# UD1: El lenguaje de programación C++

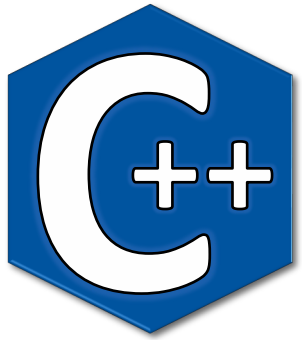
# Tema 2: Programación Basada en Objetos. Librería standard de C++

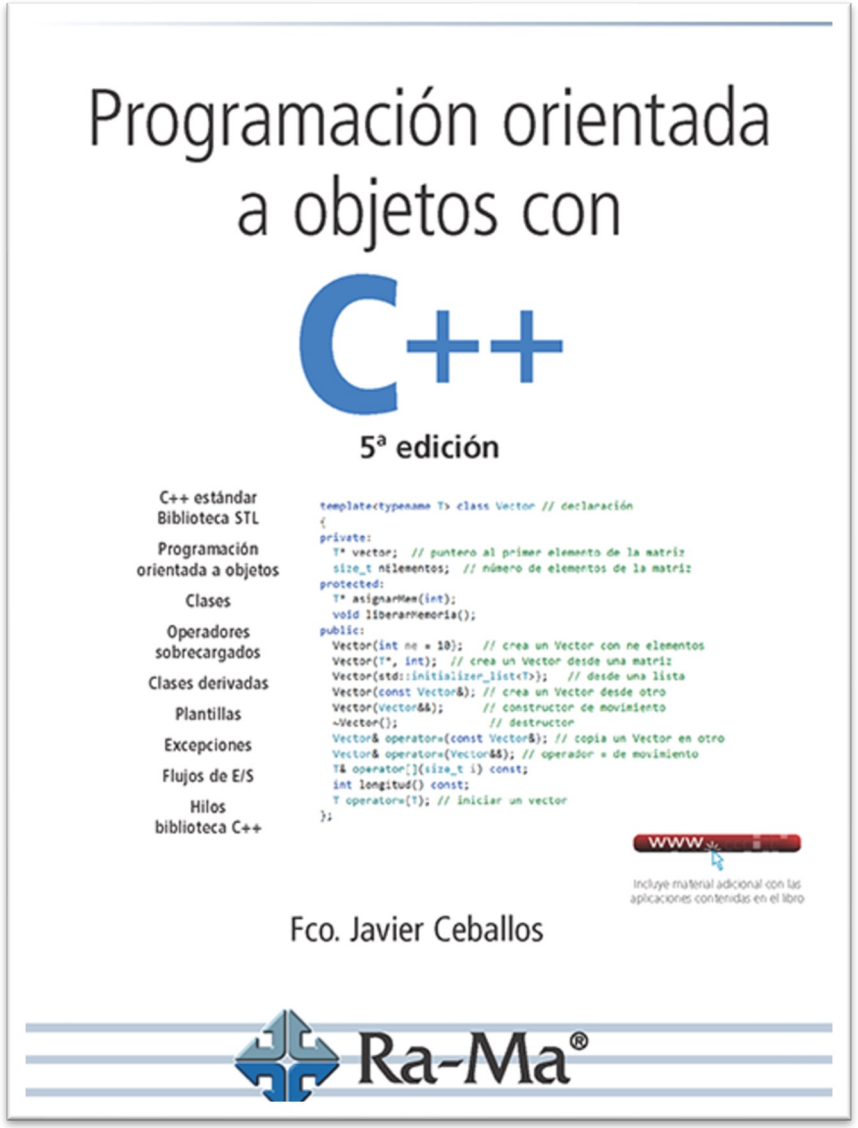

*« Me lo explicaron y lo olvidé, lo vi y lo aprendí, lo hice y lo entendí.»*

**- Confucio -**

# Bibliografía Ceballos

- Capítulos 2 (introducción a orientación a objetos, OO), 4 y 11
- El capítulo 4 describe gran parte de la librería estándar y algo de flujos. El capítulo 11 se centra en flujos. El capítulo 2 introduce términos de la orientación a objetos que son necesarios para utilizar la librería estándar



Universidad Politécnica de Cartagena

# Necesidad de Orientación a Objetos (OO)

- Las estructuras (disponibles también en C) permiten agrupar un conjunto de valores altamente relacionados para tratarlos como una unidad

```
struct datos_personales_t {
  int dia, mes, anyo, tfno;
  char nombre[30], ape1 [40], ape2[40], email[40];
};
void imprimir (struct datos_personales_t &d);
```

- La orientación a objetos amplía la idea de estructura, englobando también a las funciones que operan sobre los datos: **objeto = datos + algoritmos**

```
void copiar (struct datos_personales_t, struct datos_personales_t);
                    // facil equivocarse en el orden!
```

- En OO, el objeto sobre el que se opera está claramente separado del resto de parámetros (los valores necesarios) que requiere la función. “Ejemplo”:

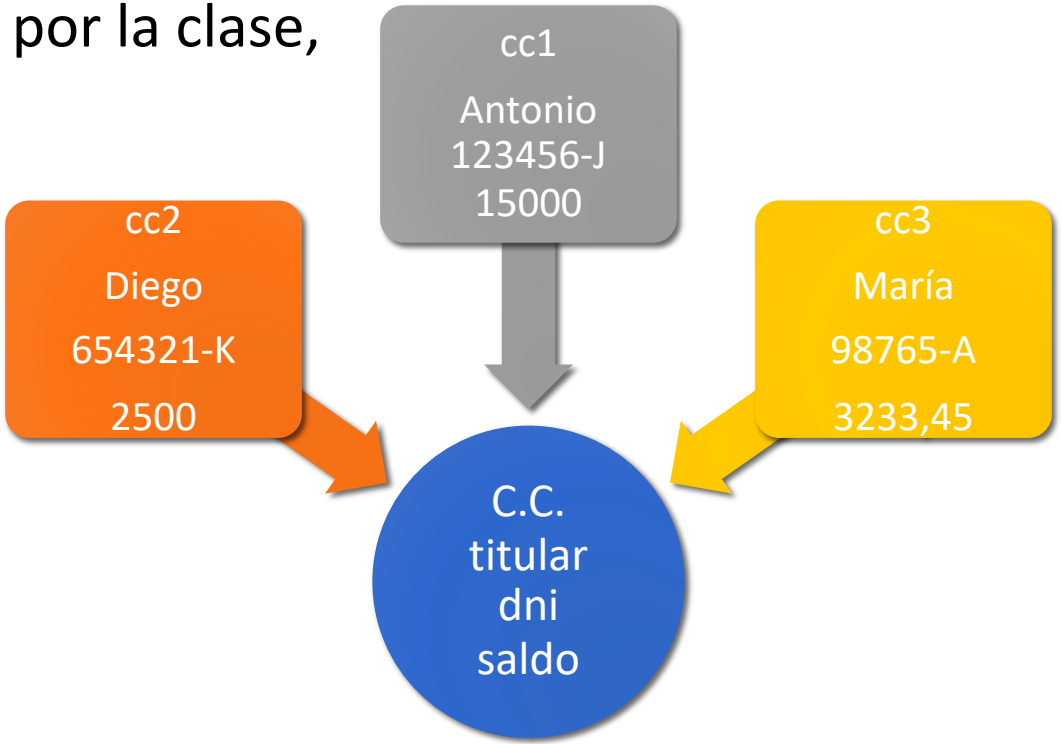
```
datos_personales_t obj1, obj2;
obj1.copiar(obj2); // se opera sobre obj1, obj2 son datos
```

# Conceptos fundamentales (I)

*Un programa es un conjunto de objetos que se comunican entre sí mediante mensajes. Una clase agrupa datos junto a las funciones (métodos) para modificarlos*

- **Clase**: definición de las propiedades y comportamiento de un nuevo tipo de dato, como si fuera un molde. La creación de un objeto a partir de una clase se llama "instanciación"
- **Objeto**: instancia concreta y diferenciable de una clase. Tiene las propiedades (variables) definidas por la clase, pero con distintos valores


https://cienciaeducacion502.blogspot.com/2014/01/introduccion-las-clases-y-los-objetos.html



Universidad Politécnica de Cartagena

# Conceptos fundamentales (II)

- **Método**: algoritmo asociado a una clase, ejecutado tras la recepción de un mensaje. Representan lo que la clase puede hacer. Un método puede cambiar las propiedades del objeto, realizar un cálculo o generar mensajes para otros objetos. Son funciones, algunas con propósito particular (constructores, destructor, operadores, etc.)

