Universidad LOYOLA

Departamento de Métodos Cuantitativos

Tema 1. C++ Avanzado

Antonio Manuel Durán Rosal (amduran@uloyola.es)

Programación Avanzada / Programación III

2º de Grado IITV / ISW

Curso 2025-2026

Septiembre 2025

Índice de contenidos

Índice de contenidos

Constructores

Establece las condiciones necesarias iniciales para crear objetos de la clase. Tiene el mismo nombre que la clase y no devuelve ningún valor. Internamente, el constructor inicializa los miembros de la clase o invoca a las funciones correspondientes para la inicialización.
Pueden existir tantos constructores como sea necesario, y si no se crean, el compilador crea de manera automática un constructor nulo.

```
class Numero{
  private: int num;
  public:
    Numero(){num=100};
    //Constructor con parámetros
    Numero(int n){num = n};
    void mostrar(){
      cout << "Num: " << num << endl
     ;
    }
};
```

```
class Numero{
  private: int num;
  public:
    Numero(){num=100};
    //Inicializador de miembros
    Numero(int n):num(n){};
    void mostrar(){
      cout << "Num: " << num << endl
     ;
    }
};
```

Creación de objetos

A través del uso de los constructores, la creación de objetos se puede realizar:

nombre_clase nombre_objeto = valor_inicialización;
nombre_clase nombre_objeto (valor, valor, valor, ...);

```
int main(){
  Numero n1 = 1;
  Numero n2(2);
  Numero *n4 = new Numero(4);
}
```

¿Cómo se destruyen los objetos?
¿Cómo se accede a los métodos en n1, n2 y n3?

Ejercicio 1

a) Defina la clase *Numero* que represente los números enteros. Defina tantos constructores como sea necesario.

b) Defina la clase *ParNumeros* que represente un par de números enteros. Defina tantos constructores como sea necesario.

c) Compruebe en un método principal las diferentes formas de inicializar objetos de cada clase.

Índice de contenidos

Objetos, referencias y punteros

- **Objeto:** región de memoria que tiene un tipo. El tipo puede ser desde primitivos (int) a complejos (clases).
- **Referencia:** es un alias. Otra forma de llamar a un objeto existente.
- **Puntero:** variable cuyo valor es una dirección de memoria de otra variable.
- **Variable:** nombre de una dirección de memoria donde hay almacenado un valor de un tipo particular.

```
Numero n1 = 20; //Objeto
int x = 10; //Valor
int &ref = x; //Referencia
int *y = &x; //Puntero
//Todos son asignados a variables: n1, x, ref, y
```

Ejercicio 2

a) Cree un programa principal haciendo uso de datos primitivos y modificándolos de distinta forma para observar las diferencias de variables, punteros y referencias.

Índice de contenidos

operator

operator es una palabra clave que declara una función especificando el significado de un símbolo de operador cuando se aplica a las instancias de una clase. Esto proporciona al operador más de un significado, o lo que llamamos como **sobrecarga**. El compilador distingue entre los diferentes significados de un operador examinando los tipos de sus **operandos**.

tipo_devuelto operator símbolo (parámetros)

Ejemplo:

```
ParNumeros operator=(ParNumeros n1, ParNumeros n2);
```

Operadores sobrecargables

+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	<<=	>>=	==	!=
<=	>=	&&	||	++	--	,	->*	->
()	[]	new	delete	new[]	delete[]			

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

Sugerencias

- El operador asignación (=) debe ser una función miembro de las clases.
- Los operadores sobrecargados <<, >> se declaran preferiblemente fuera de la clase.

Expression	Operator	Member function	Non-member function
@a	+ - * & ! ~ ++ --	A::operator@()	operator@(A)
a@	++ --	A::operator@(int)	operator@(A,int)
a@b	+ - * / % ^ & | < > == != <= >= << >> && || ,	A::operator@(B)	operator@(A,B)
a@b	= += -= *= /= %= ^= &= |= <<= >>= []	A::operator@(B)	-
a(b,c...)	()	A::operator()(B,C...)	-
a->b	->	A::operator->()	-
(TYPE) a	TYPE	A::operator TYPE()	-

Where a is an object of class A, b is an object of class B and c is an object of class C. TYPE is just any type (that operators overloads the conversion to type TYPE).

