Tema 2. Tipos y Estructuras Básicas

`http://aulavirtual.uji.es`

José M. Badía, Begoña Martínez, Antonio Morales y José M. Sanchiz

{badia, bmartine, morales, sanchiz}@icc.uji.es

Estructuras de datos y de la información

Universitat Jaume I

Índice

Bibliografía

- (Nyhoff'06), capítulos 2, 3 y 5.2
- (Orallo'02), capítulo 2.
- (Stroustrup'97), capítulo 4.

Objetivos

- Revisar tipos y estructuras de datos básicos disponibles en los principales lenguajes de programación imperativos.
- Diferenciar la definición de un tipo de la declaración de una variable.
- Saber usar estos tipos y estructuras en el lenguaje C++.
- Conocer las facilidades que aporta C++ en la librería STL.

1 Introducción

- Solucionar problemas implica manejar datos.
- Físicamente, en un ordenador los datos se codifican como secuencias de 0 y 1.
- Tipo de datos: Interpretación de los datos binarios.
- Dos grandes clases:
  - Simples (escalares). Contienen un sólo dato.
  - Compuestos (estructurados). Agrupan varios datos.

1 Introducción (II)

- **Tipos simples**: son tipos escalares, una variable $=$ un valor.

  Son: entero, real, lógico, carácter y enumerado.

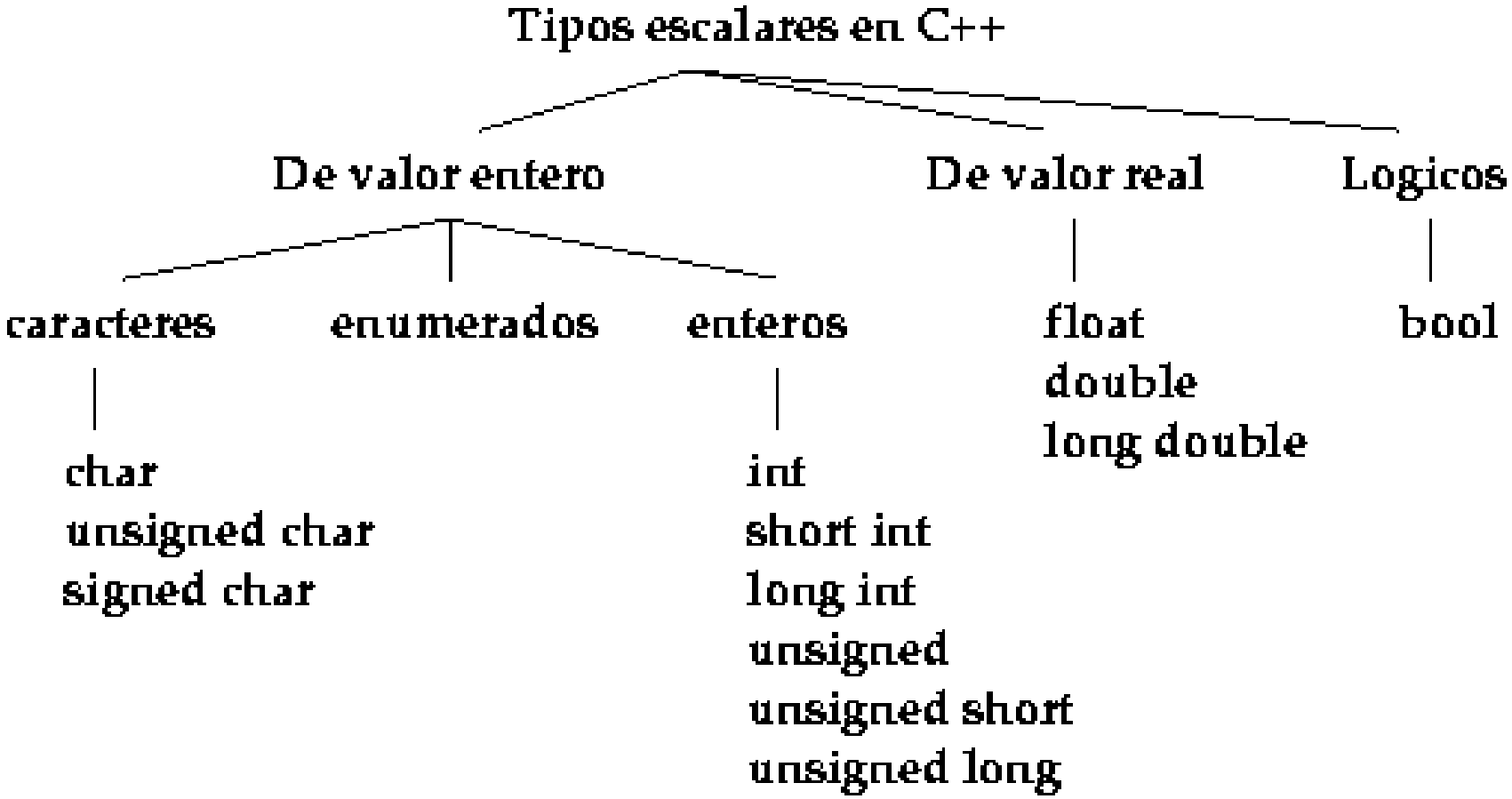
  En C++: *int, float, double, bool, char, enum*.