- **Mensaje**: invocación de un método de un objeto por parte de otro

- **Propiedad o atributo**: variables (de cualquier tipo de dato, incluido otra clase) que almacenan el estado del objeto. Es posible controlar su visibilidad y accesibilidad: públicas o privadas. Generalmente son privadas, solo modificables mediante métodos definidos en la clase

Universidad Politécnica de Cartagena

# Tipos de métodos de una clase

- **Constructores**: definen cómo crear un objeto. Se invoca automáticamente al declarar una variable de un tipo clase. Una clase suele tener múltiples constructores, incluso uno sin parámetros (*constructor por defecto*):

```
Mi_Clase mi_obj; // se invoca el constructor por defecto
```

- **Destructor**: define cómo terminar un objeto. Se invoca automáticamente por C++ (por ejemplo, cuando termina una función), aunque también podemos invocarlo (no es lo habitual). Normalmente, no es necesario preocuparse por él
- **Métodos de objeto**: constituyen el grueso de la funcionalidad del objeto. Operan sobre los atributos del objeto

```
mi_obj.met1(parms);
```

  - C++ permite redefinir operadores, que son considerados métodos de objeto

- **Métodos de clase**: no son habituales. Operan sobre los atributos de la clase

```
Mi_Clase::met3(parms);
```

Universidad Politécnica de Cartagena

# Introducción a la librería estándar de C++

- La cantidad de funcionalidad proporcionada en forma de librerías por un lenguaje es un factor cada vez más importante en su utilización:
  - Uno de los puntos débiles de C, que solo proporciona 29
  - Uno de los puntos fuertes de Java/C#, que proporcionan cientos de librerías por defecto
- Más de 80 librerías en C++: gestión de memoria, **utilidades**, gestión de excepciones, **strings, contenedores, iteradores, algoritmos**, límites numéricos, **cálculos numéricos**, **gestión de entrada/salida (streams)**, localización, expresiones regulares, **hilos**, operaciones atómicas, compatibilidad con C, etc.

Universidad Politécnica de Cartagena

# Inicialización y asignación

- Son dos acciones diferentes, aunque a menudo se confunden
- La inicialización solo se produce en el momento en que se declara una variable
- La asignación se produce cada vez que se utiliza el = (salvo en la declaración)

```
int a; // inicialización, invoca el constructor "por defecto"
a = 77; // asignación
double b = 3.14; // inicialización. El '=' es engañoso
```

- C++ introduce dos formas más de inicialización, con `()` y `{}`, que no necesitan `=`
- Se utilizan principalmente al instanciar objetos, pero también con variables de cualquier tipo: `int a {9};`
- `{}` se denomina *inicialización uniforme* (forma universal de inicialización en C++). Y lo es, salvo que la clase tenga un constructor con un parámetro de tipo `initializer_list` (no existen muchos casos, pero algunos son notables)

# Cadena de caracteres (*String*)

La clase **string**, definida en `<string>`, es una mejora del vector de caracteres (**char** `msg[]`) de C, que tiene varios defectos: deben acabar en `'\0'`, es difícil conocer su tamaño, se convierten en punteros en cuanto se pasan a una función, etc.

```
#include <iostream>
#include <string>
int main() {
 std::string str1 {"Tiempo"}, str2 {"Real"}, str3; // declaración
 std::cout << "1 -> " << str1 << '-' << str2 << '\n';
 str3 = str2; // asignación
 std::cout << "2 -> " << str1 << '-' << str3 << '\n';
 str2[0] = 'r'; // método operator[]
 std::cout << "3 -> " << str1 << '-' << str2 << '\n';
 if (str2 == str3) { std::cout << "\tIguales :-( \n"; }
 str3[0]='r';
 if (str2 == str3) { std::cout << "\tIguales! :-) \n"; }
 str3 = str1 + '-' + str2; // método concatenar
 std::cout << "4 -> " << str3 << '\n';
}
```

```
1 -> Tiempo-Real
2 -> Tiempo-Real
3 -> Tiempo-real
     Iguales! :-)
4 -> Tiempo-real
```

# <string> (colección)

Por razones de espacio, omito en muchos códigos
**#includes** y **using namespace std;**

```
std::string datos (const std::string &str) {
 return "(c: " + std::to_string(str.capacity()) +
   ", s:" + std::to_string(str.size()) + ')';
}
int main() {
 string str {"Programación en tiempo real"}; // ctor
 string subr (str.size(), '='); // ctor
 cout << "1.- " << str << "(c: " << str.capacity()
  << (str.empty()? ", Empty)": ", no Empty)")
  << endl;
 cout << "2.- " << subr << datos (subr) << endl;
 string sub_str {str, 5, 10}; // ctor
 cout << "3.- " << sub_str << datos (sub_str) << endl;
 str.front() = 'p';
 ++str.back();
 str[1] = 'R';
 str.at(5) = '9';
 cout << "4.- " << str << datos (str) << endl;
 string str2 = "Asignatura: " + str;
 cout << "5.- " << str2 << datos (str2)<< endl;
 str2.assign(5, '*');
 cout << "6.- " << str2 << datos (str2) << endl;
 str2.assign("Hola mundo!");
 cout << "7.- " << str2 << datos (str2) << endl;
 str2 = "Programacion en tiempo real";
 cout << "8.- " << str2 << datos (str2) << endl;
 std::swap (str2, subr);
 cout << "9.- " << subr << endl << str2 << endl;
 string formula="Área círculo:π·r², \u03C0·r\u00B2";
 cout << formula << endl;
}
```

**std::string** contiene un puntero a *char* (y más datos).
Gestiona la memoria para crecer si fuera necesario

```
1.- Programación en tiempo real (c: 31, no Empty)
2.- ============================(c: 31, l:28)
3.- amación e(c: 22, l:10)
4.- pRogr9mación en tiempo ream(c: 31, l:28)
5.- Asignatura: pRogr9mación en tiempo ream(c: 47, l:40)
6.- *****(c: 22, l:5)
7.- Hola mundo!(c: 22, l:11)
8.- Programacion en tiempo real(c: 31, l:27)
9.- Programacion en tiempo real
============================
Área círculo: π·r², π·r²
```

Universidad Politécnica de Cartagena

# Flujos (*streams*) (I)

- Un *stream* es un canal para recibir/enviar o manipular datos de entrada/salida
  - Internamente, un *stream* es un vector de caracteres
  - Externamente, un *stream* proporciona una interfaz universal para acceder a cualquier medio de almacenamiento. Gracias a seguir una jerarquía de herencia, es posible cambiar el flujo que se utiliza en un programa manteniendo en general el código de utilización del *stream*
- Un *stream* siempre está asociado con una fuente de datos (*input stream*) o con un sumidero de datos (*output stream*) o con ambos
- Los flujos se utilizan en C++ (y Java, C#, etc.) para leer/escribir en teclado o consola (`<iostream>`), ficheros (`<fstream>`), etc., de forma *homogénea*
- "**Strings are streams and streams are strings**". Ambos tipos están basados en vectores de caracteres, pero string ofrece acceso aleatorio al contenido y en stream el acceso es secuencial

| C | C++ |
|---|---|
| `stdin` | `std::cin` |
| `stdout` | `std::cout` |
| `stderr` | `std::cerr` |
| | `std::clog` |

# Flujos (*streams*) (II)

- C++ utiliza los métodos **`operator<<`** y **`operator>>`** para definir cómo se envía una variable a un flujo y cómo se cambia el valor de una variable a partir de los valores contenidos en un flujo, respectivamente
  - Los tipos primitivos de datos ya tienen predefinidos esos métodos
  - C++ requiere que el programador sobrecargue (redefina) estos métodos para que los tipos de datos definidos por él puedan utilizar flujos

```
istream& operator>>(istream &s, tipo &val) {}    ostream& operator>>(ostream &s, tipo &val) {}
```

- Todos los flujos comparten un conjunto de métodos:
  - `good()` : comprueba si las operaciones de E/S están disponibles (el flujo no tiene errores)
  - `eof()`: comprueba si se ha llegado al *end-of-file*
  - `fail()`: comprueba si el flujo tiene un error recuperable
  - `bad()`: comprueba si el flujo tiene un error no recuperable
  - `clear()`: elimina las marcas de error y eof, y deja el flujo en estado correcto
  - `is_open()`: comprueba si el flujo está abierto
  - `close()`: cierra el flujo y, habitualmente, el recurso asociado (fichero, socket, etc.)

