Fundamentos del Software: introducción a la programación verificada

Programación segura

Introducción al lenguaje C

Roberto Blanco[†] & Ricardo J. Rodríguez[‡]

MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

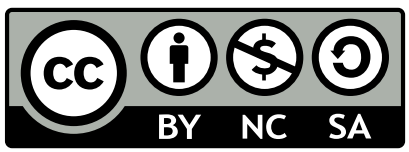
Universidad Zaragoza

[†]Max Plank Institute for Security and Privacy
Bochum, Alemania

[‡]Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza (España)

Septiembre 2020

Universidad de Zaragoza



Índice

Universidad Zaragoza

Outline

Universidad Zaragoza
1542

Introducción

Un poco de historia...

- **Diseñado por Dennis Ritchie en Bell Laboratories en los 70s**
  - Desarrollado en conjunto con UNIX OS, por la necesidad de tener un lenguaje de programación estructurado y portable
- Influenciado por los diferentes lenguajes de programación de aquellos años (e.g., ALGOL, CPL, B)
- **Muy próximo al lenguaje ensamblador** (bajo nivel)
  - Acceso directo a la memoria a través de manipulación de punteros
  - Sintaxis concisa, con un pequeño conjunto de palabras clave
- **Y con algunas cosas de lenguajes de programación de alto nivel**:
  - Estructura de bloques
  - Encapsulación de código (funciones)
  - Chequeo de tipado (débil)
- **Libro de referencia**: *The C Programming Language*, por Brian Kernighan y Dennis Ritchie, 2nd Edition, Prentice Hall. ISBN 9780131101630

Universidad Zaragoza

Introducción

Algunas cosas de interés antes de continuar...

- **C no es orientado a objetos!**
  - No existen conceptos como “private” o “protected” que permita ocultar datos de elementos
- **Problemas de portabilidad**
  - Se puede programar a bajo nivel para mejorar el rendimiento de los programas. Sin embargo, puede que no sea portable a otras plataformas
- **El compilador y el run-time a veces son un poco permisivos**: raramente evitan que el programa haga cosas estúpidas/incorrectas
  - **El chequeo de tipado es débil**
  - Por defecto, **no se comprueba en tiempo de ejecución errores en acceso a límites de arrays** (como en Java)

Universidad Zaragoza

Introducción

C estándar

- **ANSI C**: estandarizado en 1989 por American National Standards Institute
- En 1990, estandarizado por ISO (adoptado por ANSI): **C89**
- Actualizado en 1995 (C95) y en 1999 (**C99**)
- NOTA: **cada estándar puede definir diferentes comportamientos de ciertos operandos** (como operandos de desplazamiento o aritméticos)
- C++: extensión de C para programación orientada a objetos (y otras mejoras)
  - C no es un subconjunto de C++, pero puedes programar en C un código tal que es conforme con ambos estándares de C y C++

Universidad Zaragoza
1542

Introducción

C caería en...

- **Paradigma imperativo**: cada sentencia cambia directamente el estado del programa
- También es **estructurado**, dado que incorpora una estructura de programa lógica
- También es **procedural**, dado que permite programación modular (esto es, definir funciones e invocarlas)

