Programación Avanzada y Estructuras de Datos

1. Introducción a C++

Víctor M. Sánchez Cartagena

Grado en Ingeniería en Inteligencia Artificial
Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

25 de septiembre de 2024

Índice

Características del lenguaje C++ (1)

- Lenguaje compilado (Python es interpretado)
- Lenguaje de tipo imperativo y orientado a objetos (igual que Python)
  - Lenguaje imperativo: los programas son conjuntos de instrucciones que indican al computador cómo realizar una tarea
  - Lenguaje declarativo: los programas indican cuál es el resultado deseado de la tarea
- Lenguaje de **tipado fuerte** (Python es un lenguaje no tipado)

Características del lenguaje C++ (y 2)

- Desarrollado en 1980 por Bjarne Stroustroup en los laboratorios At&T
- Extensión orientada a objetos del lenguaje C: creado por Dennis Ritchie entre los años 1970-73
- Código fuente escrito en C puede compilarse como C++ (normalmente): se mantienen algunos inconvenientes del lenguaje C
- Usaremos principalmente C++11

Estructura básica de un programa en C++

```
#include <ficheros de cabecera estándar>
...
#include "ficheros de cabecera propios"
...
using namespace std; // Permite usar cout, string...
...
const ... // Constantes
...
// Variables globales: ¡¡No la usaremos en PAED!!
...
funciones ... // Declaración de funciones
...
int main() { // Función principal
...
}
```

Mi primer programa

```cpp
#include <iostream>

using namespace std;

int main() { // Función principal
    cout << "¡Hola, mundo!" << endl;
    return 0;
}
```

Compilación

- El compilador permite convertir un código fuente en un código objeto
- Usaremos el compilador GNU C++ para transformar el código fuente en C++ en un programa ejecutable
- El compilador de GNU se invoca con el programa `g++` y admite numerosos argumentos
  - `-Wall`: muestra todas as advertencias o *warnings*
  - `-g`: añade información para el depurador
  - `-o`: para indicar el nombre del ejecutable
  - `-std=c++11`: para usar el estándar C++11
  - `--version`: muestra la versión instalada

Ejemplo de uso (terminal)

```
g++ -std=c++11 -Wall -g prog.cc -o prog
```

Índice

Elementos del lenguaje

- Los identificadores son nombres de variables, constantes y funciones
- Han de comenzar por letra minúscula, mayúscula o guión bajo
- C++ distingue entre letras mayúsculas y minúsculas
- En C++ hay palabras reservadas que no se pueden utilizar como nombres definidos por el usuario

```
if while for do int friend long auto public union
...
```

  - Si las usamos como identificadores se producirá un error de compilación difícil de interpretar

Variables: definición y tipos

- Las variables permiten almacenar diferentes tipos de datos
- **Se deben declarar explícitamente**, indicando el **tipo**
- Tipos básicos o primitivos:

Tipo	Tamaño en bits CPU 32 bits	Uso
`int`	32	Número entero: `-273`, `1066`
`char`	8	Carácter: `'a'`, `'z'`, `'9'`
`float`	32	Número en coma flotante: `273.15`, `3.14`
`double`	64	Ídem de doble precisión
`bool`	8	Booleano: `true` o `false`
`void`	-	Funciones que no devuelven nada

- Se puede usar `unsigned` con int para tener solo números

Ejemplos

```
a=2
b=3
c=a+b
print(c)
```

```
int a,b,c;
a=2; b=3;
c=a+b;
cout << c << endl;
```

- ¿Qué pasa si empleamos una variable declarada sin haberle asignado valor?

```
int c;
cout << c << endl;
```

Inicialización de variables

- Las variables no inicializadas pueden tener *cualquier* valor, según del compilador, la máquina, y el estado de la memoria
- Errores en tiempo de ejecución y suspensos
- Es recomendable inicializar las variables a la vez que se declaran:

```
int c=0;
cout << c << endl;
```

- `auto`: el tipo se determina automáticamente a partir del valor (como en Python)

```
auto c=0;
cout << c << endl;
```

Ámbito

- Ámbito de una variable: parte del programa donde se puede acceder a esa variable
- Siempre dentro del bloque entre llaves donde se declaró:

```
int i=0;
int numCajas=0;
if(i<10){
    // numCajas se puede usar aquí
    int numCajas=100; // Mismo nombre pero otro ámbito
    cout << numCajas << endl; // Imprime 100
}
cout << numCajas << endl; // ¿Qué imprime? ¿0 o 100?
```

Variables locales y globales

- Variable local a una función: se declara dentro de la función y no es accesible desde fuera

```
void imprimir(){
  int i=3,j=5; // Al principio de la función
  cout << i << j << endl;
  ...
  int k=7; // En un punto intermedio
  cout << k << endl;
}
```

- Variable global: se declara fuera de cualquier función y es accesible desde cualquier parte del programa
  - Son muy peligrosas y dan lugar a programas difíciles de depurar
  - Rompen el encapsulamiento y dificultan la reutilización del código
  - No las usaremos en Programación Avanzada y Estructuras de Datos
  - Se puede conseguir una funcionalidad parecida con atributos estáticos o de clase en programación orientada a objetos

Constantes

- Tienen un valor fijo (no puede ser cambiado) durante toda la ejecución del programa
- Se declaran anteponiendo const al tipo de dato
- Son útiles para definir valores que se usen en múltiples puntos de un programa y que no cambien de valor

```
const int MAXALUMNOS=600;
const double PI=3.141592;
```