Universidad Politécnica de Cartagena

# Entrada y salida de datos (cualquier flujo)

- La **entrada** es problemática, ya que habitualmente son *cadenas de caracteres*, es decir, un texto, que proviene del teclado, un fichero, etc., a partir de las que el programa debe obtener un valor de un tipo concreto (`long`, `double`, etc.).
  - **Entrada sin formato**: simplemente, se copia el texto del flujo de entrada sin interpretar. Se utiliza la función `getline()` de la librería `<string>`

```
void getline(istream& is, string& str, char d='\n')
```

  - **Entrada con formato**: intenta obtener un valor de un tipo concreto a partir del texto de entrada. Puede fallar si la entrada no es convertible (por ejemplo, si esperamos un número pero el usuario introduce letras). Utiliza **`operator`**`>>`
- La **salida** es sencilla porque el programa conoce el tipo del valor y suele ser fácil convertir dichos valores a una cadena de caracteres para mostrarla en consola, almacenarla en fichero, etc.
- También existe la opción de realizar entrada/salida en binario

# Entrada de datos (cualquier flujo)

**Entrada datos (cadena de caracteres) del teclado, un fichero, etc.**


- Los caracteres separadores delimitan la entrada de datos en un *stream*, y son:
  - `space (0x20, ' ')`
  - `horizontal tab (0x09, '\t')`
  - `vertical tab (0x0b, '\v')`
  - `line feed (0x0a, '\n')`
  - `carriage return (0x0d, '\r')`
  - `form feed (0x0c, '\f')`
- `operator>>` descarta los caracteres separadores previos y después intenta procesar los caracteres de entrada que queden hasta encontrar un carácter separador, que no consume (lo deja en el *stream*, para la siguiente operación)
- `getline()` captura (pero no procesa) la entrada de datos hasta encontrar un carácter (`'\n'` por defecto, pero es configurable), que sí consume
- Si se utiliza `getline` tras `operator>>`, se puede obtener una cadena vacía. Solución:

Elimina separadores al inicio del flujo

```
string buffer;
getline(cin >> std::ws, buffer);
```

# Entrada de datos, mezcla operator>> y getline

```
#include <iostream>
#include <string>

using namespace std;
int main() {
  int a;
  cout << "1 numero? >>> ";
  cin >> a;
  string str1, str2, str3;
  cout << "2 palabras? >>> ";
  cin >> str1; cin >> str2;
  cout << "Str1: " <<str1<< ", str2: "
     << str2<< '\n';
  cout << "1 frase? >>> ";
  //getline(cin, str3);      // (1)
  //getline(cin>>ws, str3);   // (2)
  cout << "Str3: " << str3 << '\n';
  cout << "3 palabras? >>> ";
  cin >> str1; cin >> str2;
  cout << "Str1: " << str1
     << ", str2: " << str2 << '\n';
  cin >> str3;
  cout << "\t y quedaba para Str3: "
     << str3 << '\n';
}
```

⏎ es ENTER  Con (1)

```
1 numero? >>> 123⏎
2 palabras? >>> tiempo⏎
real⏎
Str1: tiempo, str2: real
1 frase? >>> Str3:
3 palabras? >>>    CTRL C
```

Con (2)

```
1 numero? >>> 321⏎
2 palabras? >>> tiempo⏎
real⏎
Str1: tiempo, str2: real
1 frase? >>> En un lugar de La Mancha⏎
Str3: En un lugar de a Mancha
3 palabras? >>> hola⏎
tiempo⏎
Str1: hola, str2: tiempo
real⏎
     y quedaba para Str3: real
```

# Código: arreglar error entrada de datos con formato

¿Qué pasa si el programa solicita un número pero el usuario introduce letras o símbolos?

```
void v1() {
  int a=0;
  do {
    cout << "Numero (>0)? >>> ";
    cin >> a;
    if (cin.fail()) {
      cin.clear(); // elimina error
      cin.ignore(9999, '\n'); // vacía cin
    }
  } while (a <= 0);
}
```

```
void v2() {
 int a=0;
 do {
  cout << "Numero (>0)? >>> ";
  if (!(cin >> a)) {
   cin.clear(); // elimina error
   cin.ignore(9999, '\n'); // vacía cin
  }
 } while (a <= 0);
}
```

https://en.cppreference.com/w/cpp/io/basic_ios/operator_bool

```
Numero (>0)? >>> -3⏎
Numero (>0)? >>> -2⏎
Numero (>0)? >>> tiempo real⏎
Numero (>0)? >>> @@^\nlk⏎
Numero (>0)? >>> hola44⏎
Numero (>0)? >>> 100⏎
```

Universidad Politécnica de Cartagena

# Entrada/salida en ficheros <fstream>(I)

- Acceso básico (crudo), de lectura/escritura, al contenido de un fichero: `filebuf`
- Soporte para funciones de alto nivel en el acceso a ficheros: `ifstream` (lectura); `ofstream` (escritura); `fstream` (lectura/escritura)
- Básicamente, se utilizan `ifstream` y `ofstream`, dejando `fstream` para cuando es necesario mezclar operaciones de lectura y escritura (poco habitual)
- Proporcionan varios constructores, pero destaca el que acepta la ruta (`string`) del fichero a abrir

| ifstream | ofstream |
|---|---|
| • `operator>>`: extrae datos con formato<br>• `getline`: extrae caracteres hasta encontrar un separador (u otro carácter que se le pase como parámetro)<br>• `sync`: sincroniza el contenido del fichero | • `operator<<`: inserta datos con formato<br>• `put/write`: inserta un carácter/vector de caracteres<br>• `flush`: vuelca el contenido del flujo (comando para el sistema de E/S virtual del SO) |

Universidad Politécnica de Cartagena

# Entrada/salida en ficheros <fstream> (II)

```
int main() {
 std::string nombre_fe;
 std::cout << "Nombre fichero
                entrada? ";
 getline(std::cin, nombre_fe);
 std::ifstream fe {nombre_fe};
 if (fe) { // invoca operator_bool()
  std::string nombre_fs;
  std::cout << "Nombre fichero
                 salida? ";
  getline(std::cin, nombre_fs);
  std::ofstream fs {nombre_fs};
  if (fs) {
   std::string msg, aux;
   std::cout << "Mensaje? ";
   std::cin >> msg;
   do {
      getline(fe, aux);
      fs << '['<<msg<<"] -> "
         << aux <<'\n';
    } while (!fe.eof());
   } else {
   std::cout << "[ERROR]: Fichero '"
     << nombre_fs << "' no creado\n";
   }
  } else {
   std::cout << "[ERROR]: Fichero '"
     <<nombre_fe<<"' no encontrado\n";
  }
 return 0;
}
```

# Strings como streams: <sstream>

- Permite tratar `std::string` como si fuera un flujo. Particularmente útil es el uso de **`operator`**`>>` para realizar lectura con formato del contenido del string
- Proporciona una clase para leer un string como un flujo y otra para escribir en un string como si fuera un flujo

```
int main() {
  string datos {"tiempo 77 real 3.14"};
  string salida;
  istringstream is {datos};
  ostringstream os {salida, ios_base::ate}; // ate --> at-the-end
  string txt; int ent; double r;
  is >> txt; is >> ent;
  os << ent << " " << txt;
  cout << "os: " << os.str() << '\n';
  cout << "Txt: |" << txt << "| int: " << ent;
  is >> txt; is >> r;
  cout << "\nTxt2: |" << txt << "| double: " << r;
  return 0;
}
```

```
os: 77 tiempo
Txt: |tiempo| int: 77
Txt2: |real| double: 3.14
```

# Código: tipos enumerados y flujos

```
istream& operator>>(istream &s, tipo &val){}
```

```
ostream& operator>>(ostream &s, tipo &val){}
```

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

enum class T_RGB { red, green, blue };

// funciona con cualquier stream!
istream& operator>>(istream &s, T_RGB &val) {
  string str;
  s >> str;
  if   (str == "RED")   val = T_RGB::red;
  else if (str == "GREEN") val = T_RGB::green;
  else if (str == "BLUE")  val = T_RGB::blue;
  else s.setstate(ios_base::failbit);
  return s;
}
// también puede ser string to_string(XXX)
const char* to_string (const T_RGB val) {
  switch (val) {
    case T_RGB::red   : return "RED";
    case T_RGB::green : return "GREEN";
    case T_RGB::blue : return "BLUE";
  }
}
```

Referencia (&)

```
ostream& operator<<(ostream &s, T_RGB &val) {
  s << to_string(val); // reusa función previa
  return s;
}

int main(int argc, const char *argv[]) {
  T_RGB val = T_RGB::red;
  cout << to_string(val) << endl;
  val = T_RGB::blue;
  cout << val << endl;
  cout << "Value: ";
  cin >> val;
  if( cin.fail() )
    cout << "ERROR" << endl;
  else
    cout << val << endl;
  istringstream ss {"GREEN"};
  ss >> val;
  cout << "STR: " << val << endl;
  return 0;
}
```

Inicialización uniforme { }

# Retornar más de un valor de una función

**C++ 17**

```
#include <iostream>
#include <utility>
#include <tuple>
#include <string>

std::pair<int, std::string> devuelve2 () {
  int a = 1;
  std::string str {"Tiempo real_" +
     std::to_string(a)};
  return std::make_pair (a, str);
}

std::tuple<int, double, char, std::string>
devuelve4 () {
  int a = 2; char c='@';
  double d {31415e-4};
  std::string str {"Tiempo real_" +
    std::to_string(a)};
  return std::make_tuple (a, d, c, str);
}
```

C++ crea las variables y les copia los valores retornados (*enlace estructurado*)