Universidad Zaragoza

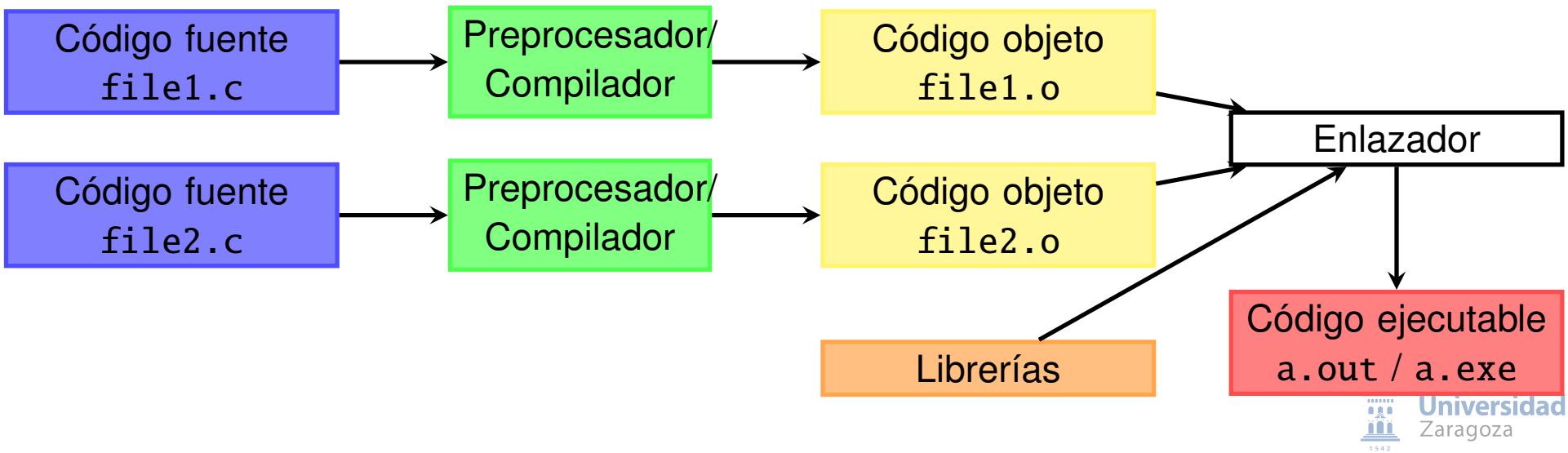
Outline

Universidad Zaragoza
1542

Compilación

Compilación separada

- **Uno (o más) ficheros de código fuente**, acabando en extensión `.c`
- **Fase de preprocesado (más luego) y fase de compilado**, obteniendo un fichero objeto (extensión del fichero `.o`)
- **Fase de enlazado**: **los ficheros objeto se unen juntos para conformar un único programa ejecutable** (extensión de fichero `.out` or `.exe`, dependiendo de tu SO!)



La compilación separada es útil!

Compilación más rápida

- Cuando se modifica un programa, típicamente se edita únicamente unas pocas líneas en algunos ficheros
- **Gracias a la compilación separada, sólo aquellos ficheros que se hayan editado son los que se recompilan cuando se vuelve a crear el ejecutable**
- **Ahorra tiempo** con programas muy grandes

Compilación

Cómo compilar un programa en C

- **Compilar y enlazar un programa C que está en un único código fuente**:

  ```
  gcc program.c
  ```

- Por defecto, `a.out` (o `a.exe`) es el nombre del fichero ejecutable generado. **Se puede cambiar con la opción** `-o` **del compilador**:

  ```
  gcc program.c -o myfancyname.out
  ```

- **Compilar y enlazar un programa C con múltiples códigos fuentes**:

  ```
  gcc program.c extras.c moreextras.c
  ```

- **Para compilar sin enlazar, hay que usar la opción** `-c`:

  ```
  gcc -c program.c
  ```

Compilación

Fase de preprocesado

- **Transforma el código fuente antes de que se produzca la compilación**, siguiendo ciertas directivas que se le da
- Una **directiva** es **una línea de código comenzando con el símbolo # symbol**
- **Allows to define macros (piece of text) that are copy/pasted in the place where the directive is defined**
- Algunos ejemplos:
  - `#define`: define una macro (útil también para definir constantes)
  - `#undef`: elimina una definición de macro
  - `#include`: inserta el texto de un fichero externo (útil para el uso de librerías)
  - `#if`: condicional basado en el valor de la expresión
  - `#ifdef`: condicional basado en si un macro está definida
  - `#ifndef`: condicional basado en si un macro no está definida
  - `#else`: alternativa
  - `#elif`: alternativa condicional

Universidad Zaragoza
1542

Compilación

Tipos de códigos fuente

- **Ficheros de códigos fuente**
  - Extensión del fichero: `.c`
  - Contiene el código real que se va a ejecutar
- **Ficheros de cabeceras**
  - Extensión del fichero: `.h`
  - Exporta definiciones de interfaz, prototipos de funciones, macros, y otras declaraciones comunes
  - Normalmente, no debe de contener nada de código fuente

En tus programas C, es casi seguro que usarás código fuente proveniente de terceros (esto es, tendrás que usar sus ficheros de cabeceras). **No debes de preocuparte de los detalles de implementación**, si no únicamente saber cómo has de llamar a las funciones que necesites: estos prototipos (o cabeceras) de las funciones es lo que se define en los ficheros de cabeceras

