Tema 4. Tipos de Datos Compuestos

Vicente Benjumea García

Introducción a la Programación
Departamento de Lenguajes y Ciencias de la Computación.
E.T.S.I. Informática. Univ. de Málaga.

Tema 4. Tipos de datos compuestos

- Tipos de datos.
- Paso de Parámetros de Tipos Compuestos.
- Cadenas de caracteres.
  - El bucle *for-each* en los `string`
- Registros.
- Arrays.
  - Arrays Predefinidos y Arrays de la Biblioteca
  - El bucle *for-each* en los arrays
  - Arrays incompletos: Listas con cantidad variable de elementos.
    - Operaciones básicas con listas
  - Arrays multidimensionales.
- Definición de constantes de tipos compuestos.
- Resolución de problemas usando tipos compuestos.

Tipos de Datos

- El **tipo de datos** define las características de los objetos: conjunto de valores que pueden tomar, operaciones que se pueden aplicar, espacio de almacenamiento, etc.
- Los **tipos de datos simples** definen conjuntos de valores formados por elementos **indivisibles** y **ordenados**.
  - Predefinidos: **bool**, **char**, **unsigned**, **int**, **double**, *etc*.
  - Definidos por el programador:
    - **enum** (tipo enumerado): define un nuevo tipo donde el conjunto de valores se especifica como una secuencia de símbolos (enumeración).
- Los **tipos de datos compuestos** (estructurados) definen conjuntos de valores formados por elementos **compuestos** que se pueden manipular como un **todo** o a partir de sus **componentes individuales**.
  - Predefinidos: el tipo **string** *(cadena de caracteres)* define una *secuencia de caracteres*, con acceso **parametrizado** a los elementos de tipo **char**.
  - Definidos por el programador:
    - **struct** (registro o estructura): define un nuevo tipo como una **agrupación** de varios componentes que pueden ser de **tipos distintos**, adecuado para procesamiento **individualizado**.
    - **array**: define un nuevo tipo como una **colección** de múltiples elementos del **mismo tipo**, con acceso **parametrizado**, adecuado para procesamiento **iterativo**.

Paso de Parámetros de Tipos Compuestos

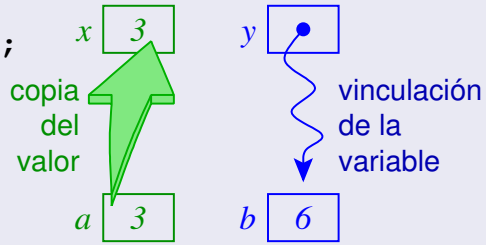
- Con los tipos simples, se utiliza el **paso por valor** y el **paso por referencia** para implementar la transferencia de información entre subprogramas.
- Para la transferencia de información de **entrada**, el **paso por valor** supone **duplicar la memoria y copiar el valor** del parámetro actual en el formal.
- En el caso de tipos **simples**, el paso por valor es adecuado, sin embargo, si el tipo del parámetro es **compuesto**, es posible que dicha **copia** implique una **alta sobrecarga**, tanto en espacio de memoria como en tiempo de ejecución.
- El **paso por referencia constante** de tipos **compuestos** permite realizar la transferencia de información de *entrada* de forma **eficiente**.
- Todos los **parámetros de entrada** de **tipos compuestos** (**struct**, **array**, **string**) se realizarán mediante el **paso por referencia constante**.

	Tipos Simples		Tipos Compuestos	
	(⇓)Ent	(⇑)Sal(⇕)E/S	(⇓)Ent	(⇑)Sal(⇕)E/S
Par. Formal	P.Valor (int x)	P.Referencia (int& x)	P.Ref.Cte (const Per& x)	P.Referencia (Per& x)

Paso de Parámetros de Tipos Compuestos

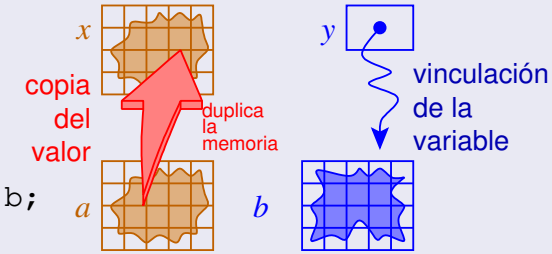
El **paso por valor** de tipos **simples** permite realizar la transferencia de información de *entrada* de forma **eficiente**.

```
void subprograma(int x, int& y)
{
    y = 2 * x;
}

int main()
{
    int a, b;
    a = 3;
    subprograma(a, b);
}
```



El **paso por valor** de tipos **compuestos** conlleva una **alta sobrecarga**, ya que se debe duplicar la memoria y copiar el contenido (de gran tamaño).

```
void subprograma(Datos x, Datos& y)
{
    // ...
}

int main()
{
    Datos a, b;
    // ...
    subprograma(a, b);
}
```



Todos los **parámetros de entrada** de **tipos simples** se realizarán mediante el **paso por valor**.

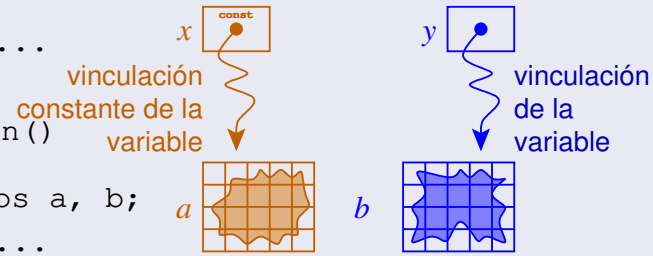
Todos los **parámetros de entrada** de **tipos compuestos** (**struct**, **array**, **string**) se realizarán mediante el **paso por referencia constante**.

> *El **paso por valor** de tipos **compuestos** **no se debe utilizar**, ya que tiene una **alta sobrecarga**.*

Todos los **parámetros de salida o entrada/salida** se realizarán mediante el **paso por referencia**.

El **paso por referencia constante** de tipos **compuestos** permite realizar la transferencia de información de *entrada* de forma **eficiente**.

```
void subprograma(const Datos& x, Datos& y)
{
    // ...
}

int main()
{
    Datos a, b;
    // ...
    subprograma(a, b);
}
```



Cadenas de Caracteres

- Las **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.

- Las **cadenas de caracteres** se representan, almacenan y manipulan mediante el tipo `string` proporcionado por la biblioteca estándar de C++.

El Tipo `string`

- Se debe incluir la biblioteca **`<string>`**, y usar el espacio de nombres **`std`**.
- El tipo **`string`** permite almacenar y manipular **cadenas de caracteres**.
- El tipo **`string`** representa una secuencia de caracteres de **longitud finita limitada** por la implementación.
- Podemos definir tanto **constantes** como **variables** y **parámetros**.
  - Las variables de tipo `string` se crean con el valor inicial de la cadena vacía (`""`).
- Una cadena de caracteres **literal** se representa mediante una sucesión de caracteres entre **comillas dobles**: **`"ejemplo"`**. La cadena de caracteres vacía se representa así: **`""`**.

```
#include <iostream>
#include <string>
using namespace std;

const string AUTOR = "Jose Luis";

int main()
{
    string nombre = "Jose Luis Lopez";
    string direccion;                    // cadena vacía ("") por defecto
}
```

Operaciones con Cadenas de Caracteres

- Asignación de *strings* completos (=).
- Paso como parámetro a subprogramas (⇑⇕*referencia* y ⇓*referencia constante*).
- Devolución por una función.
- Comparación lexicográfica (diccionario) (== , != , > , >= , < , <=).
- Concatenación de cadenas y caracteres (+ , +=):

> La concatenación (+ , +=) es **muy importante** para **añadir** caracteres a una cadena

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

    nombre = "Pepe" + "Luis";  // ERROR no es posible concatenar dos ctes "..."
    nombre += 1234;            // ERROR no es posible concatenar string con número
}
```

Operaciones con Cadenas de Caracteres

- **Longitud**. Cantidad de caracteres que componen la cadena **`nombre`**:
  - `nombre.size()` ∈ `unsigned`
  - `int(nombre.size())` ∈ `int`
- **Acceso al iésimo** carácter de la cadena **`nombre`** (de tipo **`char`**):
  - **`nombre[i]`** donde `i` ∈ { `0...nombre.size()-1` }

```
int main()
{
    string nombre = "Jose Luis";
    char primera_letra = nombre[0];                    // 'J'
    char ultima_letra  = nombre[nombre.size() - 1];    // 's'
    for (unsigned i = 0; i < nombre.size(); ++i) {
        nombre[i] = 'x';
    }
    for (int i = 0; i < int(nombre.size()); ++i) {
        cout << nombre[i] ;
    }
}
```

- En *Code::Blocks*, definir el símbolo `_GLIBCXX_DEBUG` en el apartado de `#defines` de las opciones de compilación permite comprobar los índices de acceso.

Operaciones con Cadenas de Caracteres

- **Cambiar la longitud** (la cantidad de caracteres) de una cadena:
  - `nombre.resize(sz)` cambia la cantidad de caracteres de `nombre` al valor especificado por `sz`. Si extiende, entonces rellena con el carácter *nulo*.
  - `nombre.resize(sz, 'x')` cambia la cantidad de caracteres de `nombre` al valor especificado por `sz`. Si extiende, entonces rellena con el carácter `'x'`.
- **Obtener** una nueva **subcadena** (de tipo `string`):
  - `nombre.substr(i, sz)` • `nombre.substr(i)`
    donde `i` ∈ { `0...nombre.size()` }
  - No es válida la asignación a una subcadena:

```
nombre.substr(i, sz) = "hola"; // Error
```

```
int main()
{
    string nombre = "Jose Luis";
    nombre.resize(4);                 // nombre == "Jose"
    nombre.resize(9, 'X');            // nombre == "JoseXXXXX"
    nombre.resize(nombre.size()-1); // nombre == "JoseXXXX"
    cout << nombre.substr(4) ;        // "XXXX"
    cout << nombre.substr(0, 4) ;     // "Jose"
    cout << nombre.substr(3, 2) ;     // "eX"
    cout << nombre.substr(3, 20) ;    // "eXXXX""
}
```

Operaciones con Cadenas de Caracteres. Comparación Lexicográfica

- Además de los operadores relacionales (**==** , **!=** , **>** , **>=** , **<** , **<=**) que comparan lexicograficamente (diccionario).
- El método **compare** también realiza una comparación lexicográfica, y devuelve tres posibles valores (negativo, cero, positivo):

```
int res = nombre1.compare(nombre2);
```

  - si **nombre1 < nombre2** entonces **res < 0**
  - si **nombre1 == nombre2** entonces **res == 0**
  - si **nombre1 > nombre2** entonces **res > 0**

```
#include <iostream>
#include <string>
using namespace std;
const string AUTOR = "Jose Luis";
int main()
{
    string nombre = AUTOR + "Lopez";
    if (nombre >= AUTOR) { /* ... */ }
    if (nombre.compare(AUTOR) >= 0) { /* ... */ }
}
```

Operaciones con Cadenas de Caracteres. Conversiones

- Conversión entre los tipos básicos y el tipo **string**:

```
int           valor = stoi("1234");       // Conversión de string a int
unsigned long valor = stoul("1234567");   // Conversión de string a ulong
double        valor = stod("123.456e7"); // Conversión de string a double

string cad = to_string(1234);             // Conversión de int a string
string cad = to_string(1234567);          // Conversión de unsigned a string
string cad = to_string(123.456e7);        // Conversión de double a string
```

```
int main()
{
    string nombre = "Jose Luis";
    int numero = 1234;
    nombre += to_string(numero);              // Jose Luis1234
    int valor = stoi(nombre.substr(10, 3));   // 234
    int valor = stoi("1234");                 // 1234
    int valor = stoi(" 1234 ");               // 1234
    int valor = stoi("1234xxx");              // 1234
    int valor = stoi("xxx1234xxx");           // Error
}
```

Entrada y Salida Básica de Cadenas de Caracteres

Salida de Datos <<

El **operador de salida** (<<) permite **mostrar** en el flujo de salida a pantalla (`cout`) el valor de una secuencia de caracteres de tipo **string**, o una secuencia de caracteres constante entre **comillas dobles** (`endl` ≡ *Salto-de-Línea*).
Ej: `cout << "Valor: " << nombre << " " << apellido << endl ;`

Entrada de Datos >>

El **operador de entrada** (>>) permite asignar a una **variable** de tipo **string** una secuencia de caracteres **delimitada por espacios en blanco** desde el flujo de entrada de teclado (`cin`). Ej: `cin >> nombre >> apellido ;`

- La entrada de datos (>>) se realiza **eliminado** los espacios en **blanco** y saltos de línea **iniciales** hasta encontrar algún carácter distinto de ellos.
- Después se **lee** la secuencia de caracteres hasta encontrar algún espacio en blanco o salto de línea y se **almacena** en la variable de tipo **string**.

Los espacios **delimitadores** se componen del espacio en blanco, el tabulador (`'\t'`) y el salto de línea (`'\n'`).

Entrada de Cadenas de Caracteres Avanzada

Leer línea

- La sentencia **`getline(cin, secuencia)`**, **sin eliminar** los espacios en blanco ni saltos de línea **iniciales**, **lee** del flujo de entrada (`cin`) la siguiente secuencia de caracteres hasta encontrar un **salto de línea** (que es eliminado) y la almacena en la variable de tipo **`string`**. Por ej.: `getline(cin, linea);`
- La sentencia **`getline(cin, secuencia, delim)`**, **sin eliminar** los espacios en blanco ni saltos de línea **iniciales**, **lee** del flujo de entrada (`cin`) la siguiente secuencia de caracteres hasta encontrar un carácter igual al **delimitador** especificado (de tipo **char**), que es eliminado, y la almacena en la variable de tipo **`string`**. Por ej.: `getline(cin, linea, ';');`

Eliminar caracteres

- La sentencia `cin >> ws` **elimina** los espacios en **blanco** y saltos de línea **iniciales** hasta encontrar algún carácter distinto de ellos. Por ej.: `cin >> ws;`
- La sentencia `cin.ignore(num, delim)` **elimina** los caracteres **iniciales** hasta un máximo de `num` caracteres, o hasta encontrar un carácter igual al delimitador especificado. Por ej.: `cin.ignore(100, '\n');`

Entrada de Cadenas de Caracteres Avanzada

Mezclar la entrada de datos del **operador de entrada** (>>) con la operación `getline()` causa **problemas**, ya que el operador (>>) deja en el buffer de entrada los delimitadores (espacios y saltos de línea) que posteriormente podrían afectar a `getline()`, ya que éste no elimina dichos delimitadores iniciales.

```
int main()
{
  string nombre;
  int edad;
  for (int i = 0; i < 5; ++i) {
    cout << "Introduce nombre: ";
    getline(cin, nombre);
    cout << "Introduce edad: ";
    cin >> edad ;
    cout << "Nombre: [" << nombre << "]"
         << " Edad: " << edad << endl;
  }
}
```

```
Introduce nombre: pepe luis
Introduce edad: 21
Nombre: [pepe luis] Edad: 21
Introduce nombre: Introduce edad: 32
Nombre: [] Edad: 32
Introduce nombre: Introduce edad: 45
Nombre: [] Edad: 45
Introduce nombre: Introduce edad: 12
Nombre: [] Edad: 12
Introduce nombre: Introduce edad: 23
Nombre: [] Edad: 23
```

Entrada de Cadenas de Caracteres Avanzada

El problema de mezclar la entrada de datos del **operador de entrada** (>>) con la operación `getline()` se resuelve utilizando `cin >> ws` **antes** de `getline()`, o utilizando `cin.ignore()` **después** del operador >>.

```
int main()
{
  string nombre;
  int edad;
  for (int i = 0; i < 5; ++i) {
    cout << "Introduce nombre: ";
 ▶▶ cin >> ws; ◀◀
    getline(cin, nombre);
    cout << "Introduce edad: ";
    cin >> edad ;
    cout << "Nombre: [" << nombre << "]"
         << " Edad: " << edad << endl;
  }
}
```

```
int main()
{
  string nombre;
  int edad;
  for (int i = 0; i < 5; ++i) {
    cout << "Introduce nombre: ";
    getline(cin, nombre);
    cout << "Introduce edad: ";
    cin >> edad ;
 ▶▶ cin.ignore(100, '\n'); ◀◀
    cout << "Nombre: [" << nombre << "]"
         << " Edad: " << edad << endl;
  }
}
```

Salvo algunas excepciones (en el caso de tener que leer una cadena de caracteres **vacía**), se recomienda que **siempre** se **eliminen** los espacios en **blanco** y saltos de línea **iniciales** (`cin >> ws`) antes de leer una cadena de caracteres con **getline**.

Entrada de Cadenas de Caracteres Avanzada

El problema de mezclar la entrada de datos del **operador de entrada** (>>) con la operación **getline()**. En el caso de tener que leer una cadena de caracteres **vacía**, la solución con `cin >> ws` no es adecuada, por lo que se recomienda utilizar `cin.ignore(100, '\n');`.

```
int main()
{
  string nombre;
  int edad;
  cout << "Introduce nombre: ";
  getline(cin, nombre);
  while (nombre.size() > 0) {
    cout << "Introduce edad: ";
    cin >> edad ;
 ▶▶ cin.ignore(100, '\n'); ◀◀
    cout << "Nombre: [" << nombre << "]"
         << " Edad: " << edad << endl;
    cout << "Introduce nombre: ";
    getline(cin, nombre);
  }
}
```

```
Introduce nombre: pepe luis
Introduce edad: 21
Nombre: [pepe luis] Edad: 21
Introduce nombre: juan pedro
Introduce edad: 24
Nombre: [juan pedro] Edad: 24
Introduce nombre: maria jose
Introduce edad: 32
Nombre: [maria jose] Edad: 32
Introduce nombre:
```

Cadenas de Caracteres. El bucle FOR-EACH

- El bucle ***for-each*** permite recorrer **todos** los caracteres de un *string*, y en cada iteración, **asocia** un carácter del *string* a una determinada variable de tipo `char`.
- Esta asociación puede ser **por valor** o **por referencia**.
- Como en el paso de parámetros:
  - La asociación **por valor** sólo permite **consultar** los caracteres de tipo **char** del *string*.
  - La asociación **por referencia** permite tanto la **consulta** como la **modificación** de los caracteres de tipo **char** del *string*.
- El bucle ***for-each*** es muy cómodo y seguro para realizar recorridos sobre **todos** los caracteres del *string* (eliminando la posibilidad de realizar accesos fuera del intervalo válido), sin embargo, se desconoce la posición que ocupa cada carácter.

