UNIVERSIDAD DE MÁLAGA
Dpto. Lenguajes y CC. Computación
E.T.S.I. Telecomunicación

TIPOS DE DATOS ESTRUCTURADOS

Tema 4

Programación I

Tema 4: TIPOS DE DATOS ESTRUCTURADOS

1. Paso de parámetros de tipos estructurados
2. Cadenas de caracteres
3. Registros
4. Arrays
5. Resolución de problemas usando tipos estructurados
6. Bibliografía: [DALE89a], [JOYA03]

PASO DE PARÁMETROS DE TIPOS ESTRUCTURADOS

- Los lenguajes de programación normalmente utilizan el **paso por valor** y el **paso por referencia** para implementar la transferencia de información entre subprogramas descrita en el interfaz.

- Para la transferencia de información de **entrada**, el paso por valor supone **duplicar y copiar el valor** del parámetro actual en el formal.

- En el caso de tipos **simples**, el paso por valor es adecuado, sin embargo, si el tipo de dicho parámetro es **estructurado**, es posible que dicha copia implique una **alta sobrecarga**, tanto en espacio de memoria como en tiempo de ejecución.

- El lenguaje de programación **C++** permite realizar de forma eficiente la transferencia de información de **entrada** mediante el **paso por referencia constante**.

	Tipos	
	Simples	Compuestos
(⇓) Entrada	P.Valor (`int x`)	P.Ref.Cte (`const Persona& x`)
(⇑) Salida, (⇕) E/S	P.Ref (`int& x`)	P.Ref (`Persona& x`)

Tema 4: TIPOS DE DATOS ESTRUCTURADOS

1. Paso de parámetros de tipos estructurados
2. Cadenas de caracteres
3. Registros
4. Arrays
5. Resolución de problemas usando tipos estructurados
6. Bibliografía: [DALE89a], [JOYA03]

- CADENAS DE CARACTERES

  - Representan una sucesión o **secuencia de caracteres**

  - Este tipo de datos es muy versátil, ya que sirve para representar información muy diversa:

    - Representa información textual (caracteres)

    - Entrada de datos y salida de resultados en forma de secuencia de caracteres.

    - Representa información abstracta por medio de una secuencia de caracteres

- EL TIPO `string`

  - En C++ se debe incluir la biblioteca estándar `string`.

  - El tipo `string` representa una secuencia de caracteres de **longitud finita limitada** por la implementación.

  - Podemos definir tanto constantes como variables y parámetros.

  - Una cadena de caracteres literal se representa mediante una sucesión de caracteres entre comillas dobles: `"ejemplo"`.

```
#include <iostream>
#include <string>
using namespace std;
// - Constantes ----
const string AUTOR = "Jose Luis";
// - Principal ----
int main ()
{
   string nombre = "";
}
```

- EL TIPO `string`
  - Operaciones con Cadenas de Caracteres
    - Asignación (`=`)
    - Paso como parámetro a subprogramas.
    - Devolución por una función. Esta operación, aunque es válida, no es recomendable.

- EL TIPO `string`
  - Operaciones con Cadenas de Caracteres. Entrada/Salida:

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
   string nombre, linea;
   char c;
   cin >> nombre; // Salta [espacio/ENTER] iniciales, lee hasta [espacio/ENTER]
   cin.ignore(10000, '\n'); // Salta hasta [ENTER] (elim)
   cin >> ws; // Salta [espacio/ENTER] iniciales, no lee ningun dato
   getline(cin, linea); // No salta espacios iniciales, lee hasta [ENTER] (elimina)
   cin.get(c); // No salta espacios iniciales, lee un caracter
   cout << nombre << " " << linea << endl;
}
```

- EL TIPO `string`
  - Operaciones con Cadenas de Caracteres
    - Comparación lexicográfica (`==` , `!=` , `>` , `>=` , `<` , `<=`)
      - `if (nombre >= AUTOR) {/*...*/}`
    - Concatenación de cadenas y caracteres (`+` , `+=`):

```
#include <iostream>
#include <string>
using namespace std;
// - Constantes ----
const string AUTOR = "Jose Luis";
// - Principal ----
int main ()
{
    string nombre = AUTOR + "Lopez";
    nombre += "Vazque";
    nombre += 'z';
    nombre = AUTOR + 's';
}
```

EL TIPO `string`

- Operaciones con Cadenas de Caracteres
  - Longitud. Número de caracteres que componen la cadena:
    - `nombre.size()` $\in \mathbb{N}$
    - `int(nombre.size())` $\in \mathbb{Z}$
  - Acceso al iésimo carácter de la cadena (de tipo `char`):
    - `nombre[i]` donde `i` $\in$ `[0..nombre.size() - 1]`
  - Cambiar el número de caracteres de una cadena:
    - `nombre.resize(sz)` cambia el número de caracteres de `nombre` al valor especificado por `sz`. Si extiende, entonces rellena con el carácter *nulo*.
    - `nombre.resize(sz, 'x')` cambia el número de caracteres de `nombre` al valor especificado por `sz`. Si extiende, entonces rellena con el carácter `'x'`.
- En *GNU G++*, la opción de compilación `-D_GLIBCXX_DEBUG` permite comprobar los índices de acceso.

EL TIPO `string`

- Operaciones con Cadenas de Caracteres
  - Obtener una **nueva** subcadena (de tipo `string`):
    - `nombre.substr(i, sz)` donde `i` $\in$ `[0..nombre.size()]` y `sz` $\in$ `[0..nombre.size() - i]`
    - No es válida la asignación a una subcadena:
      ~~`nombre.substr(i, sz) = "..."`~~
- En *GNU G++*, la opción de compilación `-D_GLIBCXX_DEBUG` permite comprobar los índices de acceso.

CADENAS DE CARACTERES. EJEMPLO