Outline

Universidad Zaragoza
1542

Esqueleto de un programa en C – Hello world!

```c
/*
    This is a multi-line comment.
 */
#include <stdio.h>

// And this is a single-line comment!
// You can put as many as you want :)

int main()
{
    printf("Hello world!\n");
    return 0;
}
```

- **Comentarios**: multilínea (empiezan `/*` y acaban con `*/`) o monolínea (`//`)
- `#include <stdio.h>` incluye (copia/pega) en tu fichero fuente el código del fichero `stdio.h` – librería de sistema que contiene funciones de E/S de consola y ficheros
- `int main()` define una función que devuelve un entero, llamada `main`
  - **Esta es la PRIMERA función que se ejecuta en la aplicación**
  - Por convención, se devuelve 0 cuando la ejecución es exitosa o $\neq 0$ cuando sucede un error
- **Las cadenas son secuencias de caracteres entre comillas dobles "** (más luego)

Outline

Universidad Zaragoza
1542

Tipos básicos y operadores

Tipos de datos básicos

- Tipos: `char`, `int`, `float` `double`
- Calificadores (se anteponen a los tipos): `short`, `long`, `unsigned`, `signed`, `const`
- Literales: `'c'`, `12`, `3.4`, `"A string"`

Tipos básicos y operadores

Tipos de datos básicos

- Tipos: `char`, `int`, `float` `double`
- Calificadores (se anteponen a los tipos): `short`, `long`, `unsigned`, `signed`, `const`
- Literales: `'c'`, `12`, `3.4`, `"A string"`

Tipos enumerados

- `enum WeekDay_t {Mon, Tue, Wed, Thu, Fri};`
- `enum WeekendDay_t {Sat = 0, Sun = 4};`

Tipos básicos y operadores

Tipos de datos básicos

- Tipos: `char`, `int`, `float` `double`
- Calificadores (se anteponen a los tipos): `short`, `long`, `unsigned`, `signed`, `const`
- Literales: `'c'`, `12`, `3.4`, `"A string"`

Tipos enumerados

- `enum WeekDay_t {Mon, Tue, Wed, Thu, Fri};`
- `enum WeekendDay_t {Sat = 0, Sun = 4};`

Operadores aritméticos

- `+`, `-`, `*`, `/`, `%`
- Prefijo `++` / `--`: incremento/decremento antes de usar la variable
- Postfijo `++` / `--`: incremento/decrement después de usar la variable

Tipos básicos y operadores

Tipos de datos básicos

- Tipos: `char`, `int`, `float` `double`
- Calificadores (se anteponen a los tipos): `short`, `long`, `unsigned`, `signed`, `const`
- Literales: `'c'`, `12`, `3.4`, `"A string"`

Tipos enumerados

- `enum WeekDay_t {Mon, Tue, Wed, Thu, Fri};`
- `enum WeekendDay_t {Sat = 0, Sun = 4};`

Operadores aritméticos

- `+`, `-`, `*`, `/`, `%`
- Prefijo `++` / `--`: incremento/decremento antes de usar la variable
- Postfijo `++` / `--`: incremento/decrement después de usar la variable

Otros operadores

- Relacionales y lógicos: `<`, `>`, `<=`, `>=`, `==`, `!=`, `&&`, `||`
- A nivel de bit: `&`, `|`, `^` (xor), `<<`, `>>`, `~` (negación)

Universidad Zaragoza

Tipos básicos y operadores

Precedencia de operadores en C

Precedence	Operator	Description	Associativity
1	++ -- () [] . -> (*type*){*list*}	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(C99)	Left-to-right
2	++ -- + - ! ~ (*type*) * & sizeof _Alignof	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of[note 1] Alignment requirement(C11)	Right-to-left
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13[note 2]	?:	Ternary conditional[note 3]	Right-to-Left
14	= += -= *= /= %= <<= >>= &= ^= |=	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Fuente: `https://en.cppreference.com/w/c/language/operator_precedence`

Outline

Universidad Zaragoza
1542

Definición y asignación de variables

Definición

```
char c;
int x, y;
float f;
short int z;
```

- **Primero se pone el calificador (si alguno) y luego el tipo de la variable. Por último, el identificador**
- Sobre el identificador:
  - **Secuencia de números, letras y _**
  - **No puede ser una palabra reservada ni comenzar por número**
- **Podemos definir más de una variable del mismo tipo**, separando identificadores por coma