```
void invertir(const string& palabra,
              string& inversa)
{
    inversa = "";
    for (char x : palabra) {
        inversa = x + inversa;
    }
}
```

```
void minusculas(string& palabra)
{
  for (char& x : palabra) {
    if (('A' <= x) && (x <= 'Z')) {
      x = char(int(x) - int('A') + int('a'));
    }
  }
}
```

Cadenas de Caracteres. Ejemplos

1. Programa que lee una secuencia de palabras, delimitadas por espacios en blanco y/o saltos de línea, finalizada por la palabra `"fin"`, y muestra en pantalla la cantidad de palabras que comienzan por letra vocal.
2. Programa que lee una palabra, la transforma a mayúsculas, y finalmente la muestra en pantalla.
3. 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.
4. Diseñe una función que devuelva verdadero si la palabra recibida como parámetro es "palíndromo" y falso en caso contrario.
5. Diseñe un subprograma que reemplace una parte de la cadena, especificada por un índice y una longitud, por otra cadena.

Cadenas de Caracteres. Ejemplo 1

```
#include <iostream>
#include <string>
using namespace std;

const string FIN = "fin";

bool es_vocal(char c)
{
    return (c == 'a') || (c == 'e') || (c == 'i')
        || (c == 'o') || (c == 'u');
}
void procesar(const string& palabra, int& cnt)
{
    if ((palabra.size() > 0)&&(es_vocal(palabra[0]))) {
        ++cnt;
    }
}
void leer_secuencia(int& cnt)
{
    cnt = 0;
    string palabra;
    cout << "introduce secuencia de palabras hasta " << FIN <<":" << endl;
    cin >> palabra;
    while (palabra != FIN) {
        procesar(palabra, cnt);
        cin >> palabra;
    }
}
void mostrar_resultado(int cnt)
{
    cout << "Resultado: " << cnt << endl;
}

int main()
{
    int cnt;
    leer_secuencia(cnt);
    mostrar_resultado(cnt);
}
```

Cadenas de Caracteres. Ejemplo 2

```
#include <iostream>
#include <string>
using namespace std;

void leer(string& palabra)
{
    cout << "Introduce palabra: ";
    cin >> palabra;
}
void mostrar(const string& palabra)
{
    cout << "Mayusculas: " << palabra << endl;
}
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]);
    }
}
```

```
int main()
{
    string palabra;
    leer(palabra);
    mayusculas(palabra);
    mostrar(palabra);
}
```

Cadenas de Caracteres. Ejemplo 3 (v1)

```
#include <iostream>
#include <string>
using namespace std;

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";
        }
    }
}

int main()
{
    string palabra;
    cin >> palabra;
    plural_1(palabra);
    cout << palabra << endl;
}
```

Cadenas de Caracteres. Ejemplo 3 (v2)

```
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 4 (v1)

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

```
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 4 (v2)

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

```
void invertir(const string& palabra, string& inversa)
{
    inversa = "";
    for (int i = int(palabra.size())-1; i >= 0; --i) {
        inversa = inversa + palabra[i];
    }
}
void invertir_alternativo(const string& palabra, string& inversa)
{
    inversa = "";
    for (int i = 0; i < int(palabra.size()); ++i) {
        inversa = palabra[i] + inversa;
    }
}
bool es_palindromo(const string& palabra)
{
    string inversa;
    invertir(palabra, inversa);
    return (palabra == inversa);
}
```

Cadenas de Caracteres. Ejemplo 5

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

```
void reemplazar(string& str, int pos, int sz, const string& nueva)
{
    if (0 <= pos && pos <= int(str.size()) && sz >= 0) {
        if (pos + sz < int(str.size())) {
            str = str.substr(0, pos)
                  + nueva
                  + str.substr(pos + sz);
        } else {
            str = str.substr(0, pos)
                  + nueva;
        }
    }
}
```

El Tipo Registro

- Es un **tipo definido por el programador**. El programador puede definir tantos tipos nuevos como necesite. También denominado **Estructura**.
- El tipo registro permite definir un **nuevo tipo** como una **agrupación** de varios **componentes** que pueden ser de **distintos tipos**.
- Los componentes se denominan **campos**, y su ámbito de visibilidad se restringe al propio registro definido.
- El programador debe especificar el nombre del nuevo tipo, así como el nombre y el tipo de los componentes.
- Se puede **acceder** a cada componente individual de la agrupación mediante su **identificador**. Adecuado para procesamiento **individualizado**.

```
struct NombreDelNuevoTipoRegistro {
    Tipo1 campo_1;
    Tipo2 campo_1;
    ...
    TipoN campo_n;
} ;                          // No se debe olvidar el punto-y-coma
```

Registros

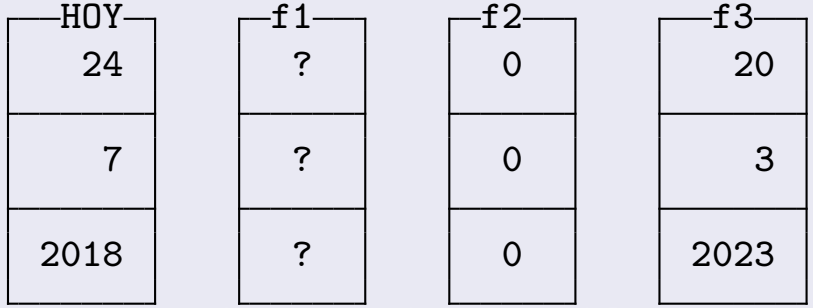
- Podemos definir un **nuevo tipo** que represente el concepto de *Fecha* como agrupación de *dia*, *mes* y *año*.

```
struct Fecha {
    int dia;
    int mes;
    int anyo;
};
```

```
// Se pueden agrupar los campos
// si son del mismo tipo
struct Fecha {
    int dia, mes, anyo;
};
```

- 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 al propio registro definido.
- Podemos declarar constantes, variables y parámetros de dicho tipo
  - Las *llaves-simples* inicializan valores y las *llaves-vacías* inicializan a cero.

```
const  Fecha HOY = { 24, 7, 2018 };
int main()
{
    Fecha f1;        // { ?, ?, ? }
    Fecha f2 = {};   // { 0, 0, 0 }
    Fecha f3 = { 20, 3, 2023};
}
```



Registros

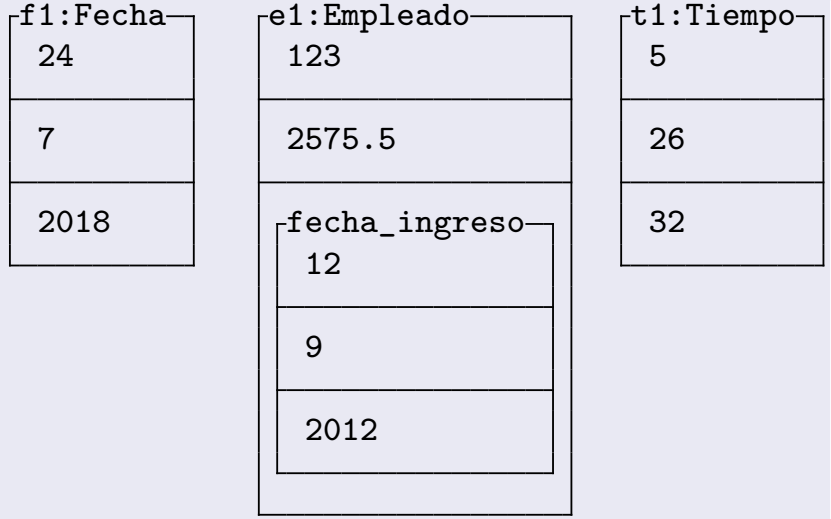
- Los campos de un registro pueden ser de cualquier tipo de datos, simple o compuesto.
  - Por ejemplo, el campo `fecha_ingreso` del tipo `Empleado` es de tipo `Fecha`.

```
struct Fecha {
    int dia;
    int mes;
    int anyo;
};
```

```
struct Empleado {
    int    codigo;
    double sueldo;
    Fecha  fecha_ingreso;
};
```

```
struct Tiempo {
    int hor;
    int min;
    int seg;
};
```

```
int main()
{
   Fecha f1 = { 24, 7, 2018 };
   Empleado e1 = { 123, 2575.5, { 12, 9, 2012 } };
   Tiempo t1 = { 5, 26, 32 };
}
```



Registros. Inicialización por Defecto de Campos

- Es posible especificar el **valor inicial** que tomarán los campos de una variable de tipo registro en caso de que no sean inicializados explícitamente.

```
struct Fecha {
    int dia = 1;
    int mes = 1;
    int anyo = 2000;
};

const  Fecha HOY = { 24, 7, 2018 };      // 24 de Julio de 2018

int main()
{
    Fecha f1 = { 25, 9, 2019 };          // 25 de Septiembre de 2019
    Fecha f2;                            // 1 de Enero de 2000
}
```

- Nótese que en el caso de especificar el valor inicial de los campos, si algún campo se deja sin inicializar entonces tomará un valor **inespecificado**.
- Especificar los valores iniciales en la definición de los campos de los registros es una *forma muy cómoda* de **evitar los errores** producidos por variables sin inicializar.

Registros. Acceso a Componentes

- 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).

```
int main()
{
    Fecha f = HOY;
    f.dia = 10;
    f.mes = 10;
    f.anyo = 2018;
    cout << HOY.dia << '/' << HOY.mes << '/' << HOY.anyo << endl;

    Empleado pepe;
    pepe.codigo = 123;
    pepe.sueldo = 1234.56;
    pepe.fecha_ingreso.dia = 27;
    pepe.fecha_ingreso.mes = 8;
    pepe.fecha_ingreso.anyo = 2018;
}
```

Registros. Operaciones

Operaciones entre registros completos.

- Asignación de *registros* completos (=).
- Paso como parámetro a subprogramas (⇑⇕*referencia* y ⇓*referencia constante*).
- Devolución por una función. Esta operación es válida, pero no es recomendable.
  - Mejor mediante un parámetro de salida (⇑) con paso por referencia.
- Cualquier otra operación debe ser definida por el programador (programada componente a componente). **No** son válidos (>> , <<, == , != , > , >= , < , <=).