```
#include <iostream>
#include <string>
using namespace std;
// - Subalgoritmos --
void mayuscula (char& letra)
{
   if ((letra >= 'a') && (letra <= 'z')) {
      letra = char(int(letra) - int('a') + int('A'));
   }
}
void mayusculas (string& palabra)
{
   for (int i = 0; i < int(palabra.size()); ++i) {
      mayuscula(palabra[i]);
   }
}
// - Principal ----
int main ()
{
   string palabra;
   cin >> palabra;
   mayusculas(palabra);
   cout << palabra << endl;
}
```

CADENAS DE CARACTERES. EJEMPLO

- Programa que lee una palabra (formada por letras minúsculas), y escribe su plural según las siguientes reglas:
  - Si acaba en vocal se le añade la letra `'s'`.
  - Si acaba en consonante se le añaden las letras `'es'`. Si la consonante es la letra `'z'`, se sustituye por la letra `'c'`
  - Suponemos que la palabra introducida es correcta.

```
#include <iostream>
#include <string>
using namespace std;
// - Subalgoritmos --
bool es_vocal (char c)
{
   return (c == 'a') || (c == 'e') || (c == 'i') || (c == 'o') || (c == 'u');
}
void plural_1 (string& palabra)
{
   if (palabra.size() > 0) {
      if (es_vocal(palabra[palabra.size() - 1])) {
         palabra = palabra + 's';
      } else {
         if (palabra[palabra.size() - 1] == 'z') {
            palabra[palabra.size() - 1] = 'c';
         }
         palabra = palabra + "es";
      }
   }
}
// - Principal ----
int main ()
{
   string palabra;
   cin >> palabra;
   plural_1(palabra);
   cout << palabra << endl;
}
```

```
// - Subalgoritmos --
void plural_2 (string& palabra)
{
   if (palabra.size() > 0) {
      if (es_vocal(palabra[palabra.size() - 1])) {
         palabra = palabra + 's';
      } else if (palabra[palabra.size() - 1] == 'z') {
         palabra = palabra.substr(0, palabra.size() - 1) + "ces";
      } else {
         palabra = palabra + "es";
      }
   }
}
void plural_3 (string& palabra)
{
   if (palabra.size() > 0) {
      if (es_vocal(palabra[palabra.size() - 1])) {
         palabra = palabra + 's';
      } else if (palabra[palabra.size() - 1] == 'z') {
         palabra.resize(palabra.size() - 1);
         palabra = palabra + "ces";
      } else {
         palabra = palabra + "es";
      }
   }
}
```

CADENAS DE CARACTERES. EJEMPLO

- Diseñe una función que devuelva verdadero si la palabra recibida como parámetro es "palíndromoz falso en caso contrario

```
// - Subalgoritmos --
bool es_palindromo (const string& palabra)
{
   bool ok = false;
   if (palabra.size() > 0) {
      int i = 0;
      int j = palabra.size() - 1;
      while ((i < j) && (palabra[i] == palabra[j])) {
         ++i;
         --j;
      }
      ok = i >= j;
   }
   return ok;
}
```

CADENAS DE CARACTERES. EJEMPLO

- Diseñe un subprograma que reemplace una parte de la cadena, especificada por un índice y una longitud, por otra cadena.

```
// - Subalgoritmos --
void reemplazar (string& str, int i, int sz, const string& nueva)
{
   if (i <= int(str.size())) {
      if (i + sz < int(str.size())) {
         str = str.substr(0, i) + nueva + str.substr(i + sz, int(str.size()) - (i + sz));
      } else {
         str = str.substr(0, i) + nueva;
      }
   }
}
//---------------------------
// str.replace(i, sz, nueva);
// str.insert(i, nueva) ≡ str.replace(i, 0, nueva)
// str.erase(i, sz) ≡ str.replace(i, sz, "")
//---------------------------
```

Tema 4: TIPOS DE DATOS ESTRUCTURADOS

1. Paso de parámetros de tipos estructurados
2. Cadenas de caracteres
3. Registros
4. Arrays
5. Resolución de problemas usando tipos estructurados
6. Bibliografía: [DALE89a], [JOYA03]

- REGISTROS

  - Composición de un número determinado de elementos que pueden ser de **distintos tipos** de datos.

  - Objetivo: Definir una nueva entidad (tipo) como **composición** de entidades menores.

  - Se puede **acceder** a cada elemento individual de la composición mediante su **identificador**.

- REGISTROS. EJEMPLO

  - Podemos definir una nueva entidad (tipo) que represente el concepto de *Fecha* como agrupación de *dia*, *mes* y *año*.

```
// - Tipos ------
struct Fecha {
   int dia;
   int mes;
   int anyo;
};
```

REGISTROS. EJEMPLO

- Los valores del tipo `Fecha` se componen de tres elementos concretos (de tipo `int` cada uno, aunque pueden ser de tipos diferentes) .

- Los identificadores `dia`, `mes` y `anyo` representan los nombres de sus elementos componentes, denominados **campos**, y su ámbito de visibilidad se restringe a la propia definición.

- Podemos declarar constantes y variables de dicho tipo

```
// - Constantes ----
const Fecha HOY = {10, 10, 2010};
// - Principal ----
int main ()
{
   Fecha f;
}
```

- REGISTROS

  - Los campos de un registro pueden ser de cualquier tipo de datos, simple o estructurado

```
// - Tipos ------
struct Empleado {
   int codigo;
   int sueldo;
   Fecha fecha_ingreso;
};
```

```
// - Tipos ------
struct Tiempo {
   int hor;
   int min;
   int seg;
};
// - Principal ----
int main ()
{
   Empleado e;
   Tiempo t1, t2;
}
```

REGISTROS

- Un objeto de tipo registro puede tratarse como un **todo**, o acceder a cada uno de sus **componentes** (campos).

- Un determinado componente podrá utilizarse en cualquier lugar en que resulten válidas las variables de su mismo tipo.

- Para **acceder** a un determinado componente se nombra el objeto seguido por el identificador del campo correspondiente ambos separados por punto ('.') (del tipo del componente especificado).