Ejercicio 3

a) Modificar la clase *ParNumeros* para sobrecargar los operadores <<, >>, +, -, =, ++.

b) Realizar un programa principal que compruebe la sobrecarga realizada.

friend

friend es una palabra clave que se utiliza para funciones, operadores y clases. Un operador/función/clase "amigo" es aquel que aún cuando no es miembro de una clase, tiene todos los privilegios de acceso a los miembros de dicha clase.

```
class Complejo{
  ...
  friend Complejo & operator+(Complejo const &c1, Complejo const
    &c2);
  friend ostream & operator<<(ostream &o, Complejo const &c);
  ...
}
ostream & operator<<(ostream & o, Complejo const &c){
  o << "(" << c.real << "," << c.imaginario << ")";
  return o;
}
```

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

Ejercicio 4

a) Modificar las variables *num1* y *num2* a privadas.

b) Hacemos funciones amigas para que el programa funcione correctamente.

Ejercicio 5

a) Con el ejercicio anterior, probamos a cambiar las funciones amigas que podamos a funciones miembro.

const

const es una palabra clave que se utiliza en la definición de funciones. El principal objetivo es no permitir a la función modificar el objeto desde el que se ha invocado. Recomendable utilizar para evitar cambios accidentales de objetos (por ejemplo, ¡proteger las variables privadas!).

```
//Objeto constante, sólo accede a miembros especificados como
    const
const nombre_clase nombre_objeto;
//Especificar que es un método const
tipo_devuelto nombre_metodo(parametros) const;
//Especificar que el método devuelve un tipo constante
const tipo_devuelto nombre_metodo(parametros) const;
//Especificar que los parámetros de entrada son const
tipo_devuelto nombre_metodo(const parametros);
```

https://en.cppreference.com/w/cpp/keyword/const

Ejercicio 6

a) Desgranar y debatir en clase el ejemplo 4 disponible en Moodle, que hace uso de punteros y el modificador **const**.

Ejercicio 7

a) Defina la clase *Persona*. Viene definida y se construye a partir del nombre. Tiene un método público que muestra por pantalla el nombre de la persona.

b) Defina un método principal que cree un objeto de tipo *Persona* y muestre por pantalla su nombre. Modifique los atributos y métodos para usar **const**.

c) También se deben usar punteros.

Índice de contenidos

Conversiones implícitas

Conversiones implícitas (o conversiones estándar): conversiones que se hacen automáticamente cuando se toma un valor de un tipo compatible. Por ejemplo: de short a entero, de float a double.

- Cuidado con las conversiones de tipos grandes a pequeños: se mantiene el valor, pero cuando es con unsigned, y se quiere convertir números negativos ¿Qué pasa?
- La conversión de bool a enteros, ¿cómo es?
- De decimal (float o double) a entero: se trunca.
- Punteros:
  - Los nulos pueden convertirse al tipo que queramos.
  - Punteros de cualquier tipo a void.
  - De clases hijas a padres (conservando el **const**).

Buscar cast en cppreference.com

Ejercicio 8

Defina un método principal donde realice las siguientes conversiones:

a) De **long** a **int**.

b) De **float** a **int**.

c) De **int** valor negativo a **unsigned int**.

d) De entero a **bool** y viceversa.

e) Cree la clase *Padre* con un atributo p, y una clase *Hija* con un atributo h que herede de Padre. Convierta un objeto tipo *Hija* a un objeto tipo *Padre*.

Conversiones implícitas entre clases

Conversiones implícitas entre clases: tres formas de hacer conversiones implícitas entre clases (dentro de clase B, para permitir conversión de clase A):

- Constructor:

  claseB (const claseA & param){...}

- Operador de asignación:

  claseB & operator=(const claseA & param){...}

- Operador de tipo:

  operator claseA(){...}

Ejercicio 9

a) Defina la clase *Persona* con los atributos nombre, apellidos, edad. Incluya un constructor con todos los parámetros, y métodos consultores y modificadores. Incluya un método que imprima por pantalla todos sus atributos.

b) Defina la clase *Estudiante* con los atributos nombre, apellidos, edad y matricula. Incluya un constructor con todos los parámetros, y métodos consultores y modificadores. Incluya un método que imprima por pantalla todos sus atributos. Además: Incluya los tres tipos de conversiones explicadas a la clase *Persona*.

c) Defina un método principal donde pruebe todos los tipos de conversiones, cree tantos objetos como sean necesario.

Conversiones explícitas

Conversiones explícitas: no las realiza el compilador automáticamente, sino que se indica de forma explícita en nuestro programa.