Conversión de tipos

- Implícita: la hace el compilador de manera automática, normalmente cuando no hay pérdida de información al convertir

Tipos	Ejemplo
char→int	`int a='A'+2; // a vale 67`
int→float	`float pi=1+2.141592;`
float→double	`double piMedios=pi/2.0;`
bool→int	`int b=true; // b vale 1`
int→bool	`bool c=77212; // c vale true`

- Explícita o *cast*: se pone el tipo deseado entre paréntesis

```
char laC=(char)('A'+2); // laC vale 'C'
int pEnteraPi=(int)pi; // pEnteraPi vale 3
```

Índice

Expresiones aritméticas

- Las expresiones aritméticas están formadas por operandos (`int`, `float` y `double`) y operadores aritméticos (`+ - * /`):

```
float i=4*5.7+3; // i vale 25.8
```

- Si aparece un operando de tipo `char` o `bool` se convierte a entero implícitamente:

```
int i=2+'a'; // i vale 99
```

- Si dividimos dos enteros el resultado es un entero:

```
cout << 7/2; // La salida es 3
```

- Si queremos que el resultado de la división entera sea un valor real hay que hacer un cast a `float` o `double`:

```
cout << (float)7/2; // La salida es 3.5
cout << (float)(7/2); // La salida es ...
```

Expresiones aritméticas

- El operador % devuelve el resto de la división entera:

```
cout << 30%7; // La salida es 2
```

- Precedencia de operadores:
  1. ++ (incremento) −− (decremento) ! (negación) − (menos unario)
  2. ⋆ (multiplicación) / (división) % (módulo)
  3. + (suma) − (resta)
- En caso de duda usar paréntesis:

```
cout << 2+3*4; // La salida es 14
// * tiene más precedencia que +
cout << 2+(3*4); // La salida es 14
cout << (2+3)*4; // La salida es ..
cout << 2+3*4%2-1/2; // La salida es ..
```

Operadores de incremento y decremento

- Los operadores ++ y −− se usan para incrementar o decrementar el valor de una variable entera en una unidad
- Preincremento/predecremento: se incrementa/decrementa antes de tomar su valor

```
int i=3,j=3;
int k=++i; // k vale 4, i vale 4
int l=--j; // l vale 2, j vale 2
```

- Postincremento/postdecremento: se incrementa/decrementa después de tomar su valor

```
int i=3,j=3;
int k=i++; // k vale 3, i vale 4
int l=j--; // l vale 3, j vale 2
```

