Universidad
Rey Juan Carlos
Escuela Técnica Superior
Ingeniería de Telecomunicación

Programación de Sistemas de Navegación

Presentación

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

Índice de contenidos

1. Presentación

- Asignatura: **Programación de Sistemas de Navegación**
- Grado: Ingeniería Aeroespacial en Aeronavegación
- Periodo de impartición: 4º, 1Q
- Tipo: OPTATIVA
- Número de créditos: 6 ECTS
- Horas/semana: 4 horas (2 horas teoría y 2 horas práctica)
- Idioma: Castellano
- Profesor: José Miguel Guerrero Hernández (josemiguel.guerrero@urjc.es)

Índice de contenidos

2. Objetivos y destrezas

- **Objetivos:**
  - Adquirir los conocimientos necesarios para el diseño e implementación de software para control del tráfico aéreo
  - Diseñar programas que resuelvan problemas de tratamiento de información
  - Presentación de herramientas para representar información y para tratarla
  - Planteamiento de metodologías para facilitar y sistematizar el análisis de problemas y el diseño de programas que los resuelvan
  - Saber analizar problemas concretos, plantear soluciones y desarrollar programas que permitan resolverlos en un computador
  - Ejercitarse como programador de forma individual y en equipo aplicando unas determinadas metodologías
- **Conocimientos y destrezas que se requieren:**
  - Conocimientos básicos de programación, adquiridos en cursos anteriores

2. Objetivos y destrezas

- **Conocimientos y destrezas que se adquieren:**
  - Técnicas de programación avanzada, usando uno de los lenguajes de programación más flexibles y potentes: C/C++
  - Programación avanzada: Orientación a Objetos, control de errores, estructuras de datos complejas, gestión avanzada de memoria y multithreading y programación en red (sockets)
  - Escribir programas con buen estilo, con una documentación adecuada, con los comentarios precisos y con las especificaciones necesarias
  - Se introducirá el uso de herramientas de programación habitual en entornos de proyectos extensos y cooperativos: Git y CMake
  - Utilizar estrategias para corregir los programas cuando no funcionan bien

Índice de contenidos

3. Temario

- **Bloque 0: Sistema Unix**

  SO, Procesos, Entorno, Metacaracteres, Ayuda, Navegación, Gestión de ficheros y procesos

- **Bloque 1: Lenguaje C**

  Introducción a C, Tipos básicos, Compilación en Linux, Estructuras de datos y estructuras de control, Manejo de la memoria, Punteros, Arrays y Strings, Funciones, Algoritmos de Ordenación y Búsqueda, Descomposición de programas en varios ficheros fuente y librerías, Depuración, Manejo de ficheros, Sockets

- **Bloque 2: Lenguaje C++**

  Diferencia entre C y C++, Namespaces, C++ strings, streams, Programación Orientacion a Objetos, Templates, Librería STL (vector, deque, list, iterators, pila, cola, maps), Manejo de excepciones

3. Temario detallado

- Bloque 0
  1. Sistema UNIX
- Bloque 1
  1. Sintaxis, Compilador, Ámbitos, Tipos, Operadores, Sentencias de control e iteración, Entrada y Salida, Errores
  2. Arrays, Strings, Struct, Punteros, Funciones
  3. Listas, Pilas, Colas, Árboles, Recursividad
  4. Ordenación, Búsqueda
  5. Argumentos, Depuración, Archivos fuente y cabecera, Compilar y enlazar, Makefile
  6. Manejo de ficheros
  7. Sockets
- Bloque 2
  1. Diferencia C y C++, POO, Sobrecarga, Namespaces, Compilar, Make
  2. Constructor, destructor, copia, sobrecarga de operadores
  3. Herencia, Polimorfismo
  4. Templates, Librería STL
  5. Librería STL vector, lista
  6. Librería STL deque, pila, cola, mapa, multimapa
  7. Manejo de excepciones

Índice de contenidos

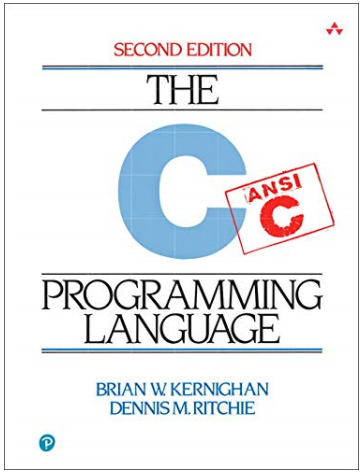
5. Metodología

- **Material:**
  - Apuntes, ejemplos y prácticas disponibles en el Aula Virtual
- **Aprendizaje:**
  - Asistir a clase habiendo leído previamente cada lección
  - Atender y participar activamente en la clase
  - Repasar después y comprender cada lección
  - Resolver los problemas propuestos
  - Programación en el ordenador: validación del código desarrollado
  - Colaborar con otros compañeros y consultarles
  - Consultar dudas al profesor

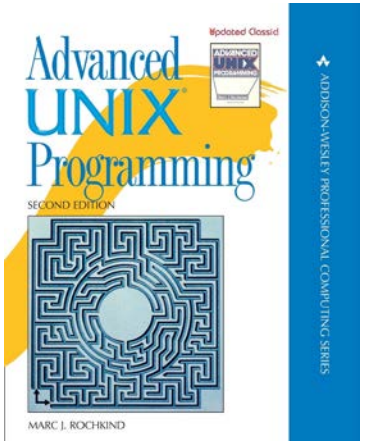
Índice de contenidos

6. Bibliografía

The C Programming Language 2nd Edition. Brian W. Kernighan, Dennis M. Ritchie. Prentice Hall Software Series

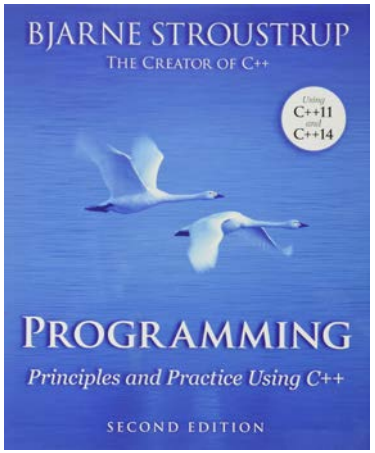


Advanced UNIX Programming. 2nd Edition. Marc J. Rochkind

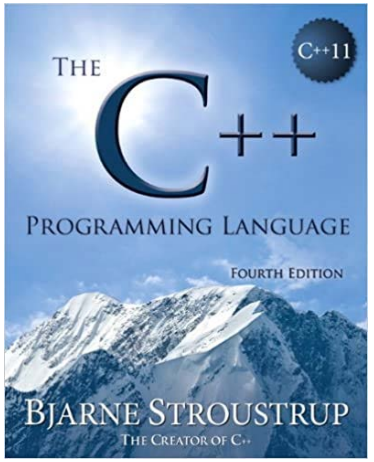


6. Bibliografía

Programming: Principles and Practice Using C++. Bjarne Stroustrup.Addison-Wesley ISBN 978-0321-992789. May 2014

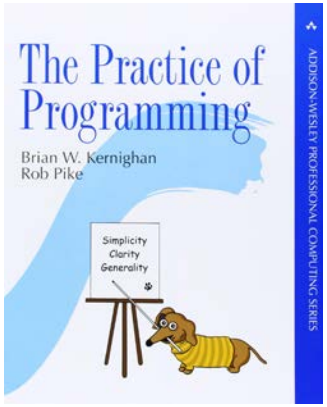


The C++ Programming Language. Addison-Wesley ISBN 978-0321563842. May 2013



6. Bibliografía

The Practice of Programming. Brian W. Kernighan, Rob Pike. Addison-Wesley Professional Computing Series



cppreference.com

http://www.cplusplus.com/reference/

http://stackoverflow.com/

Reflexión

A programar se aprende programando

Universidad Rey Juan Carlos

Escuela Técnica Superior Ingeniería de Telecomunicación

UNIX®

Programación de Sistemas de Navegación

0. Sistema UNIX

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

Índice de contenidos

1. Sistema Operativo

- Colección organizada de extensiones SW del HW
- Conjunto de rutinas de control que hacen funcionar la computadora y proporcionan un entorno de ejecución para los programas
- Interfaz HW – usuario

1. Sistema Operativo

- S.O. como **máquina virtual**
  - Gestiona los recursos del sistema
  - Se comunica con cada dispositivo en su “lenguaje particular”
  - Ofrece una interfaz sencilla y uniforme
  - Del S.O. depende tanto el uso eficiente del HW como la buena ejecución del Software
- S.O. como **administrador de recursos**
  - Reparto ordenado y equitativo de los recursos del sistema (procesador, memoria, dispositivos de E/S, ...) entre los usuarios que compiten por ellos
- La evolución de los S.O. es paralela a la historia de los computadores

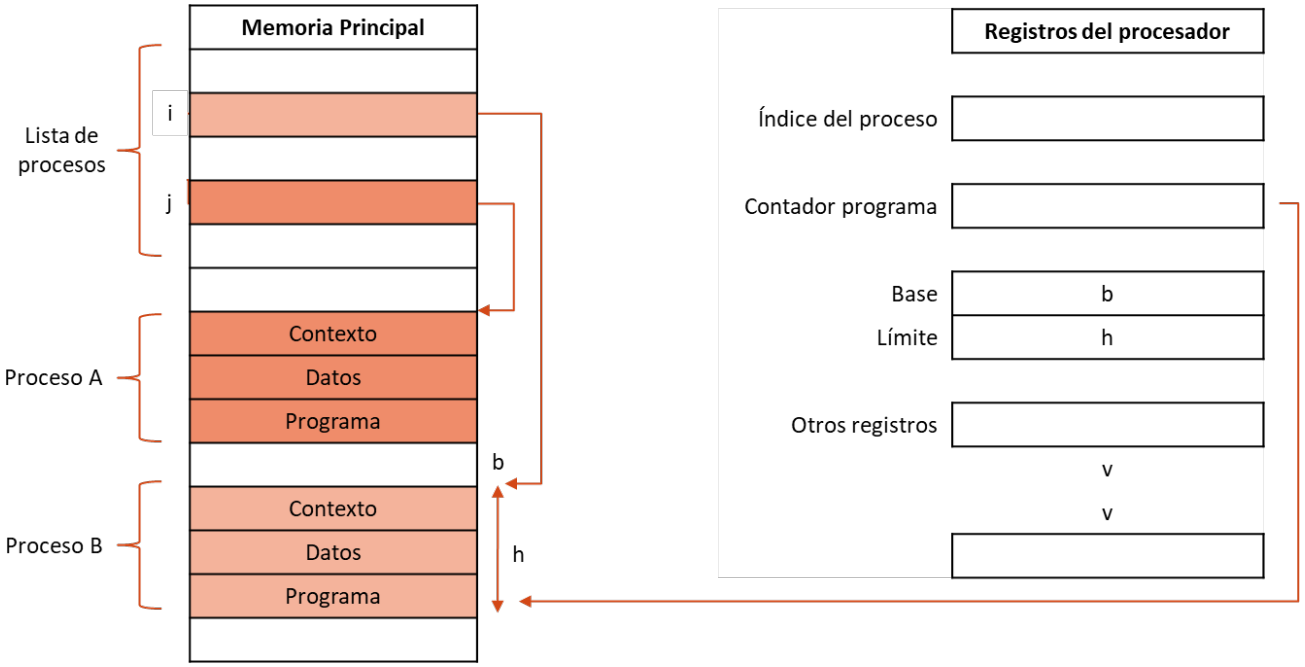
Índice de contenidos

2. Definiciones de proceso

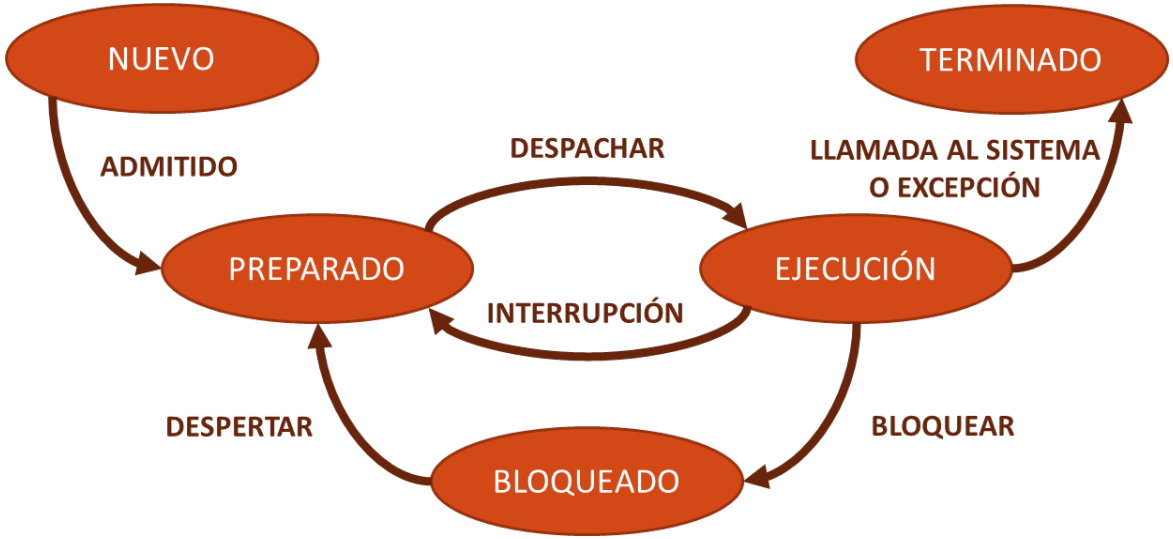
- Entidad básica que puede ser asignada a un procesador y ser ejecutada en él
- Unidad de actividad caracterizada por un sencillo tratamiento de ejecución secuencial, un estado actual, y asociada a un conjunto de recursos del sistema
- Un **proceso NO** es un **programa.** En realidad es más, **contiene**:
  - El **código** / **instrucciones** que van a ser ejecutadas en el microprocesador
  - **Datos** asociados: variables
  - **Contexto**: valores de registros, contador de programa, pila, **memoria** que ha reservado y su contenido, otra **información de planificación** para el S.O., …
- **Resumen:** un proceso es la entidad mas pequeña individualmente planificable, formada por código, datos y contexto, y caracterizada por atributos y un **estado dinámico**

2. Definiciones de proceso

- Varios procesos en memoria preparados para su ejecución
- Los valores del contexto del proceso se cargan en los registros del procesador cuando el proceso entra en ejecución



2. Estados de un proceso



- **BLOQUEAR:** ocurre cuando un proceso no puede continuar porque necesita algún recurso que no esta disponible
- **INTERRUPCIÓN:** el planificador decide que un proceso ha consumido el suficiente tiempo de CPU y es hora de dejar que otro proceso la ocupe
- **DESPACHAR:** el planificador elige al proceso que entra en ejecución
- **DESPERTAR:** un proceso pasa de estado Bloqueado a Preparado cuando el evento externo que estaba esperando se lleva a cabo

Índice de contenidos

3. Rutas absolutas y relativas

- El sistema de archivos en los sistemas operativos tipo UNIX forman un árbol a partir del directorio raíz (/).
- Poseen dos 'caracteres' especiales:
  - Para identificarse a sí mismo (.)
  - Para identificar a su directorio padre (..)
- A diferencia con los sistemas operativos tipo MSDOS (\), los nombres de los directorios se designan mediante el carácter (/)

→ Padre: usr

→ Padre: usuario

→ Padre: usuario

→ Padre: usuario

3. Rutas absolutas y relativas

- Los nombres de los archivos se pueden especificar de dos formas:
  - **Absoluta:** Escribiendo la ruta completa desde el directorio raíz (Por ejemplo: /usr/usuario/texto.txt)
  - **Relativa:** Tomando como punto de partida otro directorio distinto del raíz:
    - El directorio actual: Sólo hay que escribir el nombre del archivo (Por ejemplo: texto.txt)
    - Otro directorio: Hay que escribir las referencias correspondientes (Por ejemplo, desde el directorio /usr/usuario/directorio1/directorio2/, habría que escribir ../../texto.txt)

Índice de contenidos

4. Variables de entorno

- Una variable de entorno es un nombre asociado a una cadena de caracteres
- Dependiendo de la variable, su utilidad puede ser distinta. Algunas son útiles para no tener que escribir muchas opciones al ejecutar un programa, otras las utiliza el propio shell (PATH, PS1,…)

Variable	Descripción
DISPLAY	Donde aparecen la salidas de X-Windows
HOME	Directorio personal
HOSTNAME	Nombre de la máquina
MAIL	Archivo de correo
PATH	Lista de directorios donde buscar los programas
PS1	Prompt
SHELL	Intérprete de comandos por defecto
TERM	Tipo de terminal
USER	Nombre del usuario

4. Variables de entorno

- En el directorio raíz del usuario existen dos archivos de inicialización que contienen órdenes para que se ejecuten automáticamente cada vez que el usuario inicie una sesión
  - El archivo **.bashrc** contiene la definición de la variable de entorno prompt (utilizada para mostrar el indicativo de introducción de órdenes, por ejemplo "disco1:home/usuario1>") y las definiciones de alias (palabras equivalentes a órdenes, por ejemplo, si un usuario desea que al escribir una orden "la" puede definir un alias: alias la='ls –A')
  - El archivo **.profile** contiene la definición del tipo de terminal que se está usando, ajustes de configuración, variables de entorno, etc ...
- Estos dos archivos se leen cada vez que el usuario inicia una sesión y por lo tanto, se pueden utilizar para personalizar otra serie de variables de entorno así como para hacer que se ejecuten programas siempre que se inicie una sesión