```
void leer_fecha(Fecha& f)           // ⇑ Paso por referencia
{
    cout << "Introduce dia mes y año: ";
    cin >> f.dia >> f.mes >> f.anyo;
}
void mostrar_fecha(const Fecha& f)  // ⇓ Paso por referencia constante
{
    cout << f.dia << "/" << f.mes << "/" << f.anyo << endl;
}
int main()
{
    Fecha f1, f2;
    leer_fecha(f1);
    f2 = f1;              // Asignación de registros completos
    mostrar_fecha(f2);
}
```

Registros. Ejemplo 1

- 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 1

```
#include <iostream>
using namespace std;

const int SEGMIN = 60;
const int MINHOR = 60;
const int MAXHOR = 24;
const int SEGHOR = SEGMIN
                   * MINHOR;
struct Tiempo {
    int horas;
    int minutos;
    int segundos;
};
```

```
void leer_intervalo(int& num, int inf, int sup)
{
    cout << "Introduce valor ["
         << inf << ", " << sup << "): ";
    cin >> num;
    while ( ! ((inf <= num) && (num < sup))) {
        cout << "Error. Introduce valor["
             << inf << ", " << sup << "): ";
        cin >> num;
    }
}
void leer_tiempo(Tiempo& t)
{
    cout << "Introduce horas: ";
    leer_intervalo(t.horas, 0, MAXHOR);
    cout << "Introduce minutos: ";
    leer_intervalo(t.minutos, 0, MINHOR);
    cout << "Introduce segundos: ";
    leer_intervalo(t.segundos, 0, SEGMIN);
}
```

Registros. Ejemplo 1

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

Registros. Ejemplo 2

- Desarrolle un programa que permita introducir los datos de dos empleados (código, sueldo y fecha de ingreso) y muestre los datos del empleado con mayor antigüedad.

Registros. Ejemplo 2

```
#include <iostream>
using namespace std;

struct Fecha {
    int dia, mes, anyo;
};
struct Empleado {
    int codigo;
    double sueldo;
    Fecha fecha_ingr;
};

void leer_fecha(Fecha& f)
{
    cout << "Introduce "
         << "dia mes y año: ";
    cin >> f.dia
        >> f.mes
        >> f.anyo;
}
```

```
void leer_empleado(Empleado& e)
{
    cout << "Introduce código: ";
    cin >> e.codigo;
    cout << "Introduce sueldo: ";
    cin >> e.sueldo;
    cout << "Introduce fecha de ingreso. ";
    leer_fecha(e.fecha_ingr);
}
void mostrar_fecha(const Fecha& f)
{
    cout << f.dia << "/"
         << f.mes << "/"
         << f.anyo;
}
void mostrar_empleado(const Empleado& e)
{
    cout << e.codigo << " "
         << e.sueldo << " ";
    mostrar_fecha(e.fecha_ingr);
    cout << endl;
}
```

Registros. Ejemplo 2

```
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 {
        // fechas iguales
        ok = false;
    }
    return ok;
}
bool mas_antiguo(const Empleado& e1, const Empleado& e2)
{
    return es_menor(e1.fecha_ingr, e2.fecha_ingr);
}
```

```
int main()
{
    Empleado e1, e2;
    leer_empleado(e1);
    leer_empleado(e2);
    if (mas_antiguo(e1, e2)) {
        mostrar_empleado(e1);
    } else {
        mostrar_empleado(e2);
    }
}
```

Arrays

El Tipo Array

- Es un **tipo definido por el programador**. El programador puede definir tantos tipos nuevos como necesite.
- El tipo array permite definir un **nuevo tipo** como una **colección** (agregación) de una cantidad **fija** de **elementos** del **mismo tipo**.
- La **cantidad de elementos** de la colección debe ser **constante**, definida en **tiempo de compilación**.
- El programador debe especificar el nombre del nuevo tipo, así como el tipo de los elementos y la cantidad de elementos.
- Se puede **acceder** a cada elemento de la colección de forma **parametrizada**. Adecuado para procesamiento **iterativo**.

Arrays en C++

- **C++** ofrece la posibilidad de utilizar dos clases diferentes de arrays:
  - Arrays predefinidos de **C** (**NO** los utilizaremos en este curso):

```
typedef TipoBase NombreDelTipoArray[NUMERO_ELEMENTOS] ;
```

  - Arrays de la **biblioteca estándar** (**SÍ** los utilizaremos en este curso):

```
typedef array<TipoBase, NUMERO_ELEMENTOS> NombreDelTipoArray ;
```

  - Los arrays de la **biblioteca estándar** son más completos y robustos.

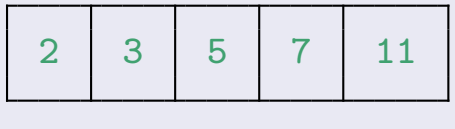
Arrays

- En la definición de un nuevo tipo Array interviene:
  - El **tipo base** es el tipo de los elementos que constituyen el array. Puede ser cualquier tipo de datos, simple o compuesto.
  - La **cantidad** de elementos que forman la colección. Debe ser **constante**.
- Se debe incluir la biblioteca **<array>**, y usar el espacio de nombres **std**.
- Por ejemplo, podemos definir un **nuevo tipo** que represente el concepto de una colección de 5 números enteros, y denominarle **Vector**.

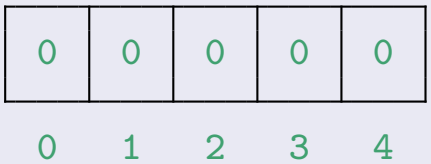
```
#include <array>
using namespace std;
const int NELMS = 5;                    // se debe definir el tamaño del array
typedef array<int, NELMS> Vector;       // como una constante simbólica
```

- Podemos declarar constantes, variables y parámetros de dicho tipo.
  - Las *llaves-dobles* inicializan valores y las *llaves-vacías* inicializan a cero.

```
const Vector PRIMOS = {{ 2, 3, 5, 7, 11 }};

int main()
{
   Vector v1;
}
```



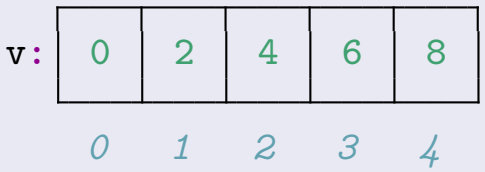

```
Vector v2 = {};
```



Arrays

- 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.
- La cantidad de elementos que componen el array:
  - `v.size()` ∈ `unsigned`
  - `int(v.size())` ∈ `int`
- **Acceso al iésimo** elemento del array `v` (del tipo base del array):
  - `v[i]` donde `i` ∈ { `0...v.size()-1` }

```
int main() {
    Vector v;
    for (int i = 0; i < int(v.size()); ++i) {
        v[i] = 2 * i;
    }
    int primer_elemento = v[0];              // 0
    int ultimo_elemento = v[v.size() - 1]; // 8
}
```



- En *Code::Blocks*, definir el símbolo `_GLIBCXX_DEBUG` en el apartado de `#defines` de las opciones de compilación permite comprobar los índices de acceso.

Arrays. Ejemplo

```cpp
#include <iostream>
#include <array>
using namespace std;

const int NELMS = 5;                  // se debe definir el tamaño del array
typedef array<int, NELMS> Vector;     // como una constante simbólica

const Vector PRIMOS = {{ 2, 3, 5, 7, 11 }};

int main()
{
    Vector v;
    for (int i = 0; i < int(v.size()); ++i) {
        v[i] = PRIMOS[i] * 2;
    }
}
```

2	3	5	7	11
0	1	2	3	4

4	6	10	14	22
0	1	2	3	4

Arrays. Procesamiento iterativo

- Los programas deben **funcionar adecuadamente** incluso cuando se **modifiquen** los valores de las **constantes** que determinan la cantidad de elementos de los arrays.
  - Los arrays se deben manipular con **procesamientos iterativos** (bucles).

```
const int NELMS = 3;                    // se debe definir el tamaño del array
typedef array<int, NELMS> Vector;       // como una constante simbólica

void procesamiento_iterativo() // procesamiento iterativo -> BIEN
{
    Vector v;                                          //
    for (int i = 0; i < int(v.size()); ++i) {          //   v: | 1 | 2 | 3 |
        v[i] = i + 1;                                  //
    }                                                  //      0   1   2
}

void procesamiento_fijo() // procesamiento fijo -> MAL
{
    Vector v; // Si cambiase el valor                  //
    v[0] = 1; // de la constante NELMS,                //   v: | 1 | 2 | 3 |
    v[1] = 2; // entonces el programa                  //
    v[2] = 3; // funcionaría MAL                      //      0   1   2
}
```

Operaciones con Arrays completos

- Asignación de *arrays* completos (=).
- Paso como parámetro a subprogramas (⇑⇕*referencia* y ⇓*referencia constante*).
- Devolución por una función. Esta operación es válida, pero no es recomendable.
  - Mejor mediante un parámetro de salida (⇑) con paso por referencia.
- Los operadores relacionales (== , != , > , >= , < , <=) sólo son aplicables si están definidos para los elementos del tipo base.
- Cualquier otra operación debe ser definida por el programador (programada elemento a elemento). **No** son válidos los op. de **E/S** (>> , <<).

```
void leer_vector(Vector& v) // ⇑ P.Ref
{
  for (int i = 0; i < int(v.size()); ++i) {
    cin >> v[i];
  }
}
void mostrar_vector(const Vector& v) // ⇓ P.Ref.Cte
{
  for (int i = 0; i < int(v.size()); ++i) {
    cout << v[i] << " ";
  }
  cout << endl;
}
```

```
int main()
{
  Vector v1, v2;
  leer_vector(v1);
  v2 = v1;              // Asignación
  if (v1 >= v2) {       // Comparación
    mostrar_vector(v1);
  } else {
    mostrar_vector(v2);
  }
}
```

Arrays. Ejemplo

```
#include <iostream>
#include <array>
using namespace std;
const int NELMS = 5;
typedef array<int, NELMS> Vector;
void leer(Vector& v)
{
   cout << "Introduce " << v.size()
        << " números: " ;
   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;
}
```