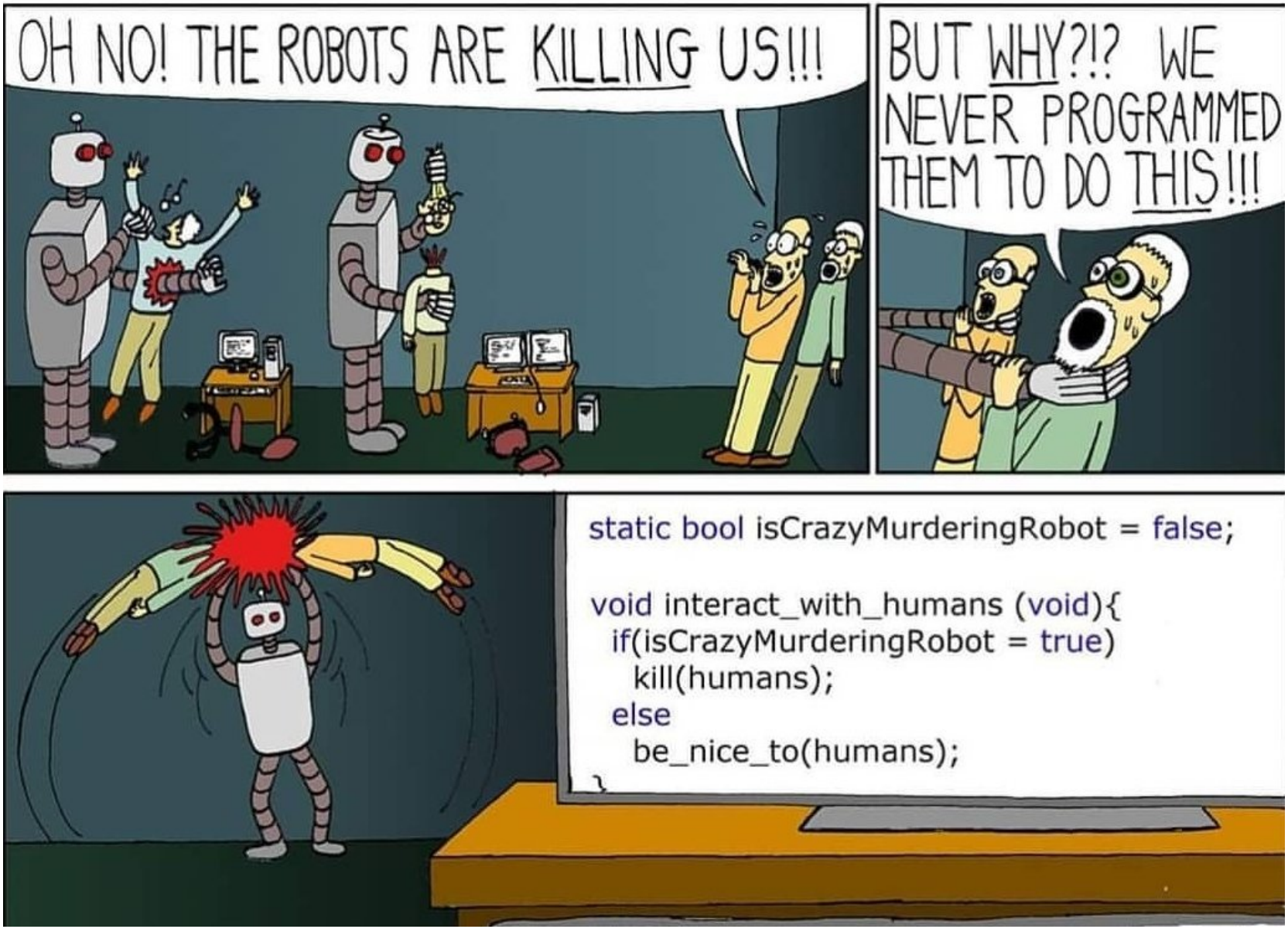
Expresiones relacionales

- Las expresiones relacionales permiten realizar comparaciones entre valores
- Operadores: == (igual), != (distinto), >= (mayor o igual), > (mayor estricto), <= (menor o igual) y < (menor estricto)
- Si los tipos de los operandos no son iguales se convierten (implícitamente) al tipo más general:

```
if(2<3.4){...} // Se transforma en: if(2.0<3.4)
```

- El resultado es 0 si la comparación es falsa y distinto de 0 si es cierta
- **Las expresiones relacionales tienen menor precedencia que las aritméticas**

Expresiones relacionales



Expresiones lógicas

- Las expresiones lógicas permiten relacionar valores booleanos y obtener un nuevo valor booleano
- Operadores: `!` (negación), `&&` (y lógico) y `||` (o lógico)
- Precedencia: `!` > aritméticas > relacionales > `&&` > `||`

```
if(a || b && c){...} // Equivale a: if(a || (b && c))
```

- Evaluación en cortocircuito:
  - Si el operando izquierdo de `&&` es falso, el operando derecho no se evalúa (`false && loquesea` es siempre `false`)
  - Si el operando izquierdo de `||` es cierto, el operando derecho no se evalúa (`true || loquesea` es siempre `true`)

Entrada y salida

- Salida por pantalla con `cout`:

```
int i=7;
cout << i << endl; // Muestra 7 y salto de línea (endl)
```

- Salida de error (por pantalla) con `cerr`:

```
int i=7;
cerr << i << endl; // Muestra 7 y salto de línea (endl)
```

- Entrada por teclado con `cin`:

```
int i;
cin >> i; // Guarda en i un número escrito por teclado
```

Índice

Condicional: `if`

- `if` evalúa una condición y toma un camino u otro
- A diferencia de Python, cada camino es un bloque entre llaves ({ y }), independientemente de cómo esté indentado el texto

```
int num=0;
cin >> num; // Leemos un número por teclado
if(num<5){ // Si num es menor que 5 ejecuta esta parte
  cout << "Camino que se toma si se cumple la condición";
  cout << "El número es menor que cinco";
}
else{ // Si no, ejecuta esta otra
  cout << "Camino que se toma si no se cumple la condición";
  cout << "El número es mayor o igual que cinco";
}
```

Condicional: `if`

- Si no hay llaves,
  sólo se considera que la primera instrucción está dentro del bloque

```
int num=0;
cin >> num; // Leemos un número por teclado
if(num<5) // Si num es menor que 5 sólo ejecuta la primera instrucció
  cout << "Camino que se toma si se cumple la condición";
  cout << "Esta línea se ejecuta siempre";
```

Iteraciones: `while` y `do-while`

- `while` ejecuta instrucciones mientras se cumpla la condición:

```
int i=10;
while(i>=0){
  cout << i << endl; // Hará una cuenta atrás del 10 a 0
  i--; // Si no decrementamos entra en un bucle infinito
}
```

- `do-while` ejecuta el cuerpo del bloque al menos una vez

```
int i=0;
do{ // Muestra el valor de i al menos una vez
  cout << "i vale: " << i << endl;
  i++;
}while(i<10);
```

Iteraciones: `for`

- `for` equivale a un `while`:

```
for(inicialización;condición;finalización){
// Instrucciones
}
```

```
inicialización;
while(condición){
  // Instrucciones
  finalización;
}
```

Iteraciones: `for`

- Tiene una sintaxis más elegante y compacta que while para iterar sobre valores numéricos
- Es una versión de bajo nivel de la estructura `for i in range(...)` de Python

```
for(int i=10;i>=0;i--){
cout << i << endl; // Hará una cuenta atrás del 10 al 0
}
```

Ejercicio

Implementa con un bucle `for`:

- Una cuenta adelante del 0 al 10
- Una cuenta adelante del 0 al 30, incrementando de 2 en 2
- Una cuenta atrás del 30 al 0, decrementando de 2 en 2

Control de flujo: `switch`

- `switch` permite ejecutar un fragmento de código u otro según el valor de una variable
- Es una especie de *`if` múltiple*
- Cuidado: cada bloque no va delimitado entre llaves, sino que acaba con la palabra clave `break`

```
char opcion;
cin >> opcion; // Leemos un carácter de teclado
switch(opcion){
  case 'a': cout << "Opción A" << endl;
  break; // Sale del switch
  case 'b': cout << "Opción B" << endl;
  break;
  case 'c': cout << "Opción C" << endl;
  break;
  default: cout << "Otra opción" << endl;
}
```