```
// - Principal ----
int main ()
{
   Fecha f = HOY;
   f.dia = 10;
   f.mes = 10;
   f.anyo = 2010;
   cout << HOY.dia << '/' << HOY.mes << '/' << HOY.anyo << endl;
}
```

- REGISTROS
  - Operaciones entre registros completos
    - Asignación (=).
    - Paso como parámetro a subprogramas.
    - Devolución por una función. Esta operación, aunque es lícita, no es recomendable.
    - Cualquier otra operación debe ser definida por el programador.

- REGISTROS. EJEMPLO (I)

  - Desarrolle un programa que calcule y muestre la diferencia de tiempo entre dos instantes de tiempo concretos, leídos de teclado, especificados de forma desglosada en *horas*, *minutos* y *segundos*.

REGISTROS. EJEMPLO (I)

```
#include <iostream>
#include <string>
using namespace std;
// - Constantes ----
const int SEGMIN = 60;
const int MINHOR = 60;
const int MAXHOR = 24;
const int SEGHOR = SEGMIN * MINHOR;
// - Tipos ------
struct Tiempo {
    int horas;
    int minutos;
    int segundos;
};
```

```
// - Subalgoritmos --
int leer_rango (int inf, int sup)
{
    int num;
    do {
        cin >> num;
    } while ( ! ((num >= inf) && (num < sup)));
    return num;
}
void leer_tiempo (Tiempo& t)
{
    t.horas = leer_rango(0, MAXHOR);
    t.minutos = leer_rango(0, MINHOR);
    t.segundos = leer_rango(0, SEGMIN);
}
```

```
// - Subalgoritmos --
void escribir_tiempo (const Tiempo& t)
{
    cout << t.horas << ":" << t.minutos << ":" << t.segundos;
}
int tiempo_a_seg (const Tiempo& t)
{
    return (t.horas * SEGHOR) + (t.minutos * SEGMIN) + (t.segundos);
}
void seg_a_tiempo (int sg, Tiempo& t)
{
    t.horas = sg / SEGHOR;
    t.minutos = (sg % SEGHOR) / SEGMIN;
    t.segundos = (sg % SEGHOR) % SEGMIN;
}
void diferencia (const Tiempo& t1, const Tiempo& t2, Tiempo& dif)
{
    seg_a_tiempo(tiempo_a_seg(t2) - tiempo_a_seg(t1), dif);
}
```

```
// - Principal ----
int main ()
{
    Tiempo t1, t2, dif;
    leer_tiempo(t1);
    leer_tiempo(t2);
    diferencia(t1, t2, dif);
    escribir_tiempo(dif);
    cout << endl;
}
```

- REGISTROS. EJEMPLO (II)

  - Desarrolle un programa que permita introducir los datos de dos personas (tanto su nombre, como su fecha de nacimiento) y muestre los datos de la persona que sea de mayor edad.

```
#include <iostream>
#include <string>
using namespace std;
// - Tipos ------
struct Fecha {
   int dia, mes, anyo;
};
struct Persona {
   string nombre;
   Fecha f_nac;
};
```

```
// - Subalgoritmos --
void leer_fecha (Fecha& f)
{
   cin >> f.dia  >>  f.mes  >>  f.anyo;
}
void leer_persona (Persona& p)
{
   cin >> ws;
   getline(cin, p.nombre);
   leer_fecha(p.f_nac);
}
void mostrar_fecha (const Fecha& f)
{
   cout << f.dia  <<  "/"  <<  f.mes  <<  "/"  <<  f.anyo;
}
void mostrar_persona (const Persona& p)
{
   cout << p.nombre << endl;
   mostrar_fecha(p.f_nac);
   cout << endl;
}
```

```
// - Subalgoritmos --
bool es_menor (const Fecha& f1, const Fecha& f2)
{
   bool ok;
   if (f1.anyo < f2.anyo) {
      ok = true;
   } else if (f1.anyo > f2.anyo) {
      ok = false;
   } else if (f1.mes < f2.mes) {
      ok = true;
   } else if (f1.mes > f2.mes) {
      ok = false;
   } else if (f1.dia < f2.dia) {
      ok = true;
   } else if (f1.dia > f2.dia) {
      ok = false;
   } else {
      ok = false;
   }
   return ok;
}
bool es_mayor (const Persona& p1, const Persona& p2)
{
   return es_menor(p1.f_nac, p2.f_nac);
}
```

```
// - Principal ----
int main ()
{
   Persona p1, p2;
   leer_persona(p1);
   leer_persona(p2);
   if (es_mayor(p1, p2)) {
      mostrar_persona(p1);
   } else {
      mostrar_persona(p2);
   }
}
```

Tema 4: TIPOS DE DATOS ESTRUCTURADOS

1. Paso de parámetros de tipos estructurados
2. Cadenas de caracteres
3. Registros
4. Arrays
5. Resolución de problemas usando tipos estructurados
6. Bibliografía: [DALE89a], [JOYA03]

- ARRAYS

  - Colección de un número determinado (definido en tiempo de compilación) de elementos de un **mismo tipo** de datos.

  - Objetivo: Definir una nueva entidad (tipo) como **agregación** de entidades menores del mismo tipo.

  - Se puede **acceder** a cada elemento individual de la colección de forma **parametrizada**.

- ARRAYS

  - En C++ se debe incluir la biblioteca `array`, así como usar el espacio de nombres `std`

  - En la definición de un tipo Array interviene:

    - El **tipo Base** es el tipo de los elementos que constituyen el array. Puede ser cualquier tipo de datos, simple o estructurado.

    - El número (cantidad) de elementos que forman la agregación.