4. Variables de entorno: bash

- La forma de definir una variable de entorno cambia con el interprete de comandos, para bash:
  - export VARIABLE=Valor
- Por ejemplo, para definir el valor de la variable DISPLAY:
  - export DISPLAY=localhost:0.0
- La variable PATH se exporta de la siguiente manera:
  - export PATH=……:$PATH
- Para ver el contenido de las variables de entorno
  - env
- Si queremos ver una variable en particular:
  - echo $VARIABLE (echo $PATH)

Índice de contenidos

5. Metacaracteres

- Son caracteres que tienen un significado especial, denominados **comodines**. A diferencia del resto de los caracteres, no tienen un significado literal
  - El carácter ***** representa a cualquier cadena de cero o más caracteres dentro del nombre de un archivo
  - El carácter **?** se corresponde con un único carácter sea éste cuál sea
  - Una secuencia de caracteres encerrada entre corchetes (**[ ]**) se corresponde con único carácter de los especificados, presentándose las siguientes excepciones:
    - Si el primer carácter de la secuencia es **!**, se corresponde con cualquier carácter excepto los especificados
    - Si aparece un **-** entre dos caracteres de la secuencia, se corresponde con cualquier carácter en el rango entre ambos

5. Metacaracteres: ejemplos

- El siguiente ejemplo borraría los archivos cuyo nombre comience y termine por J (incluido el archivo JJ), aquellos cuyo nombre empieza por la letra p seguida de otros cuatro caracteres cualquiera, los que tengan un nombre de dos caracteres ambos comprendidos entre la a y la z minúsculas y por último, los archivos cuyo nombre tenga una longitud de al menos dos caracteres tal que el primer carácter sea 1 o 2 y el segundo no sea numérico (excepto 0)
  - rm -f J*J p???? [a-z][a-z] [12][!1-9]*
- La siguiente línea invoca la función *copia_seguridad* antes presentada, pasándole como argumentos todos los archivos del directorio actual de trabajo
  - copia_seguridad *

Índice de contenidos

6. Ejecución de comandos

- Al ejecutar un programa, tenemos diferentes **opciones** (conocidas como **parámetros**) que son especificaciones que se le hacen al programa al momento de llamarlo para obtener un efecto diferente
- Los **argumentos**, son las **variables** que se le pasan al programa
- Por lo general, los comandos tienen el parámetro **-h** o **--help**, que nos muestra los parámetros disponibles para el comando y los argumentos que se le pueden pasar
- Ejemplo de ejecución:
  - programa [parámetros] [argumentos]
  - rm -f *.old

6. Ejecución: primer/segundo plano

- En Linux podemos ejecutar procesos en primer plano (foreground) o en segundo plano (background)
  - **Primer plano:** el programa lanzado en foreground monopoliza el terminal, por lo que en principio, no podremos ejecutar ningún otro programa a la vez, debemos esperar a que acabe
  - **Segundo plano:** el programa deja de monopolizar el terminal desde el que se lanzo, y este nos vuelve a mostrar el prompt
- Para lanzar en **segundo plano** añadimos **&** al final:
  - programa **&**

6. Ejecución: primer/segundo plano

- Si ejecutamos un comando de la forma habitual, lo podemos **pasar a segundo plano** hacer sin necesidad de tener que matar el proceso y volver a ejecutarlo:
  - Se suspende la ejecución del comando actual presionando "**CTRL+Z**"
  - Se lanza el proceso pausado en segundo plano invocando el comando **bg**
- Para ver el listado de procesos pendientes podemos ejecutar: **jobs**
- Para traer un proceso a primer plano, se realiza con el comando **fg**
- En caso de tener varios procesos (el número de proceso es el que aparece entre corchetes en la respuesta del comando jobs) podemos recuperar el proceso deseado con:
  - fg *identificador*

6. Ejecución: agrupamiento

- **Lista secuencial:** Se ejecutan de forma secuencial (de izquierda a derecha) todas las órdenes incluidas en la lista. El resultado es el mismo que si cada orden se hubiera escrito en una línea diferente
  - Ejemplo: Intercambia el nombre de dos archivos usando repetidamente la orden mv:

    mv arch1 aux; mv arch2 arch1; mv aux arch2

- **Lista con cauces ó tuberías (pipes |):** Se ejecutan de forma concurrente las órdenes incluidas en la lista de tal forma que la salida estándar de cada orden queda conectada a la entrada estándar del siguiente en la lista mediante un mecanismo denominado cauce (pipe)
  - Ejemplo: Muestra en orden alfabético (sort) aquellas líneas que estando entre las 10 primeras (head) contienen la palabra pepe (grep) del archivo.txt:

    head -10 archivo.txt | grep Pepe | sort

Índice de contenidos

7. Editor vi

- Aunque es bastante deseable, los sistemas operativos no siempre incorporan una interfaz gráfica, lo que limita al usuario a utilizarlos en modo texto. "**vi**" es uno de los más conocidos en los sistemas operativos tipo UNIX
- Para llamar al editor **vi**, hay que escribir la orden:

vi [*archivo1*] [*archivo2*] ...

Si existe *archivo1*, aparecerá en pantalla el texto del mismo, en caso contrario, aparecerá un archivo de texto en blanco que será creado en el caso de que se guarden los cambios

Si aparece más de un archivo en la línea de comandos ([*archivo1*] [*archivo2*] ...) éstos se editarán de forma secuencial, es decir, uno detrás de otro, dado que **vi** no tiene soporte para la edición simultánea de varios archivos

7. Editor vi

- Hay varios modos de ejecución, pero nos centraremos en los más básicos
  - Modo de órdenes: Para introducir órdenes dirigidas al editor (cambiar de modo, de línea, terminar, ...)
  - Modo de entrada de texto: Permite introducir y modificar texto directamente en el búfer del archivo
- Para introducir texto, desde el modo de órdenes pulsaremos la letra **i**
- Una vez terminemos, pulsaremos la tecla **ESCAPE** para volver al modo de órdenes
- Para guardar y cerrar el fichero, escribiremos **:wq** (**w** escribe el fichero y **q** lo cierra)
- Para ver todas las opciones de **vi**, ver el manual

Índice de contenidos

8. Comandos

- Ayuda:
  - **man** : Manual del sistema
- Navegación:
  - **cd** : Cambia el directorio de trabajo actual
  - **ls** : Lista el contenido de un directorio
  - **pwd** : Imprime el directorio de trabajo actual
- Gestión de ficheros:
  - **mkdir** : Crea un directorio
  - **rmdir** : Borra un directorio
  - **cp** : Copia archivos
  - **mv** : Mueve o renombra un archivo
  - **rm** : Borra un archivo
  - **touch** : Cambia las fechas de acceso y modificación de un archivo

8. Comandos

- Gestión de procesos:
  - **ps** : Muestra el estado de los procesos del sistema
  - **top** : Muestra una lista ordenada de procesos que están en ejecución
  - **kill** : Manda una señal a un proceso
- Otros:
  - **cal** : Muestra un calendario en la salida standard en el terminal
  - **Date** : Muestra la fecha hora actual del sistema o la establece
  - **fortune** : Muestra una frase
  - **file** : Determina el tipo de ficheros
  - **grep** : Muestra las líneas que coincide con el patrón introducido
  - **cat** : Muestra el contenido de un fichero por la salida estándar
  - **more** : Muestra el texto de un fichero largo deteniéndose por pantallas
  - **id** : Muestra la información y grupo del usuario indicado, o actual
  - **whoami** : Muestra el nombre de usuario asociado al ID del usuario actual

Índice de contenidos

9. Ejercicios

1. Determinar la utilidad en función de sus opciones para las siguientes órdenes:
   a) ls: a, b, c, C, d, f, F, g, i, l, m, p, q, R
   b) file: c, f, m
   c) find: print, group, atime, exec, newer

2. Qué orden habría que escribir para:
   a) Mostrar de forma completa todos los archivos de un directorio
   b) Buscar todos los archivos de texto (*.txt) del disco
   c) Listar el contenido de un archivo de texto (*archivo1*) que se encuentra en el mismo directorio
   d) Encontrar todos los archivos más recientes que .profile contenidos en los subdirectorios del directorio de usuario
   e) Copiar el archivo *archivo1* en el archivo *archivo2*
   f) Eliminar el archivo *archivo1*
   g) Mostrar la fecha y la hora en formato, díaSemana, día mes año, hora:minuto:segundo

9. Ejercicios

3. Crea un alias “hig”, de modo que al ejecutar “hig <patrón>”, muestre los últimos comandos ejecutados en el sistema que contienen dicho patrón introducido (por ejemplo, hig gcc):
   - “history” muestra los últimos comandos ejecutados
   - Recordar el uso del pipe (|)

4. ¿Qué contenido tiene la variable de entorno PATH?

a) Crea una carpeta en el escritorio que se llame “programas”

b) Cambia el directorio actual para entrar en la nueva carpeta

c) Genera un fichero “hola.sh” utilizando el editor vi con el siguiente contenido:

```
#!/bin/bash
echo Hola Mundo!
```

d) Cambia los permisos para poder ejecutarlo: `chmod +x hola.sh`

e) Vuelve a tu directorio home: `cd ~`

f) Ejecuta el programa “hola.sh”. ¿Funciona?

g) Realiza el apartado b) y añade a la variable PATH el nuevo directorio de trabajo

h) Realiza de nuevo los apartados e) y f)

Universidad Rey Juan Carlos
Escuela Técnica Superior Ingeniería de Telecomunicación

Programación de Sistemas de Navegación

1.1. Lenguaje C

Sintaxis, Compilador, Ámbitos, Tipos, Operadores, Sentencias de control e iteración, Entrada y Salida, Errores

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

Índice de contenidos

1. Forma general programa en C

```
directivas del preprocesador      // Precedidas por un #
declaraciones globales            // Incluyendo funciones

int main (int argc, char *argv[]) {
        secuencia de sentencias
}

tipo_devuelto funcion_1 (lista de parámetros) {
        secuencia de sentencias
}

tipo_devuelto funcion_2 (lista de parámetros) {
        secuencia de sentencias
}
...
tipo_devuelto funcion_n (lista de parámetros) {
        secuencia de sentencias
}
```

- Código fuente:
  - Formado por uno o varios ficheros
  - Debe ser compilado y enlazado

1. Compilador

- El compilador es el programa encargado de traducir un conjunto de funciones, definiciones y declaraciones de múltiples ficheros en un fichero ejecutable
- El compilador de C procesa los ficheros de código uno a uno de forma independiente, de forma que las definiciones de variables y funciones de un fichero **no las recuerda cuando procesa el fichero siguiente**. Además, el compilador realiza una única pasada por el texto, por lo que cuando se traduce un fichero, en un punto concreto del texto, sólo se dispone de las definiciones que se han encontrado hasta ese punto

1. Preprocesador

- El preprocesador es el primer programa invocado por el compilador y procesa directivas como **#include**, **#define** e **#if**

```
#include <stdio.h>

int main (void)
{
   printf ("¡Hola Mundo!\n");
   return 0;
}
```

- El preprocesador reemplaza la línea **#include <stdio.h>** por el contenido del archivo de cabecera del sistema con ese nombre. Dentro se declara, entre otras cosas, la función **printf()**
- Puede escribirse usando dobles comillas: **#include "stdio.h"**. Esta forma solía usarse para diferenciar los archivos de cabecera del sistema (<>) y los de usuario ("")

1. Comentarios

- Un comentario es una sección de texto cuyo objetivo es documentar parte de un programa. Los comentarios se ignoran por el compilador. Un comentario simple comienza con dos barras de dividir

```
// esto es un comentario
```

- Un estilo alternativo de comentarios en varias líneas es el formado por barra y asterisco

```
/*
esto es un comentario
de varias líneas
*/
```

- Los comentarios son fundamentales para la comprensión de un programa

1. Función main

- Es la función principal que debe existir en todos los programas
- La forma general es la siguiente:

```
int main (int argc, char *argv[]) { cuerpo }
```

- Donde:
  - **argc** : Valor no negativo que indica el número de argumentos enviados al programa desde el entorno donde se ejecuta
  - **argv** : Puntero al primer elemento de los argumentos pasados al programa desde el entorno de ejecución (argv[0] hasta argv[argc-1]). Se garantiza que el valor argv[argc] es un puntero nulo
  - **cuerpo** : El cuerpo de la función main

Índice de contenidos

2. Declaración de variables

- **Variable**: posición de memoria con nombre que se usa para mantener un valor que puede ser modificado por el programa. La declaración de una variable tiene la sintaxis siguiente:

```
[prefijo] tipo identificador [=expresión inicialización];
```

- Donde el prefijo puede ser:
  - **static** : indicando que sólo se mantiene una instancia de la variable
    - Si se declara en una función, el valor se mantiene en futuras ejecuciones dentro de la función
    - Si se declara fuera de una función, el valor es accesible dentro del fichero pero no es accesible para otros módulos o funciones fuera del fichero actual
  - **extern** : dota a la variable de alcance global a todo el programa (incluidas las librerías externas). Se coloca en la declaración de variables pertenecientes a otros módulos o funciones que vayan a ser utilizadas en el módulo o función actual
- Los prefijos **extern** y **static** no pueden ser utilizados en la misma declaración
- El lenguaje C es **sensible a mayúsculas y minúsculas**, además, no se pueden utilizar tildes

2. Constantes

- Las constantes pueden ser:
  - **Constantes definidas (simbólicas)**: utilizando la directiva **#define**. El preprocesador se encarga de sustituir la constante por su expresión o valor asociado

```
#define CTE valor;
```

  - **Constantes declaradas**: el cualificador **const** impide que el valor de una variable pueda ser alterado

```
const char ARROBA = '@';
const INT LIMITE = 2999;
const char ARCHIVO[] = "salida.txt";
```

  - **Constantes enumeradas**: permite crear una lista de elementos relacionados. Es posible utilizar ordinales para referenciar los elementos de la lista (empieza en 0)

```
enum DiaSemana { L, M, X, J, V, S, D };   // Tipo
enum DiaSemana diaLibre = D;              // Variable
```

2. El punto y coma

- El punto y coma se dispondrá, por regla general, al final de cada sentencia. Los delimitadores, como las llaves de apertura y cierre del bloque de código, no van seguidos de ese elemento (salvo en la declaración de clases en C++)
- Se entiende por sentencia cualquier línea o grupo de líneas de código ejecutable. No es una sentencia, por ejemplo, la cabecera de una función

```
#include <stdio.h>

int main (void)
{
   printf ("¡Hola Mundo!\n");
   return 0;
}
```

Índice de contenidos

3. Ámbito de las variables

- El **ámbito**, es la zona del programa en el que una variable es conocida
- Variables **locales**:
  - Son conocidas solo en la función en la cual están definidas (ninguna otra función tiene acceso a ellas)
  - La mayor parte de las funciones tienen una lista de parámetros. Los parámetros de una función también son variables locales
  - Las variables locales comienzan su existencia cuando la función es llamada y desaparecen cuando la función termina su ejecución
- Variables **globales**:
  - Son externas a todas las funciones, esto es variables globales que pueden ser accedidas por cualquier función
  - Pueden ser utilizadas para comunicar información entre funciones
  - Las variables externas existen permanentemente y retienen su valores aun después de que las funciones que las utilizan han terminado su ejecución

3. Ámbito de las variables

¿Por qué evitar el uso de variables globales?