Control de flujo: `break`

- La instrucción `break` también se puede utilizar para salir de cualquier bucle

```
int vec[]={1,2,5,7,6,12,3,4,9};
int i=0;
// Código equivalente sin usar break
bool encontrado=false;
while(i<9 && !encontrado){
  if(vec[i]==6)
    encontrado=true;
  else
    i++;
}
```

Control de flujo: `break`

- La instrucción `break` también se puede utilizar para salir de cualquier bucle

```
int vec[]={1,2,5,7,6,12,3,4,9};
int i=0;
// Código equivalente sin usar break
bool encontrado=false;
while(i<9 && !encontrado){
  if(vec[i]==6)
    encontrado=true;
  else
    i++;
}
```

```
int vec[]={1,2,5,7,6,12,3,4,9};
int i=0;
// Salimos del bucle al encontrar el 6 en vec
while(i<9){
  if(vec[i]==6)
    break;
  else
    i++;
}
```

Ejercicios (I)

- Escribe un programa que pida el valor de los dos lados de un rectángulo y muestre el valor de su perímetro y de su área
- Escribe un programa que pida al usuario tres números enteros e imprima el valor del máximo
- Escribe un programa que pida al usuario cinco números enteros e imprima el valor del máximo
- Escribe un programa que pida al usuario un número y compruebe si es primo
- Escribe un programa que imprima todos los números primos en un rango determinado. El primer y último elemento del rango serán pedidos por teclado al usuario

Ejercicios (II)

- Escribe un programa que pida números por teclado hasta que se introduzca un 0, y al final imprima la media de los números introducidos
- Escribe un programa que imprima la siguiente estructura. El número de líneas se leerá de entrada estándar (en el ejemplo el valor es 4):

```
*
**
***
****
```

- Ahora la estructura debe ser esta (en el ejemplo el valor es 4):

```
   *
  **
 ***
****
```

Ejercicios (III)

- Escribe un programa que imprima la siguiente estructura. El número de líneas se leerá de entrada estándar (en el ejemplo el valor es 4):

```
****
 ***
  **
   *
```

- Realiza un programa que imprima el desglose en billetes y monedas de una cantidad entera de euros, utilizando el menor número posible de monedas y billetes. Existen billetes de 500, 200, 100, 50, 20, 10, y 5 euros, y monedas de 2 y 1 euro. El programa pedirá primero al usuario una cantidad, y después imprimirá su desglose, como muestra el siguiente ejemplo:

```
Introduce la cantidad: 434
2 de 200; 1 de 20; 1 de 10; 2 de 2;
```

Índice

Arrays

- Los *arrays* almacenan múltiples valores en una única variable en posiciones de memoria contiguas
- Estos valores pueden ser de cualquier tipo que deseemos, incluso tipos de datos propios (lo veremos más adelante)
- Al declarar un array hay que especificar su tamaño (cuántos elementos almacena) mediante constantes o variables

```
// Tamaño definido mediante constantes
const int MAXALUMNOS=100;
int alumnos[MAXALUMNOS]; // Puede almacenar 100 enteros
bool gruposLlenos[5]; // Puede almacenar 5 booleanos

// Tamaño definido mediante variables
int numElementos;
cin >> numElementos; // No sabemos qué número introducirá
float listaNotas[numElementos];
```

Arrays

- Cuando
se inicializa un vector al declararlo no hace falta indicar su tamaño:

```
int numbers[]={1,3,5,2,5,6,1,2};
```

- Si un vector tiene tamaño `TAM`, el primer elemento se halla en la posición `0` y el último en la posición `TAM-1`
- Asignación y acceso a valores mediante el operador `[]`:

```
const int TAM=10;
int vec[TAM];
vec[0]=7;
vec[TAM-1]=vec[TAM-2]+1; // vec[9]=vec[8]+1;
```

- Podemos (o no) tener un fallo en tiempo de ejecución si intentamos leer o escribir en un elemento fuera del vector:

```
int vec[5];
vec[5]=7; // Puede haber fallo en tiempo de ejecución
// El último elemento válido está en vec[4]
```

Matrices

- Una matriz es un vector cuyas posiciones son, cada una de ellas, otro vector
- Hay que dar tamaño a sus dos dimensiones (filas y columnas):

```
const int TAM=10;
char tablero[TAM][TAM]; // Matriz de 10 x 10 elementos
int tabla[5][8]; // Matriz de 5 x 8 elementos
```

- Como los vectores, comienzan en `0` y acaban en `TAM-1`
- Asignación y acceso a valores mediante el operador []:

```
int matriz[8][10];
matriz[2][3]=7; // Hay que indicar fila y columna
```

Ejercicios

- Realiza un programa que imprima el desglose en billetes y monedas de una cantidad entera de euros, utilizando el menor número posible de billetes y monedas. Los valores de billetes y monedas disponibles son: 500, 200, 100, 50, 20, 10, 5, 2, 1. Intente que el código sea lo más compacto posible.
- Escribe un programa que nos diga si los elementos de un array están ordenados. Asume que el array ya tiene datos y su tamaño está almacenado en la constante `TAM`
- Escribe un programa que determine si una matriz cuadrada es triangular superior, es decir, todos los elementos por debajo de la diagonal principal son 0 (como la de abajo). Asume que la matriz ya tiene datos y su tamaño está almacenado en la constante `TAM`