```
#include <iostream>
#include <string>
#include <array>
using namespace std;
// - Constantes ----
const int NELMS = 5;
// - Tipos ------
typedef array<int, NELMS> Vector;
```

```
// - Constantes ----
const Vector PRIMOS = {{ 2, 3, 5, 7, 11 }};
// - Principal ----
int main ()
{
   Vector v;
}
```

- ARRAYS

  - Podemos declarar constantes y variables de tipo array
  - Un objeto de tipo array puede tratarse como un **todo**, o acceder a cada uno de sus **componentes** (elementos).
  - Un determinado elemento podrá utilizarse en cualquier lugar en que resulten válidas las variables de su mismo tipo Base.
  - El número de elementos que componen el array:
    - `v.size()` $\in \mathbb{N}$
    - `int(v.size())` $\in \mathbb{Z}$
  - Acceso al i-ésimo elemento del array (del tipo Base del array):
    - `v[i]` donde `i` $\in$ `[0..v.size() - 1]`
  - En *GNU G++*, la opción de compilación `-D_GLIBCXX_DEBUG` permite comprobar los índices de acceso (descargar la biblioteca de la página web de la asignatura).

ARRAYS. EJEMPLO

```
#include <iostream>
#include <string>
#include <array>
using namespace std;
// - Constantes ----
const int NELMS = 5;
// - Tipos ------
typedef array<int, NELMS> Vector;
// - Constantes ----
const Vector PRIMOS = {{ 2, 3, 5, 7, 11 }};
// - Principal ----
int main ()
{
   Vector v;
   for (int i = 0; i < int(v.size()); ++i) {
      v[i] = PRIMOS[i] * 2;
   }
}
```

- ARRAYS
  - Operaciones entre arrays completos
    - Asignación (`=`).
    - Paso como parámetro a subprogramas.
    - Devolución por una función. Esta operación, aunque es lícita, no es recomendable.
    - Los operadores relacionales (`==` , `!=` , `>` , `>=` , `<` , `<=`) son aplicables si están definidos para los elementos del tipo Base.
    - Cualquier otra operación debe ser definida por el programador.

ARRAYS. EJEMPLO

```
#include <iostream>
#include <string>
#include <array>
using namespace std;
// - Constantes ----
const int NELMS = 5;
// - Tipos ------
typedef array<int, NELMS> Vector;
// - Subalgoritmos --
void leer (Vector& v)
{
    for (int i = 0; i < int(v.size()); ++i) {
        cin >> v[i];
    }
}
int sumar (const Vector& v)
{
    int suma = 0;
    for (int i = 0; i < int(v.size()); ++i) {
        suma += v[i];
    }
    return suma;
}
```

```
// - Principal ----
int main ()
{
    Vector v1, v2;
    leer(v1);
    leer(v2);
    if (sumar(v1) == sumar(v2)) {
        cout << "Misma suma" << endl;
    }
    if (v1 < v2) {
        cout << "Menor" << endl;
    }
    v1 = v2;
    if (v1 == v2) {
        cout << "Iguales" << endl;
    }
}
```

- UTILIDAD DE LOS ARRAYS

  - Los arrays son útiles en todas aquellas circunstancias en que necesitamos tener almacenados una colección de valores (un número fijo predeterminado en tiempo de compilación) a los cuales pretendemos acceder de forma parametrizada, normalmente para aplicar un proceso iterativo.

UTILIDAD DE LOS ARRAYS. EJEMPLO 1

- Ejemplo: programa 1 de agentes de ventas

  Lee las ventas de cada .[a]gente.[ei]mprime su sueldo que se calcula como una cantidad fija (1000€) más un incentivo que será un 10 % de las ventas que ha realizado. El número de agentes es 20.

```
#include <iostream>
#include <string>
using namespace std;
// - Constantes ----
const int NAGENTES = 20;
const double SUELDO_FIJO = 1000.0;
const double INCENTIVO = 10.0;
// - Subalgoritmos --
double porcentaje (double p, double valor)
{
   return (p * valor) / 100.0;
}
// - Principal ----
int main ()
{
   for (int i = 0; i < NAGENTES; ++i) {
      cout << "Introduzca ventas del Agente " << i << ": ";
      double ventas;
      cin >> ventas;
      double sueldo = SUELDO_FIJO + porcentaje(INCENTIVO, ventas);
      cout << "Agente: " << i << " Sueldo: " << sueldo << endl;
   }
}
```

UTILIDAD DE LOS ARRAYS. EJEMPLO 2

- Ejemplo: programa 2 de agentes de ventas

  Lee las ventas de cada .[a]gente.[ei]mprime su sueldo que se calcula como una cantidad fija (1000€) más un incentivo que será un 10 % de las ventas que ha realizado. Dicho incentivo sólo será aplicable a aquellos agentes cuyas ventas superen los 2/3 de la media de ventas del total de los agentes. El número de agentes es 20.