```
int main() {
  auto [el_int, el_string] = devuelve2();
  std::cout << "Int: " << el_int
     << ", str: " << el_string << '\n';
  auto [vi, vd, vc, vs] = devuelve4();
  std::cout << "VI: " << vi << ", VD: "
     << vd << ", VC: "<< vc << ", VS: "
     << vs << '\n';
  return 0;
}
```

```
Int: 1, str: Tiempo real_1
vi: 2, vd: 3.1415, vc: @, vs: Tiempo real_2
```

- `std::pair` solo almacena 2 valores
- `std::tuple` almacena n-valores
- Se puede hacer lo mismo con una estructura, pero habría que definir el tipo previamente (el enlace estructurado sería igual)

# Retornar valor, opcionalmente, con <optional>

```
#include <iostream>
#include <optional>
#include <string>
using Tipo_t =
    std::optional<std::string>;

Tipo_t a_veces (bool con_string) {
  std::string str {"Tiempo Real"};
  if (con_string) {
    return std::make_optional(str);
  }
  return std::nullopt;
}
int main() {
  auto val = a_veces(true);
  if(val) {
    std::cout << "String :-) -> "
         << *val << '\n';
  }
  val.reset();
  val = a_veces(false);
  if(val.has_value()) {
    std::cout << "String :-( -> "
         << *val << '\n';
  } else {
    std::cout << "No hay valor :-) ";
    std::cout << val.value_or("Valor
         alternativo\n");
  }
  return 0;
}
```

Crear un alias para el tipo

optional vacío

Acceso al valor (como un puntero)

Borra el optional

Si no tiene valor, retorna el del parámetro

**C++ 17**

```
String :-) -> Tiempo Real
No hay valor :-) . Valor alternativo
```

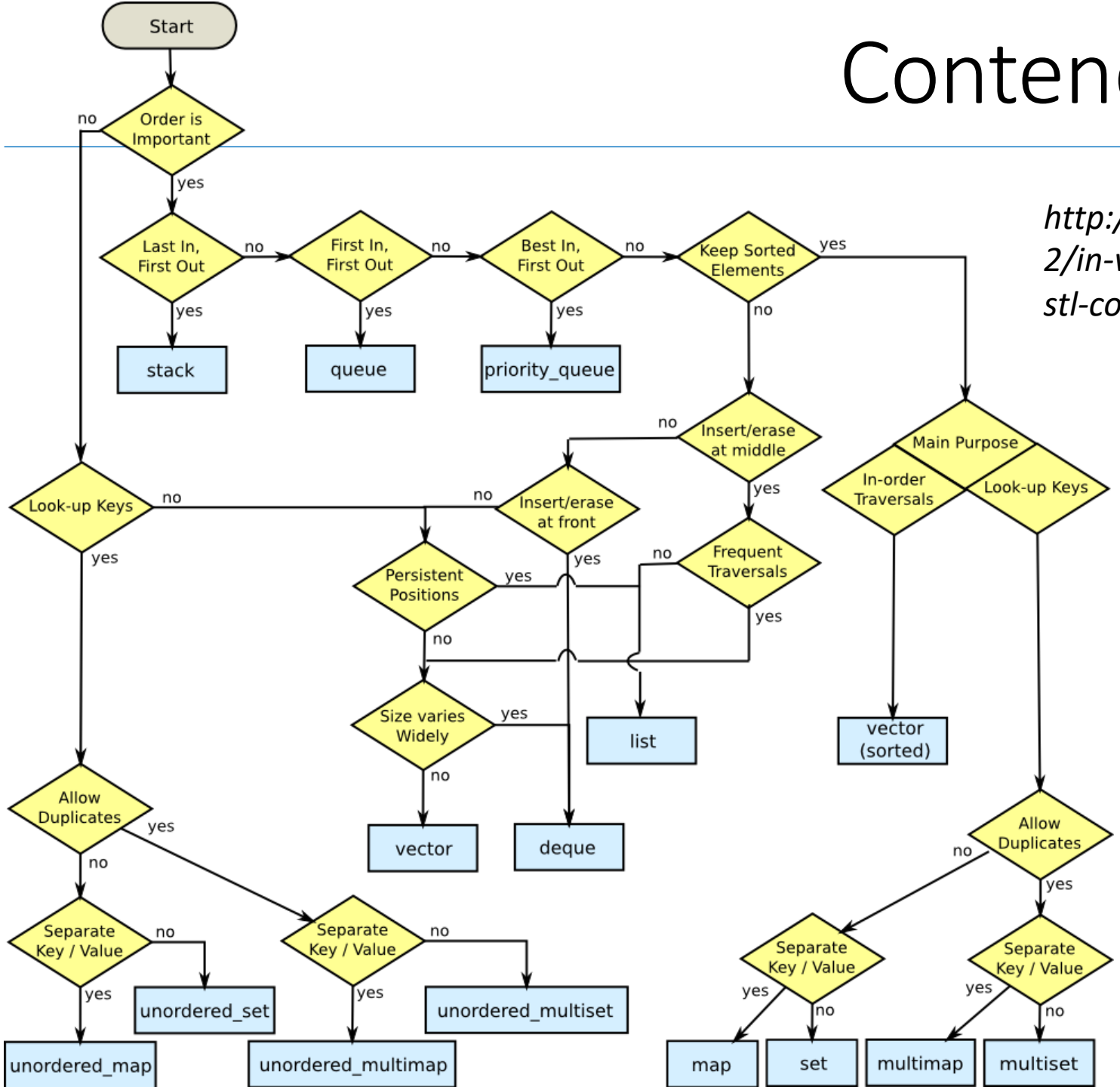
# Código: tipos enumerados y optional

```
enum class T_RGB { red, green, blue };
using Tipo_t = std::optional<T_RGB>;

istream& operator>>(istream &s, Tipo_t &val) {
  string str;
  s >> str;
  if (str == "RED")
    { val = make_optional(T_RGB::red); }
  else if (str == "GREEN")
    { val = make_optional(T_RGB::green); }
  else if (str == "BLUE")
    { val = make_optional(T_RGB::blue); }
  else {
      s.setstate(ios_base::failbit);
      val=nullopt;
  }
  return s;
}

string to_string (const T_RGB &val) {
  switch (val) {
    case T_RGB::red   : return "RED";
    case T_RGB::green : return "GREEN";
    case T_RGB::blue : return "BLUE";
    default: return "ERROR";
  }
}
```

```
ostream& operator<<(ostream &s, T_RGB &val) {
  s << to_string(val); // reusa función previa
  return s;
}

int main() {
  Tipo_t val = make_optional(T_RGB::red);
  cout << "1 -> " << to_string(*val) << endl;
  val = make_optional(T_RGB::blue);
  cout<< "2 -> "<<*val << endl;
  cout << "Value: ";
  cin >> val;
  if( cin.fail() ) {
    cout << "3 -> ERROR" << endl;
  } else {
    cout << "3 -> "<< *val << endl;
  }
  istringstream ss {"GREEN"};
  ss >> val;
  cout << "STR: " << *val << endl;
  return 0;
}
```

Universidad Politécnica de Cartagena

# Contenedores en C++




*http://stackoverflow.com/questions/47143 2/in-which-scenario-do-i-use-a-particular-stl-container/22671607#22671607*

Cuando se maneja una cantidad “no muy grande” de elementos, la arquitectura HW del sistema hace que, en general, **array**/**vector** sea la elección más eficiente

`string` también se considera un contenedor

# Plantillas (*templates*)

- C++ permite crear plantillas de funciones y plantillas de clases
- Una función parametriza el valor de los parámetros, mientras que una plantilla parametriza los tipos de dato (y también valores) de una función o una clase
  - Como si el compilador hiciera *copiar-pegar* y *buscar-reemplazar* antes de compilar
- Facilitan la programación en lenguajes con tipado fuerte:

```
template <class T> // los parámetros genéricos van entre <>
T maximo(T x, T y) {
  if (x > y) return x;
  return y;
};

int i_a, i_b;
string s_a, s_b;
maximo<int,int> (i_a, i_b);
maximo (s_a, s_b);
```

```
int maximo(int x, int y) {
  if (x > y) return x;
  return y;
};
```

```
string maximo(string x, string y) {
  if (x > y) return x;
  return y;
};
```

Normalmente, C++ infiere el tipo de los los parámetros genéricos. En algunos casos, ej. cuando no son parámetros de la función, hay que indicar el tipo concreto (`maximo<int, int>`).

# Nuevo bucle: for-de-colección

- C++11 define un nuevo bucle **for** para iterar sobre un rango de valores, habitualmente una colección:

```
for ( init; range_declaration : range_expression ) {
  loop_statements
}
```

Obligatorio
Opcional

- Recorre directamente **los valores** (¡no los índices!) almacenados en `range_expression`. C++ utiliza *iteradores* para recorrer el rango. Desde C++20, soporta inicialización

```
string texto()
  { return "Tiempo Real"; }

int main() {
  for (auto str {texto()}; auto& c : str)
    cout << '|' << c << '|';
  return 0;
}
```

```
|T||i||e||m||p||o|| ||R||e||a||l|
```

Universidad Politécnica de Cartagena

# Creación de <vector>

- Un vector es básicamente un puntero (y más información) a una zona continua del *heap* (montículo) donde se encuentran realmente los valores
- Esta clase gestiona la memoria, permitiendo que el vector crezca dinámicamente
  - Cada vez que crece, se dobla la capacidad y se copian todos los valores que había
- Ojo, ¡la interfaz de la clase permite utilizarla de varias formas!
- Cuando se declara el vector hay que pensar si se quiere almacenar los *objetos* o *referencias a los objetos*: `vector<string>` vs. `vector<string*>`
- Tiene un constructor con `initializer_list`, de forma que `{}` y `()` son distintos
  1. `vector()`: crea el vector vacío, sin inicializar la memoria asociada
  2. `vector(size_type count, const T& val)`: crea un vector con 'count' copias de 'val'
  3. `vector(size_type count)`: crea un vector con 'count' elementos inicializados por defecto
  4. `vector(initializer_list<T> init)`: crea un vector con copias de los valores de 'init'

```
std::vector<std::string> v1; // ctor. (1)
std::vector<std::string> v2 (10, "tiempo real"); // ctor. (2)
std::vector<std::string> v4 {"tiempo", "real", "c++11"}; // ctor. (4)
```

Universidad Politécnica de Cartagena

Los constructores (2), (3) y (4) reservan e inicializan la memoria asociada al vector en el *heap*. Por ejemplo:

```
std::vector<short> v {10,11,12,13,14,15};
// representación aproximada de la memora
```

**RAM - Pila**

| Dirección | MSB | 6 | 5 | 4 | 3 | 2 | 1 | LSB |
|---|---|---|---|---|---|---|---|---|
| 0x111 | Puntero al comienzo del vector 0xF11 | | | | | | | |
| 0x112 | | | | | | | | |
| 0x113 | | | | | | | | |
| 0x114 | | | | | | | | |
| 0x115 | Puntero de inserción 0xF1D | | | | | | | |
| 0x116 | | | | | | | | |
| 0x117 | | | | | | | | |
| 0x118 | | | | | | | | |
| 0x119 | Tamaño del vector 6 | | | | | | | |
| 0x11A | | | | | | | | |
| 0x11B | | | | | | | | |
| 0x11C | | | | | | | | |
| 0x11D | Capacidad del vector por ejemplo, 8 | | | | | | | |
| 0x11E | | | | | | | | |
| 0x11F | | | | | | | | |
| 0x120 | | | | | | | | |

**RAM - Montículo**

| Dirección | MSB | 6 | 5 | 4 | 3 | 2 | 1 | LSB |
|---|---|---|---|---|---|---|---|---|
| 0xF11 | v[0] → 10 | | | | | | | |
| 0xF12 | | | | | | | | |
| 0xF13 | v[1] → 11 | | | | | | | |
| 0xF14 | | | | | | | | |
| 0xF15 | v[2] → 12 | | | | | | | |
| 0xF16 | | | | | | | | |
| 0xF17 | v[3] → 13 | | | | | | | |
| 0xF18 | | | | | | | | |
| 0xF19 | v[4] → 14 | | | | | | | |
| 0xF1A | | | | | | | | |
| 0xF1B | v[5] → 15 | | | | | | | |
| 0xF1C | | | | | | | | |
| 0xF1D | | | | | | | | |
| 0xF1E | | | | | | | | |

Los punteros ocupan 4 bytes en arquitecturas de 32 bits y 8 bytes en arquitecturas de 64 bits

Universidad Politécnica de Cartagena

# Almacenamiento de elementos en `<vector>`

- Un vector tiene una capacidad (*capacity*) máxima de almacenamiento, y un tamaño (*size*, cantidad de elementos almacenados actualmente), $size \leq capacity$
- La capacidad de un vector se modifica con el método `reserve()`. En general, solo debe invocarse cuando se utilice el ctor. (1). Después, la clase autogestiona este valor
- La memoria del *heap* se tiene que inicializar para poder guardar un valor concreto:
  - Los constructores (2) a (4) inicializan los elementos del vector, pero (1) no
  - Métodos que añaden (e inicializan) elementos: al final de la secuencia (`emplace_back()` y `push_back()`) o en cualquier posición (`insert()` y `emplace()`, requieren iterador). Cuando el elemento ya existe, se usa `insert()`; cuando se quiere construir en el vector, se usa `emplace()`, al que se le pasan los valores para invocar un constructor
  - Acceso de lectura/escritura a posición ya inicializada: **`operator[]`** y método `at()` (el segundo comprueba si el índice al que se quiere acceder ha sido inicializado)
- Importante: no se debe acceder a posiciones fuera de los límites establecidos por la `capacity()` del vector. Comportamiento no definido si se accede fuera de dichos límites del vector

Universidad Politécnica de Cartagena

```
(a) std::vector<short> v;
          // memoria asociada
            //sin inicializar
```

```
(b) v.reserve (5);
            // reservo memoria
   // (pero no se inicializa)
(c) v.push_back (20);
          // inserta al final
(d) v.push_back (21);
          // inserta al final
```

| RAM – Pila (a) | |
|---|---|
| Dirección | Contenido memoria primaria |
| 0x111 | Puntero al comienzo del vector<br>? (ignoro valor) |
| 0x115 | Puntero de inserción<br>? |
| 0x119 | Tamaño del vector<br>? |
| 0x11D | Capacidad del vector<br>? |

| RAM – Montículo (a) | |
|---|---|
| Dirección | Contenido memoria primaria |
| 0xF11 | ✗ (ni siquiera está reservada) |
| 0xF13 | ✗ |
| 0xF15 | ✗ |
| 0xF17 | ✗ |
| 0xF19 | ✗ |
| 0xF1B | ✗ |
| 0xF1D | ✗ |

| RAM – Pila (b) a (d) | |
|---|---|
| Dirección | Contenido memoria primaria |
| 0x111 | Puntero al comienzo del vector<br>0xF11 |
| 0x115 | Puntero de inserción<br>0xF11 / 0xF13 / 0xF15 |
| 0x119 | Tamaño del vector<br>0 / 1 / 2 |
| 0x11D | Capacidad del vector<br>5 |

| RAM – Montículo (b) a (d) | |
|---|---|
| Dirección | Contenido memoria primaria |
| 0xF11 | ? / 20 / 20 |
| 0xF13 | ? / ? / 21 |
| 0xF15 | ? |
| 0xF17 | ? |
| 0xF19 | ? |
| 0xF1B | ✗ |
| 0xF1D | ✗ |

# Ejemplo de uso de <vector> (I)

```
#include <iostream>
#include <vector>
#include <string>

string print_info(const vector<string> &v) {
 string tmp {"Tam:"+to_string(v.size())};
 tmp += ", cap:" + to_string(v.capacity());
 return tmp;
}

int main () {
 vector<string> vs;
 vs.reserve(20);
 vs.emplace_back("string 1");
 cout << "1.- " << print_info(vs) << '\n';
 string str {"string 2"};
 vs.push_back (str);
 cout << "2.- " << print_info(vs) << '\n';
 vs[7] = "string-7";
 cout << "3.- " << print_info(vs) << '\n';
 vs.emplace_back("string x");
 cout << "4.- " << print_info(vs) << '\n';
 cout << "vs[0] --> " << vs[0] << '\n';
 cout << "vs[1] --> " << vs[1] << '\n';
 cout << "vs[5] --> " << vs[5] << '\n';
 cout << "vs[7] --> " << vs[7] << '\n';
 cout << "vs[2] --> " << vs[2] << '\n';
 cout << "vs[23] --> " << vs[23] << '\n';
 cout << "vs.at(11) --> " << vs.at(11)
      << '\n';
 return 0;
}
```

Acceso a elemento fuera de límites. '*undefined behaviour*'

Acceso a elemento no inicializado. Excepción

```
1.- Tam: 1, cap: 20
2.- Tam: 2, cap: 20
3.- Tam: 2, cap: 20
4.- Tam: 3, cap: 20
vs[0] --> string 1
vs[1] --> string 2
vs[5] -->
vs[7] --> string-7
vs[2] --> string x
vs[23] -->
```

Se ha modificado el valor, pero es '*undefined behaviour*'. Además, no se ha modificado el tamaño

# Ejemplo de uso de <vector> (II)