(nombre_tipo) variable o expresión;

```
int n;
sqrt((double)(n+2));
```

Clases

- La palabra reservada **template** indica al compilador que el código que sigue es una plantilla o patrón de clase y que se utilizará *Tipo* como tipo genérico.
- Los parámetros de *Tipo* aparecen entre operadores relacionales desigualdad "<>".

```
template <class Tipo>
class NombreClase
{
  ...
}
```

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

Clases

- Para la definición de cada función miembro:

```
template <class Tipo>
TipoDeResultado NombreClase<Tipo>::nombreDeFuncion(...)
```

- Para la correcta compilación el fichero .cpp se incluye con un #include “fichero.cpp” al final del fichero.h
- También se puede definir todo dentro del fichero .h en lugar de separarlo en dos ficheros.
- Haciendo uso de **export**.

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

Funciones

- Para compilar se usan los mismo métodos que se han comentado en la diapositiva anterior.
- La sintaxis es:

```
template <typename Tipo>
TipoDeResultado NombreDeFuncion(Tipo parametro1, ...)
```

- Si *Tipo* es un clase entonces es preferible cambiar **typename** por **class**.

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

Ejercicio 10

a) Se implementa una plantilla de clase para un *Punto2D* (x,y) de forma que se puedan declarar objetos con Tipos **int** o **float**.

b) La clase será dotada de constructores, selectores, modificadores, sobrecarga del operador asignación y funciones de E/S.

c) Se codifican plantillas de funciones para el cálculo de diferentes distancias entre puntos de este tipo.

Índice de contenidos

¿Qué es una excepción?

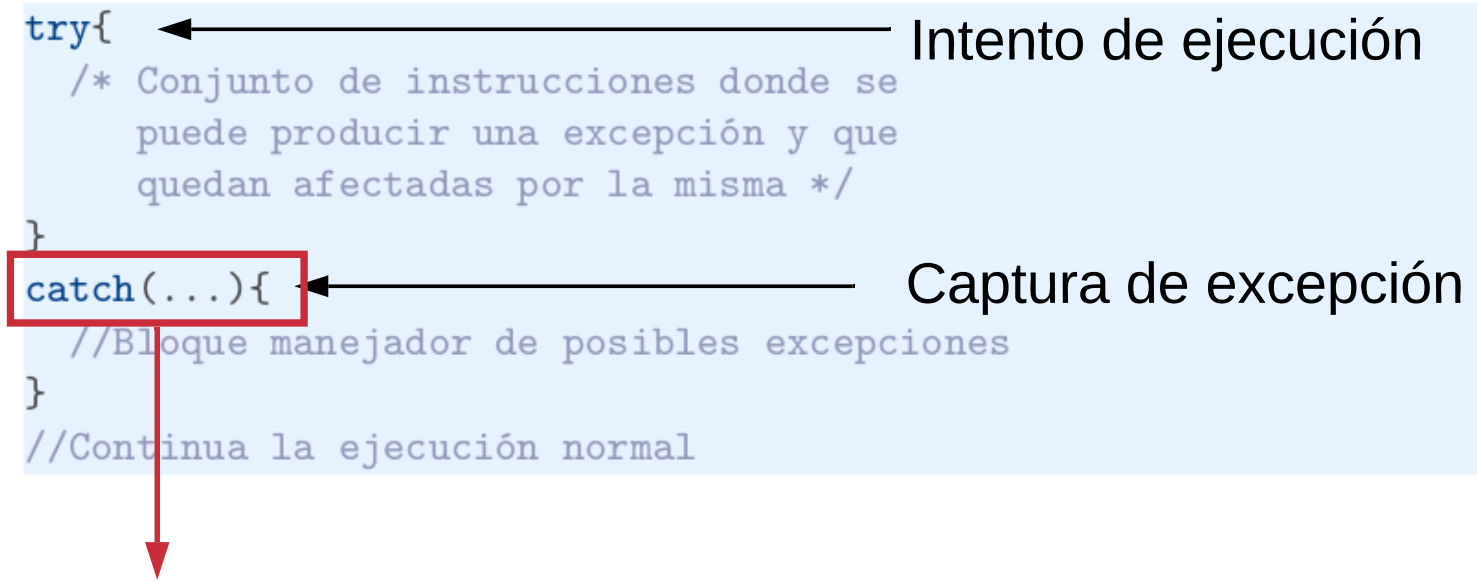
Al conjunto de situaciones *imprevistas* se la denomina **excepción**, y al mecanismo dispuesto para reaccionar ante esas situaciones se le denomina **manejador de excepciones**. Distinguimos entre:

a) **Excepciones Síncronas:** ocurren dentro del programa. Por ejemplo, se agota la memoria, valores nulos, ...

b) **Excepciones Asíncronas:** tienen origen fuera del programa, a nivel de Sistema Operativo. Por ejemplo, se pulsan las teclas *Ctrl+C*, ...