Universidad Zaragoza

Definición y asignación de variables

Asignación

```
char c = 'a';
int x, y = 5;
```

- **El valor se asigna usando el operador de asignación** = (existen algunas variantes, ver diapositiva de precedencia de operadores)
- **Se pueden inicializar las variables cuando se definen. Cuando no se inicializan, su valor inicial está indefinido (desconocido) – depende del compilador**
- Desde luego, se puede asignar valor a una variable en cualquier otro momento en el código
- En el ejemplo anterior, `y` toma el valor 5 mientras que `x` está indefinida

Universidad Zaragoza

Outline

Universidad Zaragoza

Tipos de datos complejos

Punteros

- **Un puntero no es más que una dirección de memoria**
- Para definir un puntero **usamos el operador ***. Por ejemplo,
  `int value, *x;`
  - `value` es una variable de tipo entero
  - `int *x` ha de leerse como "puntero a entero", y significa que la dirección almacenada en esta variable ha de ser interpretada como que contiene un entero

```
int value = 3;
int *x;

x = &value;    // ahora, x apunta a la dirección de la variable value
*x = 4;        // ahora, value es 4 (se ha modificado su valor!)
```

Tipos de datos complejos

Punteros

- **Para cualquier tipo T, se puede definir un puntero a T**
  - Los punteros pueden referenciar también a una función (o a un objeto en C++)
  - **Valor del puntero**: **dirección del objeto o función correspondiente**
- **Operadores puntero**:
  - **Operador * de-referencia a un puntero**
  - **Operador & crea un puntero** (referencia a)

```
int i = 3; int *j = &i;
*j = 4; printf("i = %d\n", i); // imprime i = 4

int myfunc (int arg);
int (*fptr)(int) = myfunc;
i = fptr(4); //  lo mismo que llamar a myfunc(4);
```

Tipos de datos complejos

Punteros

- **Para cualquier tipo T, se puede definir un puntero a T**
  - Los punteros pueden referenciar también a una función (o a un objeto en C++)
  - **Valor del puntero**: **dirección del objeto o función correspondiente**
- **Operadores puntero**:
  - **Operador * de-referencia a un puntero**
  - **Operador & crea un puntero** (referencia a)

```
int i = 3; int *j = &i;
*j = 4; printf("i = %d\n", i); // imprime i = 4

int myfunc (int arg);
int (*fptr)(int) = myfunc;
i = fptr(4); //  lo mismo que llamar a myfunc(4);
```

- **Puntero genérico**: `void *`
  - **No puede ser de-referenciado o usado en operaciones aritméticas – útil para reducir errores de programación**
- **Puntero nulo**: `NULL` or `0`
  - Siempre (repito, SIEMPRE) conviene inicializar punteros a `NULL`

Universidad Zaragoza

Tipos de datos complejos

Arrays (vectores)

- Una variable declarada como array **representa una región contigua en memoria donde se almacenan los elementos del array**
- **Hay que especificar el tamaño del array entre corchetes**. Este operando también se usa para acceder a un determinado elemento en el vector

```
int x[10]; // define un array de 10 elementos (de 4 bytes cada uno)
x[2] = 0;  // pone un 0 en el tercer elemento del array
```

- **El primer índice del vector es 0**

Universidad Zaragoza

Tipos de datos complejos

Arrays (vectores)

- Una variable declarada como array **representa una región contigua en memoria donde se almacenan los elementos del array**
- **Hay que especificar el tamaño del array entre corchetes**. Este operando también se usa para acceder a un determinado elemento en el vector

```
int x[10]; // define un array de 10 elementos (de 4 bytes cada uno)
x[2] = 0;  // pone un 0 en el tercer elemento del array
```

- **El primer índice del vector es 0**
- **Un identificador array es equivalente a un puntero que referencia al primer elemento del array**:

```
int x[5], *ptr;
ptr = &x[0]; // equivalente a ptr = x;
```

Universidad Zaragoza

Tipos de datos complejos

Arrays (vectores)

- Una variable declarada como array **representa una región contigua en memoria donde se almacenan los elementos del array**
- **Hay que especificar el tamaño del array entre corchetes**. Este operando también se usa para acceder a un determinado elemento en el vector

```
int x[10]; // define un array de 10 elementos (de 4 bytes cada uno)
x[2] = 0;  // pone un 0 en el tercer elemento del array
```