```
template <class T>
vector<T> orden_inverso (const vector<T> &v) {
 const auto tam = v.size();
 vector<T> tmp (tam);
 for (int i = 1; i<tam-1; ++i)
  tmp[i] = v.at(tam-i-1);
 tmp.front() = v.back();
 tmp.back() = v.front();
 return tmp;
}
template <class T>
std::ostream& operator<< (ostream& os,
 const vector<T> &v) {
 os.put('[');
 for (const auto& i : v)
  os << ' ' << i;
 return os << "] (c: " << v.capacity() << ", s: "
  << v.size() << ')';
}

int main () {
  vector<string> v_str {"Programación", "en",
     "tiempo", "real"}; // ctor
  vector<string> v_rev {orden_inverso(v_str)}; // ctor
  vector<string> vv (5, "** "); // ctor
  cout << v_str << '\n'<< v_rev << '\n'<< vv << '\n';
  vv = v_str;
  cout << vv << '\n';
  vv.assign(5, "** ");
  vv.reserve(200);
  cout << vv << endl;
  cout << (v_str>v_rev? "v_str":"v_rev")<<" mayor\n";
  v_rev[0][0] = 'A';
  cout << (v_str>v_rev? "v_str":"v_rev")<<" mayor\n";
  vv.clear();
  cout << v_str << endl << v_rev << endl << vv;
  return 0;
}
```

Invoca esta función

```
[ Programación en tiempo real] (c: 4, s: 4)
[ real tiempo en Programación] (c: 4, s: 4)
[ ** ** ** ** ** ] (c: 5, s: 5)
[ Programación en tiempo real] (c: 5, s: 4)
[ ** ** ** ** ** ] (c: 200, s: 5)
v_rev mayor
v_str mayor
[ Programación en tiempo real] (c: 4, s: 4)
[ Aeal tiempo en Programación] (c: 4, s: 4)
[] (c: 200, s: 0)
```

# Crecimiento de <vector>

```
#include <iostream>
#include <vector>
string print_info(const vector<int> &v) {
 string tmp {"Tam:"+to_string(v.size())};
 tmp += ", cap:" + to_string(v.capacity());
 return tmp;
}
int main(){
  vector<int> v {1,2,3};
  cout << print_info(v) << '\n';
  cout << "1. pos v[0]: " << &v[0] << ", pos v: "<< &v <<'\n';
  v.push_back(4);
  cout << print_info(v) << '\n';
  cout << "2. pos v[0]: " << &v[0] << ", pos v: "<< &v <<'\n';
  v.push_back(5);
  cout << print_info(v) << '\n';
  cout << "3. pos v[0]: " << &v[0] << ", pos v: "<< &v <<'\n';
  v.push_back(6);
  v.push_back(7);
  cout << print_info(v) << '\n';
  cout << "4. pos v[0]: " << &v[0] << ", pos v: "<< &v <<'\n';
  return 0;
}
```

```
Tam:3, cap:3
1. pos v[0]: 0x557868b47eb0, pos v: 0x7ffcbca1f310
Tam:4, cap:6
2. pos v[0]: 0x557868b482e0, pos v: 0x7ffcbca1f310
Tam:5, cap:6
3. pos v[0]: 0x557868b482e0, pos v: 0x7ffcbca1f310
Tam:7, cap:12
4. pos v[0]: 0x557868b48300, pos v: 0x7ffcbca1f310
```

El vector va creciendo automáticamente conforme se van insertando elementos. v[0] cambia de posición cada vez que aumenta la capacidad del vector porque se reserva memoria y se copian los valores anteriores.
La variable que almacena la información del vector, v, nunca cambia

# <vector> de estructura

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
struct pod_agenda {
    int num_movil;
    string nombre;
    string email;
};
pod_agenda crear_contacto () {
     pod_agenda aux;
     cout << "Nombre?>> ";
     getline(cin>>ws, aux.nombre);
     cout << "E-mail?>> ";
     getline(cin>>ws, aux.email);
     cout << "Teléfono?>> ";
     cin >> aux.num_movil;
     return aux;
}
```

```
int main() {
    vector<pod_agenda> agenda;
    for (int i=0; i<2; ++i) {
        agenda.emplace_back
            (crear_contacto());
    }
    for (const auto &[m,n,e] : agenda) {
        cout << n << " -> " << e
            << ", tfno: " << m << endl;
    }
    return 0;
}
```

```
Nombre?>> Diego⏎
E-mail?>> diego@upct.es⏎
Teléfono?>> 123⏎
Nombre?>> Ana Ros⏎
E-mail?>> ana@upct.es⏎
Teléfono?>> 321⏎
Diego -> diego@upct.es, tfno: 123
Ana Ros -> ana@upct.es, tfno: 321
```

# <vector> multidimensional

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

template <class T>
ostream& operator<< (ostream& os,
 const vector<T> &v) {
  for (const auto &i : v) {
    cout << i << " ";
  }
  cout << '\n';
  return os;
}

int main () {
  vector<vector<int>> mat {
       {1,2,3},
       {4,5,6},
       {7,8,9}    };
  auto transpuesta {mat};
  cout << "*** MATRIZ ***\n";
  for (const auto &i : mat) {
    for (const auto &j : i) {
      cout << j << " ";
    }
    cout << '\n';
  }
  cout << "\n*** TRANSPUESTA ***\n";
  for (int i=0; i<mat.size(); ++i) {
    for (int j=0; j<mat[0].size();++j) {
      transpuesta[i][j] = mat[j][i];
    }
  }
  cout << transpuesta << '\n';
  return 0;
}
```

Se utiliza esta función doblemente, ya que son dos vector<int> anidados

- Vector de vectores
- Ojo, no se especifican las dimensiones, como sucede en C
- Ojo, puede crecer dinámicamente si no se tiene cuidado

```
*** MATRIZ ***
1 2 3
4 5 6
7 8 9

*** TRANSPUESTA ***
1 4 7
2 5 8
3 6 9
```

# `<set>` y `<map>` (colecciones, elementos en *heap*)

- Otras colecciones para almacenar y organizar datos son **std::set** (elementos no repetidos y ordenados, similar a vector) y **std::map** (colección de pares *clave-valor*, como si fuera un set de `std::pair` *clave-valor* ordenados por la *clave*)

```
std::set<std::string> v_unicos;
std::map<std::string, double> nota_asignaturas;
```

- Piense en `map` como un `set` (vector) donde los índices ¡pueden ser de cualquier tipo!. Las claves están internamente ordenadas y no se pueden repetir
- Ambos contenedores proporcionan `insert()` y `emplace()` para añadir elementos. Otros métodos útiles: `count()`, `find()`, `contains()`
- `<map>` además permite gestionar los elementos con `operator[]`, que inicializa el elemento si no existía previamente. El método `at()` lanza excepción si el elemento no estaba inicializado.

# Ejemplo de <map> (I)

```
#include <iostream>
#include <map>
#include <string>
#define ESTUDIANTES 40
using namespace std;

double nota_media(const map<string,double> &notas)
{
 double total=0.0;
 for (const auto &[c, v] : notas)
  { total += v; }
 return total/notas.size();
}

int main() {
 map<string, double> notas;
 string nombre;
 double nota;
 for (int i=0; i<ESTUDIANTES; ++i) {
  cout << "Nombre? >> ";
  getline(cin>>ws, nombre);
  cout << "Nota? >> ";
  cin >> nota;
  if (notas.count(nombre) != 0) {
   cout << "ERROR, nombre ya introducido\n";
  } else { notas[nombre] = nota; }
 }
 cout << "Consultar nota de >> ";
 getline(cin>>ws, nombre);
 if (notas.count(nombre) != 0) {
  cout << nombre << " tiene un "
       << notas[nombre] << '\n';
 } else { cout << "ERROR, no existe\n"; }
 cout << "Nota media curso: "
      << nota_media(notas) << '\n';
 return 0;
}
```

Programa que almacena y recupera la nota de estudiantes, calcula la nota media

```
Nombre? >> Lucas Perez⏎
Nota? >> 6⏎
Nombre? >> Ana Ros⏎
Nota? >> 8⏎
Nombre? >> Lucas Perez⏎
Nota? >> 3⏎
ERROR, nombre ya introducido
...
Consultar nota de >> Ana Ros⏎
Ana Ros tiene un 8
Nota media curso: 6.66667
```

# Ejemplo de <map> (II)

Programa que calcula histograma de letras de frase

```
#include <cctype>
int main() {
  string msg;
  map<char, int> letras;
  cout << "Introduce texto >>> ";
  getline(cin, msg);
  for (const auto &l : msg) {
    if (isalpha(l)) { ++letras[l]; }
  }
  cout << "Hay " << letras.size()
     << " letras distintas\n";
  int value = 0;
  char key;
  for (const auto &[c, v] : letras) {
    cout << "\tLetra '" << c
    << "' : " << v << " veces\n";
    if (v > value) {
      value = v;
      key = c;
    }
  }
  cout << "'" <<key<<"' es la más repetida, "
   <<value<<" veces. La borro y reimprimo\n";
  letras.erase(letras.find(key));
  for (const auto &[c,v] : letras)
    cout << "\tLetra '" << c << "' : "
      << v << " veces\n";
}
```