```cpp
#include <iostream>
#include <string>
#include <array>
using namespace std;
// - Constantes ----
const int NAGENTES = 20;
const double SUELDO_FIJO = 1000.0;
const double INCENTIVO = 10.0;
const double PROMEDIO = 2.0 / 3.0;
// - Tipos ------
typedef array<double, NAGENTES> Ventas;
// - Subalgoritmos --
double calc_media (const Ventas& v)
{
    double suma = 0.0;
    for (int i = 0; i < int(v.size()); ++i) {
        suma += v[i];
    }
    return suma / double(v.size());
}
```

```cpp
// - Subalgoritmos --
double porcentaje (double p, double valor)
{
    return (p * valor) / 100.0;
}
void leer_ventas (Ventas& v)
{
    for (int i = 0; i < int(v.size()); ++i) {
        cout << "Introduzca ventas del Agente "
             << i << ": ";
        cin >> v[i];
    }
}
```

```
// - Subalgoritmos --
void imprimir_sueldos (const Ventas& v)
{
    double umbral = PROMEDIO * calc_media(v);
    for (int i = 0; i < int(v.size()); ++i) {
        double sueldo = SUELDO_FIJO;
        if (v[i] >= umbral) {
            sueldo += porcentaje(INCENTIVO, v[i]);
        }
        cout << "Agente: " << i << " Sueldo: " << sueldo << endl;
    }
}
// - Principal ----
int main ()
{
    Ventas ventas;
    leer_ventas(ventas);
    imprimir_sueldos(ventas);
}
```

LISTAS o SECUENCIAS DE ELEMENTOS

- Hay situaciones en las que debemos almacenar una **lista** o **secuencia** de elementos, donde el número de elementos puede **variar** durante la ejecución del programa, pero nunca sobrepasará un determinado límite máximo.

- Usualmente, la opción más adecuada para gestionar esta estructura de datos suele ser definir un tipo registro que contenga:

  - El número de elementos que actualmente contiene la lista.

  - Un array, del tamaño adecuado al límite máximo de la lista, que almacene los elementos consecutivamente al principio.

- Tiene diversas denominaciones, entre otras, arrays incompletos, listas o secuencias con número variable de elementos, etc. Hay que entender el concepto, que puede surgir en diferentes contextos y denominaciones, y aplicarlo adecuadamente.

UTILIDAD DE LAS LISTAS. EJEMPLO 3

- Ejemplo: programa 3 de agentes de ventas

  Lee las ventas de cada .[a]gente.[ei]mprime su sueldo que se calcula como una cantidad fija (1000€) más un incentivo que será un 10 % de las ventas que ha realizado. Dicho incentivo sólo será aplicable a aquellos agentes cuyas ventas superen los 2/3 de la media de ventas del total de los agentes. El número **máximo** de agentes es 20, pero el número **actual** de agentes será leído de teclado.

```
#include <iostream>
#include <string>
#include <array>
using namespace std;
// - Constantes ----
const int MAX_AGENTES = 20;
const double SUELDO_FIJO = 1000.0;
const double INCENTIVO = 10.0;
const double PROMEDIO = 2.0 / 3.0;
// - Tipos ------
typedef array<double, MAX_AGENTES> Datos;
struct Ventas {
   int nelms;
   Datos elm;
};
// - Subalgoritmos --
double calc_media (const Ventas& v)
{
   double suma = 0.0;
   for (int i = 0; i < v.nelms; ++i) {
      suma += v.elm[i];
   }
   return suma / double(v.nelms);
}
```

```
// - Subalgoritmos --
double porcentaje (double p, double valor)
{
   return (p * valor) / 100.0;
}
void imprimir_sueldos (const Ventas& v)
{
   double umbral = PROMEDIO * calc_media(v);
   for (int i = 0; i < v.nelms; ++i) {
      double sueldo = SUELDO_FIJO;
      if (v.elm[i] >= umbral) {
         sueldo += porcentaje(INCENTIVO,
                              v.elm[i]);
      }
      cout << "Agente: " << i
           << " Sueldo: " << sueldo << endl;
   }
}
```

```
// - Subalgoritmos --
void leer_ventas (Ventas& v)
{
    int nag;
    cout << "Introduzca total de agentes: ";
    cin >> nag;
    if (nag > int(v.elm.size())) {
        v.nelms = 0;
        cout << "Error" << endl;
    } else {
        v.nelms = nag;
        for (int i = 0; i < v.nelms; ++i) {
            cout << "Introduzca ventas agente "
                 << i << ": ";
            cin >> v.elm[i];
        }
    }
}
// - Principal ----
int main ()
{
    Ventas ventas;
    leer_ventas(ventas);
    imprimir_sueldos(ventas);
}
```

```
// - Subalgoritmos --
void leer_ventas_alternativo (Ventas& v)
{
    int valor;
    v.nelms = 0;
    cout << "Intr. ventas agente (0 fin): ";
    cin >> valor;
    while ((v.nelms < int(v.elm.size()))
           && (valor > 0)) {
        v.elm[v.nelms] = valor;
        ++v.nelms;
        if (v.nelms < int(v.elm.size())) {
            cout << "Intr. ventas agente(0 fin):";
            cin >> valor;
        }
    }
}
```

UTILIDAD DE LAS LISTAS. EJEMPLO 4

- Ejemplo: programa 4 de agentes de ventas

  Lee el nombre y las ventas de cada .ªgente", calcula su sueldo como una cantidad fija (1000 €) más un incentivo que será un 10 % de las ventas que ha realizado. Dicho incentivo sólo será aplicable a aquellos agentes cuyas ventas superen los 2/3 de la media de ventas del total de los agentes. Finalmente muestra los datos de los agentes (nombre, ventas y sueldo). El número **máximo** de agentes es 20, pero el número **actual** de agentes será leído de teclado.

