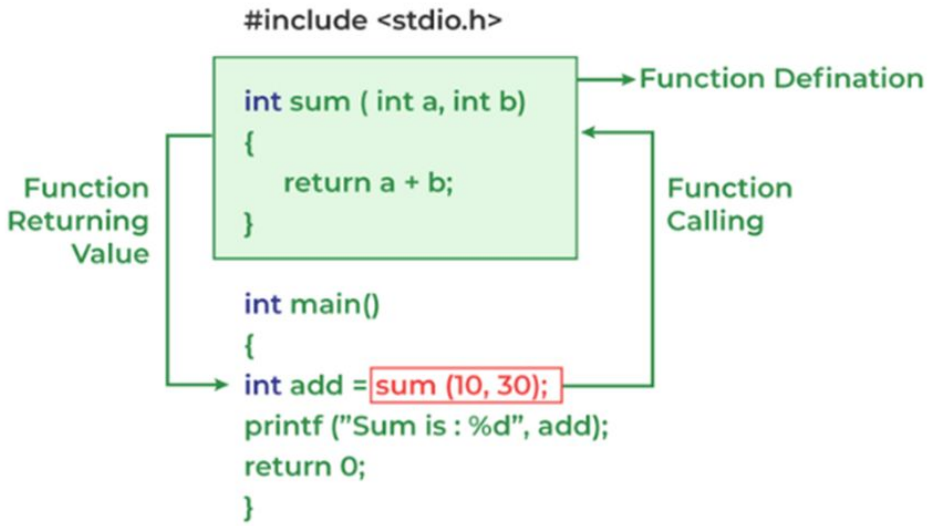
2. Definición: la definición consiste en el set de instrucciones que serán ejecutadas cuando la función sea llamada.

Etapas en la creación de una función

3. Llamada: una llamada es una instrucción que le dice al compilador que ejecute una función. En la misma utilizamos tanto el nombre como los parámetros de la función.

Etapas en la creación de una función

3. Llamada: una llamada es una instrucción que le dice al compilador que ejecute una función. En la misma utilizamos tanto el nombre como los parámetros de la función.



Tipos de funciones

1. Library Functions: también llamadas ‘built-in-function’ son funciones que están contenidas dentro de un paquete de compilador, cada una con un significado específico. Tienen la ventaja de que pueden ser usadas sin necesidad de ser definidas por el usuario y están optimizadas para mejor performance. En este caso no siempre tenemos acceso a la definición de la función.

i.e. pow(), printf(), sqrt(), strcomp(), strcpy(), etc.

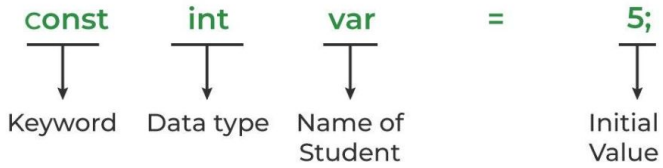
2. User Defined Functions: estas son funciones creadas por el programador. También son llamadas ‘tailor-made-functions’ ya que están creadas para trabajar con un programa en particular, adecuándose a las necesidades del programador en el momento de su creación. Si bien son modificables, es necesario declararlas y definirlas.

Constants y macros

- Constants: En C, las constantes son valores de solo lectura que no cambian durante la ejecución. Pueden ser enteras, flotantes, de texto o caracteres. Solo es posible asignarle un valor en el momento de su declaración

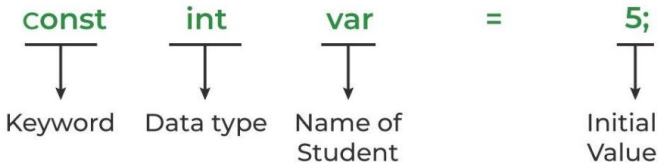
Constants y macros

- Constants: En C, las constantes son valores de solo lectura que no cambian durante la ejecución. Pueden ser enteras, flotantes, de texto o caracteres. Solo es posible asignarle un valor en el momento de su declaración



Constants y macros

- Constants: En C, las constantes son valores de solo lectura que no cambian durante la ejecución. Pueden ser enteras, flotantes, de texto o caracteres. Solo es posible asignarle un valor en el momento de su declaración



- Macros: En C, una macro es un nombre simbólico que representa un valor, expresión o código. Se define con *#define* y el preprocesador la reemplaza por su contenido.