Las implementaciones de C++ sólo consideran las excepciones síncronas.

Bloque try-catch



- Relanzar la misma excepción.
- Saltar a una etiqueta.
- Terminar su ejecución normalmente (alcanzar la llave de cierre).

Bloque try-catch

```
1 try{
2   /* Conjunto de instrucciones donde se
3      puede producir una excepción y que
4      quedan afectadas por la misma */
5 }
6 catch(...){
7   //Bloque manejador de posibles excepciones
8 }
9 //Continua la ejecución normal
```

Intento de ejecución

Captura de excepción

Ejemplo:

INTENTA { → **Try**
1.- Abrir el fichero → ¿Error de apertura/permisos?
2.- Guardar la información
3.- Cerrar el fichero
}
SI HAY UN ERROR CAPTURALO Y EJECUTA → **Catch**
{
4.- Informa al usuario
}

Evitar pasos 2 y 3

Bloque try-catch

```
try{
  /* Conjunto de instrucciones donde se
     puede producir una excepción y que
     quedan afectadas por la misma */
}
catch(...){
  //Bloque manejador de posibles excepciones
}
//Continua la ejecución normal
```

Ejemplo:

```
try{
  /* código y lanzamiento de excepciones */
}
catch(int param){ cout << "Excepcion de tipo int" << endl; }
catch(char param){ cout << "Excepcion de tipo char" << endl; }
catch(...){ cout << "Excepcion por defecto" << endl; }
```

throw

```
1 try{
2   ...
3   throw ExceptionType;
4   ...
5 }
6 catch(ExceptionType e){
7   //Bloque manejador de posibles excepciones
8 }
9 //Continua la ejecución normal
```

Lanza excepción

Siempre Bloque Try

Debe haber un bloque catch que capture cada excepción lanzada

Ejemplo de excepción tipo int

```
1 try{
2   throw 20;
3 }
4 catch(int e){
5   cout <<"An exception occurred. Exception Nr. " << exc << endl;
6 }
7 //Continua la ejecución normal
```

throw

```
1 try{
2   ...
3   throw ExceptionType;
4   ...
5 }
6 catch(ExceptionType e){
7   //Bloque manejador de posibles excepciones
8 }
9 //Continua la ejecución normal
```

La Clase tipo ExceptionType debe contener la información necesaria para conocer la naturaleza de la circunstancia excepcional (probablemente un error).

Ejercicio 11

a) Defina la clase *ZeroException* con un método público que muestre por pantalla el mensaje "*No puede introducir un valor 0*".

b) Defina método suma que solicite por pantalla dos números y devuelva la suma de ambos números. Si uno de los números introducidos por el usuario tiene valor 0, debe lanzar la excepción *ZeroException*.

c) Defina método principal que realice la suma de dos valores obtenidos por teclado, y lance una excepción en caso de valor igual a 0.

Ejercicio 12

a) Defina la clase *ExceptionA* con un método público que muestre por pantalla el mensaje "*Excepción padre A*".

b) Defina la clase *ExceptionB* que herede de *ExceptionA* con un método público que muestre por pantalla el mensaje "*Excepción hija B*".

c) Defina método principal que lance una excepción de tipo *ExceptionB* y capture:

- Primero *ExceptionA* y luego *ExceptionB*.
- Primero *ExceptionB* y luego *ExceptionA*.

¿Qué ocurre?

Ejercicio 13

a) Defina en el método principal un array de 10 mil millones de enteros y asigne un valor al elemento de la posición 50.

b) Para ello declare una variable **long int** que especificará el tamaño del array.

¿Qué ocurre?¿Cómo puede solucionarlo?

Excepciones estándar en C++

Excepción	Descripción
bad_alloc	Fallo de alojamiento de memoria.
bad_cast	Falla el **dynamic_cast**.
bad_exception	Lanzada por ciertos especificadores de excepciones dinámicas.
bad_typeid	Lanzada por **typeid**.
bad_function_call	Se llama a una función que es un objeto vacío.
runtime_error	Error detectado en tiempo de ejecución.