```cpp
#include <iostream>
#include <string>
#include <array>
using namespace std;
// - Constantes ----
const int MAX_AGENTES = 20;
const double SUELDO_FIJO = 1000.0;
const double INCENTIVO = 10.0;
const double PROMEDIO = 2.0 / 3.0;
// - Tipos ------
struct Agente {
   string nombre;
   double ventas;
   double sueldo;
};
typedef array<Agente, MAX_AGENTES> AAgentes;
struct LAgentes {
   int nags;
   AAgentes ag;
};
// - Subalgoritmos --
double porcentaje (double p, double valor)
{
   return (p * valor) / 100.0;
}
// - Subalgoritmos --
void leer_agente (Agente& a)
{
   cout << "Introduzca el nombre del agente: ";
   cin >> ws;
   getline(cin, a.nombre);
   cout << "Introduzca las ventas del agente: ";
   cin >> a.ventas;
   a.sueldo = 0;
}
void leer_agentes (LAgentes& v)
{
   int nag;
   cout << "Introduzca total de agentes: ";
   cin >> nag;
   if (nag > int(v.ag.size())) {
      v.nags = 0;
      cout << "Error" << endl;
   } else {
      v.nags = nag;
      for (int i = 0; i < v.nags; ++i) {
         leer_agente(v.ag[i]);
      }
   }
}
```

```
// - Subalgoritmos --
double calc_media (const LAgentes& v)
{
   double suma = 0.0;
   for (int i = 0; i < v.nags; ++i) {
      suma += v.ag[i].ventas;
   }
   return suma / double(v.nags);
}
void calcular_sueldo (Agente& a, double m)
{
   a.sueldo = SUELDO_FIJO;
   if (a.ventas >= PROMEDIO * m) {
      a.sueldo += porcentaje(INCENTIVO,a.ventas);
   }
}
void calcular_sueldos (LAgentes& v)
{
   double media = calc_media(v);
   for (int i = 0; i < v.nags; ++i) {
      calcular_sueldo(v.ag[i], media);
   }
}
```

```
// - Subalgoritmos --
void mostrar_agente (const Agente& a)
{
   cout << "Nombre: " << a.nombre << endl;
   cout << "Ventas: " << a.ventas << endl;
   cout << "Sueldo: " << a.sueldo << endl;
}
void mostrar_agentes (const LAgentes& v)
{
   for (int i = 0; i < v.nags; ++i) {
      mostrar_agente(v.ag[i]);
   }
}
// - Principal ----
int main ()
{
   LAgentes agentes;
   leer_agentes(agentes);
   calcular_sueldos(agentes);
   mostrar_agentes(agentes);
}
```

ARRAYS MULTIDIMENSIONALES

- El **tipo Base** de un array puede ser tanto simple como estructurado, por lo tanto puede ser otro *array*, dando lugar a arrays con **múltiples dimensiones**.

- Así, cada elemento de un array puede ser a su vez un array.

```
#include <iostream>
#include <string>
#include <array>
using namespace std;
// - Constantes ----
const int NFILAS = 3;
const int NCOLUMNAS = 5;
// - Tipos ------
typedef array<int, NCOLUMNAS> Fila;
typedef array<Fila, NFILAS> Matriz;
```

```
// - Principal ----
int main ()
{
   Matriz m;
   for (int f = 0; f < int(m.size()); ++f) {
      for (int c = 0; c < int(m[f].size()); ++c) {
         m[f][c] = (f * m[0].size()) + c;
      }
   }
   Fila fil = m[0];
   int n = m[2][4];
}
```

ARRAYS MULTIDIMENSIONALES. EJEMPLO

- Algoritmo que lee una matriz 3x5 de enteros (fila a fila), almacenandolos en un array bidimensional .a", Finalmente imprime la matriz según el siguiente formato:

a	a	a	a	a	b
a	a	a	a	a	b
a	a	a	a	a	b
c	c	c	c	c	

donde "brepresenta el resultado de sumar todos los elementos de la fila y çrepresenta el resultado de sumar todos los elementos de la columna donde se encuentran.

```cpp
#include <iostream>
#include <string>
#include <array>
using namespace std;
// - Constantes ----
const int NFILAS = 3;
const int NCOLUMNAS = 5;
// - Tipos ------
typedef array<int, NCOLUMNAS> Fila;
typedef array<Fila, NFILAS> Matriz;
// - Subalgoritmos --
```

```cpp
// - Subalgoritmos --
int sumar_fila (const Fila& fil)
{
    int suma = 0;
    for (int c = 0; c < int(fil.size()); ++c) {
        suma += fil[c];
    }
    return suma;
}
int sumar_columna (const Matriz& m, int c)
{
    int suma = 0;
    for (int f = 0; f < int(m.size()); ++f) {
        suma += m[f][c];
    }
    return suma;
}
```

```
// - Subalgoritmos --
void escribir_fila (const Fila& fil)
{
   for (int c = 0; c < int(fil.size()); ++c) {
      cout << fil[c] << " ";
   }
}
void escribir_matriz_formato (const Matriz& m)
{
   for (int f = 0; f < int(m.size()); ++f) {
      escribir_fila(m[f]);
      cout << sumar_fila(m[f]);
      cout << endl;
   }
   for (int c = 0; c < int(m[0].size()); ++c) {
      cout << sumar_columna(m, c) << " ";
   }
   cout << endl;
}
void leer_matriz (Matriz& m)
{
   cout << "Escribe fila a fila" << endl;
   for (int f = 0; f < int(m.size()); ++f) {
      for (int c = 0; c < int(m[f].size()); ++c) {
         cin >> m[f][c];
      }
   }
}
```

```
// - Principal ----
int main ()
{
   Matriz m;
   leer_matriz(m);
   escribir_matriz_formato(m);
}
```

ARRAYS CONSTANTES. EJEMPLO (I)

- Array constante de cadenas de caracteres.