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 12 & 6 \\ 0 & 0 & -3 \end{pmatrix}$$

Índice

Funciones

- Una función es un bloque de código que realizan una tarea
- Permite agrupar operaciones comunes en un bloque reutilizable
- Puede opcionalmente tener parámetros de entrada y devolver un valor como salida

```
tipoRetorno nombreFuncion(parametro1,parametro2,...){
  tipoRetorno ret;
  instruccion1;
  instruccion2;
  ...
  return ret;
}
```

Funciones

- Se puede utilizar más de un `return` en el cuerpo de una función si eso simplifica el código

```
bool buscar(int vec[],int n){ // Dos parámetros
  bool encontrado=false;
  for(int i=0;i<TAM && !encontrado;i++){
    if(vec[i]==n)
       encontrado=true;
  }
  return encontrado; // Un único return
}
```

```
bool buscar(int vec[], int n){
  for(int i=0;i<TAM;i++){
    if(vec[i]==n)
      return true; // Primer return
  }
  return false; // Segundo return
}
```

Paso de parámetros

- Se permite paso de parámetros por valor o por referencia (con &)
  - Por valor: se hace una copia de la variable → las modificaciones en el interior de la función no son visibles fuera
  - Por referencia: no se hace copia → las modificaciones en el interior de la función SÍ son visibles fuera

```
// a y b se pasan por valor, c por referencia
void funcion(int a,int b,bool &c){
  c=a<b; // c mantiene este valor al acabar la función
}
```

- Es recomendable pasar por referencia variables muy grandes cuya copia puede ralentizar el programa
  - Con el modificador `const` indicamos que la variable no se va a modificar dentro de la función (para evitar errores inesperados)

```
void funcion(const string &s){
  // El compilador no hace copia de s, pero si
  // intentamos modificarlo nos da un error
}
```

Pasando arrays y matrices

- Los arrays y matrices sólo se pueden pasar por referencia
- Hay que indicarlo con `[]` en la declaración de la función, omitiendo el tamaño en la primera dimensión

```
void sumar(int v[],int m[][TAM]){
  // En m no se pone el tamaño de la primera dimensión
  ...
}
...
// No se ponen corchetes en la llamada a la función
sumar(v,m);
```

Prototipos

- A veces es necesario utilizar una función antes de que aparezca su código
- En esos casos se debe poner el prototipo de la función

```
void miFuncion(bool,char,double[]); // Prototipo

char otraFuncion(){
  double vr[20];
  // Todavía no se ha declarado miFuncion
  // pero podemos usarla gracias al prototipo
  miFuncion(true,'a',vr);
}

// Declaración de la función
void miFuncion(bool exist,char opt,double vec[]){
  ...
}
```

Ejercicios

- Escribe un programa que imprima todos los números primos en un rango determinado. El primer y último elemento del rango serán pedidos por teclado al usuario. Emplea funciones para simplificar el código del programa.

Índice

Arrays de caracteres

- Para representar cadenas de caracteres, podemos emplear arrays de tipo `char`
- Desventajas:
  - No podemos cambiar el tamaño de la cadena
  - No podemos saber fácilmente su longitud
  - Cualquier operación implica iterar elemento a elemento

```
const int TAM=4;
char cadena[TAM];
cadena[0]='h';cadena[1]='o';cadena[2]='l';cadena[3]='a';

for(int i=0; i< TAM; i++)
   cout<<cadena[i];
cout << endl;
```

Cadenas de caracteres de C

- Último carácter de cada cadena sea `'\0'`
- Fácil manipulación con funciones de `#include<string.h>`

```
#include<string.h>
char cadena[]="hola";
//Las líneas de abajo son equivalentes a la anterior
char cadena[5];
cadena[0]='h';cadena[1]='o';
cadena[2]='l';cadena[3]='a';cadena[4]='\0';

cout<<cadena<<endl;
//Imprime: La longitud es 4
cout << "La longitud es" << strlen(cadena) << endl;
```

- Pero todavía tenemos el problema del tamaño fijo
- No las usaremos en la asignatura

La clase `string`

- Representa una cadena de tamaño variable
- Se usa igual que un tipo básico
- No es necesario preocuparse del `'\0'`

```
string vacia; //Cadena vacía por defecto
string s="hola"; // Almacena 4 caracteres
s="hola a todo el mundo"; // Almacena 20 caracteres*
s="ok"; // Almacena 2 caracteres
```

- Salida por pantalla con cout y cerr:

```
string s="Nota";
int num=10;
cout << s << " -> " << num; // Muestra "Nota -> 10"
```

La clase `string`

- Lectura de entrada estándar con operador >>: lee hasta el primer espacio en blanco

```
string s;
cin >> s;
// El usuario escribe "hola"
// La variable s almacena "hola"
...
// El usuario escribe "buenas tardes"
// La variable s almacena "buenas"
```

- Con `getline`: lee hasta el primer salto de línea

```
string s;
getline(cin,s);
// Si el usuario introduce "buenas tardes"
// en s se almacena "buenas tardes"
```

Métodos de `string`

- `length` devuelve el número de caracteres de la cadena:

```
// unsigned int length()
string s="hola, mundo";
cout << s.length(); // Imprime 11
```