- Varios módulos o funciones pueden interferir entre sí
  - Al programar somos predecibles. ¿Cuántas veces no escribimos programas con variables como i, tmp, num, etc.? ¿Cuántas otras funciones que utilicemos, hechas por terceros, no tendrán nombres de variables similares? Si utilizamos variables globales tendremos que cuidarnos manualmente de no interferir con las variables que utilicen otros fragmentos del programa
- Código más difícil de mantener:
  - El manejar variables globales nos obliga a documentarlas también de manera global, no sólo para evitar los problemas que mencionamos, sino para facilitar la extensibilidad a futuro. Utilizar variables de ámbito más limitado nos permite documentarlas siempre al inicio de cada función o bloque
- Mayor uso de memoria:
  - El sistema operativo reclama el espacio utilizado por los datos tan pronto como estos dejan de ser necesarios. Sin embargo, cuando las variables son globales, no se puede predecir que los datos de una variable no volverán a ser necesitados
  - Al utilizar ámbitos más limitados, el sistema reclamará el espacio tan pronto salgamos del bloque en que fueron creadas
- Salvo causa justificada siempre **evitaremos el uso de variables globales**

Índice de contenidos

4. Tipos y modificadores

- Existen cinco **tipos básicos**, el resto de los tipos son combinaciones:
  - **char** : carácter (char)
  - **int** : entero (y su variante extendida long int)
  - **float** : coma flotante (float)
  - **double** : coma flotante de doble precisión (double)
  - **void** : nulo
- Existen otros tipos que son combinaciones de los anteriores, o bien utilizan modificadores:
  - **long** : Aplicable sobre (**int**, **double**), dobla el espacio de almacenamiento utilizado para almacenar la variable
  - **short** : Aplicable sobre (**int**), se utiliza cuando se desea una variable menor que un **int**
  - **signed** : Aplicable sobre (**int**, **char**, **long**, **short**), indica que el valor de una variable numérica puede ser positivo o negativo
  - **unsigned** : Aplicable sobre (**int**, **char**, **long**, **short**), se utiliza cuando la variable sea siempre positiva

4. Operadores

- Seis tipos distintos de **operadores**:
  - **Aritméticos** : + - * / % → Suma, Resta, Multiplicación, División, Módulo
  - **Lógicos** : ! && || → Negación, Conjunción Y, Disyunción O
  - **Relacionales** : == != > < >= <= → Igual, Distinto, Mayor/menor (igual) que
  - **Asignación** : = ++ -- += -= *= /= %=
  - **Dirección** : * (operador de contenido o indirección)
    & (operador de referencia o dirección)
  - **Movimiento** : << >>, destinados a operaciones sobre bits

Índice de contenidos

5. Sentencias de control: if/else

- La sentencia **if/else** permite comprobar si se da una determinada condición:

```
if (expresión_condicional) {
        /* las declaraciones se ejecutarán si la expresión booleana es verdadera */
}
else {
        /* las declaraciones se ejecutarán si la expresión booleana es falsa */
}
```

- Ejemplo:

```
meses mes; // meses: tipo enumerado con los meses del año
...
if ((mes >= Junio) && (mes <= Agosto)) {
        esVerano = 1;
}
else {
        esVerano = 0;
}
```

5. Sentencias de control: switch

- La sentencia **switch/case/break** se utiliza a la hora de comparar una variable con múltiples valores. La condición switch se debe establecer mediante un entero, carácter, o un valor enumerado. Si la sentencia break se omite, se ejecutará el siguiente case/default

```
switch (condición) {
  case condición1:
        declaraciones;
        break;
  case condición2:
        declaraciones;
        break;
...
  default:
        declaraciones;
}
```

Ejemplo:

```
switch (mes) {
  case Enero:
        maxTemp = 20; minTemp = 0;
        break;
  case Febrero:
        maxTemp = 30;  minTemp = 10;
        break;
...
  default:
        maxTemp = 60;
        minTemp = 20;
}
```

Índice de contenidos

6. Sentencias de iteración: while

- Los bucles, permiten la ejecución de un código reiteradas veces
- La condición debe de ser una expresión lógica, similar a la de la sentencia if. Primero se evalúa la condición. Si el resultado es verdadero, se ejecuta el bloque de código. Luego se vuelve a evaluar la condición, y en caso de dar verdadero se vuelve a ejecutar el bloque. El bucle se corta cuando la condición da falso

```
while (condición) {
        declaraciones;
}
```

- Ejemplo: imprimir los números de 0 a 99

```
int i = 0;
while (i < 100) {
        printf("%d\n", i);
        i = i + 1;
}
```

6. Sentencias de iteración: do...while

- Similar a **while**, pero con la diferencia de que el bucle **do...while** es un se ejecuta al menos una vez

```
do {
        declaraciones;
} while (condición)
```

- Ejemplo: imprimir los números de 0 a 99

```
int i = 0;
do {
        printf("%d\n", i);
        i = i + 1;
} while (i < 100);
```

6. Sentencias de iteración: for

- El bucle **for** es muy flexible y potente. Tiene diferentes formas de implementación, aunque la más común es la siguiente:

```
for ( inicialización; condición; incremento ) {
        declaraciones;
}
```

  - **Inicialización:** en esta parte se inicia la variable que controla el bucle y es la primera sentencia que ejecuta el bucle. Sólo se ejecuta una vez
  - **Expresión condicional:** al igual que en el bucle while, esta expresión determina si el bucle continuará ejecutándose o no
  - **Incremento:** es una sentencia que ejecuta al final de cada iteración del bucle. Por lo general, se utiliza para incrementar la variable con que se inicio el ciclo. Luego de ejecutar el incremento, el bucle revisa nuevamente la condición, si es verdadera tiene lugar una ejecución más, si es falsa se termina

- Ejemplo: imprimir los números de 0 a 99

```
int i;
for (i=0; i < 100; i = i + 1) {
        printf("%d\n", i);
}
```

Índice de contenidos

7. Funciones de E/S formateada

- Las funciones de E/S sirven para la transferencia de información entre el ordenador y los dispositivos de entrada/salida estándar, tales como un teclado y un monitor
- Como norma general, el archivo de cabecera requerido para la entrada/salida estándar se llama **stdio.h**

```
#include <stdio.h>
```

- La función **scanf** permite leer varios tipos de datos de una sola vez, tales como enteros, números decimales o cadenas de caracteres. Cada nombre de variable debe ser precedido por un ampersand (&), salvo en el caso de los arrays

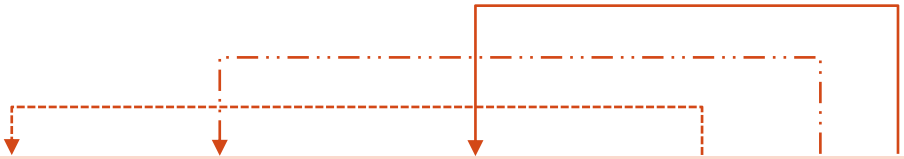
```
scanf(cadena de control,arg1,arg2,...,argN);
```

- La función **printf** permite escribir cualquier combinación de valores numéricos, caracteres sueltos y cadenas de caracteres

```
printf(cadena de control,arg1,arg2,...,argN);
```

7. Funciones de E/S formateada

- Ejemplo salida:



Código
```
printf("Nombre: %s; Edad: %d; Altura: %5.2f cm", "Juan", 23, 1.70);
```

Salida
```
Nombre: Juan; Edad: 23; Altura: 1.70 cm
```

- Ejemplo entrada:

Código
```
char nombre[10];
int edad;
float altura;

scanf("%s %d %f", nombre, &edad, &altura);
```

Carácter de conversión	Significado del dato
c	Carácter
d	Entero decimal
e	Coma flotante
f	Coma flotante
g	Coma flotante
h	Entero corto
o	Entero octal
s	Cadena de caracteres
u	Entero decimal sin signo

Índice de contenidos

8. Compilación

- Para la realización de las prácticas de este curso se utilizará el compilador **gcc** (compilador C de GNU, http://gcc.gnu.org)
- **gcc** permite realizar y controlar las etapas de preproceso, compilación, ensamblado y enlazado necesarias para generar el código objeto de un programa
- La orden **gcc** genera por defecto, como resultado de la compilación, un archivo con el nombre **a.out** con el código ejecutable
- Para especificar otro archivo destino se puede utilizar el flag o

```
gcc prueba.c -o ejecutable
```

  - devuelve, en caso de una compilación sin errores, un archivo ejecutable con el nombre **ejecutable**, resultado de compilar **prueba.c**

```
gcc prog1.c prog2.c prog3.c  -o salida
```

  - compila **prog1.c**, **prog2.c** y **prog3.c**, y los enlaza generando **salida**

- **gcc** decide qué hacer con cada archivo (preproceso, enlazado, ...) dependiendo de la extensión del archivo origen: .c, .o, ...

8. Errores

- **Compilación**:
  - Son errores que impiden la ejecución del programa
  - Los errores son detectados por el compilador antes de producir el resultado
  - Suelen ser errores de sintaxis, mala escritura del código fuente
  - Los errores más comunes son: comandos mal escritos, orden incorrecto de las operaciones, variables con tipos no coincidentes y omisión de los elementos necesarios

- **Ejecución**:
  - Son errores que ocurren mientras un programa se ejecuta
  - Son más difíciles de detectar y corregir
  - Pueden ocasionar que un programa se bloquee, por ejemplo con un bucle infinito, o que deje de ejecutarse prematuramente, por ejemplo una división por 0

Índice de contenidos

9. Ejercicios

1. Escribe un programa que escriba en la primera línea su nombre, en la segunda su apellido y lo muestre por pantalla
2. Escribe un programa que te pida tu edad y tu nombre, y te calcule el año en que naciste. Muestra los datos por pantalla
3. Escribe un programa que indique si un número introducido por teclado es par o impar
4. Escribe un programa que indique a partir de un número introducido por teclado, el nombre del mes correspondiente utilizando **switch**
5. Escribe un programa que muestre por pantalla los números del 10 al 100 utilizando **for**, **while** y **do…while**
6. Dados tres números enteros introducidos por teclado, escribe un programa que determine cuál es el mayor

9. Ejercicios

7. Escribe un programa que pida por teclado cuatro números y calcule y muestre la media de los cuatro

8. Escribe un programa que pida el diámetro y la altura por teclado, y calcule y muestre el área de un cilindro:

$$Área = 2 \cdot \pi \cdot r \cdot (r + h)$$

9. Escribe un programa que muestre las tablas de multiplicar de los números pares

$2x0 = 0$ $\quad\quad$ $8x0 = 0$
$2x1 = 2$ $\quad$ ……. $\quad$ $8x1 = 8$
$2x2 = 4$ $\quad\quad$ $8x2 = 16$
…… $\quad\quad$ ……

9. Ejercicios

10. Escribe un programa que pregunte el precio, el tanto por ciento de descuento, y te diga el precio con descuento. Por ejemplo, si el precio que introduce el usuario es 300 y el descuento 20, el programa dirá que el precio final con descuento es de 240

11. Escribe un programa que pregunte al usuario los dos lados de un rectángulo y presente por pantalla el cálculo del perímetro (suma de los lados) y el área (base por altura)

12. Suponiendo que 1 euro = 1.11343 dólares. Escribe un programa que pida al usuario un número de dólares y calcule y muestre el cambio en euros de forma continua. El programa se cerrará cuando el usuario introduzca un número negativo

13. Escribir un programa que pida por teclado los tres coeficientes (a, b y c) de la ecuación $ax^2+bx+c=0$ y calcule y muestre las dos soluciones, suponiendo que ambas serán reales (es decir que la raíz queda positiva)

Nota: $x_{1,2}=(a\pm sqrt(b^2-4ac))/2$, **sqrt** es una función que devuelve la raíz cuadrada, para poder invocarla es necesario poner en la cabecera del programa: **#include <math.h>**

Universidad Rey Juan Carlos
Escuela Técnica Superior Ingeniería de Telecomunicación

Programación de Sistemas de Navegación

1.2. Lenguaje C

Arrays, Strings, Struct, Punteros, Funciones

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

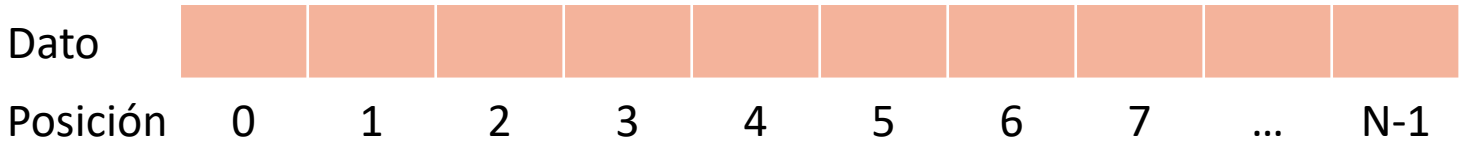
EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

Índice de contenidos

1. Arrays

- Colección finita de valores del mismo tipo
- En C, las posiciones comienzan en 0



- La última posición accesible siempre será N-1

```
int temperatura[12]; // declaración de un array con 12 enteros
// asignación de los valores
temperatura[0] = 20;
temperatura[1] = 32;
temperatura[2] = temperatura[3] + 5;
….
temperatura[11] = temperatura[4] * temperatura[7];
```

1. Arrays

- Los vectores de mayores dimensiones se pueden crear de la siguiente forma:

```
double matriz[10][20];
```

- Los arrays también pueden ser inicializados a la vez que se declaran

```
char abecedario[26] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N',
                       'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z' };
```

- Cuando un array se usa como argumento, no se introduce el tamaño del array en este, sino como un argumento adicional

```
int suma_array(int values[], int n);
```

Índice de contenidos

2. Strings

- Los **strings** no son más que un **array de caracteres** que termina en un carácter nulo (**\0**), que significa fin de cadena. En este caso los caracteres van entre **comillas simples**:

```
char saludo[5] = {'H', 'O', 'L', 'A', '\0'};
```

- El caso anterior es útil cuando se sabe la longitud de la cadena. Si no se sabe la longitud, es posible declararlo utilizando **comillas dobles**. El siguiente ejemplo muestra cómo se declara el mismo string sin indicar el tamaño:

```
char saludo[] = "HOLA";
```

- Cuando se declara un string de la segunda forma, el compilador se encarga de añadir el carácter de fin de cadena

2. Operaciones con strings

- Para poder utilizar funciones que trabajan con cadenas de caracteres, es necesario importar la librería **string.h**

```
#include <string.h>
```

- Operaciones comunes:
  - **strcpy**(str1, str2) : Copia string2 en string1
  - **strcat**(str1, str2) : Concatena string2 al final de string1
  - **strlen**(str) : Devuelve el número de caracteres que tiene la cadena (sin el '\0')
  - **strcmp**(str1, str2) : Devuelve 0 si ambas cadenas son iguales
  - **strchr**(str1, ch) : Devuelve un puntero a la primera aparición de ch en str1
  - **strstr**(str1, str2) : Devuelve un puntero a la primera aparición de str2 en str1

Índice de contenidos

3. Tipos de datos estructurados

- Son estructuras de datos que agrupan campos de otros tipos
- Esta construcción define un nuevo tipo de datos. Es decir, un tipo de dato estructurado tiene la misma entidad que el tipo "**int**" o "**float**"
- El nombre del nuevo tipo estructurado es "**struct nombre_struct**"

```
struct nombre_struct
{
   tipo_1 campo1;
   tipo_2 campo2;
   ...
   tipo_N campoN;
};
```

```
#define NOMBRE_SIZE 100
#define APELLIDOS_SIZE 200
#define CONTACTOS_NUM 100

/* Definición de la estructura */
struct informacion_persona
{
   char nombre[NOMBRE_SIZE];
   char apellidos[APELLIDOS_SIZE];
   unsigned int telefono;
   unsigned int edad;
};

/* Declaración de variables con esta estructura */
struct informacion_persona persona1, persona2, listado[CONTACTOS_NUM];
```

3. Tipos de datos estructurados

- El **acceso** a los campos de una variable estructurada se denota por el nombre de la variable seguido de un **punto** y del nombre del campo

```
int main (int argc, char *argv[])
{
   /* Declaración de variables con esta estructura */
   struct informacion_persona persona1;

   persona1.nombre[0] = 'C';
   persona1.nombre[1] = 'D';
   persona1.apellidos[0] = 'A';
   persona1.apellidos[1] = 'B';
   persona1.telefono = 12314223;
   persona1.edad = 24;
}
```

3. Tipos de datos estructurados

- Los tipos estructurados pueden anidarse. El único requisito es que **la definición de un tipo preceda a su uso**
- Los accesos a estructuras anidadas se realizan a través de puntos

```
struct direccion
{
   char calle[20];
   int numero, cp;
   char pais[10];
};

struct informacion_persona
{
   char nombre[20];
   int edad;
   struct direccion dir;
};
```

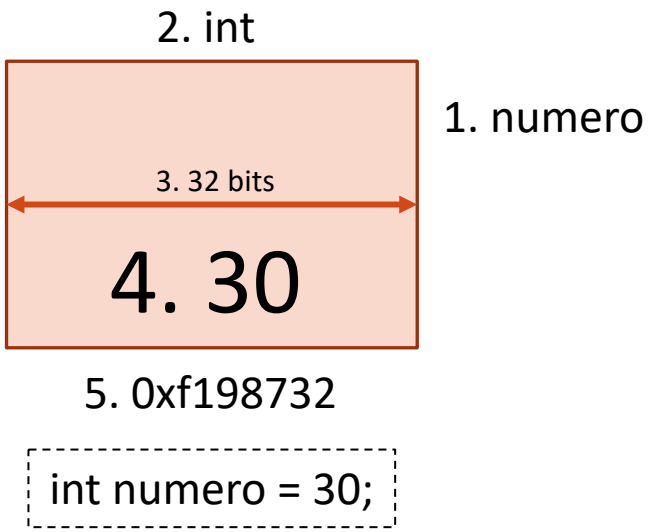
```
int main (int argc, char *argv[])
{
   struct informacion_persona persona1;
   persona1.nombre[0] = 'C';
   persona1.nombre[1] = 'D';
   persona1.edad = 24;
   persona1.dir.calle[4] = 'M';
   persona1.dir.cp = 28850;
}
```

Índice de contenidos

4. Punteros

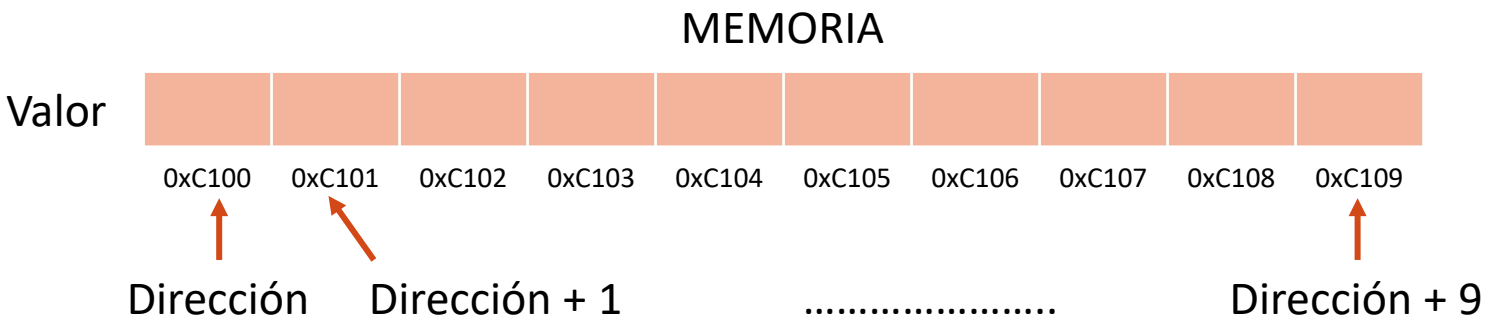
- Los punteros permiten manipular la memoria del ordenador de forma eficiente