```
#include <iostream>
#include <string>
#include <array>
using namespace std;
// - Constantes ----
const int NELMS = 7;
// - Tipos ------
typedef array<string, NELMS> Vector;
// - Constantes ----
const Vector DIAS = {{
   "Lunes", "Martes", "Miercoles", "Jueves", "Viernes", "Sabado", "Domingo"
}};
// - Principal ----
int main ()
{
   // ...
}
```

ARRAYS CONSTANTES. EJEMPLO (II)

- Array constante de personas.

```
// - Constantes ----
const int NELMS = 3;
// - Tipos ------
struct Fecha {
   int dia;
   int mes;
   int anyo;
};
struct Persona {
   string nombre;
   Fecha fnac;
};
typedef array<Persona, NELMS> Datos;
```

```
// - Constantes ----
const Datos DATOS = {{
   { "Lola", { 20, 4, 2010 } } ,
   { "Pepe", { 12, 8, 2011 } } ,
   { "Luis", { 24, 9, 2012 } }
}};
// - Principal ----
int main ()
{
   // ...
}
```

ARRAYS CONSTANTES. EJEMPLO (III)

- Array constante de dos dimensiones.

```
#include <iostream>
#include <string>
#include <array>
using namespace std;
// - Constantes ----
const int NFILAS = 3;
const int NCOLUMNAS = 4;
// - Tipos ------
typedef array<int, NCOLUMNAS> Fila;
typedef array<Fila, NFILAS> Matriz;
// - Constantes ----
const Matriz MATRIZ = {{
   {{ 00, 01, 02, 03 }} ,
   {{ 10, 11, 12, 13 }} ,
   {{ 20, 21, 22, 23 }}
}};
```

Tema 4: TIPOS DE DATOS ESTRUCTURADOS

1. Paso de parámetros de tipos estructurados
2. Cadenas de caracteres
3. Registros
4. Arrays
5. Resolución de problemas usando tipos estructurados
6. Bibliografía: [DALE89a], [JOYA03]

- RESOLUCIÓN DE PROBLEMAS USANDO TIPOS ESTRUCTURADOS

  - Problema 1: Producto de 2 matrices (máximo 10x10).

```
#include <iostream>
#include <string>
#include <cassert>
#include <array>
using namespace std;
// - Constantes ----
const int MAX = 10;
// - Tipos ------
typedef array<double, MAX> Fila;
typedef array<Fila, MAX> Tabla;
struct Matriz {
   int n_fil;
   int n_col;
   Tabla datos;
};
```

```
// - Subalgoritmos --
void leer_matriz (Matriz& m)
{
    cout << "Dimensiones?: ";
    cin >> m.n_fil >> m.n_col;
    cout << "Escribe valores fila a fila:" << endl;
    for (int f = 0; f < m.n_fil; ++f) {
        for (int c = 0; c < m.n_col; ++c) {
            cin >> m.datos[f][c];
        }
    }
}
```

```
// - Subalgoritmos --
void escribir_matriz (const Matriz& m)
{
    for (int f = 0; f < m.n_fil; ++f) {
        for (int c = 0; c < m.n_col; ++c) {
            cout << m.datos[f][c] << " ";
        }
        cout << endl;
    }
}
```

```
// - Subalgoritmos --
double suma_fila_por_col (const Matriz& x, const Matriz& y, int f, int c)
{
    assert(x.n_col == y.n_fil); // PRECOND
    double suma = 0.0;
    for (int k = 0; k < x.n_col; ++k) {
        suma += x.datos[f][k] * y.datos[k][c];
    }
    return suma;
}
```

```
// - Subalgoritmos --
void mult_matriz (Matriz& m, const Matriz& a, const Matriz& b)
{
   assert(a.n_col == b.n_fil); // PRECOND
   m.n_fil = a.n_fil;
   m.n_col = b.n_col;
   for (int f = 0; f < m.n_fil; ++f) {
      for (int c = 0; c < m.n_col; ++c) {
         m.datos[f][c] = suma_fil_por_col(a, b, f, c);
      }
   }
}
// - Principal ----
int main ()
{
   Matriz a,b,c;
   leer_matriz(a);
   leer_matriz(b);
   if (a.n_col != b.n_fil) {
      cout << "No se puede multiplicar." << endl;
   } else {
      mult_matriz(c, a, b);
      escribir_matriz(c);
   }
}
```

- APLICACIÓN: Agenda Personal

  - La información personal que será almacenada es la siguiente: Nombre, Teléfono, Dirección, Calle, Número, Piso, Código Postal y Ciudad

  - Las operaciones a realizar con dicha agenda serán:

    1. Añadir los datos de una persona
    2. Acceder a los datos de una persona a partir de su nombre.
    3. Borrar una persona a partir de su nombre.
    4. Modificar los datos de una persona a partir de su nombre.
    5. Listar el contenido completo de la agenda.

```
#include <iostream>
#include <string>
#include <cassert>
#include <array>
using namespace std;
// - Constantes ----
const int MAX_PERSONAS = 50;
const int OK = 0;
const int AG_LLENA = 1;
const int NO_ENCONTRADO = 2;
const int YA_EXISTE = 3;
// - Tipos ------
struct Direccion {
    int num;
    string calle;
    string piso;
    string cp;
    string ciudad;
};
struct Persona {
    string nombre;
    string tel;
    Direccion direccion;
};
```

```
// - Tipos ------
typedef array<Persona, MAX_PERSONAS> Personas;
struct Agenda {
    int n_pers;
    Personas pers;
};
// - Subalgoritmos --
void Inicializar (Agenda& ag)
{
    ag.n_pers = 0;
}
```

```
// - Subalgoritmos --
void Leer_Direccion (Direccion& dir)
{
    cin >> ws;
    getline(cin, dir.calle);
    cin >> dir.num;
    cin >> dir.piso;
    cin >> dir.cp;
    cin >> ws;
    getline(cin, dir.ciudad);
}
//---------------------------
void Leer_Persona (Persona& per)
{
    cin >> ws;
    getline(cin, per.nombre);
    cin >> per.tel;
    Leer_Direccion(per.direccion);
}
```