```
Introduce texto >>>
Programacion @ en tiempo 55
rea_l⏎
Hay 13 letras distintas
    Letra 'P' : 1 veces
    Letra 'a' : 3 veces
    Letra 'c' : 1 veces
    Letra 'e' : 3 veces
    Letra 'g' : 1 veces
    Letra 'i' : 2 veces
    Letra 'l' : 1 veces
    Letra 'm' : 2 veces
    Letra 'n' : 2 veces
    Letra 'o' : 2 veces
    Letra 'p' : 1 veces
    Letra 'r' : 3 veces
    Letra 't' : 1 veces
'a' es la más repetida, 3
veces. La borro y reimprimo
    Letra 'P' : 1 veces
    Letra 'c' : 1 veces
    Letra 'e' : 3 veces
    Letra 'g' : 1 veces
    Letra 'i' : 2 veces
    Letra 'l' : 1 veces
    Letra 'm' : 2 veces
    Letra 'n' : 2 veces
    Letra 'o' : 2 veces
    Letra 'p' : 1 veces
    Letra 'r' : 3 veces
    Letra 't' : 1 veces
```

# Ejemplo vector de mapas

Histograma de las páginas de un libro, contenidas en ficheros diferentes

```
#include <iostream>
#include <map>
#include <vector>
#include <fstream>

using namespace std;

map<string, int> hacer_histograma
 (ifstream &ifs) {
  map<string, int> aux;
  while(!ifs.eof()) {
    string tmp;
    ifs >> tmp;
    aux[tmp]++;
  }
  return aux;
}
```

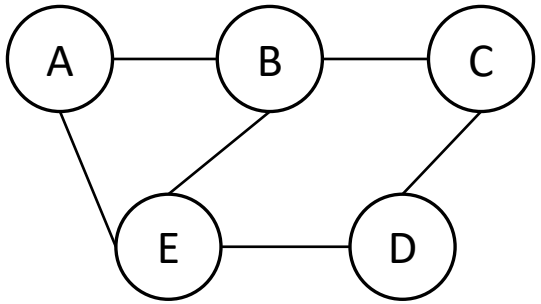
Ctor. (1) de vector, no reserva ni inicializa memoria

```
int main() {
  vector<map<string, int>> histograma_palabras;
  int n_pags;
  cout << "¿Cuántas páginas? >> ";
  cin >> n_pags;
  histograma_palabras.reserve(n_pags);
  string n_fichero;
  cout << "¿Prefijo fichero? >> ";
  cin >> n_fichero;
  for (int i=0; i<n_pags; ++i) {
    ifstream fichero {n_fichero+to_string(i)};
    if (fichero) {
      histograma_palabras.push_back(
        hacer_histograma(fichero) );
    }
  }
  return 0;
}
```

Hay que inicializar las posiciones del vector al insertar los valores

# Ejemplo mapa de vectores

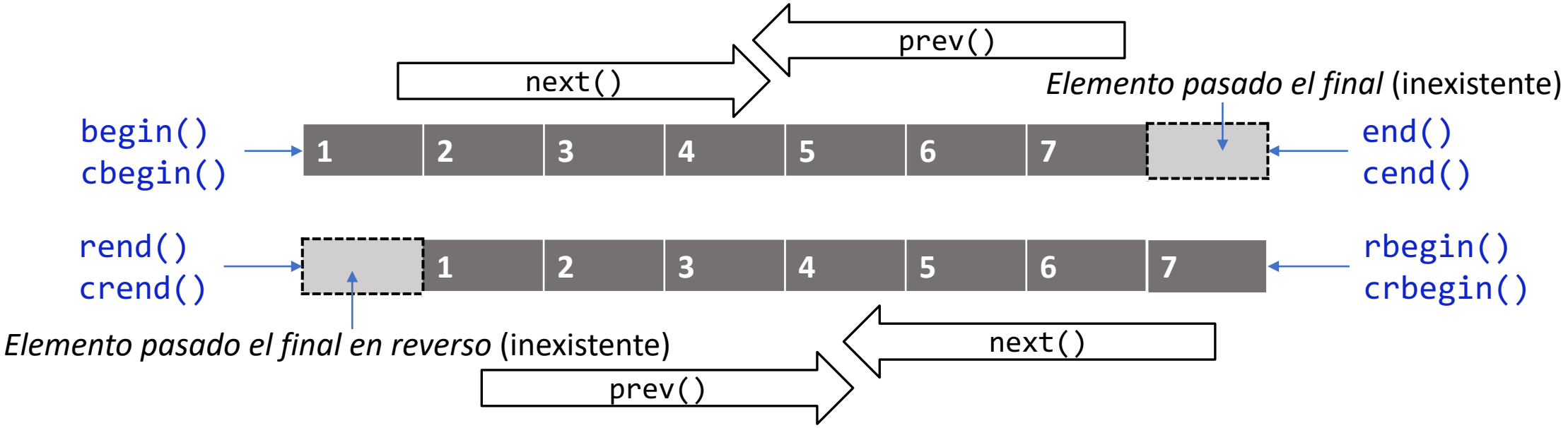
```
#include <iostream>
#include <map>
#include <vector>
using namespace std;

int main() {
  map<char, vector<char>> grafo {
   {'a', {'b', 'e'}},
   {'b', {'a', 'c', 'e'}},
   {'c', {'b', 'd'}},
   {'d', {'c', 'e'}},
   {'e', {'a', 'b', 'd'}}
  };
  vector<char> max;
  int max_arcos=0;
  for (const auto &[c, v] : grafo) {
    if (v.size() > max_arcos) {
      max_arcos = v.size();
      max.clear();
      max.push_back(c);
    }
    else if (v.size() == max_arcos)
      { max.push_back(c); }
  }
  cout << "Nodo/s con máximo/s arco/s ("
     << max_arcos << ") son: ";
  for (const auto &v : max)
    { cout << v << ", "; }
  cout << '\n';
  return 0;
}
```




```
Nodo/s con máximo/s arco/s (3) son: b, e,
```

# Iteradores <iterator>

- El **Iterador** es una forma de acceder a los elementos de una colección sin exponer su estructura interna (e.g., memoria continua, referencias entre nodos, etc.)
- El operador * (indirección) se utiliza para acceder al elemento al que apunta el iterador
- Funciones similares: `begin()`, `end()`, `next()`, `prev()`, `distance()`, etc. → homogeneidad de uso
- Si modificamos una colección es posible que algunos iteradores queden invalidados




- Intentar acceder al iterador devuelto por `Xend()` es '*undefined behaviour*'

# Código: iteradores (I)

solo por esta vez, para demostrar que funciona

```
template <class A, class B>
ostream& operator<< (ostream &os,
 const pair<A, B> &v) {
  os << v.first << ',' << v.second;
  return os;
}
template <class B>
void print_collection (const B &ppio,
   const B &final) {
 B it {ppio};
 while (it != final) {
  cout << *it << '|';
  it = next(it); // '++it' es posible,
   // aunque no siempre funciona
 }
 cout << '\n';
}
```

Tipo de los valores almacenados en el <map>. Necesario para imprimir los valores del mapa

```
int main () {
 vector<long> v {1,10,100,1000,10000};
 long a[] {-1, -10, -100, -1000};
 map<string, long> m {{"uno", 1},
  {"dos", 2}, {"tres", 3}, {"cuatro", 4}};
 print_collection (cbegin(v), cend(v));
 print_collection (v.cbegin(), v.cend());
 print_collection (cbegin(a), cend(a));
 print_collection (crbegin(a), crend(a));
 print_collection (begin(m), end(m));
 auto it = rbegin(m);
 advance(it,2);
 cout << *it << '|';
 it = prev(it);
 cout << *it;
}
```

Mejor utilizar la versión `funcIterador(coll)` que `coll.funcIterador()`

```
1|10|100|1000|10000|
1|10|100|1000|10000|
-1|-10|-100|-1000|
-1000|-100|-10|-1|
cuatro,4|dos,2|tres,3|uno,1|
dos,2|tres,3
```

Equivalente a: `map<string, long>::reverse_iterator it = rbegin(m);`

Universidad Politécnica de Cartagena

# Código: iteradores (II)