  enum en un mecanismo para definir nuevos tipos escalares.

- **Estructuras básicas**: vectores, cadenas de caracteres, y registros.

  En C++: *array, string, struct, union*.

2 Tipos Escalares



2.1 Tipos Lógicos

- **En C**
  - No existe
  - Se usan valores enteros
    - 0 equivale a falso
    - distinto de 0 equivale a verdadero
- **En C++**
  - Tipo predefinido: `bool`
  - Valores: true y false
  - También se usan enteros

2.2 Tipos Enumerados

- **Valores**. Lista ordenada de valores conceptuales dados por el usuario.
- **Operaciones**. Asignación, relacionales.
- **Implementación**.

  Los valores se codifican con números enteros comenzando por 0 (por defecto).
- **Definición en C++**:

```
enum TEstado {APAGADO=2, CONECTADO, ENLINEA};
```

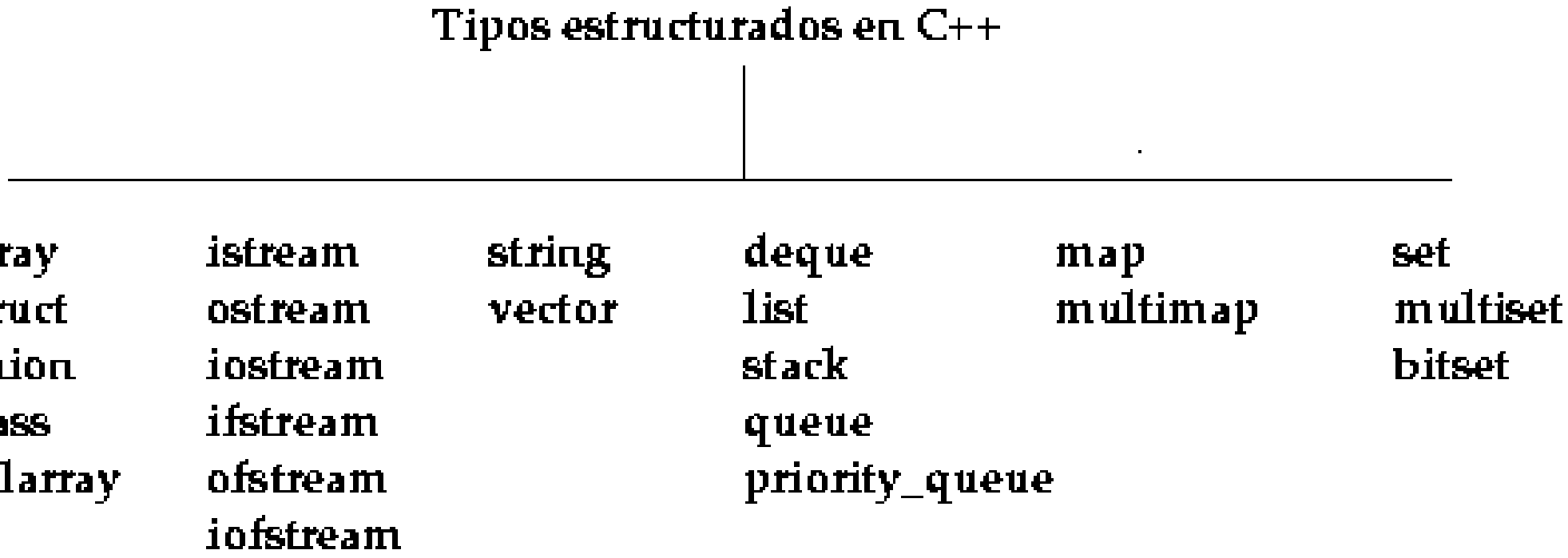
- **Usos**:
  - Permite usar conceptos como valores
  - Sus valores pueden usarse como constantes enteras
  - Cuidado con la lectura y escritura de enumerados

Ejemplo

```
int e;
...
  switch(e) {
    case 2:
      ...
    case 3:
      ...
    case 4:
      ...
  }
```

```
TEstado e;
...
  switch(e) {
    case APAGADO:
      ...
    case CONECTADO:
      ...
    case ENLINEA:
      ...
  }
```

3 Tipos Estructurados



3.1 Vectores

Vector: colección ordenada de elementos del mismo tipo.