- Una variable en C tiene 5 elementos:
  1. Nombre de la variable
  2. Tipo de dato
  3. Tamaño
  4. Valor
  5. Dirección de memoria



4. Punteros: memoria

- La memoria es un conjunto de celdas contiguas donde se almacenan datos
- Cada celda tiene dos valores asociados: **dirección y contenido**
- La unidad más pequeña de memoria es el **bit** (0 o 1)
- El conjunto de 8 bits es lo que se denomina **byte** (00000000 …. 11111111)
- El lugar donde se almacena cada byte es único, y es su dirección
- Si los bytes son consecutivos la dirección se incrementa secuencialmente



4. Punteros

- Un puntero es una variable que contiene la dirección física (dirección de memoria) de un elemento determinado del programa

- Ejemplo:

```
int *p = 32;
```

- Para operar con punteros C dispone de los operadores * (valor) y **&** (dirección)

4. Punteros: operadores * y &

- Todas las **variables** se almacenan en una posición de memoria que se puede obtener con el operador ampersand (**&**). Este operador significa "**la dirección de**"
  - Por ejemplo, si declaramos una variable int num = 5, **&num** sería **la dirección de** memoria de num, mientras que num es el valor (contenido)

- Si en lugar de utilizar una variable se utiliza un **puntero**, este almacena la dirección de memoria, no el valor. Para acceder directamente al valor almacenado en la variable apuntada se utiliza el operador asterisco (*). Este operador significa "**valor apuntador por**"
  - Así, si declaramos un puntero int *num = 5, num contiene la dirección de memoria, y ***num** sería **el valor apuntado por** num, su contenido

4. Punteros: operadores * y &

- Comparativa entre declarar una variable con o sin punteros:

Sin punteros	Con punteros
int num = 5; num → valor de num, 5 &num → dirección de memoria de num	int *num = 5; num → dirección de memoria de num *num → valor de num, 5

- Importante, las direcciones de memoria no son fijas, varían con cada ejecución del programa

4. Punteros: tipos de datos

- Al igual que una variable, un puntero puede ser:
  - **char*** : carácter
  - **int*** : entero
  - **float*** : coma flotante
  - **double*** : coma flotante de doble precisión
  - **void*** : nulo
- La forma general para declarar un puntero es:

```
tipo *nombre_puntero ;
```

- Ejemplos:

```
int *num ;
float numf1, *numf2;
double *numd1, *numd2 ;
```

Se pueden declarar variables y punteros en la misma línea

4. Punteros: asignaciones

- Se puede asignar una variable a un puntero:

```
int edad = 10;
int *ptr = &edad;
```

- Así como un puntero a otro puntero, siempre que sean **compatibles**:

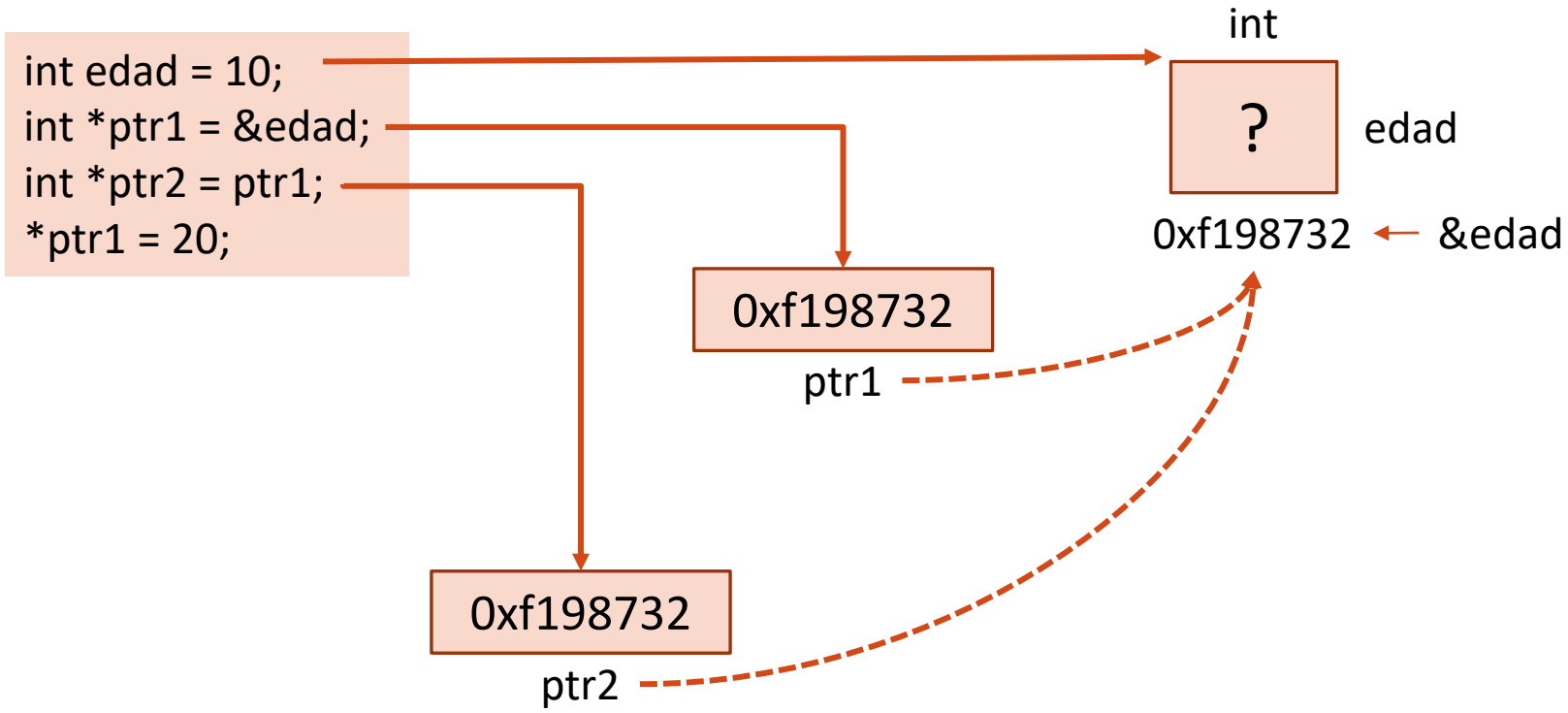
```
int edad = 10;
int *ptr1 = &edad;
int *ptr2 = ptr1;
*ptr1 = 20;
```

No hay que poner &, ptr1 ya contiene la dirección de memoria

- ¿Cuál es el valor de edad, *ptr1 y *ptr2?

4. Punteros: asignaciones

- ¿Cuál es el valor de edad, *ptr1 y *ptr2?



4. Punteros: operaciones

- A un puntero es posible asignarle el valor **NULL** (el puntero no apunta a ninguna dirección, dirección nula 0x000)

```
int *ptr;
ptr = NULL;
```

- Es posible **sumar y restar** una cantidad entera **N** a una variable puntero. Hay que tener en cuenta que la nueva dirección de memoria difiera de la original en **N * sizeof(tipo apuntado)**

```
int *ptr;  // El tamaño sizeof(int) de un int es 4 bytes
ptr += 4; // Sumando 4, la nueva dirección se ha incrementado en 16 bytes
ptr -= 1;  // Restando 1, la nueva dirección se ha decrementado en 4 bytes
```

4. Punteros: operaciones

- Es posible comparar dos variables puntero si estas son compatibles (apuntan datos de igual tipo)

```
ptr1 == ptr2    ptr1 != ptr2    ptr1 == NULL    ptr1 < ptr2    ptr1 >= ptr2
```

- Se pueden realizar **operaciones** sobre los datos apuntados a través de notación de punteros

```
*ptr1 > *ptr2    *ptr1++    (*ptr1)--
```

- ¿Cuál es la diferencia entre *ptr1++ y (*ptr1)++?

4. Punteros: memoria

- Todo puntero tiene que ser correctamente inicializado. Es decir, su valor tiene que ser nulo o apuntar a la dirección de memoria de una variable de su mismo tipo de dato:

```
int *ptr1;
*ptr1 = 20;  // Falla porque ptr1 no está inicializado
```

- ¿Qué ocurre si no sé la variable a la que tiene que apuntar o está declarada? Hay que **reservar el espacio de memoria** necesario para que albergue el tipo de dato requerido
- Para ello existe la función **malloc**, de la librería **stdlib.h**. Su forma genérica es:

```
tipo_dato * nombre_variable = (tipo_dato*)malloc(sizeof(tipo_dato));
```

- El ejemplo anterior se solucionaría con la inicialización:

```
int * ptr1 = (int*)malloc(sizeof(int));
```

4. Punteros: memoria

- También existe la función **calloc**, que al igual que malloc reserva la memoria pero la inicializa a cero
- Y ahora, ¿qué ocurre con esta memoria que ha sido asignada por el usuario?. Hay que liberarla cuando ya no sea necesaria. Para ello existe la función free:

```
free (variable_puntero) ;
```

4. Punteros: arrays

- Dado que un array es una sucesión de datos continuos en memoria del mismo tipo, es posible declararlo utilizando un puntero y reservando la memoria para cada una de sus N celdas:

```
tipo_dato * nombre_variable = (tipo_dato*)malloc(sizeof(tipo_dato) * N) ;
```

- Ejemplos equivalentes:

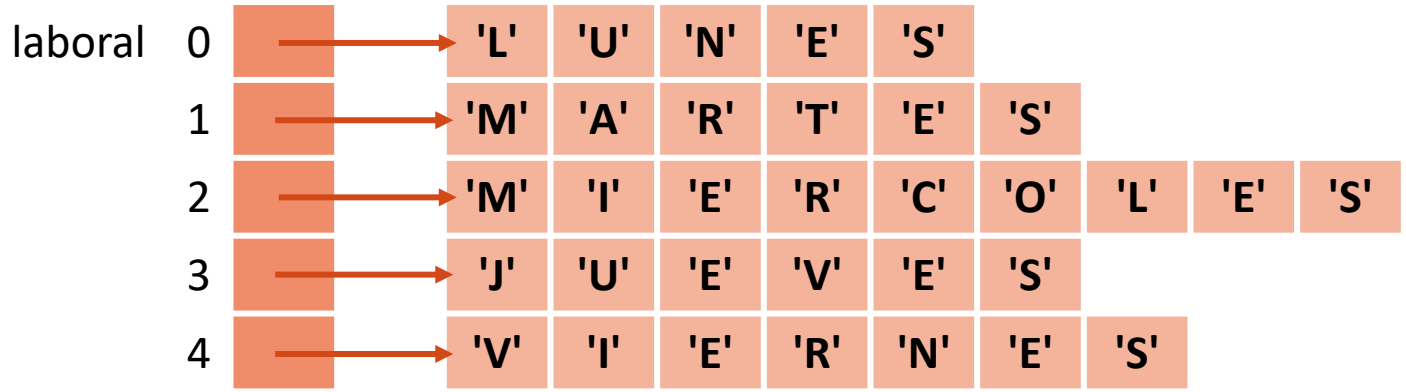
```
char ptr[15] ;
```

```
char * ptr = (char*)malloc(sizeof(char) * 15) ;
```

4. Punteros: arrays de punteros

- Un array de punteros funciona como un array de cualquier otro tipo de datos, con la diferencia de que cada elemento apunta a un valor del tipo declarado previamente
- Si queremos un **array de strings**, no es posible hacerlo con char, hay que utilizar un puntero

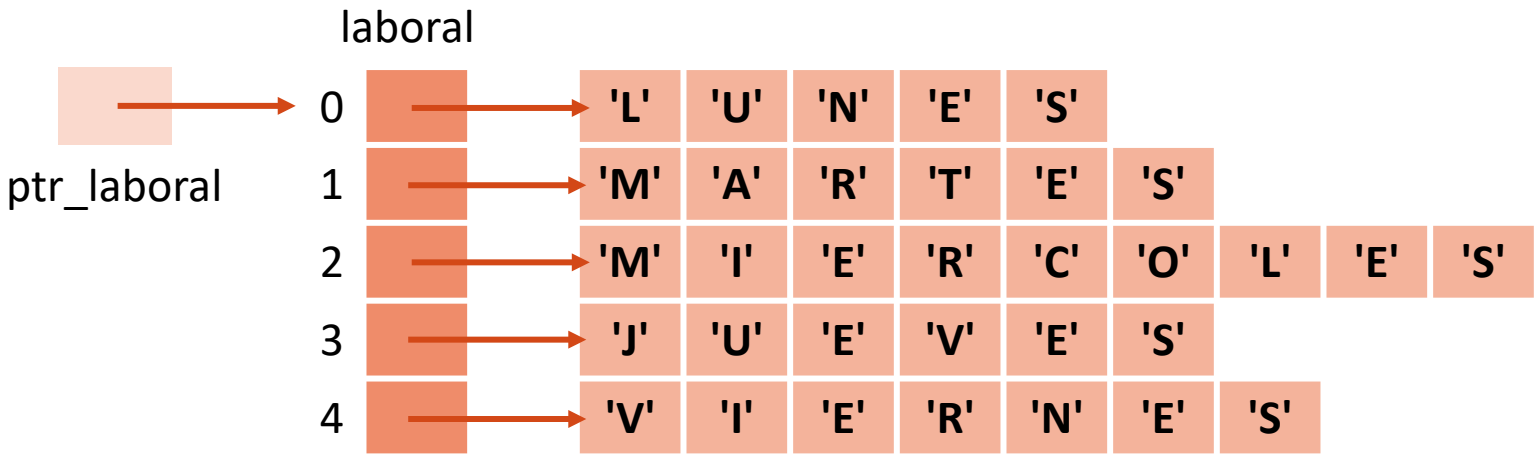
```
char* laboral[5] = {"LUNES", "MARTES", "MIERCOLES", "JUEVES", "VIERNES"} ;
```



4. Punteros: punteros a punteros

- Un puntero a puntero funciona de forma similar a otro puntero. Éste apunta a otras variables puntero y se usa mucha veces para apuntar a un array de punteros a char

```
char* laboral[5] = {"LUNES", "MARTES", "MIERCOLES", "JUEVES", "VIERNES"} ;
char** ptr_laboral = laboral ;
```



4. Punteros: acceso a struct

- Ya sabemos que para acceder a un elementro de una estructura, se utiliza el operador punto. Por ejemplo:

```
struct persona persona1;
persona1.edad = 50;
```

```
struct persona {
        char nombre[50];
        int edad;
};
```

- Pero, ¿qué ocurre si persona es un puntero a una structura?

```
struct persona *ptr_persona1 = (struct persona *)malloc( sizeof( struct persona ) );
```

- En este caso aplicando la forma anterior, el acceso se haría como:

```
(*ptr_persona1).edad = 50;
```

- Pero existe una alternativa:

```
ptr_persona1 -> edad = 50;
```

- Ambos métodos son equivalentes e intercambiables. **PUNTERO -> ATRIBUTO** es un método abreviado de **(*PUNTERO).ATRIBUTO**

4. Punteros: problemas comunes

- Olvidar poner el ampersand & al inicializar el puntero:

```
int edad = 20;
int *ptr = edad; // Hay que apuntar a la dirección de la variable &edad
```

- Los punteros tienen que ser declarados según el tipo de dato al que apuntan:

```
int edad = 20;
char *ptr = &edad; // Debe apuntar a 1 byte pero intenta apuntar a 4 bytes
```

Índice de contenidos

5. Funciones

- Las funciones en C tienen el siguiente formato:

```
tipo_del_resultado NOMBRE(tipo_param1 param1, tipo_param2 param2, ... )
{
        /* Cuerpo de la función */
}
```

- Cuando se invoca una función se asignan valores a sus parámetros y comienza a ejecutar el cuerpo hasta que se llega al final o se encuentra la instrucción **return**. Si la función devuelve un resultado, esta instrucción debe ir seguida del dato a devolver

```
int suma (int numero1, int numero2)
{
        int suma=0;
        suma = numero1 + numero2;
        return suma;
}
```

```
int suma (int numero1, int numero2)
{
        return numero1 + numero2;
}
```

5. Funciones: parámetros

- Los parámetros que se le pasan a una función pueden ser de dos tipos:
  - **Por valor**: la función recibe una copia de la variable, por lo que cualquier modificación solo afectará a dicha copia, **no se modifica la variable original**
  - **Por referencia**: la función recibe la variable original. En este caso, si realizamos algún cambio en el parámetro de la función, **sí se modifica la variable original**. En este caso se puede pasar un puntero, o bien la referencia de la variable con &

valor

```
void incremento (int n) {
        n++;
}

int main (void) {
        int numero = 0;
        incremento(numero);
        printf("%d", numero);
        return 0;
}
```

referencia puntero

```
void incremento (int* n) {
        *n+=10;
}

int main (void) {
        int numero = 0;
        int *ptr = &numero;
        incremento(ptr);
        printf("%d", numero);
        return 0;
}
```

referencia &

```
void incremento (int* n) {
        *n+=10;
}

int main (void) {
        int numero = 0;
        incremento(&numero);
        printf("%d", numero);
        return 0;
}
```