Estos son sólo unos ejemplos de las excepciones que ya están definidas en C++, para más información, se recomienda visitar:
https://en.cppreference.com/w/cpp/error/exception

Especificación de excepciones en C++

- La función sólo lanza tipos de excepciones concretos:
  tipo nombre_función (parámetros) throw (int,float)
- La función no lanza excepción:
  tipo nombre_función (parámetros) throw ()
  tipo nombre_función (parámetros) noexcept
  tipo nombre_función (parámetros) noexcept (true)
- La función lanza cualquier tipo de excepción:
  tipo nombre_función (parámetros) → **Manejo Formal**.
  tipo nombre_función (parámetros) noexcept (false)

Eliminado desde C++17

Ejercicio 14

a) Defina una función que:

- Lance cualquier excepción.
- No lance ninguna excepción.
- Haga un manejo formal.

b) Defina una función principal que utilice cada una de las funciones definidas en el apartado anterior.

Ejercicio 15

a) Defina una clase *ExceptionGenerica* que herede de **exception** que imprima por pantalla el mensaje "*Error: tipoExcepción*". Para ello:

- Defina una clase *ExceptionGenerica* que herede de **exception**.
- Redefina los métodos que considere necesarios para mostrar el mensaje por pantalla: "*Error: tipoExcepción*".

b) Cree un método principal que compruebe que se muestra el mensaje de la clase *ExceptionGenerica* en caso de que ocurra un error de tipo **exception**.

Índice de contenidos

Introducción

STL hace uso de la mayoría de lo que nosotros utilizamos:

- Funciones, clases, estructuras.
- Plantillas de funciones y clases.
- Typedefs, y tipos asociados.

Objetivos:

- Ver cómo están implementados los STL Containers.
- Ver cómo las diferencias en implementación afectan a su uso.

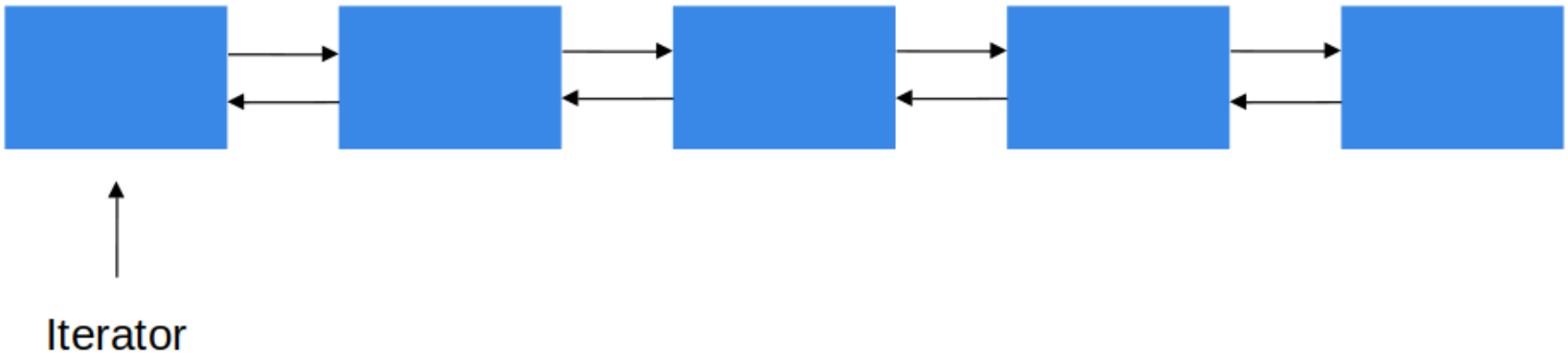
https://en.cppreference.com/w/cpp/container

Contenedores más utilizados

- **Contenedores secuenciales:** array, vector, deque, forward_list, **list**.
- **Adaptaciones de los anteriores:** **stack**, **queque**, priority_queue.
- **Contenedores asociativos:** **set**, **map**, **multiset**, **multimap**.
- **Contenedores asociativos *hash*:** **unordered_set**, **unordered_map**, **unordered_multiset**, **unordered_multimap**.

https://en.cppreference.com/w/cpp/container

STL: list




```
list <tipo> nombre_lista;
```

```
list <tipo>::iterator nombre_iterador;
```