```
// - Subalgoritmos --
void Escribir_Direccion (const Direccion& dir)
{
    cout << dir.calle << ", ";
    cout << dir.num << " ";
    cout << dir.piso << endl;
    cout << dir.cp << " ";
    cout << dir.ciudad << endl;
}
//---------------------------
void Escribir_Persona (const Persona& per)
{
    cout << per.nombre << endl;
    cout << per.tel << endl;
    Escribir_Direccion(per.direccion);
}
```

```
// - Subalgoritmos --
// Busca una Persona en la Agenda
// Devuelve su posicion si se encuentra, o bien  >= ag.n_pers en otro caso
int Buscar_Persona (const string& nombre, const Agenda& ag)
{
   int i = 0;
   while ((i < ag.n_pers)  && (nombre != ag.pers[i].nombre)) {
      ++i;
   }
   return i;
}
//---------------------------
void Anyadir (Agenda& ag, const Persona& per)
{
   ag.pers[ag.n_pers] = per;
   ++ag.n_pers;
}
//---------------------------
void Eliminar (Agenda& ag, int pos)
{
   ag.pers[pos] = ag.pers[ag.n_pers - 1];
   --ag.n_pers;
}
```

```
// - Subalgoritmos --
void Anyadir_Persona (const Persona& per, Agenda& ag, int& ok)
{
    int i = Buscar_Persona(per.nombre, ag);
    if (i < ag.n_pers) {
        ok = YA_EXISTE;
    } else if (ag.n_pers == int(ag.pers.size())) {
        ok = AG_LLENA;
    } else {
        ok = OK;
        Anyadir(ag, per);
    }
}
//---------------------------
void Borrar_Persona (const string& nombre, Agenda& ag, int& ok)
{
    int i = Buscar_Persona(nombre, ag);
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO;
    } else {
        ok = OK;
        Eliminar(ag, i);
    }
}
```

```
// - Subalgoritmos --
void Modificar_Persona (const string& nombre, const Persona& nuevo, Agenda& ag, int& ok)
{
    int i = Buscar_Persona(nombre, ag);
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO;
    } else {
        Eliminar(ag, i);
        Anyadir_Persona(nuevo, ag, ok);
    }
}
//---------------------------
void Imprimir_Persona (const string& nombre, const Agenda& ag, int& ok)
{
    int i = Buscar_Persona(nombre, ag);
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO;
    } else {
        ok = OK;
        Escribir_Persona(ag.pers[i]);
    }
}
```

```
// - Subalgoritmos --
void Imprimir_Agenda (const Agenda& ag, int& ok)
{
   for (int i = 0; i < ag.n_pers; ++i) {
      Escribir_Persona(ag.pers[i]);
   }
   ok = OK;
}
//---------------------------
char Menu ()
{
   char opcion;
   cout << endl;
   cout << "a. -  Anadir Persona" << endl;
   cout << "b. -  Buscar Persona" << endl;
   cout << "c. -  Borrar Persona" << endl;
   cout << "d. -  Modificar Persona" << endl;
   cout << "e. -  Imprimir Agenda" << endl;
   cout << "x. -  Salir" << endl;
   do {
      cout << "Introduzca Opcion: ";
      cin >> opcion;
   } while ( ! (((opcion  >=  'a')  && (opcion  <=  'e'))  ||  (opcion  ==  'x')));
   return opcion;
}
```

```
// - Subalgoritmos --
void Escribir_Cod_Error (int cod)
{
    switch (cod) {
    case OK:
        cout << "Operacion correcta" << endl;
        break;
    case AG_LLENA:
        cout << "Agenda llena" << endl;
        break;
    case NO_ENCONTRADO:
        cout << "La persona no se encuentra en la agenda" << endl;
        break;
    case YA_EXISTE:
        cout << "La persona ya se encuentra en la agenda" << endl;
        break;
    }
}
```

```
// - Principal ----
int main ()
{
    Agenda ag;
    char opcion;
    Persona per;
    string nombre;
    int ok;
    Inicializar(ag);
    do {
        opcion = Menu();
        switch (opcion) {
        case 'a':
            cout << "Introduzca los datos de la Persona" << endl;
            cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl;
            Leer_Persona(per);
            Anyadir_Persona(per, ag, ok);
            Escribir_Cod_Error(ok);
            break;
        case 'b':
            cout << "Introduzca Nombre" << endl;
            cin >> nombre;
            Imprimir_Persona(nombre, ag, ok);
            Escribir_Cod_Error(ok);
            break;
```

```
        case 'c':
            cout << "Introduzca Nombre" << endl;
            cin >> nombre;
            Borrar_Persona(nombre, ag, ok);
            Escribir_Cod_Error(ok);
            break;
        case 'd':
            cout << "Introduzca Nombre" << endl;
            cin >> nombre;
            cout << "Nuevos datos de la Persona" << endl;
            cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl;
            Leer_Persona(per);
            Modificar_Persona(nombre, per, ag, ok);
            Escribir_Cod_Error(ok);
            break;
        case 'e':
            Imprimir_Agenda(ag, ok);
            Escribir_Cod_Error(ok);
            break;
        }
    } while (opcion  !=  'x' );
}
```

Tema 4: TIPOS DE DATOS ESTRUCTURADOS

1. Paso de parámetros de tipos estructurados
2. Cadenas de caracteres
3. Registros
4. Arrays
5. Resolución de problemas usando tipos estructurados
6. Bibliografía: [DALE89a], [JOYA03]