5. Funciones: parámetros

- En el caso de los **arrays**, existen dos formas:

```
tipo_dato nombre_variable []
```

```
tipo_dato * nombre_variable
```

- Por ejemplo:

```
void imprime(char lista[], int n)
```

```
void imprime(char* lista, int n)
```

- Los **arrays de punteros** también pueden pasarse de dos formas:

```
tipo_dato * nombre_variable []
```

```
tipo_dato ** nombre_variable
```

- Por ejemplo:

```
void imprime(char* lista[], int n)
```

```
void imprime(char** lista, int n)
```

Índice de contenidos

6. Ejercicios

1. Crea una **estructura** que almacene los datos necesarios para identificar un **vehículo**, estos son: **tipo** (coche, moto, camión, autobús), **marca y color**. Crea un array de vehículos, recorre el array y muestra el contenido de cada vehículo
2. Crea un array como en el ejercicio anterior, pero ahora el contenido de cada posición será un **puntero a vehículo**
3. Escribe un programa que solicite introducir una **cadena de caracteres** y te diga si es un palíndromo o no (se lee igual de izquierda a derecha que de derecha a izquierda)
4. Escribe una función con la cabecera **void intercambia(int*, int*)** que intercambie el valor de dos variables numéricas usando punteros
5. Escribe un programa que inicialmente solicite un número, y después solicite tantos números como el valor anteriormente introducido. Estos valores serán almacenados en un array de punteros. Crea una función que calcule la media del array

6. Ejercicios

6. Escribe un programa que inicialmente solicite un número, y después solicite tantos números como el valor anteriormente introducido. Estos valores serán almacenados en un array de punteros. Crea una función que calcule la media del array

7. Escribe una función con la cabecera **void imprime(float*, int)** que muestre los valores almacenados del ejercicio anterior

8. Teniendo en cuenta que los elementos de un array son almacenados en memoria de forma consecutiva, escribe la función con la cabecera **void invertir(float*, int)** que devuelva el array anterior de forma inversa utilizando punteros

Universidad Rey Juan Carlos
Escuela Técnica Superior Ingeniería de Telecomunicación

Programación de Sistemas de Navegación

1.3. Lenguaje C

Listas, Pilas, Colas, Árboles, Recursividad

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

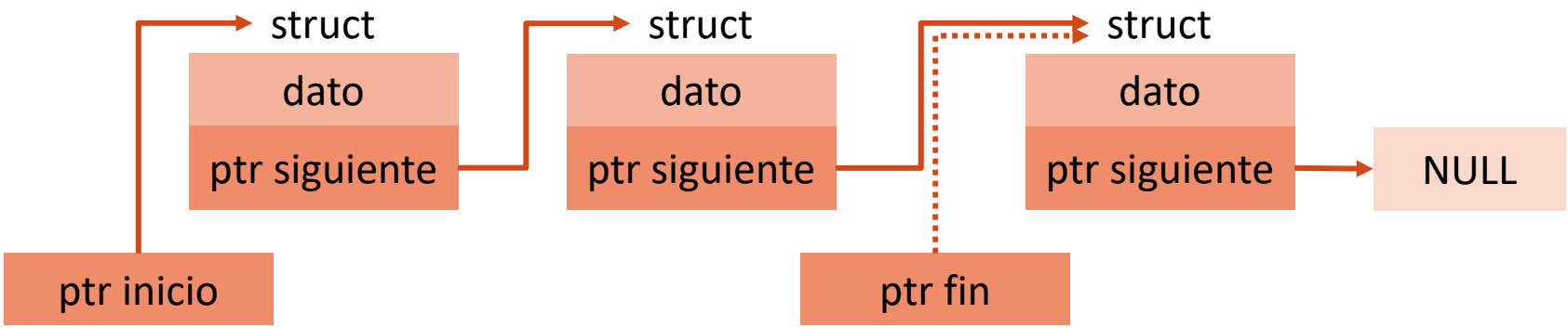
Índice de contenidos

1. Listas enlazadas

- Un **array no es muy apropiado cuando el tamaño del mismo cambia dinámicamente** a lo largo de la ejecución del programa en un rango relativamente amplio. En estos casos la lista (linked list) suele ser la mejor alternativa
- La **lista** es una abstracción de datos que adquiere sentido en aquellos problemas en los que se necesita mantener una colección de datos, pero, o bien **no se sabe a priori el número de elementos a almacenar**, o dicho **número varía muy ampliamente**
- La idea de la lista es mantener los elementos de la colección en una cadena en la que cada elemento está enlazado tiene un puntero hacia el elemento siguiente. Si el elemento no apunta a ningún otro, su puntero tiene un valor nulo

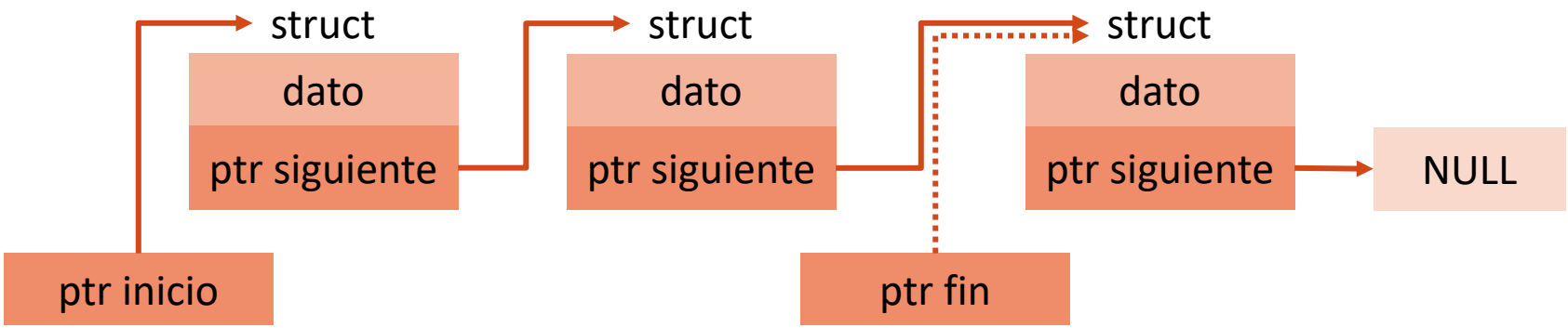
1. Listas enlazadas

- Mientras que en un array los **elementos** están contiguos en la **memoria**, en una lista los elementos están **dispersos**. La asignación de memoria es hecha durante la ejecución
- En una lista los **elementos** son **contiguos** en lo que concierne al **enlazado**. El enlace entre los elementos se hace mediante un puntero. Mientras que en un array el acceso se hace a través de un índice, el acceso en las listas se hace a través de un puntero



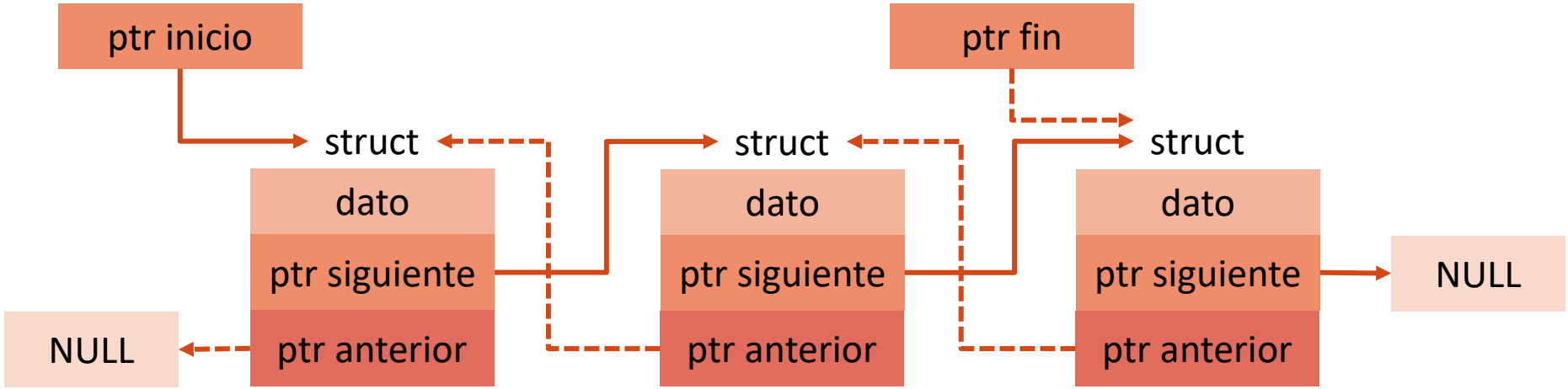
1. Listas enlazadas

- El puntero siguiente del último elemento debe apuntar a NULL (fin de la lista)
- Para acceder a un elemento, la lista es recorrida comenzando por el inicio, el puntero siguiente permite el desplazamiento hacia el próximo elemento
- El desplazamiento se hace en una sola dirección, del primer al último elemento
- Las listas enlazadas pueden ser utilizadas cuando se necesitan hacer varias operaciones de inserción y eliminación de elementos



1. Listas doblemente enlazadas

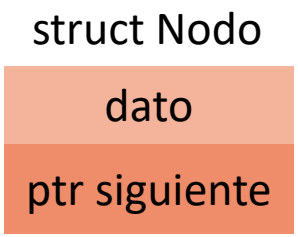
- La principal diferencia, es que contienen un puntero al elemento anterior, de manera que nos permite iterar en las dos direcciones
- El puntero anterior del primer elemento debe apuntar a NULL (inicio de lista)



1. Construir una lista

- Para establecer un elemento de la lista, será utilizado el tipo struct. El elemento de la lista tendrá un campo dato y un puntero siguiente
- El puntero siguiente tiene que ser del mismo tipo que el elemento, si no, no podrá apuntar hacia el elemento. El puntero siguiente permitirá el acceso al próximo elemento

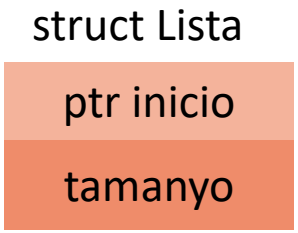
```
typedef struct TNodo {
        tipo_dato dato;
        struct TNodo *siguiente;
} Nodo;
```



1. Construir una lista (opcional)

- Para tener el control de la lista es preferible guardar determinados elementos: el primer elemento, el último (opcional), el número de elementos. Para ello será empleado otra estructura (no es obligatorio, pueden ser utilizadas variables)

```
typedef struct DatosLista {
        Nodo *inicio;
        int tamanyo;
} Lista;
```

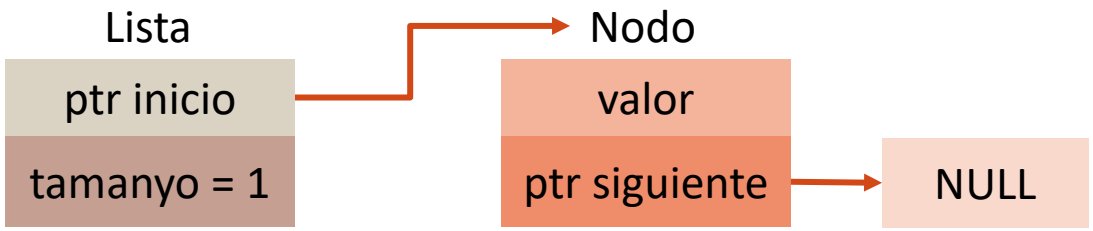


- El puntero inicio tendrá la dirección del primer elemento de la lista. El puntero fin albergará la dirección del último elemento de la lista. La variable tamaño contiene el número de elementos
- Cualquiera sea la posición en la lista, los punteros inicio y fin apuntan siempre al primer y último elemento. El campo tamaño contendrá al numero de elementos de la lista cualquiera sea la operación efectuada sobre la lista

1. Operaciones con listas

- Inicialización de una lista:
  - Esta operación debe ser hecha antes de otra operación sobre la lista
  - Se crea el primer nodo con el valor indicado en su campo y apuntando a NULL
  - La lista comienza con el puntero inicio apuntando al nodo, y el tamaño con valor 1

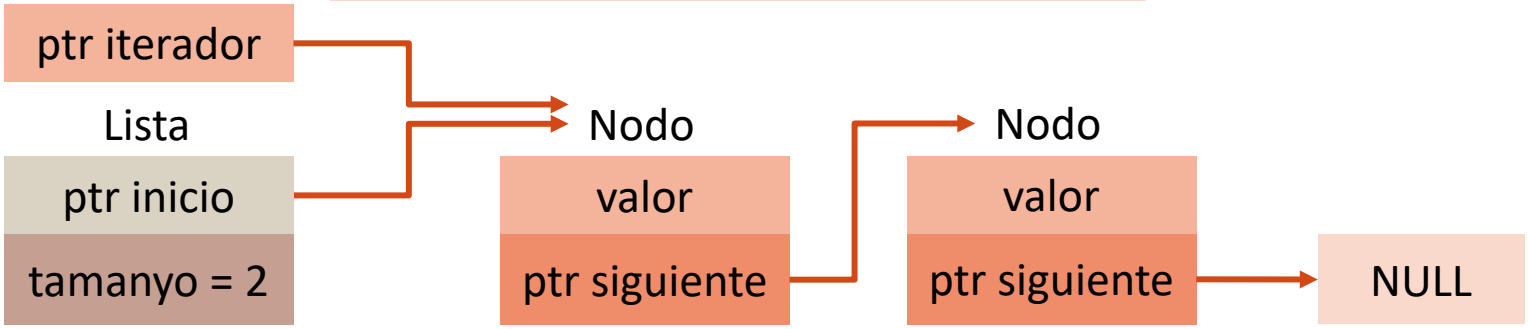
```
void inicializarLista (Lista *lista, tipo_dato valor) {
        Nodo* nodo = (Nodo*)malloc(sizeof(Nodo));
        nodo -> dato = valor;
        nodo -> siguiente = NULL;

        lista -> inicio = nodo;
        lista -> tamanyo = 1;
}
```



1. Operaciones con listas

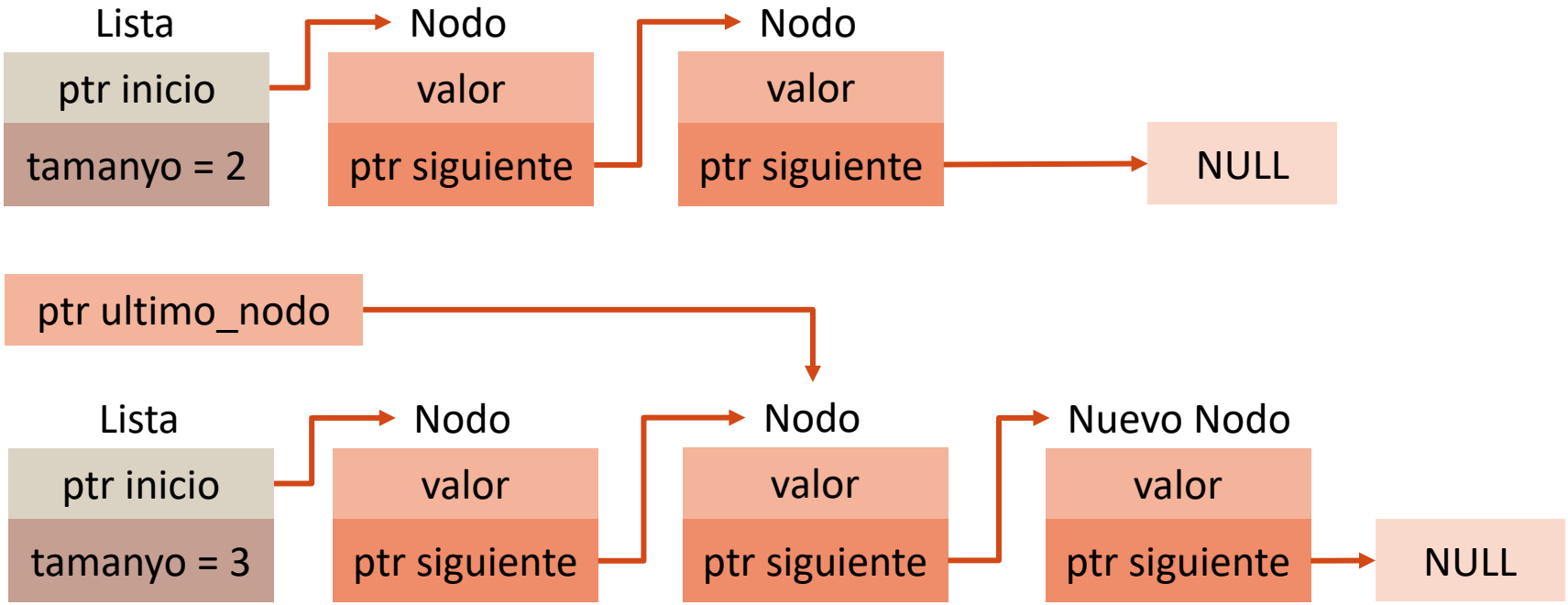
- Obtener el último elemento (si la lista no contiene el puntero fin):
  - Esta operación devuelve un puntero al último nodo de la lista
  - Recorre toda la lista y devuelve el último
  - ¡Cuidado! Si la lista está vacía (iterator == NULL) el acceso fallaría

```
Nodo* ultimo_nodo (Lista *lista) {
        Nodo* iterador = lista -> inicio;
        while (iterador -> siguiente != NULL) {
                iterador = iterador -> siguiente;
        }
        return iterador;
}
```