```
void mostrar(const Vector& v)
{
   for (int i = 0; i < int(v.size()); ++i) {
      cout << v[i] << " ";
   }
   cout << endl;
}
int main()
{
    Vector v1, v2;
    leer(v1);
    leer(v2);
    if (sumar(v1) > sumar(v2)) {
        mostrar(v1);
    } else {
        mostrar(v2);
    }
}
```

Arrays Predefinidos y Arrays de la Biblioteca

- Los **arrays predefinidos** son compatibles con el lenguaje de programación C, y por lo tanto son útiles en aquellas circunstancias en las que es adecuado interactuar con bibliotecas de C o del sistema operativo.
- Los **arrays** proporcionados por la **biblioteca** estándar de C++ ofrecen un mecanismo **más simple, consistente y robusto**, por lo que son más adecuados para la mayoría de las situaciones.
- Entre las ventajas que ofrecen los arrays proporcionados por la biblioteca estándar de C++, podemos destacar las siguientes:
  - **Simplicidad** y **consistencia**: el tratamiento de los arrays de la biblioteca es consistente respecto a los otros tipos de datos.
    - Inicialización (valores y variables), asignación, comparación, devolución de función y paso de parámetros (valor, referencia y referencia constante).
  - **Robustez** respecto a errores de programación.
    - Los arrays de la biblioteca no decaen al tipo puntero, por lo que son más robustos ante numerosos tipos de errores.
    - Si se establecen las opciones de compilación adecuadas, entonces se comprueban los accesos indexados a los elementos de los arrays durante el tiempo de ejecución.

Arrays. El bucle FOR-EACH

- El bucle ***for-each*** permite recorrer **todos** los elementos de un array, y en cada iteración, **asocia** un elemento del array a una determinada variable del mismo tipo base de los elementos del array.
- Esta asociación puede ser **por valor**, **por referencia** o **por referencia constante**.
- Como en el paso de parámetros:
  - La asociación **por valor** sólo permite **consultar** los valores de **tipos simples** de los elementos del array.
  - La asociación **por referencia constante** sólo permite **consultar** los valores de **tipos compuestos** de los elementos del array.
  - La asociación **por referencia** permite tanto la **consulta** como la **modificación** de los elementos del array, tanto de **tipos simples** como **compuestos**.
- El bucle ***for-each*** permite realizar recorridos sobre **todos** los elementos del array de forma cómoda y segura (elimina la posibilidad de realizar accesos fuera del intervalo válido), sin embargo, se desconoce la posición que ocupa cada elemento.

```
void leer(Vector& v)
{
   for (int& x : v) {
      cin >> x;
   }
}
```

```
void mostrar(const Vector& v)
{
   for (int x : v) {
      cout << x << " ";
   }
   cout << endl;
}
```

Arrays. Ejemplo con FOR-EACH

```cpp
#include <iostream>
#include <array>
using namespace std;
const int NELMS = 5;
typedef array<int, NELMS> Vector;
void leer(Vector& v)
{
   cout << "Introduce " << v.size()
        << " números: " ;
   for (int& x : v) {
      cin >> x;
   }
}
int sumar(const Vector& v)
{
   int suma = 0;
   for (int x : v) {
      suma += x;
   }
   return suma;
}
```

```cpp
void mostrar(const Vector& v)
{
   for (int x : v) {
      cout << x << " ";
   }
   cout << endl;
}
int main()
{
    Vector v1, v2;
    leer(v1);
    leer(v2);
    if (sumar(v1) > sumar(v2)) {
        mostrar(v1);
    } else {
        mostrar(v2);
    }
}
```

Utilidad de los Arrays

Los arrays son útiles en todas aquellas circunstancias en que necesitamos tener **almacenados una colección de valores** (una cantidad fija predeterminada en tiempo de compilación) a los cuales pretendemos **acceder de forma parametrizada**, normalmente para aplicar un **procesamiento iterativo**.

Ejemplo 1. Programa de notas de alumnos (v1)

Lee la nota de cada alumno (la cantidad de alumnos es 20) y muestra si está aprobado o suspenso, considerando que el alumno está aprobado si su nota es mayor o igual a 5.

Utilidad de los Arrays. Ejemplo 1

```
#include <iostream>
using namespace std;

const int NALUMNOS = 20;
const double APROBADO = 5.0;

int main()
{
    for (int i = 0; i < NALUMNOS; ++i) {
        cout << "Introduzca nota del alumno " << i << ": ";
        double nota;
        cin >> nota;
        if (nota >= APROBADO) {
            cout << "Alumno: " << i << " Aprobado" << endl;
        } else {
            cout << "Alumno: " << i << " Suspenso" << endl;
        }
    }
}
```

Utilidad de los Arrays. Ejemplo 2

Ejemplo 2. Programa de notas de alumnos (v2)

Lee la nota de cada alumno (la cantidad de alumnos es 20) y muestra si está aprobado o suspenso, considerando que el alumno está aprobado si su nota es mayor o igual **a la nota media de todos los alumnos**.

Utilidad de los Arrays. Ejemplo 2

```cpp
#include <iostream>
#include <array>
using namespace std;

const int NALUMNOS = 20;

typedef array<double, NALUMNOS> Notas;

void leer_notas(Notas& v)
{
    for (int i = 0; i < int(v.size()); ++i) {
        cout << "Introduzca nota del alumno " << i << ": ";
        cin >> v[i];
    }
}
```

```cpp
double calc_media(const Notas& v)
{
  double suma = 0.0;
  for (int i = 0; i < int(v.size()); ++i) {
    suma += v[i];
  }
  return suma / double(v.size());
}
```

```cpp
double calc_media(const Notas& v)
{
  double suma = 0.0;
  for (double x : v) { // FOR-EACH
    suma += x;
  }
  return suma / double(v.size());
}
```

Utilidad de los Arrays. Ejemplo 2

```
void mostrar_notas(const Notas& v, double umbral)
{
    for (int i = 0; i < int(v.size()); ++i) {
        if (v[i] >= umbral) {
            cout << "Alumno: " << i << " Aprobado" << endl;
        } else {
            cout << "Alumno: " << i << " Suspenso" << endl;
        }
    }
}
int main()
{
    Notas notas;
    leer_notas(notas);
    double media = calc_media(notas);
    mostrar_notas(notas, media);
}
```

Arrays Incompletos: Listas con Cantidad Variable de Elementos

- Hay situaciones en las que debemos almacenar una **lista** de elementos, donde la cantidad 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:
  - La **cantidad de elementos** válidos que **actualmente** contiene la lista.
  - Un **array**, del tamaño adecuado al límite máximo de la lista, que almacene los elementos **consecutivamente** al principio.
  - En ocasiones, el registro puede tener más datos.
- Tiene diversas denominaciones, hay que entender el concepto, que puede surgir en diferentes contextos y denominaciones, y aplicarlo adecuadamente.
- Usualmente, **NO** se podrá utilizar el bucle **for-each** sobre el array, ya que no se procesarán todos sus elementos, solo aquellos que realmente tengan datos válidos.

```
const int MAX_ELEMENTOS = 100;
typedef array<TipoBase, MAX_ELEMENTOS> Elementos;
struct Lista {
    int nelms = 0; // la cantidad de elementos inicialmente es cero (lista vacía)
    Elementos elm;
};
```

Utilidad de las Listas. Ejemplo 3

Ejemplo 3. Programa de notas de alumnos (v3)

- Lee la nota de cada alumno (la cantidad **máxima** de alumnos es 20) y muestra si está aprobado o suspenso, considerando que el alumno está aprobado si su nota es mayor o igual **a la nota media de todos los alumnos**.
- **La cantidad máxima de alumnos es 20, pero la cantidad actual de alumnos será leída de teclado.**

Utilidad de las Listas. Ejemplo 3

```cpp
#include <iostream>
#include <array>
using namespace std;

const int MAX_ALUMNOS = 20;

typedef array<double, MAX_ALUMNOS> Datos;
struct Notas {
    int nelms = 0;  // aseguramos que se inicializa correctamente a cero (lista vacía)
    Datos elm;
};

double calc_media(const Notas& v)
{
    double media = 0.0;
    if (v.nelms > 0) {
        double suma = 0.0;
        for (int i = 0; i < v.nelms; ++i) {
            suma += v.elm[i];
        }
        media = suma / double(v.nelms);
    }
    return media;
}
```

Utilidad de las Listas. Ejemplo 3

```
void leer_notas(Notas& v)
{
    cout << "Introduzca total de alumnos: ";
    cin >> v.nelms;
    if (v.nelms <= 0 || v.nelms > int(v.elm.size())) {
        v.nelms = 0;  // lista vacía
        cout << "Error" << endl;
    } else {
        for (int i = 0; i < v.nelms; ++i) {
            cout << "Introduzca nota del alumno " << i << ": ";
            cin >> v.elm[i];
        }
    }
}
```

Utilidad de las Listas. Ejemplo 3

```
void mostrar_notas(const Notas& v, double umbral)
{
    for (int i = 0; i < v.nelms; ++i) {
        if (v.elm[i] >= umbral) {
            cout << "Alumno: " << i << " Aprobado" << endl;
        } else {
            cout << "Alumno: " << i << " Suspenso" << endl;
        }
    }
}
int main()
{
    Notas notas;
    leer_notas(notas);
    double media = calc_media(notas);
    mostrar_notas(notas, media);
}
```

Utilidad de las Listas. Ejemplo 3

```
void anyadir_elemento(Notas& v, double valor)
{
    if (v.nelms < int(v.elm.size())) {  // Si el nuevo elemento cabe en el array
        v.elm[v.nelms] = valor;         // Añade el elemento al final de la lista
        ++v.nelms;                      // Incrementa la cuenta de elementos
    } else {
        cout << "Error" << endl;
    }
}
void leer_notas_alternativo(Notas& v)
{
    double valor;
    v.nelms = 0;  // lista vacía
    cout << "Introduzca nota del alumno " << v.nelms << " (0 para fin): ";
    cin >> valor;
    while (valor > 0) {
        anyadir_elemento(v, valor);
        cout << "Introduzca nota del alumno " << v.nelms << " (0 para fin): ";
        cin >> valor;
    }
}
```

Búsqueda Lineal o Secuencial (v1)

- Adecuada como mecanismo de **búsqueda general** en colecciones de datos **sin organización** conocida.
- Si encuentra el elemento buscado, entonces devuelve el índice donde se encuentra el elemento en el array, en otro caso devuelve un índice con valor fuera de límites.
  - Procede comparando consecutivamente el elemento a buscar con todos los elementos de la colección, hasta que lo encuentre, o hasta que haya comparado todos los elementos.