Tipos de Macros

1. Object-like macros → `#define DATE 31`

Tipos de Macros

1. Object-like macros → #define DATE 31

```
#define DATE 31

int main(){
   printf("Lockdown will be extended"
          " upto %d-MAY-2020",
          DATE);

Output:
Lockdown will be extended upto 31-MAY-2020
```

Tipos de Macros

1. Object-like macros → `#define DATE 31`
2. Chain macros →
```
#define INSTAGRAM FOLLOWERS
 #define FOLLOWERS 10
```

Tipos de Macros

1. Object-like macros → `#define DATE 31`
2. Chain macros → `#define INSTAGRAM FOLLOWERS`
   `#define FOLLOWERS 10`

```
#define INSTAGRAM FOLLOWERS
#define FOLLOWERS 10

int main(){
    printf("Geeks for Geeks have %dK"
           " followers on Instagram",
           INSTAGRAM);

Output:
Geek for Geeks have 10K followers on Instagram
```

Tipos de Macros

1. Object-like macros → `#define DATE 31`
2. Chain macros →
```
#define INSTAGRAM FOLLOWERS
 #define FOLLOWERS 10000
```

3. Multi-line macros →
```
#define ELE  1, \
              2, \
              3
```

Tipos de Macros

1. Object-like macros → `#define DATE 31`
2. Chain macros → `#define INSTAGRAM FOLLOWERS`
   `#define FOLLOWERS 10000`

3. Multi-line macros → `#define ELE 1, \`
   `2, \`
   `3`

```
#define ELE 1, \
            2, \
            3
int main(){
    // Array arr[] with elements defined in macros
    int arr[] = { ELE };
    for (int i = 0; i < 3; i++) {
        printf("%d  ", arr[i]);
    }
    return 0;
}
```

Output:

1 2 3

Tipos de Macros

1. Object-like macros → `#define DATE 31`
2. Chain macros →
```
#define INSTAGRAM FOLLOWERS
 #define FOLLOWERS 10000
```
3. Multi-line macros →
```
#define ELE  1, \
              2, \
              3
```
4. Function-like macros → `#define CUBE(b) b*b*b`

Tipos de Macros

1. Object-like macros → `#define DATE 31`
2. Chain macros → `#define INSTAGRAM FOLLOWERS`
   `#define FOLLOWERS 10000`
3. Multi-line macros → `#define ELE 1, \`
   `2, \`
   `3`
4. Function-like macros → `#define CUBE(b) b*b*b`

```
#define CUBE(b) b*b*b
int main() {
     printf("%d", CUBE(1+2));
     return 0;
}
Output:
¿ 27 ? ¿ 7 ?
```

Tipos de Macros

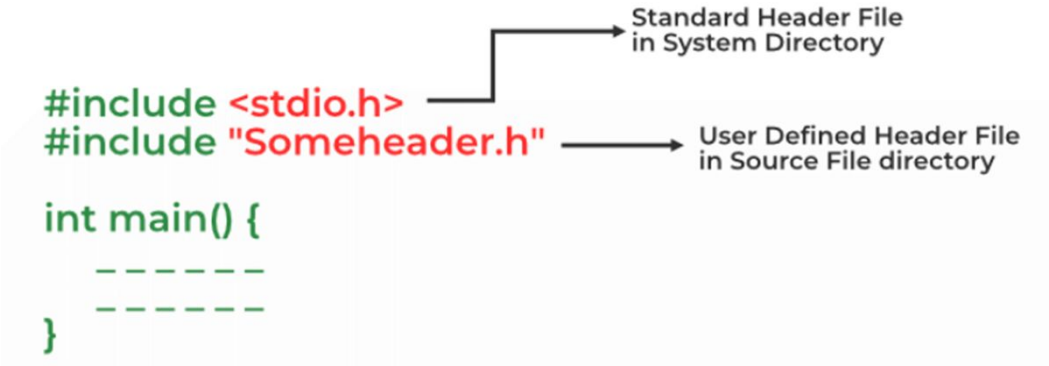
1. Object-like macros → `#define DATE 31`
2. Chain macros → `#define INSTAGRAM FOLLOWERS`
   `#define FOLLOWERS 10000`
3. Multi-line macros → `#define ELE 1, \`
   `2, \`
   `3`
4. Function-like macros → `#define CUBE(b) b*b*b`

```
#define CUBE(b) b*b*b
int main() {
     printf("%d", CUBE(1+2));
     return 0;
}
Output:
¿ 27 ? ¿ 7 ?
```

```
#define SUM(a,b,c) a+b+c
int main() {
     printf("%d", SUM(1,,2));
     return 0;
}
Output:
¿ error ? ¿ 3 ?
```