- **El primer índice del vector es 0**
- **Un identificador array es equivalente a un puntero que referencia al primer elemento del array**:

```
int x[5], *ptr;
ptr = &x[0]; // equivalente a ptr = x;
```

- **Aritmética de punteros y un array**:
  - `x[2]` es lo mismo que `*(x + 2)`, **el compilador asume que quieres decir 2 elementos pasado el elemento `x`** (esto es, ese operador suma incrementa el tamaño del elemento cada vez!)
- **Una cadena en C se representa como un array de caracteres**
  - En C, el byte nulo (`0x00`) indica el final de la cadena

Universidad Zaragoza 1542

Tipos de datos complejos

Registros

```
struct CartesianCoordinate{
        float x;
        float y;
};

// Definición de una variable de ese tipo
struct CartesianCoordinate myCoordinate;

// También puede definirse como un nuevo tipo para usarse posteriormen
typedef struct{
        float x;
        float y;
} CCoordinate_type;
// o
// typedef struct CartesianCoordinate CCoordinate_type;

CCoordinate_type otherCoordinate;
CCoordinate_type *pointer; // Variable puntero
```

- **Acceso a cada campo del registro mediante el operando .**: `myCoordinate.x = 5;`
- **Cuando se trabaja con punteros a registro, hay que usar el operador ->**: `pointer = &myCoordinate; pointer -> x = 3;`

Outline

Universidad Zaragoza
1542

Composición secuencial

- **C es un lenguaje imperativo**: cada sentencia cambia el estado global del programa
- **Las sentencias se separan con ;**
- **Las sentencias se ejecutan secuencialmente**
  - **La localización en el código determina el orden de ejecución**
  - **Una sentencia comienza la ejecución cuando la anterior acaba**
  - **Cada efecto de una sentencia se realiza justo después de su ejecución**

Universidad Zaragoza
1542

Composición secuencial

- Recuerda que **hay operadores que pueden tener efecto antes/después de la ejecución de la sentencia** (operadores pre/post incremento o decremento):

```
int x = 3, result;
result = x++ + 2; // result es 5 y x es 4
// x se incrementa DESPUÉS de realizar la suma x + 2

result = ++x + 2; // result es 6 y x es 4
// x se incrementa ANTES de realizar la suma x + 2
```

Composición secuencial

- Recuerda que **hay operadores que pueden tener efecto antes/después de la ejecución de la sentencia** (operadores pre/post incremento o decremento):

```
int x = 3, result;
result = x++ + 2; // result es 5 y x es 4
// x se incrementa DESPUÉS de realizar la suma x + 2

result = ++x + 2; // result es 6 y x es 4
// x se incrementa ANTES de realizar la suma x + 2
```

- **Las llaves se pueden usar para definir bloques de sentencias**
  - **Definen el ámbito/visibilidad de las variables**: las variables definidas dentro del bloque son **desconocidas fuera de ese bloque**

```
char x, y;
{
   // aquí dentro, puedo definir otra vez x e y, incluso con otro tipo
   // si las redefino, no podré usar las variables x, y previas
   int x, y;
   x = 3;
   y = 4 + y;
}
// y aquí puedo usar las variables carácter x, y otra vez
// (las variables enteras x, y se desconocen aquí)
```

Composición condicional

```
if(condition)
{
   // Bloque de sentencias a ejecutar si condition es ci
}else{ // opcional
  // Bloque de sentencias a ejecutar si condition es fal
}
```

- **Condición que se va a evaluar entre paréntesis**
  - En C, todo distinto de 0 es cierto, mientras que 0 es falso
  - Esto es, podríamos tener algo como `if(n)` y asumir que se evaluará a cierto si $n \neq 0$, falso en otro caso
- **Las sentencias a ejecutar cuando la condición es cierta se ponen entre llaves, seguidas de la condición**
  - **Estas llaves son opcionales si se trata de una sentencia única**
- **Las sentencias a ejecutar cuando la condición es falsa se ponen entre llaves, seguidas de la palabra clave `else`**
  - Esta parte es **opcional**
  - Como antes (y como siempre desde ahora), **estas llaves son opcionales si se trata de una sentencia única**

Composición condicional

```
switch(expression)
{
    case 1: // bloque de sentencias a ejecutar si variable = 1
            break; // si esto no aparece, la ejecución continúa secuencialmente
    case 2:
    case 3: // bloque de instrucciones a ejecutar si variable = 2 o variable = 3
            break;
    // puedes poner un número de casos arbitrario
    default: // opcional, se puede especificar para recoger la no coincidencia
             // este siempre es el último caso, no es necesario el break
}
```