1. Operaciones con listas

- Insertar un dato en una lista (push):
  - Esta operación inserta el dato al final de la lista y actualiza los punteros de la lista
  - Genera un nuevo nodo con el dato y lo inserta detrás del último nodo de la lista



1. Operaciones con listas

- Insertar un dato en una lista (push):
  - Esta operación inserta el dato al final de la lista y actualiza los punteros de la lista
  - Genera un nuevo nodo con el dato y lo inserta detrás del último nodo de la lista

```
void push (Lista *lista, tipo_dato valor) {
        // Se genera el nuevo nodo
        Nodo* nodo = (Nodo*)malloc(sizeof(Nodo));
        nodo -> dato = valor;
        nodo -> siguiente = NULL;

        // Se obtiene el último nodo
        Nodo* ultimo = ultimo_nodo(lista);
        ultimo -> siguiente = nodo;

        // Se incrementa en 1 el número de elementos
        lista -> tamanyo += 1;
}
```

1. Operaciones con listas

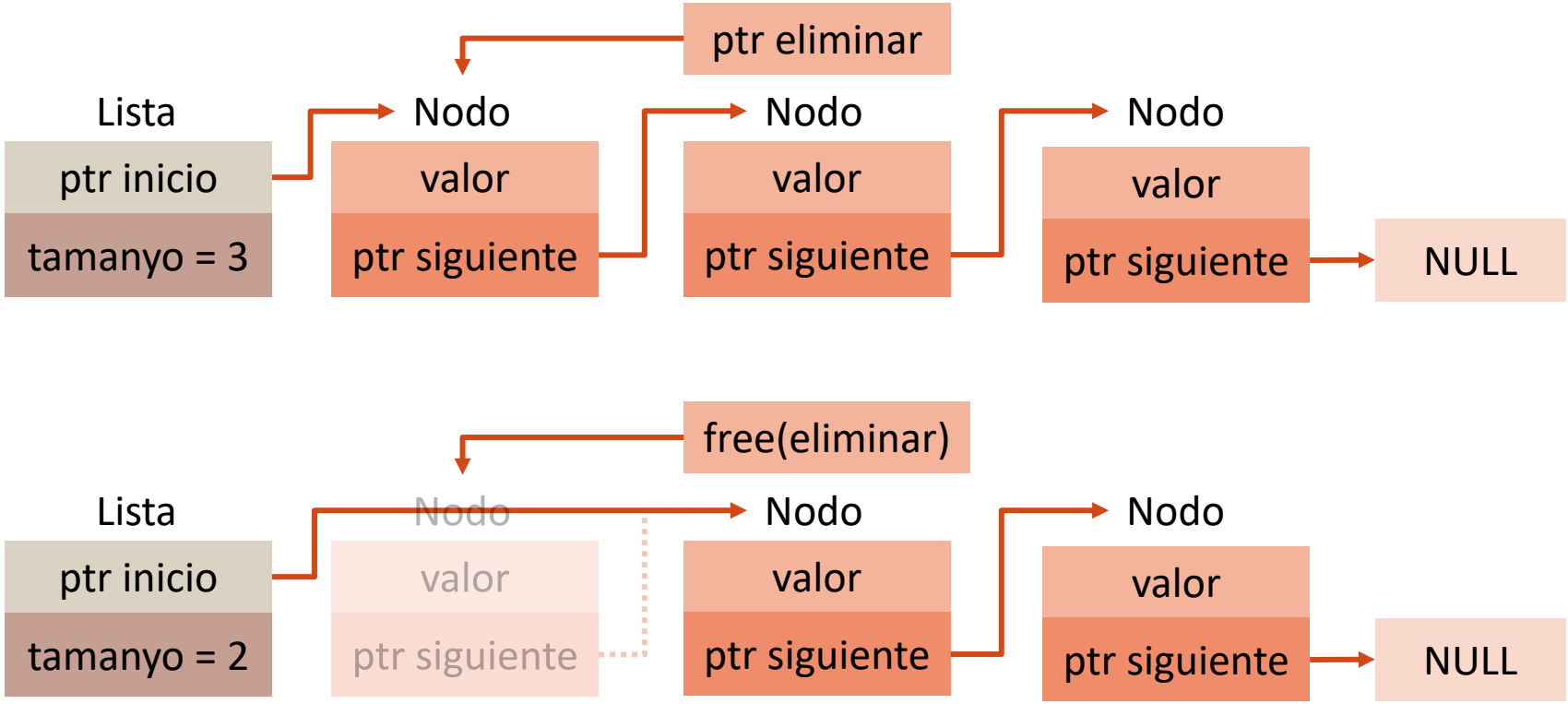
- Eliminar un dato de una lista en una posición:

```
void eliminar (Lista *lista, int posicion) ;
```

  - Esta operación elimina el nodo de la posición y **actualiza los punteros** de la lista
  - **Liberar la memoria (free)** del nodo a eliminar que está en dicha posición
  - Hay que diferenciar si el nodo es el primero o no

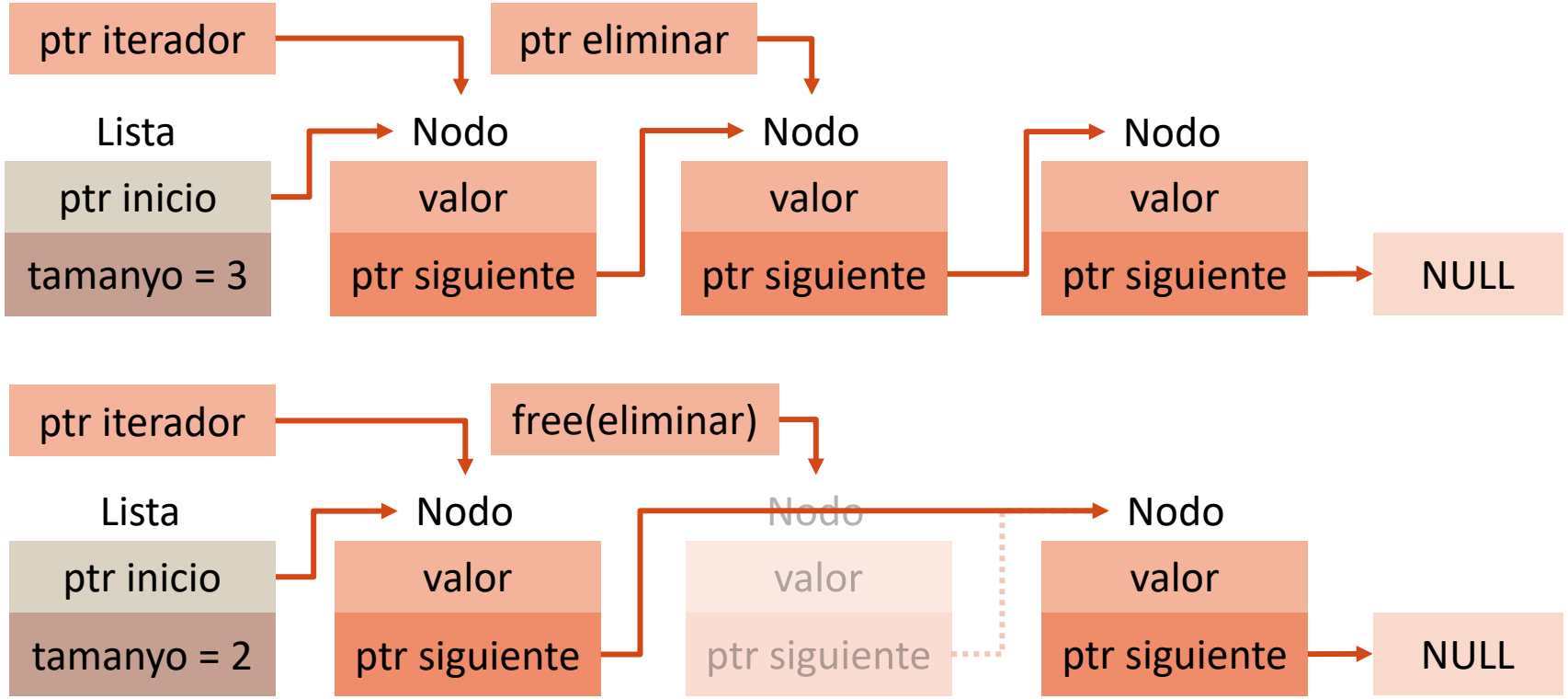
1. Operaciones con listas

- Si es el primer elemento:



1. Operaciones con listas

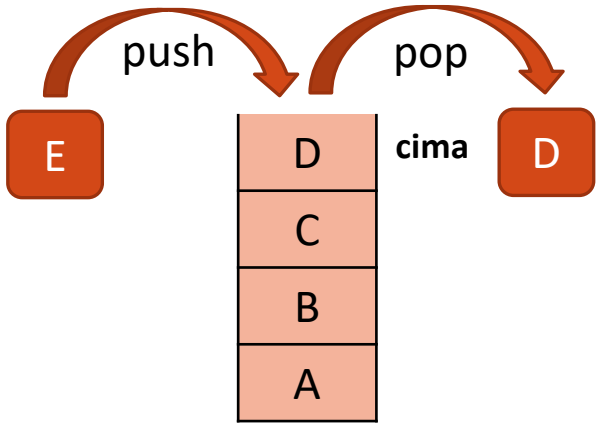
- Si no es el primer elemento (ejemplo posición 2):



Índice de contenidos

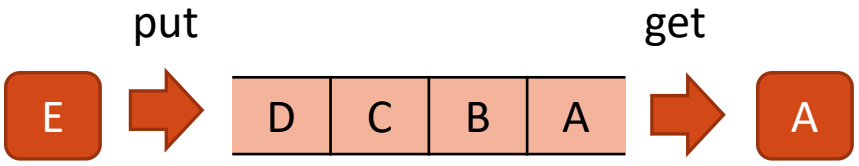
2. Pilas

- La **pila** es una estructura de datos que permite almacenar datos en el orden LIFO (Last In First Out, "último en entrar, primero en salir")
- La recuperación de los datos es realizada en el orden inverso de su inserción
- Las operaciones que pueden hacerse son:
  - **push**: inserta un elemento al final de la pila (en la cima)
  - **pop**: elimina el último elemento insertado de la pila (cima)
  - **empty**: devuelve si la pila está vacía o no
  - **full**: devuelve si la pila está llena
  - **print**: muestra los valores de la pila



2. Colas

- La **pila** es una estructura de datos que permite almacenar datos en el orden FIFO (First In First Out, “primero en entrar, primero en salir")
- La recuperación de los datos es realizada en el mismo orden que el de su inserción
- Las operaciones que pueden hacerse son:
  - **put**: inserta un elemento al final de la cola
  - **get**: elimina el primer elemento insertado de la cola
  - **empty**: devuelve si la cola está vacía o no
  - **full**: devuelve si la cola está llena
  - **print**: muestra los valores de la cola

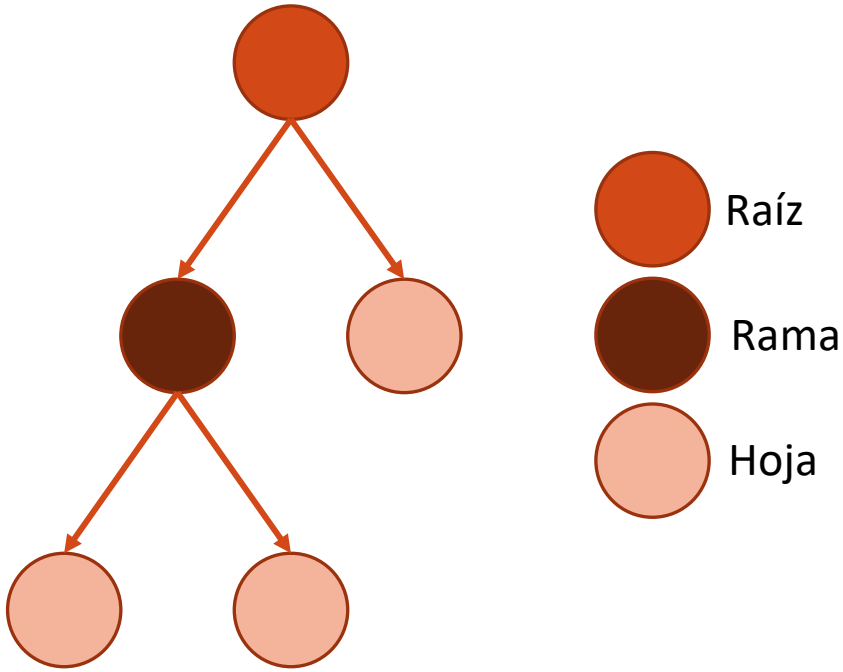


Índice de contenidos

3. Árboles

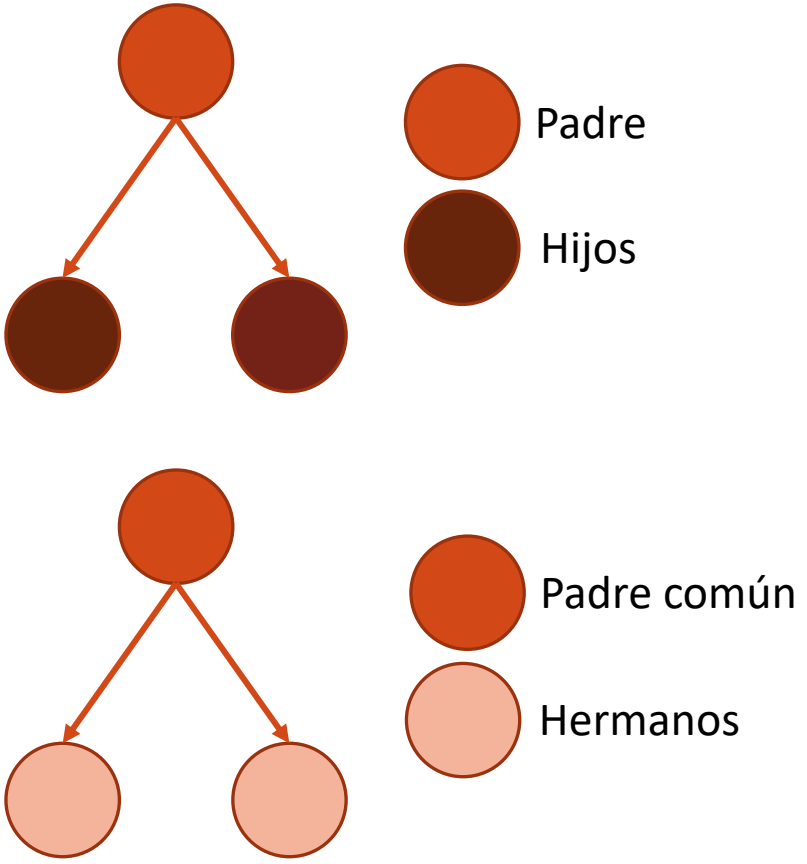
- Los árboles son estructuras que almacenan sus nodos en forma jerárquica y no en forma lineal

- Tipos de nodos:
  - **Raíz**: es el primer nodo de un Árbol. Solo puede haber uno
  - **Rama**: nodos que no son la raíz y que además tiene al menos un hijo
  - **Hoja**: nodos que no tienen hijos, siempre se encuentran en los extremos de la estructura



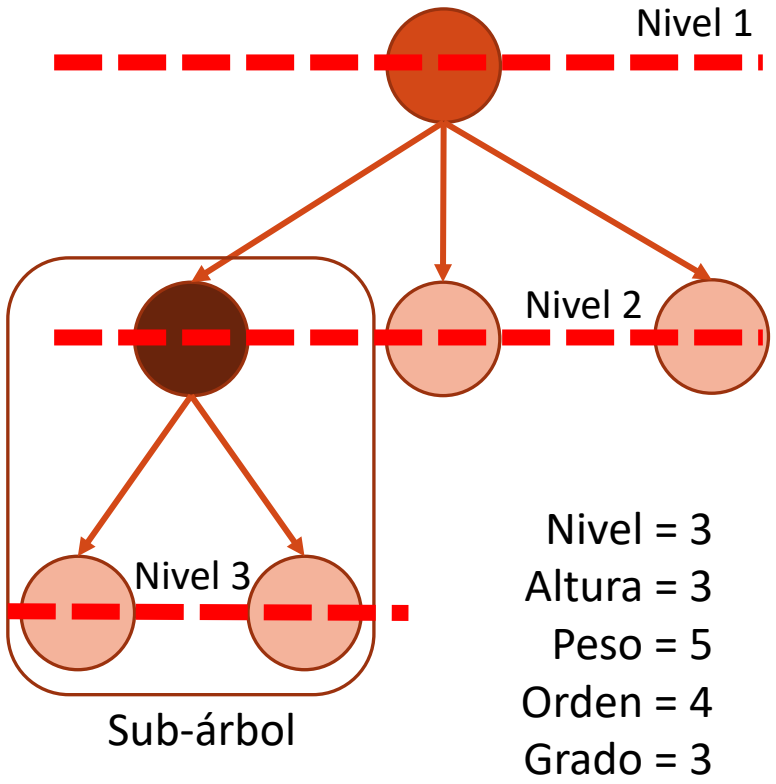
3. Árboles

- Más tipos de nodos:
  - **Padre**: todos aquellos nodos que tiene al menos un hijo
  - **Hijo**: todos aquellos nodos que tiene un padre
  - **Hermano**: aquellos nodos que comparten a un mismo padre en común dentro de la estructura