```
int main () {
  vector<string> vs;
  vs.reserve(15);
  string str = "string 1";
  vs.push_back(str);
  vs.emplace_back("string 2");
  vs.emplace_back(10, '*');
  cout << "1.- ";
  print_collection(begin(vs), end(vs));
  vs.emplace(begin(vs), "primero");
  cout << "2.- ";
  print_collection(begin(vs), end(vs));
  vs.emplace(end(vs), "ultimo");
  cout << "3.- ";
  print_collection(begin(vs), end(vs));
  str.assign("en medio");
  auto it2 = begin(vs);
  advance(it2, 2);
  it2 = vs.insert(it2, str);
  cout << "4.- ";
  print_collection(begin(vs), end(vs));
  cout << "5.- ";
  print_collection(it2, end(vs));
}
```

```
1.- string 1|string 2|**********|
2.- primero|string 1|string 2|**********|
3.- primero|string 1|string 2|**********|ultimo|
4.- primero|string 1|en medio|string 2|**********|ultimo|
5.- en medio|string 2|**********|ultimo|
```

Universidad Politécnica de Cartagena

# <algorithm>

Define funciones que operan sobre ¡colecciones de (casi) cualquier tipo! utilizando iteradores

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
using namespace std;
bool es_impar(const int &i) { return i&1; }
bool es_par(const int &i) { return not es_impar(i); }
bool f (const int &i) { return (i > 5 && i < 17); }
void imp (const int &i) { cout << i << ','; }
int main() {
  vector<int> v{{2, 4, 6, 8, 10}},
     v_M5_m17(v.size());
  cout << "Entre los numeros: ";
  for_each(cbegin(v), cend(v), imp);
  cout << '\n';
  if (all_of(cbegin(v), cend(v), es_par))
    { cout << "Todos son pares\n"; }
  if (none_of(cbegin(v), cend(v), es_impar))
    { cout << "Ninguno es impar\n"; }
  copy_if(cbegin(v), cend(v), begin(v_M5_m17), f);
  // La colección de destino tiene que tener
  // suficiente espacio para acomodar los elementos
  for_each(cbegin(v_M5_m17), cend(v_M5_m17), imp);
  cout << '\n';
  sort(begin(v), end(v));
  for_each(cbegin(v), cend(v), imp);
  cout << '\n';
  sort(rbegin(v), rend(v));
  for_each(cbegin(v), cend(v), imp);
  return 0;
}
```

*Todos los rangos se definen siempre **[primero, último)***

Con **funciones lambda** es posible incluir el código de, p.ej. `es_par()`, en la llamada al algoritmo

```
Entre los numeros: 2,4,6,8,10,
Todos son pares
Ninguno es impar
6,8,10,0,0,
2,4,6,8,10,
10,8,6,4,2,
```

Universidad Politécnica de Cartagena

# <random>

- Mejora sustancial en el soporte a la creación de números aleatorios
- Proporciona generadores de números aleatorios (GNA) y distribuciones probabilísticas (DP)

| Generadores de números aleatorios | | Distribuciones | |
|---|---|---|---|
| minstd_rand0 | ranlux48 | uniform_int_distribution | gamma_distribution |
| minstd_rand | knuth_b | uniform_real_distribution | normal_distribution |
| mt19937 | | bernoulli_distribution | chi_squared_distribution |
| mt19937_64 | **default_random_engine** | binomial_distribution | cauchy_distribution |
| ranlux24_base | **std::random_device** | geometric_distribution | fisher_f_distribution |
| ranlux48_base | | poisson_distribution | student_t_distribution |
| ranlux24 | | exponential_distribution | |

- Los GNA se utilizan como punto de entrada para muestrear la DP
- Proporciona gran soporte y flexibilidad para diseñar GNAs y permite también cambiar su semilla y guardar y recuperar de disco su estado.
- Recuerde: si no se cambia el valor de la semilla, un programa produce siempre la misma secuencia de valores aleatorios. **std::random_device** proporciona una forma sencilla de cambiar la semilla

# Código: quiniela aleatoria

```
#include <iostream>
#include <random>
using namespace std;
enum quiniela { uno, equis, dos };
ostream& operator<< (ostream &os,
 const quiniela q) {
  if (q==uno) os << '1';
  else if (q==equis) os << 'x';
  else os << '2';
  return os;
}
quiniela to_quiniela (const int v) {
  if (v==0) return uno;
  else if (v==1) return equis;
  else return dos;
}
template <class Dist>
void generar_quiniela (Dist &dist,
 const string &msg) {
  random_device rd;
  default_random_engine eng(rd());
  string tmp {"Quiniela "+msg};
  string subr (tmp.size(), '=');
  cout << tmp << '\n' << subr << '\n';
  for (int i=1; i<=15; ++i) {
    cout << "Partido " << i;
    quiniela res = to_quiniela(dist (eng));
    cout << " -> " << res << '\n';
  }
}
int main () {
  uniform_int_distribution<> uniforme (0,2);
  discrete_distribution<> hacia_casa
       ({45,35,20});
  generar_quiniela(uniforme,
        "misma probabilidad");
  cout << '\n';
  generar_quiniela(hacia_casa,
        "favoreciendo a los locales");
  return 0;
}
```

Invoca operador<<

Así se genera el valor aleatorio

Universidad Politécnica de Cartagena

# Código: random

```
#include <iostream>
#include <map>
#include <random>
#include <iomanip>
using namespace std;
int main () {
  random_device rd;
  default_random_engine eng(rd());
  uniform_int_distribution<> u_d(1, 6);
  int mean = u_d(eng);
  cout << "Randomly-chosen mean: " << mean;
  normal_distribution<> normal_dist(mean, 2);
  map<int, int> hist, hist_ud;
  for (int n = 0; n < 10000; ++n) {
    ++hist[round(normal_dist(eng))];
    ++hist_ud[round(u_d(eng))];
  }
  cout <<"\nNormal distribution around " << mean;
  for (const auto& p : hist) {
    cout << fixed << setprecision(1) << setw(2)
     <<p.first<<' '<<string(p.second/200, '*')<<endl;
  }
  cout <<"\nUniform distribution [1,6]:\n" ;
  for (const auto& [clave, valor] : hist_ud) {
    cout << clave << ' ' << string(valor/200, '*') << endl;
  }
}
```

```
Randomly-chosen mean: 3
Normal distribution around 3-4
-3
-2
-1 *
 0 ***
 1 ******
 2 ********
 3 *********
 4 ********
 5 ******
 6 ***
 7 *
 8
 9
10
11

Uniform distribution [1,6]:
 1 ********
 2 ********
 3 ********
 4 ********
 5 ********
 6 ********
```

Universidad Politécnica de Cartagena

# Consideraciones de rendimiento

***"Early optimization is the root of all evil"*** -- Donald Knuth

- Con la tecnología de compiladores actual, solo hay que preocuparse de diseñar o elegir los algoritmos/tipos de dato correctos. El compilador se encarga del resto
- Cuando haya dudas sobre la velocidad de un código, hay que probarlo y perfilarlo (*profiler*) con herramientas para estar seguro de dónde está el cuello de botella
- La arquitectura HW es muy compleja y hay muchos factores que influyen en el rendimiento del código. El rendimiento suele variar entre máquinas y SOs, y también depende de la carga del sistema
- Ejemplo. Ordenar 9999 veces en orden creciente y decreciente, alternativamente, un vector/array de 9999 enteros. Siempre se pasan las variables por valor

| | **Sin optimizar, sin *copy ellision*** | **Sin optimizar (-O0)** | **Optimizado (-O3)** |
|---|---|---|---|
| `array` | real 0m9.565s | real 0m9.229s | real 0m1.674s |
| `vector` | real 0m27.061s | real 0m18.671s | real 0m1.356s |

https://en.cppreference.com/w/cpp/language/copy_elision

```
#define TAM 9'999
#define ITS 9’999
using namespace std;
using ais = array<int, TAM>;
using vis = vector<int>;
random_device rd;
default_random_engine gen(rd());
uniform_int_distribution ud (1, TAM-3);

template <class T>
void print_c (const T &c) {
  for (const auto& i : c)
    cout << i << ',';
  cout << "\n=======\n\n";
}
bool en_decreciente (int& i1, int& i2) {
  return (i1>i2);
}
template <class T>
T ordenar (T c) {
  static bool en_creciente=false;
  if (en_creciente) sort(begin(c), end(c));
  else sort(begin(c), end(c), en_decreciente);
  en_creciente = !en_creciente;
  return c;
}
```

```
void f_array (const bool print) {
  ais a;
  for (int i=0; i<a.size(); ++i)
    a[i] = i*2;
  if (print) print_c(a);
  for (int i=0; i<ITS; ++i) {
    a = ordenar(a);
    a[ud(gen)] = ud(gen);
  }
  if (print) print_c(a);
}

void f_vector (const bool print) {
  vis v (TAM);
  for (int i=0; i<v.size(); ++i)
    v[i] = i*2;
  if (print) print_c(v);
  for (int i=0; i<ITS; ++i) {
    v = ordenar(v);
    v[ud(gen)] = ud(gen);
  }
  if (print) print_c(v);
}
```