- `find` devuelve la posición en la aparece una subcadena dentro de una cadena

```
// size_t find(const string &s,unsigned int pos=0)
cout << s.find("mundo"); // Imprime 6
// Si no encuentra la subcadena devuelve string::npos
```

Métodos de `string`

- `replace` sustituye una cadena (o parte de ella) por otra

```
// string& replace(unsigned int pos,unsigned int tam,
//const string &s)
string s="hola mundo";
s.replace(0,4,"hello"); // s vale "hello mundo"
```

- `erase` permite eliminar parte de una cadena

```
// string& erase(unsigned int pos=0,unsigned int tam=
//string::npos);
string cad="hola mundo";
cad.erase(4,3); // cad vale "holando"
```

- `substr` devuelve una subcadena de la cadena original

```
// string substr(unsigned int pos=0,unsigned int tam=
//string::npos) const;
string cad="hola mundo";
string subcad=cad.substr(2,5); // subcad vale "la mu"
```

Operadores de `string`

- Comparaciones: == (igual), != (distinto), > (mayor estricto), >= (mayor o igual), < (menor estricto) y <= (menor o igual)

```
string s1,s2;
cin >> s1; cin >> s2;
if(s1==s2) // La comparación es en orden lexicográfico
cout << "Son iguales" << endl;
```

- Asignación

```
string s1="hola";
string s2;
s2=s1;
```

- Concatenación

```
string s1="hola";
string s2="mundo";
string s3=s1+","+s2; // s3 vale "hola, mundo"
```

Operadores de `string`

- Acceso a componentes como si fuera un array de caracteres

```
string s="hola";
char c=s[3]; // s[3] vale 'a'
s[0] = 'H';
cout << s << ":" << c << endl ; // Imprime "Hola:a"
```

- Ejemplo de recorrido de un string carácter a carácter

```
string s="hola, mundo";
for(unsigned int i=0;i<s.length();i++)
    s[i]='f'; // Sustituye cada carácter por 'f'
```

Conversiones de/a `string`

- Convertir un número entero o real a `string`:

```
#include <string> // ¡Ojo! No es lo mismo que <string.h>
...
int num=100;
string s=to_string(num);
```

- Convertir un string a número entero:

```
string s="100";
int num=stoi(s);
```

- Convertir un string a número real:

```
string s="10.5";
float num=stof(s);
```

Ejercicios

- Escribe un programa que indique si la letra de un número de DNI introducido por teclado es correcta. La letra se obtiene dividiendo el número entre 23 y quedándonos con su resto. Esta tabla muestra la letra correspondiente según el resto:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

- Escribe un programa que nos devuelva el número de dígitos que contiene una cadena introducida por teclado
- Escribe un programa que lea cadenas por teclado, una por línea. El programa acabará cuando se introduzca la cadena vacía. Por cada cadena introducida, el programa deberá imprimir cuántos dígitos tiene.

Ejercicios

- Escribe un programa que pida al usuario dos cadenas e imprima el prefijo común más largo. Por ejemplo, para `polinomio` y `polinización` debe imprimir `polin`.
- Escribe una función que determine si una cadena es un palíndromo: se lee igual de izquierda a derecha que de derecha a izquierda.

Índice

Ficheros

- Ficheros (o archivos): forma en la que C++ permite acceder a la información almacenada en disco
- Su tamaño puede variar durante la ejecución del programa según los datos que almacena
- Dos maneras de guardar la información: **ficheros de texto** y ficheros binarios
- Ficheros de texto:
  - Secuencias de caracteres, tal como se mostrarían por pantalla: el valor entero 19 se guardará en fichero como los caracteres 1 y 9
  - Modo de lectura/escritura secuencial

Declaración

- Un tipo de dato más en C++
- Biblioteca `fstream`

```
#include <fstream>
```

- Tres tipos de datos básicos para trabajar con ficheros:

```
ifstream ficheroLec; // Leer de fichero
ofstream ficheroEsc; // Escribir en fichero
fstream ficheroLecEsc; // Leer y escribir en fichero [NO LO USAREMOS]
```

Apertura y cierre

- `open` asocia variable a fichero físico

```
ifstream fichero; // Vamos a leer del fichero
fichero.open("miFichero.txt");
// Ahora ya podemos leer de "miFichero.txt"

//también podemos utilizar strings
string nombreFichero="mifichero.txt"
fichero.open(nombreFichero);
```

Apertura y cierre

- `open` tiene un segundo parámetro que determina el modo de apertura
  - Lectura: `ios::in`
  - Escritura: `ios::out`
  - Añadir al final: `ios::out | ios::app`

```
ifstream ficheroLec;
ofstream ficheroEsc;
// Abrimos solo para leer
ficheroLec.open("miFichero.txt",ios::in);
// Abrimos para añadir información al final
ficheroEsc.open("miFichero.txt",ios::out|ios::app);
```

- Consideraciones:
  - Por defecto, el tipo `ifstream` se abre para lectura y el `ofstream` para escritura
  - Si abrimos un fichero que ya existe para escritura (`ios::out`), se borra todo su contenido
  - Si abrimos con `ios:app`, se añade la nueva información al final
  - Si el fichero no existe, se creará uno nuevo con tamaño inicial 0