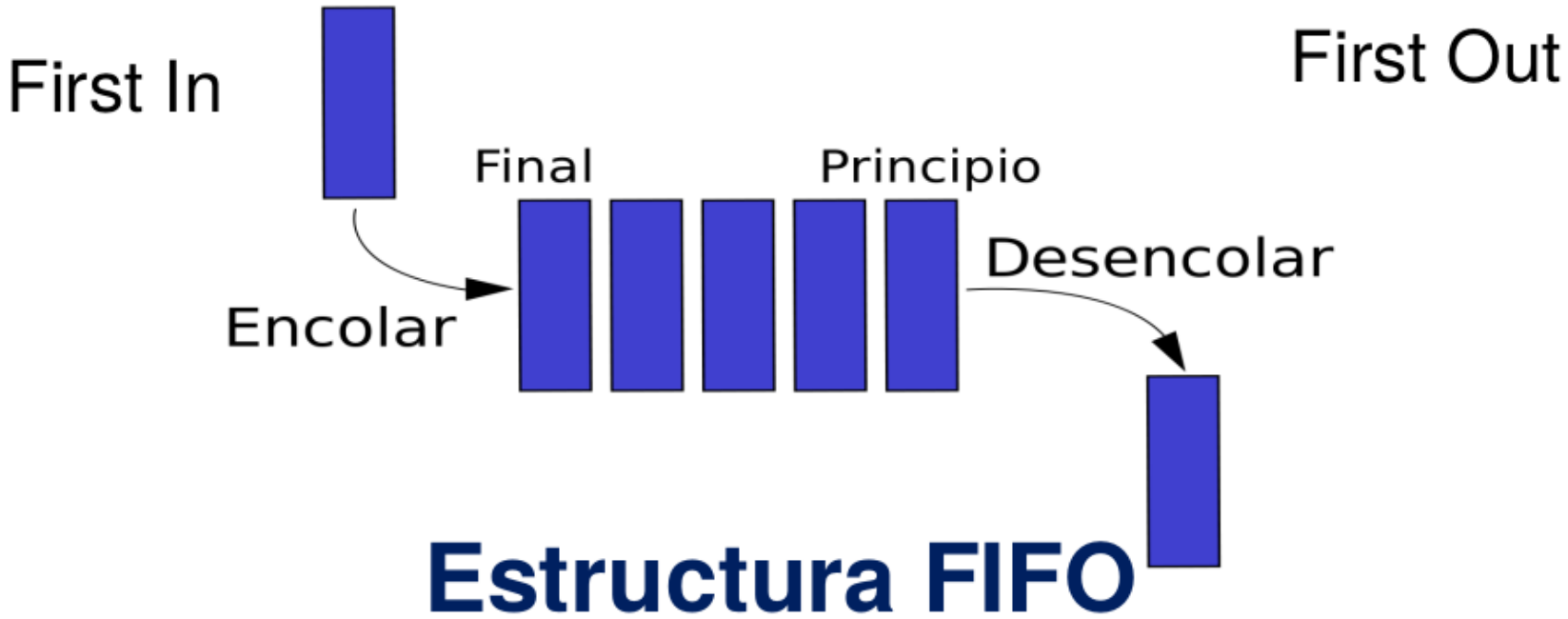
https://en.cppreference.com/w/cpp/container/list

Ejercicio 16

a) Mostrar el uso de la clase **list**.

https://en.cppreference.com/w/cpp/container/list

STL: queue




```
queue <tipo> nombre_cola;
```

No se usan iteradores.