TAD vector

Usa tipoindice, tipobase

Operaciones:

CreaVector: $\longrightarrow$ vector

Almacena: vector x tipoindice x tipobase $\longrightarrow$ vector

Consulta: vector x tipoindice $\longrightarrow$ tipobase

Axiomas: $\forall$ i, j $\in$ tipoindice, $\forall$ e, f $\in$ tipobase

1) Consulta(CreaVector,i) = *error*

2) Consulta(Almacena(v,i,e),j) = *si* i=j *entonces* e
sino Consulta(v,j)

3) Almacena(Almacena(v,i,e),j,f) = *si* i=j *entonces* Almacena(v,j,f)
sino Almacena(Almacena(v,j,f),i,e)

3.1 Vectores (II)

Propiedades:

- Número fijo de elementos.
- Elementos ordenados (primero, segundo, etc).
- Elementos del mismo tipo.
- Acceso directo a un elemento por medio de su posición.

Vectores en C/C++

- **Declaración**:

```
tipoElemento nombreVector[CAPACIDAD];
```

Implementación

- Los elementos ocupan bloques de memoria contiguos, cada bloque del tamaño necesario para que quepa un elemento.
- La variable vector es un puntero al tipo base: Contiene la dirección inicial del vector, dirección del elemento 0.
- El elemento `i` está en la dirección:

$$direccionBase + i \times tamanoElemento$$

3.1 Vectores (IV)

- **Vectores bidimensionales**: Pueden considerarse como una matriz.

  `tipoElemento nombreVector[NUM_FILAS][NUM_COLUMNAS];`

  Ejemplos:

```
const int FILAS=30, COLUMNAS=5;
typedef double TMatriz[FILAS][COLUMNAS];
TMatriz m;
...
double tabla[2][3] = { {0.5, 0.6, 0.3},
                       {0.6, 0.4, 0.5} };
```

  El elemento **`[i][j]`** está en la dirección:

$$direccionBase + (i \times NUM\_COLUMNAS + j) \times tamanoElemento$$

- **Vectores como parámetros**:

  En realidad se pasa la dirección base del vector. En el caso de un vector unidimensional

  ```
  typedef double VECTOR[CAPACIDAD];
  ```

  Son equivalentes:

  ```
  void Imprime(double v[CAPACIDAD],  int n);
  void Imprime(double v[], int n);
  void Imprime(double *v, int n);
  void Imprime(VECTOR v, int n);
  ```

3.1 Vectores (VI)

- **Vectores como parámetros** (cont.):

  En vectores de dos o más dimensiones

  ```
  typedef double TABLA[FILAS][COLUMNAS];
  ```

  las siguientes declaraciones

  ```
  void Imprime(double t[][], int n, int m);
  void Imprime(double **t, int n, int m);
  ```

  NO funcionan, pues el compilador no puede calcular la dirección de memoria del elemento `[i][j]`. Es necesario hacer:

  ```
  void Imprime(double t[FILAS][COLUMNAS], int n, int m);
  void Imprime(double t[][COLUMNAS], int n, int m);
  void Imprime(TABLA t, int n, int m);
  ```

Problemas con los vectores de C

- Limitación en ciertas operaciones: Se tratan elemento a elemento
  - Asignación
  - Operaciones relacionales
- Tamaño fijo. Su capacidad no puede cambiar en tiempo de ejecución.
- No hay comprobación de rango.
- No son objetos autocontenidos. No incorporan toda la información necesaria para describirlos y operar con ellos (p.e. número de elementos).

3.1 Vectores (VIII)

- **Vectores en la STL**:

  La *Standard Templates Library* (STL) es parte de C++ desde 1994. Aporta una colección de tipos de datos (clases o *contenedores*), con algoritmos para realizar las operaciones más usuales.

  Incluye la clase **`vector`**, mucho más flexible que los vectores clásicos de C, vistos hasta ahora. Entre otras cosas, a los objetos **`vector`** se les puede modificar el tamaño en tiempo de ejecución, y se comprueba automáticamente que los índices estén dentro del rango correcto.

  Se dará más información de la STL más adelante.

3.2 Cadenas

Cadena: Secuencia ordenada de caracteres.