Apertura y cierre

- Antes de leer/escribir, se debe comprobar con `is_open` si el fichero se ha abierto correctamente (true) o no (false)
- Al terminar de usar el fichero, se debe liberar con close

```
ifstream fl("miFichero.txt");
if(fl.is_open()){
// Ya podemos trabajar con el fichero
...
fl.close(); // Cerramos el fichero
}
else // Mostrar error de apertura
```

Lectura carácter a carácter

- Función `get`: lee un carácer (no descarta blancos)
- Devuelve `false` cuando se ha llegado al final del fichero: debemos ignorar lo que ha leído

```
ifstream fl("miFichero.txt");
if(fl.is_open()){
  char c;
  while(fl.get(c)){
    cout << c;
  }
  fl.close();
}
else{
  cout << "Error al abrir el fichero" << endl;
}
```

Lectura línea a línea

- Función `getline`: lee una línea completa (no descarta blancos)
- Devuelve `false` cuando se ha llegado al final del fichero: debemos ignorar lo que ha leído

```
ifstream fl("miFichero.txt");
if(fl.is_open()){
  string s;
  while(getline(fi,s)){
    cout << s << endl;
  }
  fl.close();
}
else{
  cout << "Error al abrir el fichero" << endl;
}
```

Lectura con el operador >>

- Con >> se pueden leer datos de cualquier tipo igual que con `cin`
- `eof()` nos indica si se ha alcanzado el final de fichero
  - Al leer datos que estarían fuera del fichero, devuelve `true`
  - Pero cuando se leen los últimos datos válidos, devuelve `false`
- Por ejemplo, si tenemos un fichero que contiene en cada línea una cadena y dos enteros (ej. `Hola 1032 124`)

```
ifstream fl("miFichero.txt");
if(fl.is_open()){
  string s;
  int num1,num2;
  fl >> s;
  while(!fl.eof()){ // Lee el string
    fl >> num1; // Lee el primer número
    fl >> num2; // Lee el segundo número
    cout << s << "," << num1 << "," << num2 << endl;
    fl >> s;
  }
fl.close();
}
```

Ejercicios

- Implementa un programa que lea un fichero `fichero.txt` e imprima por pantalla las líneas del fichero que contienen la cadena `Hola`.
- Diseña una función `inicioFichero` que reciba dos parámetros: el primero debe ser un número entero positivo `n` y el segundo el nombre de un fichero de texto. La función debe mostrar por pantalla las `n` primeras líneas del fichero. El programa también debe funcionar con ficheros menores de *n* líneas.
- Diseña una función `finFichero` que reciba dos parámetros: el primero debe ser un número entero positivo `n` y el segundo el nombre de un fichero de texto. La función debe mostrar por pantalla las `n` últimas líneas del fichero. El programa también debe funcionar con ficheros menores de *n* líneas.

Escritura

- Utilizamos el operador << igual que con `cout`

```
ofstream fe("miFichero.txt");
if(fe.is_open()){
  int num=10;
  string s="Hola, mundo";
  fe << "Un numero entero: " << num << endl;
  fe << "Un string: " << s << endl;
  fe.close();
}
else{
  cout << "Error al abrir el fichero" << endl;
}
```

Índice

Organización de la memoria

- Los datos que guardamos en variables y constantes se almacenan en la memoria del ordenador
- Cada dato se guarda una dirección o posición de memoria numerada
- Variables estáticas (las que hemos usado hasta ahora)
  - Siempre asociadas a la misma posición de memoria
  - La posición de memoria queda libre cuando salen de ámbito

```
int i=0;
char c;
float vf[3]={1.0,2.0,3.0};
```

Organización de la memoria

```
int mi_funcion(){
    int a,b;
    a=1;
    b=2;
    int c = a+b;
    return c;
}

void main(){
    int a=0;
    a=mi_funcion();
    cout << a;
}
```

Ejercicio

Haz una traza del contenido de la memoria en este fragmento de código

Punteros

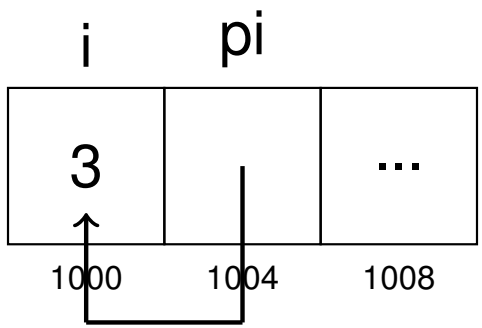
- Un puntero almacena la dirección de memoria donde se encuentra otro dato
- Se dice que el puntero “apunta” a ese dato
- Los punteros se declaran usando el carácter *
- El dato al que apunta el puntero será de un tipo concreto que deberá indicarse al declarar el puntero:

```
int *punteroEntero; // Puntero a entero
char *punteroChar; // Puntero a carácter
int *vecPunterosEntero[20]; // Array de punteros a entero
double **doblePunteroReal; // Puntero a puntero a real
```

Punteros

- El operador * permite acceder al contenido de la variable a la que apunta el puntero
- El operador & permite obtener la dirección de memoria en la que está almacenada una variable:

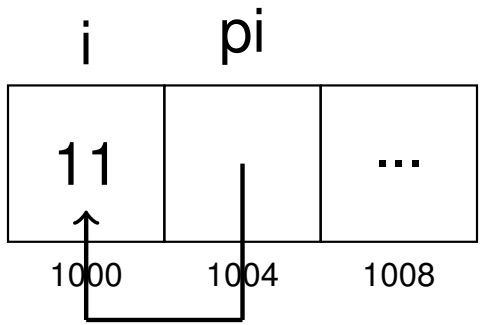
```
int i=3;
int *pi;
pi=&i; // pi contiene la dirección de memoria de i
*pi=11; // Contenido de pi es "11". Por lo tanto i = 11
```



Punteros

- Los punteros y las direcciones de memoria se pueden imprimir, aunque no suele ser necesario

```
int i=11;
int *pi;
pi=&i;
cout << pi << endl; // Muestra "1000"
cout << *pi << endl; // Muestra "11"
cout << &pi << endl; // Muestra "1004"
```



Punteros

- Para indicar que un puntero no apunta todavía a ninguna dirección de memoria, le asignamos el valor `nullptr`
- Es muy recomendable inicializar siempre los punteros con `nullptr`:

```
int i=11;
int *pi=nullptr;

if(...){ // Si se cumple alguna condición
    pi=&i;
}

if(pi != nullptr)
   cout << *pi << endl; // Muestra "11"
```

Punteros

```
int *p;
int mi_funcion(){
  int a,b;
  a=1; b=2;
  p =&a;*p = 3;
  p= &b;*p=4;
  int c = a+b;
  return c;
}

void main(){
   int a=0;
   a=mi_funcion();
   cout << a;
   cout << *p;
}
```

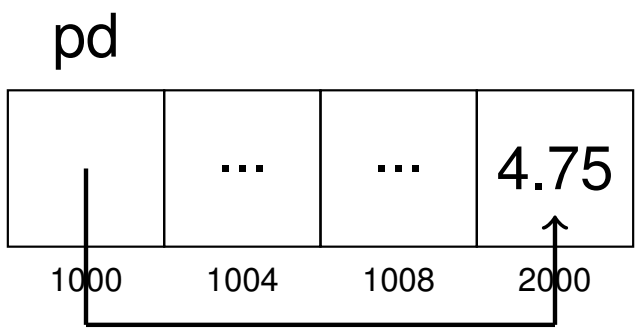
Ejercicio

Haz una traza del contenido de la memoria en este fragmento de código. ¿Hay algún error?

Reserva y liberación de memoria

- El operador `new` permite reservar memoria de manera dinámica
- Devuelve la dirección del bloque de memoria reservado, que se debe almacenar en un puntero
- Si no hay suficiente memoria para la reserva, devuelve `nullptr`

```
double *pd = new double;
if(pd!=nullptr){ // Comprueba que se ha podido reservar
  *pd=4.75;
  cout << *pd << endl; // Muestra "4.75"
}
```



Reserva y liberación de memoria

- El operador `delete` permite liberar memoria reservada con `new`
- **Siempre que se reserva con `new` hay que liberar con `delete`**

```
double *pd;
pd=new double; // Reserva memoria
...
delete pd; // Libera la memoria apuntada por pd
pd=nullptr; // Conveniente si vamos a seguir usando pd
```

- Un puntero se puede reutilizar tras liberar su contenido y reservar memoria otra vez:

```
double *pd;
pd=new double; // Reserva memoria
...
delete pd; // Libera la memoria apuntada por pd
pd=new double; // Reservamos de nuevo memoria
...
```

Reserva y liberación de memoria

```
int *p=NULL;
int mi_funcion(){
    int a,b;
    a=1;
    b=2;
    p =new int;
    *p=3;
    int c = a+b + *p;
    return c;
}

void main(){
    int a=0;
    a=mi_funcion();
    cout << *p;
}
```

Ejercicio

Haz una traza del contenido de la memoria en este fragmento de código. ¿Echas algo de menos?

Punteros y arrays

- Una variable de tipo array es en realidad un puntero al primer elemento del array
- Siempre apunta al primer elemento del array y no se puede modificar

```
int vec[5]={4,5,2,8,12};
cout << vec << endl; // Muestra la dirección de memoria
// del primer elemento del array
cout << *vec << endl; // Muestra "4"
```

- También se pueden obtener punteros a otros elementos del array

```
int vec[20];
int *pVec=vec; // Ambos son punteros a entero
*pVec=58; // Equivalente a vec[0]=58;
pVec=&(vec[7]);
*pVec=117; // Equivalente a vec[7]=117;
```

Punteros y arrays

- Los punteros pueden usarse para crear arrays dinámicos
- Se reservan añadiendo corchetes a `new`
- Se liberan añadiendo corchetes a `delete`

```
int *pv;
pv=new int[10]; // Reserva memoria para 10 enteros
pv[0]=585; // Accedemos como en un array estático
...
delete [] pv; // Liberamos toda la memoria reservada
```

- **Nos permiten cambiar el tamaño del array sin tener que declarar otra variable**

Ejercicio

Completa el siguiente código para que se puedan asignar al array tantos números en cualquier posición

```
void imprimir(int* array, int tamano){
    for(int i=0; i < ?????; i++)
        cout << ????? << " ";
}
void asignar(int* array, int &tamano, int posicion, int numero){
    ??????
}
//Main:
int tam=10;
int* arr = new int[tam];
asignar(arr,tam, 0, 1);
asignar(arr,tam, 1 , 2);
...
asignar(arr,tam, 19,15);
asignar(arr,tam, 20,42);
imprimir(arr,tam);
```