```
int buscar(const Lista& v, int x)
{
    int idx = -1;
    for (int i = 0; (i < v.nelms) && (idx < 0); ++i) {
        if (x == v.elm[i]) {
            idx = i;
        }
    }
    return idx;
}
```

Búsqueda Lineal o Secuencial (v2 y v3)

Buscar Alternativo (v2)

```
int buscar(const Lista& v, int x)
{
    int idx = -1;
    bool encontrado = false;
    for (int i = 0; (i < v.nelms) && ( ! encontrado); ++i) {
        if (x == v.elm[i]) {
            idx = i;
            encontrado = true;
        }
    }
    return idx;
}
```

Buscar Alternativo (v3)

```
int buscar(const Lista& v, int x)
{
    int i = 0;
    // Evaluación en CORTOCIRCUITO
    while ((i < v.nelms) && (x != v.elm[i])) {
        ++i;
    }
    if (i == v.nelms) {
        i = -1;
    }
    return i;
}
```

Operaciones Básicas con Listas (I)

Tipo Lista

```
const int MAX_ELEMENTOS = 100;
typedef array<int, MAX_ELEMENTOS> Elementos;
struct Lista {
    int nelms = 0;  // aseguramos que se inicializa correctamente a cero
    Elementos elm;
};
```

Añadir elemento al final

```
void anyadir(Lista& v, int valor, bool& ok)
{
    if (v.nelms < int(v.elm.size())) {
        v.elm[v.nelms] = valor;
        ++v.nelms;
        ok = true;
    } else {
        ok = false;
    }
}
```

Operaciones Básicas con Listas (II)

Eliminar elemento desordenando

```
void eliminar(Lista& v, int valor, bool& ok)
{
    int pos = buscar(v, valor);
    if (0 <= pos && pos < v.nelms) {
        v.elm[pos] = v.elm[v.nelms-1];
        --v.nelms;
        ok = true;
    } else {
        ok = false;
    }
}
```

Operaciones Básicas con Listas (III)

Añadir elemento ordenado

```
int buscar_posicion(const Lista& v, int valor)
{
    int pos = v->nelms;
    for (int i = 0; (i < v.nelms) && (pos == v.nelms); ++i) {
        if (valor < v.elm[i]) {
            pos = i;
        }
    }
    return pos;
}
void anyadir_ord(Lista& v, int valor, bool& ok)
{
    if (v.nelms < int(v.elm.size())) {
        int pos = buscar_posicion(v, valor);
        for (int i = v.nelms-1; i >= pos; --i) {
            v.elm[i+1] = v.elm[i];
        }
        v.elm[pos] = valor;
        ++v.nelms;
        ok = true;
    } else {
        ok = false;
    }
}
```

Operaciones Básicas con Listas (IV)

Eliminar elemento manteniendo el orden

```
void eliminar_ord(Lista& v, int valor, bool& ok)
{
    int pos = buscar(v, valor);
    if (0 <= pos && pos < v.nelms) {
        for (int i = pos+1; i < v.nelms; ++i) {
            v.elm[i-1] = v.elm[i];
        }
        --v.nelms;
        ok = true;
    } else {
        ok = false;
    }
}
```

Operaciones Básicas con Listas (V)

Eliminar todos los elementos que cumplen condición

```
// Elimina todos los elementos que cumplen una determinada condición.
// En este caso, elimina todos los elementos que son iguales al valor
// recibido como parámetro.

void eliminar_todos(Lista& v, int valor)
{
    int j = 0;  // j contiene la posición donde almacenar los elementos válidos
    for (int i = 0; i < v.nelms; ++i) {  // para cada elemento original en pos i
        if ( v.elm[i] != valor ) {       // si el elemento original es válido
            v.elm[j] = v.elm[i];         // almacenar elemento original en pos j
            ++j;                         // incrementa pos j para siguiente válido
        }
    }
    v.nelms = j;  // j contiene la cantidad de elementos válidos guardados
}
```

Operaciones Básicas con Listas (VI)

Ordenación por Burbuja

```
// Código simplificado para facilitar su estudio

void ordenar_burbuja(Lista& v)
{
    for (int k = 0; k < v.nelms-1; ++k) {
        for (int i = 0; i < v.nelms-1; ++i) {
            if (v.elm[i] > v.elm[i+1]) {
                // intercambiar(v.elm[i], v.elm[i+1]);
                int aux = v.elm[i];
                v.elm[i] = v.elm[i+1];
                v.elm[i+1] = aux;
            }
        }
    }
}
```

Arrays Multidimensionales

- El **tipo base** de un array puede ser tanto simple como compuesto, por lo tanto puede ser **otro array**, dando lugar a arrays con **múltiples dimensiones**.
- En un array con **dos dimensiones**:
  - La **primera dimensión** corresponde a las **FILAS**.
  - La **segunda dimensión** corresponde a las **COLUMNAS**.
  - Muchos alumnos definen **MAL** el tipo, con las dimensiones intercambiadas.

```
const int NFILAS = 3;
const int NCOLUMNAS = 5;

typedef array<int, NCOLUMNAS> Fila;
typedef array<Fila, NFILAS> Matriz;

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];  // acceso a la primera fila (0) de tipo Fila
    int n = m[2][4];  // última fila (2), última columna (4), elemento 14
}
```

COLUMNAS

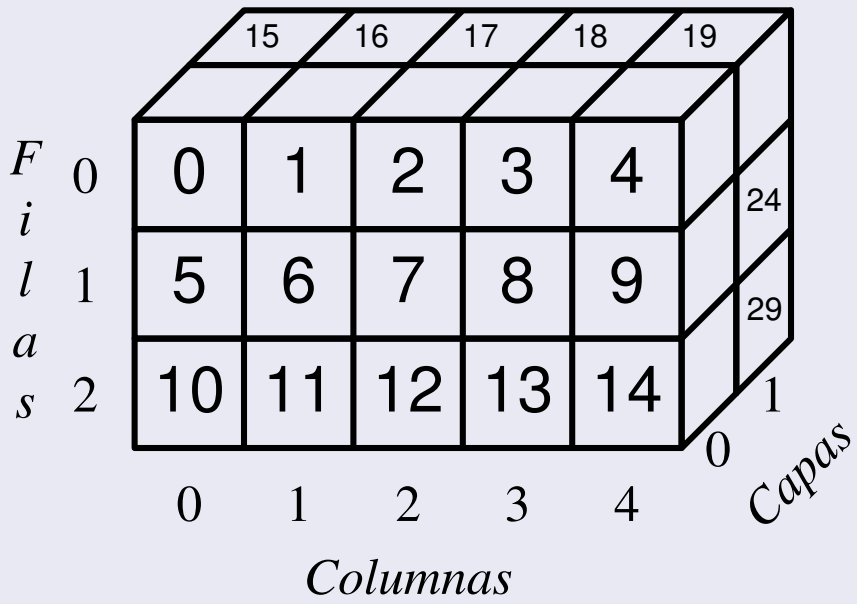
m:	0	1	2	3	4
FILAS 0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14

Arrays Multidimensionales

- También es posible definir un array de tres o más dimensiones.

```
const int NFILAS = 3;
const int NCOLUMNAS = 5;
const int NCAPAS = 2;

typedef array<int, NCOLUMNAS> Fila;
typedef array<Fila, NFILAS> Matriz;
typedef array<Matriz, NCAPAS> Cubo;
```

```
Cubo c;

c[0][0][0] = 0;    c[1][0][0] = 15;
c[0][0][4] = 4;    c[1][0][4] = 19;
c[0][2][0] = 10;   c[1][1][4] = 24;
c[0][2][4] = 14;   c[1][2][4] = 29;
```



Entrada / Salida de un Array 2D

Mostrar el contenido de un Array 2D

```
void mostrar(const Matriz& m)
{
  for (int f = 0; f < int(m.size()); ++f) {
    for (int c = 0; c < int(m[f].size()); ++c) {
      cout << setw(3) << m[f][c] << " "; // espacio
    }                                    // separación
    cout << endl;
  }
}
```

```
void mostrar(const Matriz& m)
{
  for (const Fila& fila : m) {
    for (int x : fila) {
      cout << setw(3) << x << " "; // espacio
    }                              // separación
    cout << endl;
  }
}
```

Leer el contenido de un Array 2D

```
void leer(Matriz& m)
{
  cout << "Introduce " << m.size() << " x "
       << m[0].size() << " números" << endl;
  for (int f = 0; f < int(m.size()); ++f) {
    for (int c = 0; c < int(m[f].size()); ++c) {
      cin >> m[f][c];
    }
  }
}
```

```
void leer(Matriz& m)
{
  cout << "Introduce " << m.size() << " x "
       << m[0].size() << " números" << endl;
  for (Fila& fila : m) {
    for (int& x : fila) {
      cin >> x;
    }
  }
}
```

Búsqueda Lineal o Secuencial en Array 2D (v1)

- Si encuentra el elemento buscado, entonces devuelve los índices donde se encuentra el elemento en el array 2D, en otro caso devuelve un valor fuera de límites en los parámetros de los índices de la fila y columna.