- **Útil cuando se tienen sentencias de comparación en la condición del `if`**
- **`expression` debe de ser un tipo enumerable**

Composición iterativa

```
while(condition){
    // Sentencias a ejecutar cuando la condición es cierta
}
```

- **Si `condition` se evalúa a cierto, el bloque de sentencias entre llaves se ejecutará**
  - **Recuerda que si es una única sentencia, las llaves son opcionales**
  - Al finalizar la ejecución de la última sentencia del bloque, se reevalúa `condition`
  - Recuerda también la regla de las condiciones de C: 0 es falso, todo lo demás es verdad
- **Cuidado: asegúrate que la condición del bucle está afectada (de alguna manera) por el bloque de sentencias**
  - **Si no, tendrás un bucle infinito!** (lo que normalmente no es muy práctico)

Universidad Zaragoza

Composición iterativa

```
do{
    // Bloque de sentencias a ejecutar mientras que conditi
}while(condition);
```

- **La condición se evalúa al final!**
- **Cuando es cierta, la ejecución del bloque de sentencias se repite. Si no, el bucle finaliza**
- **Diferencia con el bucle `while`: `do-while` ejecuta el bloque de sentencias por lo menos una vez** (la condición se evalúa al final en vez de al principio)

Universidad Zaragoza

Composición iterativa

```
for(initialization ; loop condition ; statement step){
      // Bloque de sentencias a ejecutar en el bucle
}
```

- **Útil para iterar cuando el número de iteraciones está acotado**
- **Tres elementos entre los paréntesis, separados por ';'**:
  - **Initialization**: aquí podemos declarar una (o más) variables de iteración e iniciarlas debidamente. Puede ser vacía
  - **Loop condition**: condición a evaluar. Si es cierta, el bloque de sentencias (entre llaves) se ejecuta. Puede ser vacía también
  - **Statement step**: se ejecuta después de la ejecución del bloque de sentencias. Puede ser vacía también

Composición iterativa

```
for(initialization ; loop condition ; statement step){
      // Bloque de sentencias a ejecutar en el bucle
}
```

- **Útil para iterar cuando el número de iteraciones está acotado**
- **Tres elementos entre los paréntesis, separados por ';'**:
  - **Initialization**: aquí podemos declarar una (o más) variables de iteración e iniciarlas debidamente. Puede ser vacía
  - **Loop condition**: condición a evaluar. Si es cierta, el bloque de sentencias (entre llaves) se ejecuta. Puede ser vacía también
  - **Statement step**: se ejecuta después de la ejecución del bloque de sentencias. Puede ser vacía también
- Como una buena práctica de programación, **la variable de iteración no debería de modificarse dentro del bloque de sentencias (únicamente en el statement step)**

```
for(int i = 0; i <= 10; i++){
    // hacer otras cosas aquí
}
```

```
for(;;){
   // Cuántas iteraciones hace aquí? }
```

```
for(int i = 0, j = 3;
        i <= 10 && (j % 2) == 0;
        i++, j *= 2){
    // hacer más cosas aquí
```

Outline

Universidad Zaragoza
1542

Macros

- **Útil para definir constantes o funciones** a través de la directiva **`#define`**
- Recuerda: **el pre-procesador insertará el código allí donde se use la macro**

Macros

- **Útil para definir constantes o funciones** a través de la directiva **`#define`**
- Recuerda: **el pre-procesador insertará el código allí donde se use la macro**

```
#define mymult(a,b) a*b
```

```
#define mymult(a,b) (a)*(b)
```

Considera el código `k = mymult(i-1, j+5)`. Qué ocurre entonces?