Header files

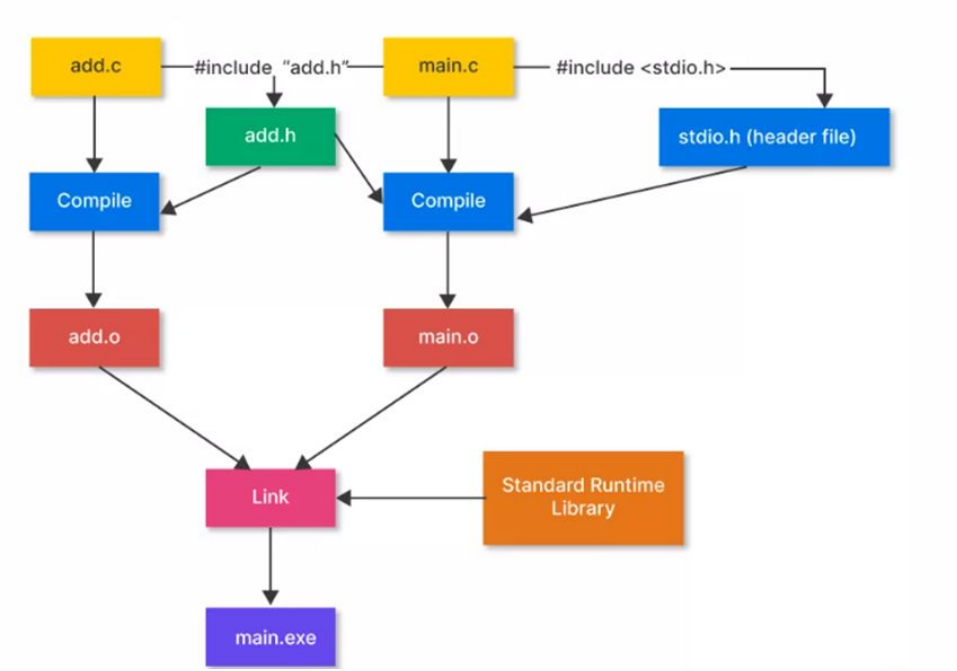
- Los header files son archivos de texto, que contienen definiciones de funciones, variables y macros que son compartidos a lo largo de muchos source files. Tienen la función de proveer una interfaz que agrupe funciones y estructuras de datos que pueden ser usadas por otras partes del programa. Están definidos por el sufijo .h



Header files - usos

1. Agrupan funciones, typedefs y macros, facilitando mantenimiento.
2. Declaran funciones para que sean usadas en varios archivos, evitando que haya duplicados y asegurando consistencia.
3. Ayudan en la modularización y reutilización del código.
4. Ayudan al compilador a detectar *typedef mismatches* y otros errores
5. Definen la interfaz pública de las librerías.

Header files - safeguard



•Cuando la estructura de archivos se complejiza puede que tengamos 2 *source files* que incluyan un mismo .h, lo que causa errores. Para evitar esto, se usa un safeguard.

```
#ifndef HEADER_FILE_NAME
#define HEADER_FILE_NAME
the entire header file
#endif
```

Múltiples archivos - best practices

1. Auto-contenimiento: cada .c o .h debe incluir todas sus dependencias
2. Comentarios: que expliquen el objetivo / funcionalidad de cada macro o función
3. Minimalismo: que los .h tengan lo justo y necesario para funcionar
4. Nombres consistentes (Coding Style Guide)

Múltiples archivos - best practices

1. Auto-contenimiento: cada .c o .h debe incluir todas sus dependencias
2. Comentarios: que expliquen el objetivo / funcionalidad de cada macro o función
3. Minimalismo: que los .h tengan lo justo y necesario para funcionar
4. Nombres consistentes (Coding Style Guide)

IMPORTANTE:

independientemente del compilador que utilicen, la ubicación de los archivos es importante. **En nuestro caso** les pedimos que por favor todos los archivos .c, .h, etc estén siempre dentro de la misma carpeta.