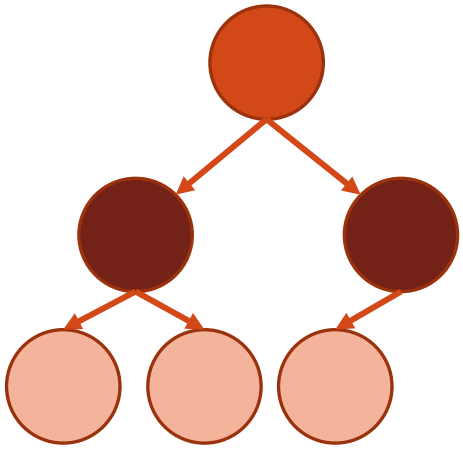
3. Árboles

- Definiciones:
  - **Nivel**: Cada generación dentro del árbol. El nivel 0 es un árbol vacío
  - **Altura**: número máximo de niveles de un árbol
  - **Peso**: número de nodos que tiene un árbol
  - **Orden**: número máximo de hijos que puede tener un nodo
  - **Grado**: número mayor de hijos que tiene alguno de los nodos del árbol y esta limitado por el Orden
  - **Sub-árbol**: todo árbol generado a partir de una sección determinada del árbol

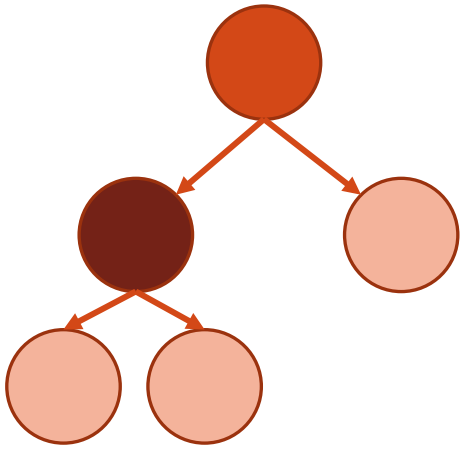


3. Árboles n-arios

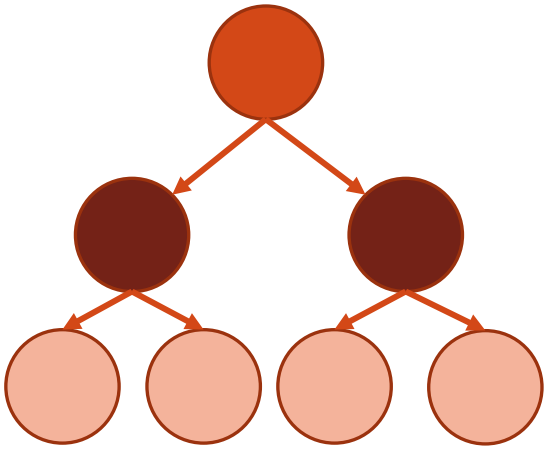
- **Árboles n-arios**: aquellos arboles donde el número máximo de hijos por nodo es de N
- **Árboles binarios**: son árboles n-arios de grado 2
  - **Árbol binario lleno**: aquel donde todos los nodos tienen cero o 2 hijos, con excepción de la Raíz
  - **Árbol binario perfecto**: es árbol lleno donde todas las hojas están al mismo nivel



binario no lleno



binario lleno y no perfecto

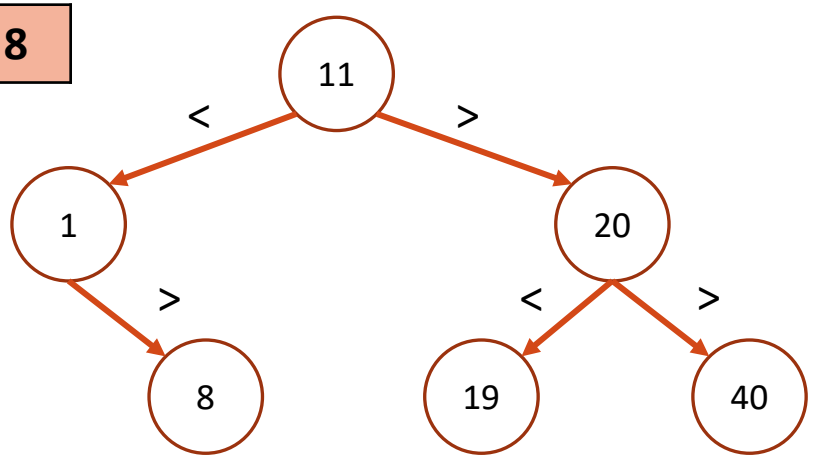


binario perfecto

3. Árboles binarios ordenados

- Para cada nodo de un árbol binario de búsqueda debe cumplirse la propiedad:
  - Las claves de los nodos del subárbol **izquierdo** deben ser **menores** que la clave de la raíz
  - Las claves de los nodos del subárbol **derecho** deben ser **mayores** que la clave de la raíz

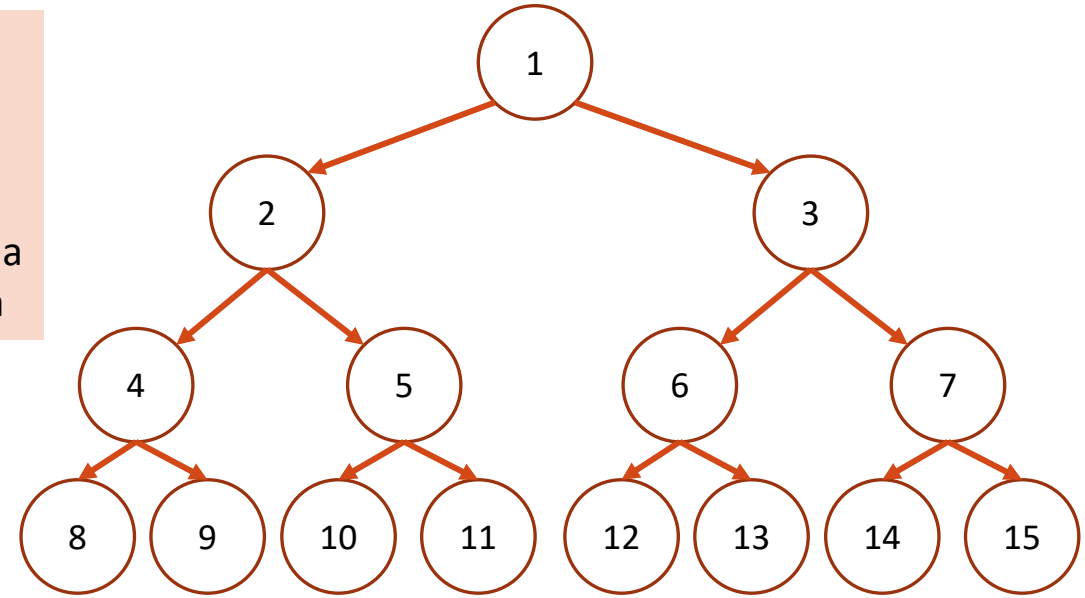
11	20	19	40	1	1	8



3. Árboles: recorridos

- En **amplitud**:
  - Se recorre primero la raíz, luego se recorren los demás nodos ordenados por el nivel al que pertenecen en orden de izquierda a derecha
  - Es necesario el uso de una **cola**

- Introduzco la raíz a la cola
- Mientras la cola no esté vacía:
  - Saco elemento de la cola
  - Evalúo nodo sacado
  - Introduzco en la cola hijo izquierda
  - Introduzco en la cola hijo derecha

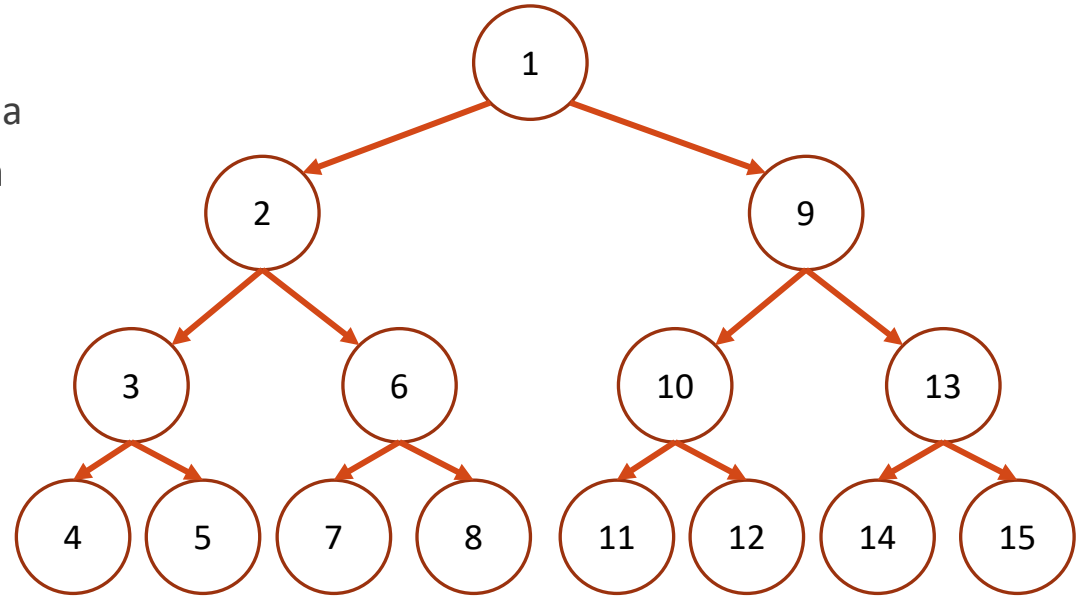


3. Árboles: recorridos

- En **profundidad**:
  - Se recorre primero la raíz, luego se recorren los demás nodos de forma **recurrente**, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa, de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado
  - Existen varios recorridos según el orden de análisis:
    - Pre-Orden (actual, izquierda, derecha)
    - Post-Orden (izquierda, derecha, actual)
    - In-Orden (izquierda, actual, derecha)

3. Árboles: recorridos

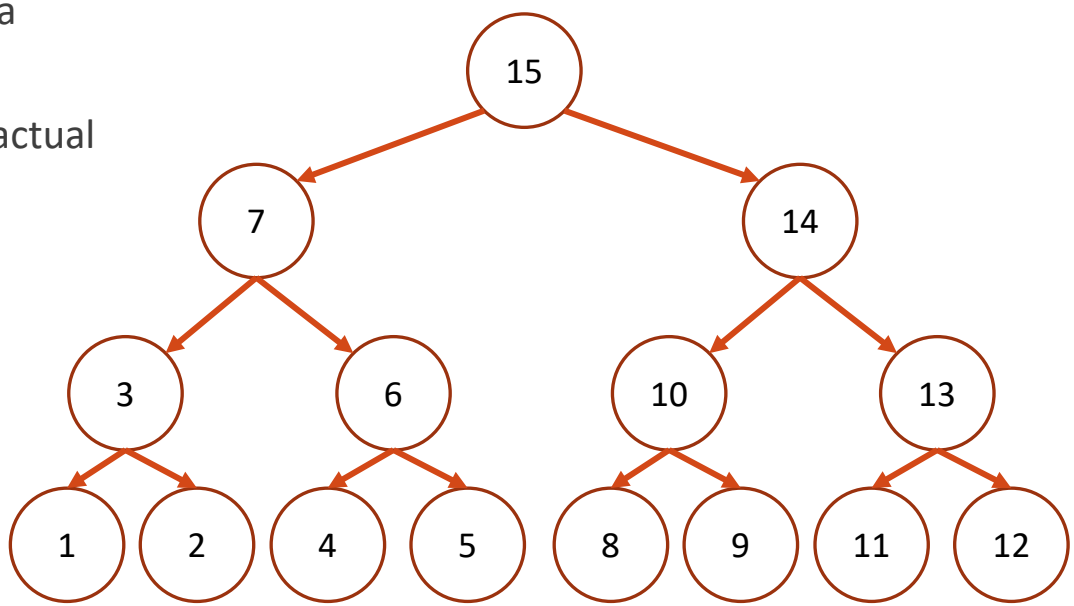
- En profundidad → Pre-Orden:
  - El recorrido inicia en la Raíz y luego se recorre en pre-orden cada unos de los sub-árboles de izquierda a derecha
  - Evalúo nodo actual
  - Evalúo hijo subárbol izquierda
  - Evalúo hijo subárbol derecha



3. Árboles: recorridos

- En profundidad → Post-Orden:
  - Se recorre en post-orden cada uno de los sub-árboles y al final se recorre la raíz

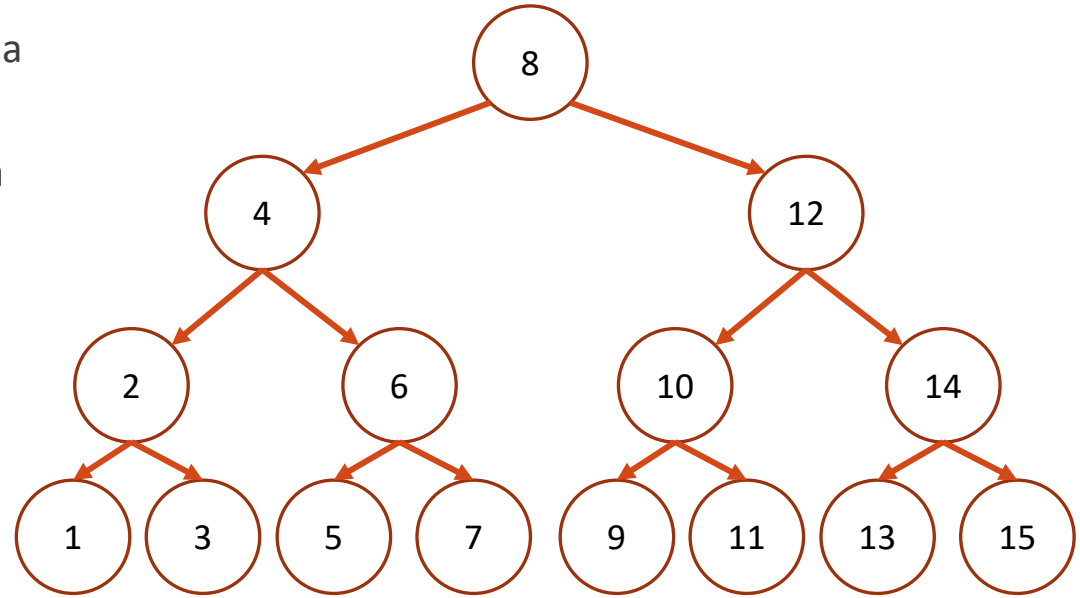
  - Evalúo hijo subárbol izquierda
  - Evalúo hijo subárbol derecha
  - Si no hay hijos, evalúo nodo actual



3. Árboles: recorridos

- En profundidad → In-Orden:
  - Se recorre en in-orden el primer sub-árbol, luego se recorre la raíz y al final se recorre en in-orden los demás sub-árboles

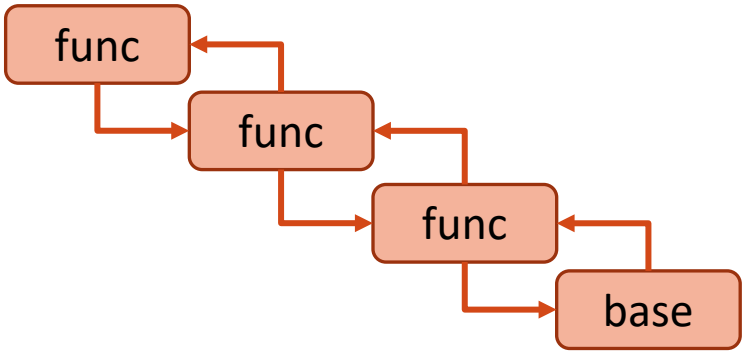
  - Evalúo hijo subárbol izquierda
  - Evalúo nodo actual
  - Evalúo hijo subárbol derecha



Índice de contenidos

4. Funciones recursivas

- Las funciones recursivas son aquellas que se llaman así mismas. Estas funciones están divididas en dos partes:
  - **Caso base**: condición de parada. Es aquel que para su solución, no se requiere de la función que está definiendo
  - **Llamada recurrente**: sí que requieren utilizar la función que se está definiendo



- Es muy **importante** que siempre exista un **caso base** que al menos se cumpla en alguna ocasión. De otro modo, la llamada a una función de este tipo genera un **bucle infinito** de llamadas así misma

4. Funciones recursivas: ejemplos

- Para calcular el factorial de un número podemos hacerlo de forma iterativa, o recursiva:

```
void factorial (int num) {
        int fact = 1;
        for (int i=num; i > 0; i--) fact = fact * i;
}
```

```
int factorial (int num) {
        if (num == 0) return 1;
        return num*factorial(num-1);
}
```