Macros

- **Útil para definir constantes o funciones** a través de la directiva **`#define`**
- Recuerda: **el pre-procesador insertará el código allí donde se use la macro**

```
#define mymult(a,b) a*b
```

```
#define mymult(a,b) (a)*(b)
```

Considera el código `k = mymult(i-1, j+5)`. Qué ocurre entonces?

```
k = i - 1 * j + 5;
```

```
k = (i - 1)*(j + 5);
```

Universidad Zaragoza

Macros

- **Útil para definir constantes o funciones** a través de la directiva **#define**
- Recuerda: **el pre-procesador insertará el código allí donde se use la macro**

```
#define mymult(a,b) a*b
```

```
#define mymult(a,b) (a)*(b)
```

Considera el código `k = mymult(i-1, j+5)`. Qué ocurre entonces?

```
k = i - 1 * j + 5;
```

```
k = (i - 1)*(j + 5);
```

- **Ojo con los efectos secundarios!**

```
#define mysq(a) (a)*(a)

k = mysq(i++)
```

Macros

- **Útil para definir constantes o funciones** a través de la directiva **#define**
- Recuerda: **el pre-procesador insertará el código allí donde se use la macro**

```
#define mymult(a,b) a*b
```

```
#define mymult(a,b) (a)*(b)
```

Considera el código `k = mymult(i-1, j+5)`. Qué ocurre entonces?

```
k = i - 1 * j + 5;
```

```
k = (i - 1)*(j + 5);
```

- **Ojo con los efectos secundarios!**

```
#define mysq(a) (a)*(a)

k = mysq(i++)
```

- **El pre-procesado del post hará `k = (i++)*(i++)`**
- Alternativa: **funciones inline**

```
inline int mysq(int a) {return a*a};
```

Universidad Zaragoza

Funciones

- Buenas prácticas: **siempre escribir código de manera modular**
  - **Diseña un algoritmo que resuelva tu problema**
  - **Define funciones para resolver tu problema paso a paso**
  - **Cada función resuelve un paso del problema!**
  - **Más fácil para depurar e identificar errores**

Universidad Zaragoza
1542

Funciones

- Buenas prácticas: **siempre escribir código de manera modular**
  - **Diseña un algoritmo que resuelva tu problema**
  - **Define funciones para resolver tu problema paso a paso**
  - **Cada función resuelve un paso del problema!**
  - **Más fácil para depurar e identificar errores**
- **Ámbito y visibilidad de las sentencias**: cada sentencia sólo *conoce* aquello que se ha definido por encima (no por debajo) de su ámbito actual. Esto aplica tanto a las funciones como a las variables globales o locales
  - **NUNCA uses variables globales!!** (a no ser que no te quede otra opción)

Universidad Zaragoza

Funciones

- Buenas prácticas: **siempre escribir código de manera modular**
  - **Diseña un algoritmo que resuelva tu problema**
  - **Define funciones para resolver tu problema paso a paso**
  - **Cada función resuelve un paso del problema!**
  - **Más fácil para depurar e identificar errores**
- **Ámbito y visibilidad de las sentencias**: cada sentencia sólo *conoce* aquello que se ha definido por encima (no por debajo) de su ámbito actual. Esto aplica tanto a las funciones como a las variables globales o locales
  - **NUNCA uses variables globales!!** (a no ser que no te quede otra opción)
- **Consejo de programación: usar prototipos de funciones**
  - Declarados en un fichero de cabecera aparte (o justo encima del `main`)
  - El código de las funciones en un fichero fuente aparte (or justo debajo de `main`)

Universidad Zaragoza

Funciones

- Buenas prácticas: **siempre escribir código de manera modular**
  - **Diseña un algoritmo que resuelva tu problema**
  - **Define funciones para resolver tu problema paso a paso**
  - **Cada función resuelve un paso del problema!**
  - **Más fácil para depurar e identificar errores**
- **Ámbito y visibilidad de las sentencias**: cada sentencia sólo *conoce* aquello que se ha definido por encima (no por debajo) de su ámbito actual. Esto aplica tanto a las funciones como a las variables globales o locales
  - **NUNCA uses variables globales!!** (a no ser que no te quede otra opción)
- **Consejo de programación: usar prototipos de funciones**
  - Declarados en un fichero de cabecera aparte (o justo encima del `main`)
  - El código de las funciones en un fichero fuente aparte (or justo debajo de `main`)
- **Los parámetros de una función se pasan por valor**: se hace una copia del valor del parámetro
  - Si quieres que la función modifique el valor del parámetro y que esa modificación sea visible fuera del ámbito de la función, hay que pasar el parámetro por referencia (como puntero)

Universidad Zaragoza
1542

Funciones

```
int my_function(int p1, int p2); // prototipo de la función

int main(){
    // código aquí para hacer algo
    my_function(a, b);
    // cuando se ejecuta, p1 coge el valor de a
    // y p2 coje el valor de b
 }

 // cuerpo de la función
  int my_function(int p1, int p2){
              return (p1 + p2)*(p1 - p2);
  }
```