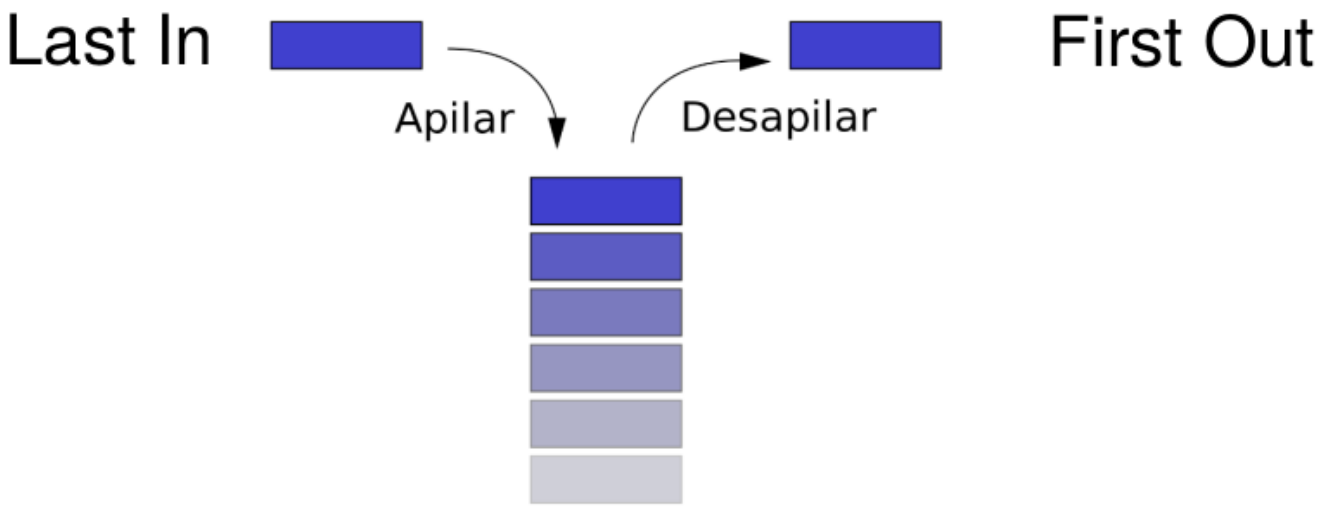
https://en.cppreference.com/w/cpp/container/queue

Ejercicio 17

a) Mostrar el uso de la clase **queue**.

https://en.cppreference.com/w/cpp/container/queue

STL: stack




```
stack <tipo> nombre_pila;
```

No se usan iteradores.

https://en.cppreference.com/w/cpp/container/stack

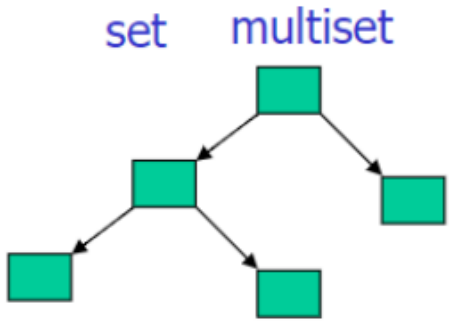
Ejercicio 18

a) Mostrar el uso de la clase **stack**.

https://en.cppreference.com/w/cpp/container/stack

STL: set, multiset

- Representan conjuntos de elementos ordenados:
  - **set:** elementos no repetidos.
  - **multiset:** elementos repetidos.
- Ambos se organizan en árboles binarios.
- No se permiten modificación de elementos.




```
set <tipo> nombre_conjunto;
multiset <tipo> nombre_conjunto;
```

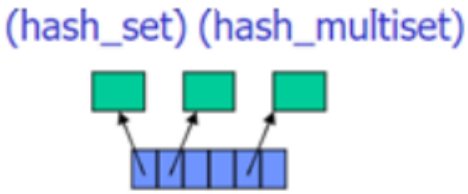
```
set <tipo>::iterator nombre_iterador;
multiset <tipo>::iterator nombre_iterador;
```

https://en.cppreference.com/w/cpp/container/set

https://en.cppreference.com/w/cpp/container/multiset

STL: unordered_(set,multiset)

- Representan conjuntos de elementos no ordenados:
  - **unordered_set:** elementos no repetidos.
  - **unordered_multiset:** elementos repetidos.



- Ambos se organizan en **Tablas hash**.
- No se permiten modificación de elementos.

```
unordered_set <tipo> nombre_conjunto;
unordered_multiset <tipo> nombre_conjunto;
```

```
unordered_set <tipo>::iterator nombre_iterador;
unordered_multiset <tipo>::iterator nombre_iterador;
```

https://en.cppreference.com/w/cpp/container/unordered_set

https://en.cppreference.com/w/cpp/container/unordered_multiset

Ejercicio 19

a) Mostrar el uso de los contenedores **set**, **multiset**, **unordered_set** y **unordered_multiset**.