```
void buscar2d(const Matriz& m, int x, int& ff, int& cc)
{
    ff = -1;
    cc = -1;
    for (int f = 0; (f < int(m.size())) && (ff < 0); ++f) {
        for (int c = 0; (c < int(m[f].size())) && (cc < 0); ++c) {
            if (x == m[f][c]) {
                ff = f;
                cc = c;
            }
        }
    }
}
```

Búsqueda Lineal o Secuencial en Array 2D (v2)

```
void buscar2d(const Matriz& m, int x, int& ff, int& cc) {
    ff = -1;
    cc = -1;
    bool encontrado = false;
    for (int f = 0; (f < int(m.size())) && ( ! encontrado); ++f) {
        for (int c = 0; (c < int(m[f].size())) && ( ! encontrado); ++c) {
            if (x == m[f][c]) {
                ff = f;
                cc = c;
                encontrado = true;
            }
        }
    }
}
```

Arrays Multidimensionales. Ejemplo

- Algoritmo que lee una matriz 3x5 de enteros (fila a fila), almacenandolos en un array bidimensional `a`. Finalmente muestra 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 `b` representa el resultado de sumar todos los elementos de la fila y `c` representa el resultado de sumar todos los elementos de la columna donde se encuentran.

Arrays Multidimensionales. Ejemplo

```
#include <iostream>
#include <array>
using namespace std;

const int NFILAS = 3;
const int NCOLUMNAS = 5;

typedef array<int, NCOLUMNAS> Fila;
typedef array<Fila, NFILAS> Matriz;

void leer_matriz(Matriz& m)
{
    cout << "Introduce " << m.size() << " x "
         << m[0].size() << " números" << endl;
    for (int f = 0; f < int(m.size()); ++f) {
        for (int c = 0; c < int(m[f].size()); ++c) {
            cin >> m[f][c];
        }
    }
}
```

Arrays Multidimensionales. Ejemplo

```
int sumar_fila_alternativo(const Matriz& m, int f)
{
    int suma = 0;
    for (int c = 0; c < int(m[f].size()); ++c) {
        suma += m[f][c];
    }
    return suma;
}
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;
}
```

Arrays Multidimensionales. Ejemplo

```
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 << sumar_fila_alternativo(m, f);
        cout << endl;
    }
    for (int c = 0; c < int(m[0].size()); ++c) {
        cout << sumar_columna(m, c) << " ";
    }
    cout << endl;
}
int main()
{
    Matriz m;
    leer_matriz(m);
    escribir_matriz_formato(m);
}
```

Arrays Multidimensionales. Ejemplo: Difuminar Matriz (I)

Análisis de vecinos: difuminar matriz

- Se debe leer una matriz de 3×4 números enteros, posteriormente se debe difuminar la matriz, y finalmente de mostrará el contenido de la matriz difuminada.
- Para difuminar una matriz, se debe reemplazar cada número por la media (entera) del propio número junto con sus vecinos, considerando que los números pueden tener 8, 5 o 3 vecinos, dependiendo de su posición en la matriz (esquinas y laterales).
- Se debe tener cuidado que los números reemplazados no afecten a los cálculos.
- Por ejemplo, para la esquina superior derecha, $(76 + 43 + 32 + 12) \ / \ 4 = 40.75$

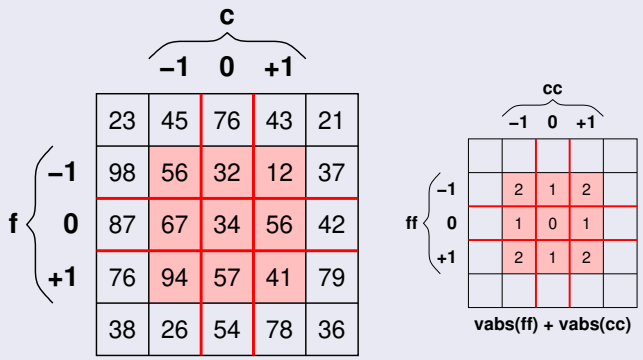
```
Introduce 3 x 4 números
23 45 76 43
98 56 32 12
87 67 34 56

 55  55  44  40
 62  57  46  42
 77  62  42  33
```

Arrays Multidimensionales. Ejemplo: Difuminar Matriz (II)

```
#include <iostream>
#include <iomanip>
#include <array>
using namespace std;
const int NFILS = 3;
const int NCOLS = 4;
typedef array<int, NCOLS> Fila;
typedef array<Fila, NFILS> Matriz;

void leer(Matriz& m)
{
    cout << "Introduce " << m.size() << " x " << m[0].size() << " números" << endl;
    for (int f = 0; f < int(m.size()); ++f) {
        for (int c = 0; c < int(m[f].size()); ++c) {
            cin >> m[f][c];
        }
    }
}
void mostrar(const Matriz& m)
{
    for (int f = 0; f < int(m.size()); ++f) {
        for (int c = 0; c < int(m[f].size()); ++c) {
            cout << setw(3) << m[f][c] << " ";   // espacio de separación
        }
        cout << endl;
    }
}
```

Arrays Multidimensionales. Ejemplo: Difuminar Matriz (III)

```
bool es_valido(const Matriz& m, int f, int c, int ff, int cc)
{
    return (0 <= f+ff && f+ff < int(m.size())) && (0 <= c+cc && c+cc < int(m[f+ff].size()));
}       // && (vabs(ff)+vabs(cc) == 2); // -> selecciona solo los vecinos de las esquinas
        // && (vabs(ff)+vabs(cc) == 1); // -> selecciona solo los vecinos horizontal y vertical
        // && (vabs(ff)+vabs(cc) > 0);  // -> selecciona solo los vecinos (no central)

double media_vecinos(const Matriz& m, int f, int c)
{
    int nvec = 0;
    int suma = 0;
    for (int ff = -1; ff <= +1; ++ff) {
        for (int cc = -1; cc <= +1; ++cc) {
            if (es_valido(m, f, c, ff, cc)) {
                suma += m[f+ff][c+cc];
                ++nvec;
            }
        }
    }
    return double(suma) / double(nvec);
}

void difuminar(Matriz& m)
{
    Matriz aux = m;
    for (int f = 0; f < int(m.size()); ++f) {
        for (int c = 0; c < int(m[f].size()); ++c) {
            m[f][c] = media_vecinos(aux, f, c);
        }
    }
}
```




```
int main ()
{
    Matriz m;
    leer(m);
    difuminar(m);
    mostrar(m);
}
```

Definición de Constantes de Tipos Compuestos. Ejemplo 1

- Array constante de cadenas de caracteres.

```
#include <iostream>
#include <string>
#include <array>
using namespace std;

const int NELMS = 7;

typedef array<string, NELMS> Vector;

const Vector DIAS = {{
    "Lunes", "Martes", "Miercoles", "Jueves", "Viernes", "Sabado", "Domingo"
}};

int main()
{
    Vector dias = DIAS;
    // ...
}
```

Definición de Constantes de Tipos Compuestos. Ejemplo 2

- Lista constante de números.

```
const int NELMS = 20;
typedef array<int, NELMS> Datos;
struct Lista {
    int nelms = 0;
    Datos elm;
};

const Lista DATOS = { 3, {{ 1, 2, 3 }} }; // valor cero para el resto de elementos

int main()
{
    Lista datos = DATOS;
    // ...
}
```

Definición de Constantes de Tipos Compuestos. Ejemplo 3

- Array constante de personas.

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

const Datos DATOS = {{
    { "Lola", { 20, 4, 2010 } } ,
    { "Pepe", { 12, 8, 2011 } } ,
    { "Luis", { 24, 9, 2012 } }
}};

int main()
{
    Datos datos = DATOS;
    // ...
}
```

Definición de Constantes de Tipos Compuestos. Ejemplo 4

- Lista constante de personas.

```
const int NELMS = 20;
struct Fecha {
    int dia;
    int mes;
    int anyo;
};
struct Persona {
    string nombre;
    Fecha fnac;
};
typedef array<Persona, NELMS> Datos;
struct Lista {
    int nelms = 0;
    Datos elm;
};

const Lista DATOS = { 3,
    {{
        { "Lola", { 20, 4, 2010 } } ,
        { "Pepe", { 12, 8, 2011 } } ,
        { "Luis", { 24, 9, 2012 } }
    }}
};

int main()
{
    Lista datos = DATOS;
    // ...
}
```

Definición de Constantes de Tipos Compuestos. Ejemplo 5

- Array constante de dos dimensiones.

```
#include <iostream>
#include <array>
using namespace std;

const int NFILAS = 3;
const int NCOLUMNAS = 4;

typedef array<int, NCOLUMNAS> Fila;
typedef array<Fila, NFILAS> Matriz;

const Matriz MATRIZ = {{
    {{  0,  1,  2,  3 }} ,
    {{ 10, 11, 12, 13 }} ,
    {{ 20, 21, 22, 23 }}
}};

int main()
{
    Matriz m = MATRIZ;
    // ...
}
```

Resolución de Problemas Usando Tipos Compuestos

Programa de notas de alumnos (v4)

- Lee el **nombre y la nota** de cada alumno (la cantidad **máxima** de alumnos es 20) y muestra el nombre y si está aprobado o suspenso, considerando que el alumno está aprobado si su nota es mayor o igual **a la nota media de todos los alumnos**.
- **La cantidad máxima de alumnos es 20, pero la cantidad actual de alumnos será leída de teclado.**

Resolución de Problemas Usando Tipos Compuestos (I)

```cpp
#include <iostream>
#include <string>
#include <array>
using namespace std;
const int MAX_ALUMNOS = 20;
struct Alumno {
    string nombre;
    double nota = 0;    // aseguramos que se inicializa correctamente a cero
};
typedef array<Alumno, MAX_ALUMNOS> AAlumnos;
struct LAlumnos {
    int nelms = 0;      // aseguramos que se inicializa correctamente a cero (lista vacía)
    AAlumnos elm;
};
double calc_media(const LAlumnos& v) {
    double media = 0.0;
    if (v.nelms > 0) {
        double suma = 0.0;
        for (int i = 0; i < v.nelms; ++i) {
            suma += v.elm[i].nota;
        }
        media = suma / double(v.nelms);
    }
    return media;
}
```

Resolución de Problemas Usando Tipos Compuestos (II)

```
void leer_alumno(Alumno& a)
{
    cout << "Introduzca el nombre del alumno: ";
    cin >> ws;
    getline(cin, a.nombre);
    cout << "Introduzca la nota del alumno: ";
    cin >> a.nota;
}
void leer_alumnos(LAlumnos& v)
{
    cout << "Introduzca total de alumnos: ";
    cin >> v.nelms;
    if (v.nelms <= 0 || v.nelms > int(v.elm.size())) {
        v.nelms = 0;  // lista vacía
        cout << "Error" << endl;
    } else {
        for (int i = 0; i < v.nelms; ++i) {
            leer_alumno(v.elm[i]);
        }
    }
}
```