TAD cadena

Usa caracter, nat, bool

Operaciones:

CreaCadena: $\longrightarrow$ cadena

AñadeCaracter: cadena x caracter $\longrightarrow$ cadena

Concatena: cadena x cadena $\longrightarrow$ cadena

Longitud: cadena $\longrightarrow$ nat

Compara: cadena x cadena $\longrightarrow$ bool

Consulta: cadena x nat $\longrightarrow$ caracter

Axiomas: $\forall$ c, c1, c2 $\in$ cadena, $\forall$ e $\in$ caracter, $\forall$ i $\in$ nat

1) Concatena(c,CreaCadena) = c

2) Concatena(c1, AñadeCaracter(c2,e)) = AñadeCaracter(Concatena(c1,c2),e)

3) Longitud(CreaCadena) = *cero*

4) Longitud(AñadeCaracter(c,e)) = succ(Longitud(c))

3.2 Cadenas (II)

En C

```c
#include <cstring>
...
...
char cad1[20], cad2[20];
strcpy(cad1, "Hola");
scanf("%s", cad2);
if (!strcmp(cad1, cad2))
  printf("Son iguales \n");
else {
  strcat(cad1, cad2);
  printf("%s\n", cad1);
}
```

En C++

```c
#include <string>
#include <iostream>
...
string cad1, cad2;
cad1 = "Hola";
cin >> cad2;
if (cad1 == cad2)
  cout << "Son iguales" << endl;
else {
  cad1 = cad1 + cad2;
  cout << cad1 << endl;
}
```

3.3 Registros

Registro: colección de datos de distintos tipos

TAD registro

Usa tipobase_i

Operaciones:

$\text{CreaRegistro}: \rightarrow \text{registro}$

$\text{Almacena}_i: \text{registro x tipobase}_i \rightarrow \text{registro}$

$\text{Consulta}_i: \text{registro} \rightarrow \text{tipobase}_i$

... (Tantas operaciones *Almacena* y *Consulta* como campos)

Axiomas: $\forall$ r $\in$ registro, $\forall$ $e_i \in \text{tipobase}_i$, $\forall$ $e_j \in \text{tipobase}_j$

1) $\text{Consulta}_j(\text{CrearRegistro}) = \text{error}$

2) $\text{Consulta}_j(\text{Almacenar}_i(r,e_i)) =$ *si* $\{i\} = \{j\}$ *entonces* e_i
sino $\text{Consulta}_j(r)$

3) $\text{Almacenar}_j(\text{Almacenar}_i(r,e_i),e_j) =$
si $\{i\} = \{j\}$ *entonces* $\text{Almacenar}_j(r,e_j)$
sino $\text{Almacenar}_i(\text{Almacenar}_j(r,e_j),e_i)$

4) $\text{Almacenar}_i(\text{CrearRegistro},e_i) = \text{Almacenar}_i(\text{CrearRegistro},e_i)$

3.3 Registros (II)

Definición de tipos

```
struct Tpunto {
  float x, y;
};
struct Ttemperatura {
  double grados;
  char escala;
} temp1, q; // Además declara dos variables
```

Declaración de variables

En C:

```
struct Tpunto a, b;
```

En C++ :

```
Tpunto a, b;
```

4 Librería STL

STL: Standard Template Library

Contiene definiciones de clases: plantillas de datos (contenedores) + algoritmos.

- Contenedores: vectores, listas, mapas, conjuntos, colas.
- Algoritmos: ordenación, búsqueda, recorrido, permutación, montículos, etc.
- Iteradores: para recorrido de objetos compuestos.

```
#include <vector>
...
vector<double>v1(100, -3.14);
vector<double>v2(50, 2.17);
v1.swap(v2); //intercambio eficiente de la STL
sort(v1.begin(), v1.end());
```

5 Definición vs. Declaración

- **Definir un tipo**
  - Especifica un nombre y una estructura para un nuevo tipo de datos.
  - NO reserva espacio para ninguna variable.

```
struct Tcomplejo{
  float preal, pimag;
};
```

- **Declarar variables**
  - Reserva y da un nombre a una zona de memoria.
  - Su contenido se interpretará como de un tipo de datos dado.

```
float f;
Tcomplejo c;
```

5 Definición vs. Declaración (II)

Definición de tipos con `typedef`

- Creando un alias para otro tipo existente

```
typedef int Tindice;
typedef float * Tpuntero;
```

- Construyendo un nuevo tipo a partir de otro existente

```
typedef char Tcadena[10];
typedef Tcadena Tmatriz[5];
```

5 Definición vs. Declaración (III)

- Registros

```
struct Tfecha {
  int dia, mes, anyo;
};
class Tpersona {
  string nombre, dni;
  Tfecha fnacimiento;
};
```

- Enumerados

```
enum Tdia {lunes, martes, miercoles, jueves, viernes};
```