https://en.cppreference.com/w/cpp/container/set

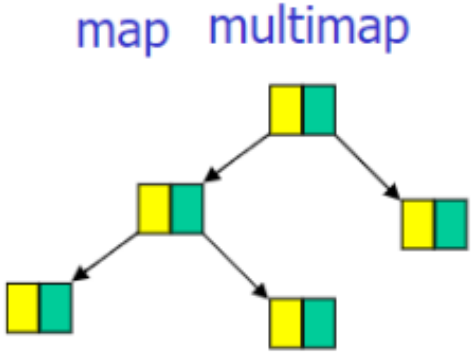
https://en.cppreference.com/w/cpp/container/multiset

https://en.cppreference.com/w/cpp/container/unordered_set

https://en.cppreference.com/w/cpp/container/unordered_multiset

STL: map, multimap

- Contienen pares clave-valor ordenados por clave:
  - **map:** claves únicas.
  - **multimap:** claves no únicas.
- Ambos se organizan en árboles binarios.




```
map <tipo clave, tipo valor> nombre_conjunto;
multimap <tipo clave, tipo valor> nombre_conjunto;
```

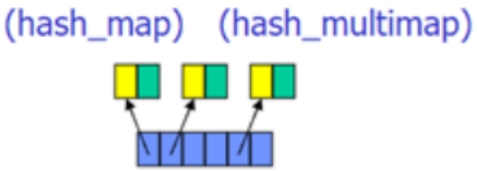
```
map <tipo clave, tipo valor>::iterator nombre_iterador;
multimap <tipo clave, tipo valor>::iterator nombre_iterador;
```

https://en.cppreference.com/w/cpp/container/map

https://en.cppreference.com/w/cpp/container/multimap

STL: unordered_(map,multimap)

- Contienen pares clave-valor sin ordenar:
  - **unordered_map:** claves únicas.
  - **unordered_multimap:** claves no únicas.
- Ambos se organizan en **Tablas hash**.




```
unordered_map <tipo clave, tipo valor> nombre_conjunto;
unordered_multimap <tipo clave, tipo valor> nombre_conjunto;

unordered_map <tipo clave, tipo valor>::iterator nombre_iterador;
unordered_multimap <tipo clave, tipo valor>::iterator nombre_it;
```

https://en.cppreference.com/w/cpp/container/unordered_map

https://en.cppreference.com/w/cpp/container/unordered_multimap

Ejercicio 20

a) Mostrar el uso de los contenedores **map**, **multimap**, **unordered_map** y **unordered_multimap**.

https://en.cppreference.com/w/cpp/container/map

https://en.cppreference.com/w/cpp/container/multimap

https://en.cppreference.com/w/cpp/container/unordered_map

https://en.cppreference.com/w/cpp/container/unordered_multimap

Índice de contenidos

Función de comparación en C++

Hay cuatro formas de definir una comparación:

- Definir operador $<$ (**operator<()**).
- Definir una función específica.
- Definir operador () (**operator()()**).
- Librería **functional**.

Veremos las tres primeras formas.

operator <

bool operator < (T b) const

Ejemplo:

```
class Player
{
  public:
    int id;
    string name;
    Player(int playerId, string playerName){
      id=playerId;
      name=playerName;
    }
    bool operator < (Player const & playerObj) const{
      return id < playerObj.id;
    }
};
```

Función específica

bool nombre_función(T a, T b)

Ejemplo:

```
bool compare_by_name(Player const & p1, Player const & p2) {
  return p1.name < p2.name;
}
```

operator ()

bool operator () (T a, T b)

Ejemplo:

```
class PlayerComparator {
  bool operator ()(Player const & p1, Player const & p2) {
    if (p1.name == p2.name)
      return p1 < p2;
    return p1.name < p2.name;
  }
};
```

Ejercicio 21

a) Mostrar el uso de las distintas formas de definir comparaciones.

Referencias

Recursos electrónicos

- cppreference. (2025). Referencia C++: https://en.cppreference.com/w/.
- Goalkicker.com (2025). C++: Note for Professionals.
- Goalkicker.com (2025). Algorithms: Notes for Professionals.
- Joyanes, L. (2006). Programación en C++: Algoritmos, Estructuras de datos y Objetos.

Referencias

Libros:

- Brassard, G., Bratley, P. (1997). Fundamentos de Algoritmia.
- Guerequeta, R., Vallecillo, A. (1998). Técnicas de diseño de algoritmos.
- Joyanes, L. (2008). Fundamentos de programación: algoritmos, estructura de datos y objetos.
- Knuth, D. (2011). The Art of Computer Programming, Volumes 1-4. Third Edition.
- Martí, N., Ortega, Y., Verdejo, J.A. (2013). Estructuras de datos y métodos algorítmicos: Ejercicios resueltos.
- Parker, A. (1993). Algorithms and Data Structures in C++. CRC Press.

¿Preguntas?