Resolución de Problemas Usando Tipos Compuestos (III)

```
void mostrar_alumno(const Alumno& a, double umbral)
{
    if (a.nota >= umbral) {
        cout << "Alumno: " << a.nombre << " Aprobado" << endl;
    } else {
        cout << "Alumno: " << a.nombre << " Suspenso" << endl;
    }
}
void mostrar_alumnos(const LAlumnos& v, double umbral)
{
    for (int i = 0; i < v.nelms; ++i) {
        mostrar_alumno(v.elm[i], umbral);
    }
}
int main()
{
    LAlumnos alumnos;
    leer_alumnos(alumnos);
    double media = calc_media(alumnos);
    mostrar_alumnos(alumnos, media);
}
```

Resolución de Problemas Usando Tipos Compuestos (IV)

```
void anyadir_elemento(LAlumnos& v, const Alumno& a)
{
    if (v.nelms < int(v.elm.size())) {  // Si el nuevo elemento cabe en el array
        v.elm[v.nelms] = a;             // Añade el elemento al final de la lista
        ++v.nelms;                      // Incrementa la cuenta de elementos
    } else {
        cout << "Error" << endl;
    }
}
void leer_alumno_alternativo(Alumno& a)
{
    cout << "Introduzca el nombre del alumno (fin para terminar): ";
    cin >> ws;
    getline(cin, a.nombre);
    if (a.nombre != "fin") {
        cout << "Introduzca la nota del alumno: ";
        cin >> a.nota;
    }
}
void leer_alumnos_alternativo(LAlumnos& v)
{
    Alumno a;
    v.nelms = 0;  // lista vacía
    leer_alumno_alternativo(a);
    while (a.nombre != "fin") {
        anyadir_elemento(v, a);
        leer_alumno_alternativo(a);
    }
}
```

Procesamiento de Secuencia de Números

- Se debe leer una secuencia de números terminada en **cero**, de longitud inespecificada, y para cada número de la secuencia que se repita, mostrará la mayor distancia entre las posiciones consecutivas en que aparece cada número.
- La cantidad máxima de números distintos es 10, pero podrían ser menos.
- Es importante tener en cuenta que en un procesamiento secuencial no sabemos cuantos elementos se van a introducir en la secuencia, y por lo tanto, **no** se pueden almacenar **TODOS** los elementos de la secuencia en un array, ya que no podemos determinar su capacidad en tiempo de compilación.

```
Introduzca secuencia de numeros: 12 34 56 12 26 44 34 12 56 12 34 7 0
El resultado es:
 12 4
 34 5
 56 6
```

Resolución de Problemas. Procesamiento Secuencial (I)

```
#include <iostream>
#include <array>
using namespace std;

const int FIN = 0;
const int MAXDIST = 10;
struct Dato {
    int numero = -1;    // valores iniciales por defecto
    int ult_pos = -1;   // valores iniciales por defecto
    int max_dst = -1;   // valores iniciales por defecto
};
typedef array<Dato, MAXDIST> ADatos;
struct Datos {
    int nelms = 0;      // aseguramos que se inicializa correctamente a cero (lista vacía)
    ADatos elm;
};

void inic(Datos& dt)
{
    dt.nelms = 0; // lista vacía
}
int buscar(const Datos& dt, int num)
{
    int i = 0;
    while ((i < dt.nelms)&&(dt.elm[i].numero != num)) {
        ++i;
    }
    return i;
}
```

Resolución de Problemas. Procesamiento Secuencial (II)

```
void registrar(Datos& dt, int num, int pos)
{
    int i = buscar(dt, num);
    if (0 <= i && i < dt.nelms) {
        int dst = pos - dt.elm[i].ult_pos;
        if (dst > dt.elm[i].max_dst) {
            dt.elm[i].max_dst = dst;
        }
        dt.elm[i].ult_pos = pos;
    } else if (dt.nelms < int(dt.elm.size())) {
        dt.elm[dt.nelms].numero = num;
        dt.elm[dt.nelms].ult_pos = pos;
        dt.elm[dt.nelms].max_dst = -1;
        ++dt.nelms;
    }
}
void leer_y_registrar(Datos& dt)
{
    int numero;
    int pos = 0;
    inic(dt);
    cout << "Introduzca secuencia de numeros: " ;
    cin >> numero;
    while (numero != FIN) {
        ++pos;
        registrar(dt, numero, pos);
        cin >> numero;
    }
}
```

Resolución de Problemas. Procesamiento Secuencial (III)

```
void mostrar_resultado(const Datos& dt)
{
    cout << "El resultado es: " << endl;
    for (int i = 0; i < dt.nelms; ++i) {
        if (dt.elm[i].max_dst >= 0) {
            cout << " " << dt.elm[i].numero << " " << dt.elm[i].max_dst << endl;
        }
    }
    cout << endl;
}
int main()
{
    Datos dt;
    leer_y_registrar(dt);
    if (dt.nelms == 0) {
        cout << "Error" << endl;
    } else {
        mostrar_resultado(dt);
    }
}
```

Resolución de Problemas. Multiplicación de Matrices (I)

- Multiplicación de Matrices: producto de 2 matrices (de máximo 10x10 elementos).

```
#include <iostream>
#include <cassert>
#include <array>
using namespace std;

const int MAX = 10;

typedef array<double, MAX> Fila;
typedef array<Fila, MAX> Tabla;
struct Matriz {
    int n_fil = 0;
    int n_col = 0;
    Tabla datos;
};
```

Resolución de Problemas. Multiplicación de Matrices (II)

```
void leer_matriz(Matriz& m) {
    cout << "Dimensiones?: ";
    cin >> m.n_fil >> m.n_col;
    if ((m.n_fil <= 0)||(m.n_fil > int(m.datos.size()))
        ||(m.n_col <= 0)||(m.n_col > int(m.datos[0].size()))) {
        cout << "Error: tamaño erróneo" << endl;
        m.n_fil = 0;
        m.n_col = 0;
    } else {
        cout << "Introduce 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];
            }
        }
    }
}
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;
    }
}
```

Resolución de Problemas. Multiplicación de Matrices (III)

```
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;
}
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_fila_por_col(a, b, f, c);
        }
    }
}
```

Resolución de Problemas. Multiplicación de Matrices (IV)

```
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);
        cout << "Resultado:" << endl;
        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.

Aplicación: Agenda Personal (I)

```
#include <iostream>
#include <string>
#include <array>
using namespace std;

const int MAX_PERSONAS = 50;
const int OK = 0;
const int AG_LLENA = 1;
const int NO_ENCONTRADO = 2;
const int YA_EXISTE = 3;


struct Direccion {
    int num = 0;
    string calle;
    string piso;
    string cp;
    string ciudad;
};
struct Persona {
    string nombre;
    string tel;
    Direccion direccion;
};
```

```
typedef array<Persona, MAX_PERSONAS> Personas;
struct Agenda {
    int n_pers = 0;  // lista vacía
    Personas pers;
};


void inicializar(Agenda& ag)
{
    ag.n_pers = 0;   // lista vacía
}
```

Aplicación: Agenda Personal (II)

```
void leer_dir(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_dir(per.direccion);
}
```

```
void escribir_dir(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_dir(per.direccion);
}
```

Aplicación: Agenda Personal (III)

```
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 eliminar(Agenda& ag, int pos, int& ok)
{
    if (0 <= pos && pos < ag.n_pers) {
        ag.pers[pos] = ag.pers[ag.n_pers - 1];
        --ag.n_pers;
        ok = OK;
    } else {
        ok = NO_ENCONTRADO;
    }
}
void borrar_persona(const string& nombre, Agenda& ag, int& ok)
{
    int i = buscar_persona(nombre, ag);
    eliminar(ag, i, ok);
}
```

Aplicación: Agenda Personal (IV)

```
void anyadir(Agenda& ag, const Persona& per, int& ok)
{
    if (ag.n_pers == int(ag.pers.size())) {
        ok = AG_LLENA;
    } else {
        ag.pers[ag.n_pers] = per;
        ++ag.n_pers;
        ok = OK;
    }
}
void anyadir_persona(const Persona& per, Agenda& ag, int& ok)
{
    int i = buscar_persona(per.nombre, ag);
    if (0 <= i && i < ag.n_pers) {
        ok = YA_EXISTE;
    } else {
        anyadir(ag, per, ok);
    }
}
```

Aplicación: Agenda Personal (V)

```
void modificar_persona(const string& nombre, const Persona& nuevo,
                       Agenda& ag, int& ok)
{
    int i = buscar_persona(nuevo.nombre, ag);
    if (0 <= i && i < ag.n_pers) {
        ok = YA_EXISTE;
    } else {
        borrar_persona(nombre, ag, ok);
        if (ok == OK) {
            anyadir(ag, nuevo, ok);
        }
    }
}

void mostrar_persona(const string& nombre, const Agenda& ag, int& ok)
{
    int i = buscar_persona(nombre, ag);
    if (0 <= i && i < ag.n_pers) {
        escribir_persona(ag.pers[i]);
        ok = OK;
    } else {
        ok = NO_ENCONTRADO;
    }
}
```

Aplicación: Agenda Personal (VI)

```
void mostrar_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. - Mostrar Agenda" << endl;
    cout << "x. - Salir" << endl;
    do {
        cout << "Introduzca Opcion: ";
        cin >> opcion;
    } while ( ! (((opcion >= 'a') && (opcion <= 'e')) || (opcion == 'x')));
    return opcion;
}
```

Aplicación: Agenda Personal (VII)

```
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;
    }
}
```

Aplicación: Agenda Personal (VIII)

```
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 >> ws;
            getline(cin, nombre);
            mostrar_persona(nombre, ag, ok);
            escribir_cod_error(ok);
            break;
```

Aplicación: Agenda Personal (IX)

```
        case 'c':
            cout << "Introduzca Nombre" << endl;
            cin >> ws;
            getline(cin, nombre);
            borrar_persona(nombre, ag, ok);
            escribir_cod_error(ok);
            break;
        case 'd':
            cout << "Introduzca Nombre" << endl;
            cin >> ws;
            getline(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':
            mostrar_agenda(ag, ok);
            escribir_cod_error(ok);
            break;
        }
    } while (opcion != 'x' );
}
```