- **Primero hay que especificar el tipo que se devuelve. Después, un identificador de la función**
- **Entre paréntesis, el conjunto de parámetros esperados**
  - Separados por coma, **primero se indica el tipo del parámetro y luego un identificador**
- **Cuando se llama a la función, se copian de izquierda a derecha**
  - **Debe de coincidir el número de parámetros**
  - **Debe de coincidir el tipo de cada parámetro** (dependiendo del tipo de dato puede aparecer una advertencia o un error en compilación)

Outline

Universidad Zaragoza
1542

Argumentos de línea de comandos

```c
int main(int argc, char *argv[]){
   printf("There is %d arguments\n", argc);
   for(int i = 0; i < argc; i++)
      printf("Argument %d: %s\n", i, argv[i]);

   return 0;
}
```

```
$ gcc example.c
$ ./a.out this is a test
There is 5 arguments
Argument 0: ./a.out
Argument 1: this
Argument 2: is
Argument 3: a
Argument 4: test
```

- **Argumentos de la función `main`: `int argc, char *argv[]`**
- `argc` **indica el número de argumentos pasados por la línea de comandos**
  - Por lo menos siempre existe uno: el propio ejecutable!
- `*argv[]` **es un vector de vectores de caracteres** (i.e., un vector de cadenas). En cada posición tenemos un argumento – observa el ejemplo!

Outline

Universidad Zaragoza
1542

Otras librerías y funciones interesantes

- `stdio.h`: E/S de ficheros y consola (`perror`, `printf`, `open`, `close`, `read`, `write`, `scanf`, etc.)
- `stdlib.h`: utilidades comunes (`malloc`, `calloc`, `strtol`, `atoi`, etc.)
- `string.h`: manipulación de cadenas y de bytes (`strlen`, `strcpy`, `strcat`, `memcpy`, `memset`, etc.)
- `ctype.h`: funciones para tipos carácter (`isalnum`, `isprint`, `isupport`, `tolower`, etc.)
- `errno.h`: define `errno`, usado para gestión de errores
- `math.h`: funciones matemáticas (`ceil`, `exp`, `floor`, `sqrt`, etc.)
- `signal.h`: funciones de gestión de señales (`raise`, `signal`, etc.)
- `stdint.h`: funciones auxiliares de enteros (`intN_t`, `uintN_t`, etc.)
- `time.h`: funciones relacionadas con tiempo (`asctime`, `clock`, `time_t`, etc.)

Outline

Universidad Zaragoza
1542

Ejercicios

Escribe un programa que, dado un número entero por el usuario, diga si es par o impar.

Ejemplo de ejecución:

```
Number? 2
The number 2 is even
```

Espera de 5 a 10 minutos para continuar (haz el ejercicio!)

Universidad Zaragoza

Ejercicios

```c
#include <stdio.h>

int main(){
    int n;

    printf("Number? ");
    scanf("%d", &n);

    printf("The number %d is ", n);
    if(n % 2) // equivalent to (n % 2) == 1
        puts("odd.");
    else
        puts("even.");

    return 0;
}
```

Universidad Zaragoza

Ejercicios

Escribe un programa en C que, dado un número entero por el usuario, lo escriba en binario. Usa programación modular.

Ejemplo:

```
Number? 47
47 in binary form is 101111
```

Espera de 5 a 10 minutos para continuar (haz el ejercicio!)

Universidad Zaragoza

Ejercicios

```c
#include <stdio.h>

long to_binary(int d);

int main()
{
    long binary;
    int decimal;

    printf("Number? ");
    scanf("%d",&decimal);
    binary = to_binary(decimal);
    printf("%d in binary form is %ld\n", decimal, binary);

    return 0;
}

long to_binary(int d)
{
    long bno = 0, remainder, f = 1;
    while(d != 0)
    {
        remainder = d % 2;
        bno = bno + remainder*f;
        f = f*10;
        d = d/2;
    }
    return bno;
}
```

Fundamentos del Software: introducción a la programación verificada

Programación segura

Introducción al lenguaje C

Roberto Blanco[†] & Ricardo J. Rodríguez[‡]

MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

Universidad Zaragoza

[†]Max Plank Institute for Security and Privacy
Bochum, Alemania

[‡]Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza (España)

Septiembre 2020

Universidad de Zaragoza