- Suponiendo que tenemos un árbol cuyos nodos se componen de un valor entero (dato) y dos punteros a nodos (hijoizquierdo, hijoderecho). El siguiente ejemplo de función recursiva recorre el árbol en pre-orden y muestra el valor de cada nodo

```
void preorden (Nodo* nodo) {
        // Condición de parada
        if (nodo == NULL) return;
        // Recorrido en pre-orden
        printf("Valor nodo: %d\n", nodo -> dato);
        preorden(nodo -> hijoizquierdo);
        preorden(nodo -> hijoderecho);
}
```

Índice de contenidos

5. Ejercicios

1. Genera la estructura de una lista cuyos nodos tienen que almacenar un número entero, para ello, crea las estructuras **Lista** y **Nodo** vistas en clase
2. Implementa la función **void inicializarLista(Lista* lista, int dato)** que inicialice la lista con un nodo que contenga el valor introducido
3. Implementa la función **void push(Lista* lista, int dato)** que inserte un nodo al final de la lista que contenga el valor introducido
4. La sucesión de Fibonacci es: 0,1,1,2,3,5,8,13,21,... Sus dos primeros términos son 0 y 1, y los restantes se obtienen sumando los dos anteriores. Crea una función recursiva **fibonacci(int n)** que dado un número n, el valor del n-ésimo término, muestre la sucesión hasta dicha posición (ej: la llamada a fibonacci(8) debe mostrar [13])
5. Implementa una función recursiva **length(Nodo* nodo)** que recibe el puntero al primer elemento de una lista y devuelva su longitud

Universidad Rey Juan Carlos
Escuela Técnica Superior Ingeniería de Telecomunicación

Programación de Sistemas de Navegación

1.4. Lenguaje C

Ordenación, Búsqueda

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

Índice de contenidos

1. Algoritmos de ordenación

- Los **algoritmos de ordenación** permiten organizar de forma creciente o decreciente una colección de elementos, bien sean de una lista o un array
- La principal diferencia es la eficiencia, es decir, cómo se comportan al ordenar una gran entrada de datos
  - Los lentos se comportan en un orden cuadrático, es decir, $O(n^2)$
  - Los algoritmos rápidos se comportan, en un caso promedio, en un orden logarítmico O(n log n)

1. Algoritmos de ordenación

- **Método de la burbuja o intercambio directo:**
  - Se compara cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Se necesita revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada
  - Se trata de un algoritmo sencillo de implementar pero muy lento con listas grandes

```
for (int i=1;i<numel; i++) {
        for (int j=0; j < numel-i ; j++) {
                if (array[j] > array[j+1]) {
                        aux = array[j];
                        array[j] = array[j+1];
                        array[j+1] = aux;
                }
        }
}
```

20	**17**	**62**	**33**	**43**	**23**
17	**20**	33	43	23	**62**
17	20	33	23	**43**	62
17	20	23	**33**	43	62
…	**…**	**…**	**…**	**…**	**…**
17	**20**	**23**	**33**	**43**	**62**

1. Algoritmos de ordenación

- **Quicksort:**
  - Basado en la técnica de divide y vencerás
  - Algoritmo eficiente y rápido
- Algoritmo:
  1. Se inicia eligiendo el primer elemento al que llamaremos pivote
  2. Se utilizan dos índices 'izquierda' (2nd elemento) y 'derecha' (último elemento)
  3. El punto en que se cruzan los índices es la posición adecuada para colocar el pivote (a un lado los elementos son todos menores y al otro son todos mayores). Para ello, mientras los índices no se crucen:
     - Si elemento(izquierda) > elemento(pivote) > elemento(derecha):
       - Intercambiar elementos, avanzar izquierda y retroceder derecha
     - Sino:
       - Si elemento(pivote) >= elemento(izquierda): avanzar izquierda
       - Si elemento(pivote) <= elemento(derecha): retroceder derecha
  4. Intercambiar elemento(pivote) con elemento (derecha)
  5. Repetir el punto 3 para cada conjunto de elementos a cada lado del pivote

1. Algoritmos de ordenación

- **Quicksort:**

pivote izq der

15	67	8	27	35

Mientras izq <= der:
    Si elem(izq) > elem(piv) > elem(der):
        intercambiar elem(izq) y elem(der)
        izq++ y der--
    Sino:
        si elem(pivote) >= elem(izq):
            izq++
        si elem(pivote) <= elem(der):
            der--
Intercambiar elem(pivote) y elem(der)

¿izq <= der? Si:
    ¿ (67 > 15) && (15 > 35) ? No
    ¿ (15 >= 67) ? No
    ¿ (15 <= 35) ? Si: derecha--

1. Algoritmos de ordenación

- **Quicksort:**

pivote izq der

15	67	8	27	35

pivote izq der

15	67	8	27	35

Mientras izq <= der:
    Si elem(izq) > elem(piv) > elem(der):
        intercambiar elem(izq) y elem(der)
        izq++ y der--
    Sino:
        si elem(pivote) >= elem(izq):
            izq++
        si elem(pivote) <= elem(der):
            der--
Intercambiar elem(pivote) y elem(der)

¿izq <= der? Si:
    ¿ (67 > 15) && (15 > 35) ? No
    ¿ (15 >= 67) ? No
    ¿ (15 <= 35) ? Si: derecha--

1. Algoritmos de ordenación

- **Quicksort:**

pivote izq der

15	67	8	27	35

pivote izq der

15	67	8	27	35

pivote izq der

15	67	8	27	35

```
Mientras izq <= der:
    Si elem(izq) > elem(piv) > elem(der):
        intercambiar elem(izq) y elem(der)
        izq++ y der--
    Sino:
        si elem(pivote) >= elem(izq):
            izq++
        si elem(pivote) <= elem(der):
            der--
Intercambiar elem(pivote) y elem(der)
```

```
¿izq <= der? Si:
    ¿ (67 > 15) && (15 > 27) ? No
    ¿ (15 >= 67) ? No
    ¿ (15 <= 27) ? Si: derecha--
```

1. Algoritmos de ordenación

- **Quicksort:**

pivote	izq			der
15	**67**	**8**	**27**	**35**

pivote	izq		der	
15	**67**	**8**	**27**	**35**

pivote	izq	der		
15	**67**	**8**	**27**	**35**

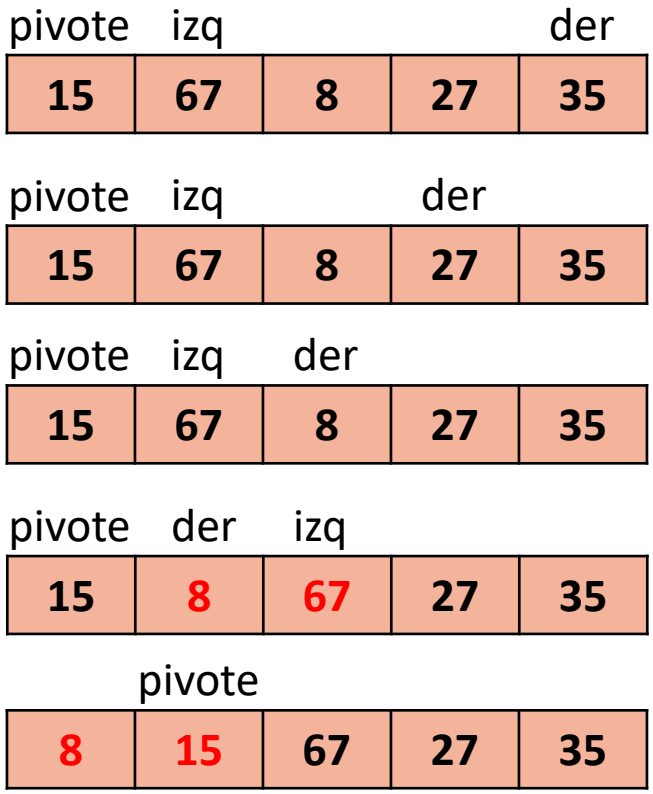
pivote	der	izq		
15	**8**	**67**	**27**	**35**

```
Mientras izq <= der:
        Si elem(izq) > elem(piv) > elem(der):
                intercambiar elem(izq) y elem(der)
                izq++ y der--
        Sino:
                si elem(pivote) >= elem(izq):
                        izq++
                si elem(pivote) <= elem(der):
                        der--
Intercambiar elem(pivote) y elem(der)
```

```
¿izq <= der? Si:
        ¿ (67 > 15) && (15 > 8) ? Si:
                intercambiar
                izquierda++ y derecha--
```

1. Algoritmos de ordenación

- **Quicksort:**

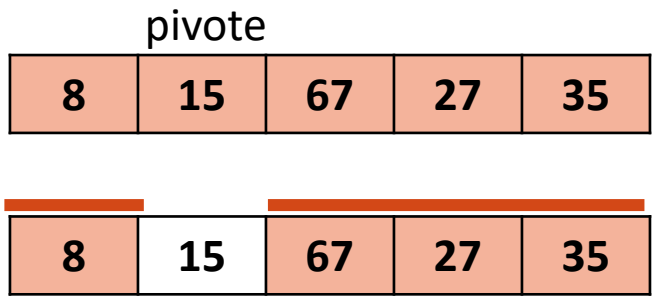


Mientras izq <= der:
    Si elem(izq) > elem(piv) > elem(der):
        intercambiar elem(izq) y elem(der)
        izq++ y der--
    Sino:
        si elem(pivote) >= elem(izq):
            izq++
        si elem(pivote) <= elem(der):
            der--
Intercambiar elem(pivote) y elem(der)

¿izq <= der? No
    intercambiar elem(piv) y elem(der)
Procesar lado izquierdo pivote
Procesar lado derecho pivote

1. Algoritmos de ordenación

- **Quicksort**:

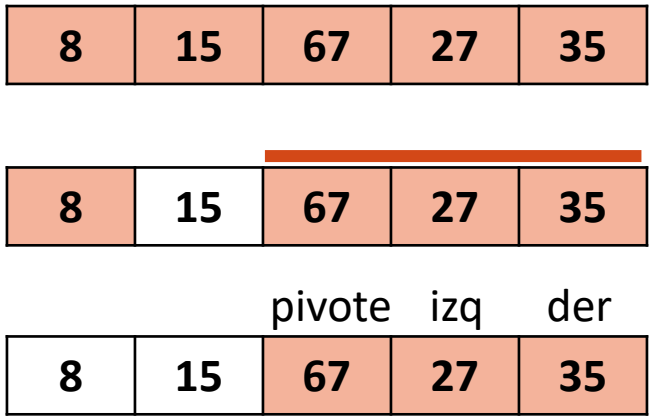


Mientras izq <= der:
    Si elem(izq) > elem(piv) > elem(der):
        intercambiar elem(izq) y elem(der)
        izq++ y der--
    Sino:
        si elem(pivote) >= elem(izq):
            izq++
        si elem(pivote) <= elem(der):
            der--
Intercambiar elem(pivote) y elem(der)

Procesar lado izquierdo pivote:
    solo 1 elem, ordenado
Procesar lado derecho pivote

1. Algoritmos de ordenación

- **Quicksort:**

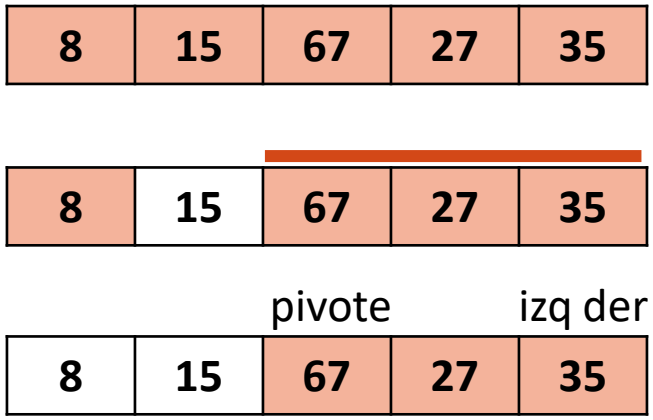


Mientras izq <= der:
    Si elem(izq) > elem(piv) > elem(der):
        intercambiar elem(izq) y elem(der)
        izq++ y der--
    Sino:
        si elem(pivote) >= elem(izq):
            izq++
        si elem(pivote) <= elem(der):
            der--
Intercambiar elem(pivote) y elem(der)

Procesar lado derecho pivote

1. Algoritmos de ordenación

- **Quicksort:**

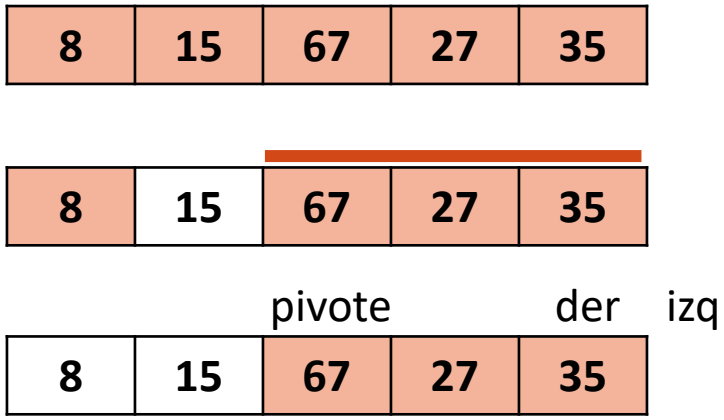


Mientras izq <= der:
    Si elem(izq) > elem(piv) > elem(der):
        intercambiar elem(izq) y elem(der)
        izq++ y der--
    Sino:
        si elem(pivote) >= elem(izq):
            izq++
        si elem(pivote) <= elem(der):
            der--
Intercambiar elem(pivote) y elem(der)

¿izq <= der? Si:
    ¿ (27 > 67) && (67 > 35) ? No:
    ¿ (67 >= 27) ? Si: izquierda++
    ¿ (67 <= 35) ? No

1. Algoritmos de ordenación

- **Quicksort:**

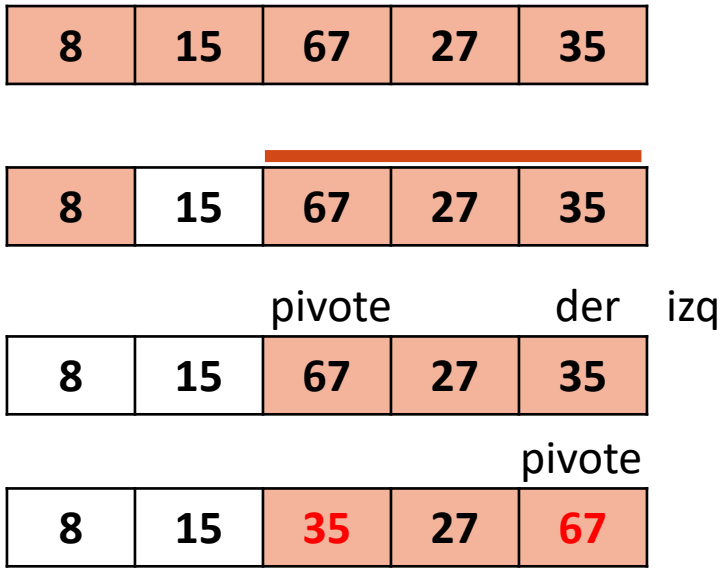


Mientras izq <= der:
    Si elem(izq) > elem(piv) > elem(der):
        intercambiar elem(izq) y elem(der)
        izq++ y der--
    Sino:
        si elem(pivote) >= elem(izq):
            izq++
        si elem(pivote) <= elem(der):
            der--
Intercambiar elem(pivote) y elem(der)

¿izq <= der? Si:
    ¿ (35 > 67) && (67 > 35) ? No:
    ¿ (67 >= 35) ? Si: izquierda++
    ¿ (67 <= 35) ? No

1. Algoritmos de ordenación

- **Quicksort:**

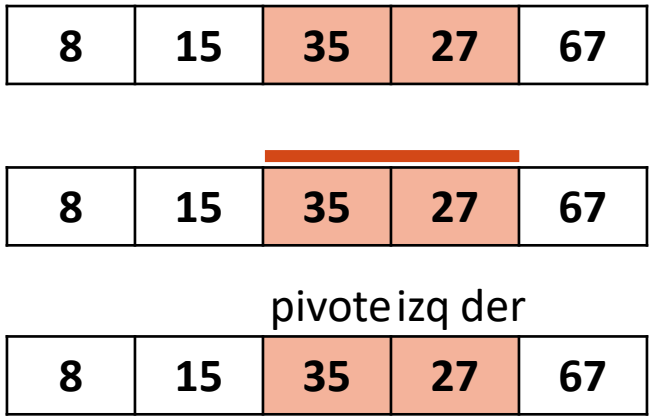



```
Mientras izq <= der:
    Si elem(izq) > elem(piv) > elem(der):
        intercambiar elem(izq) y elem(der)
        izq++ y der--
    Sino:
        si elem(pivote) >= elem(izq):
            izq++
        si elem(pivote) <= elem(der):
            der--
Intercambiar elem(pivote) y elem(der)
```

```
¿izq <= der? No:
    intercambiar elem(piv) y elem(der)
Procesar lado izquierdo pivote
Procesar lado derecho pivote
```

1. Algoritmos de ordenación

- **Quicksort:**



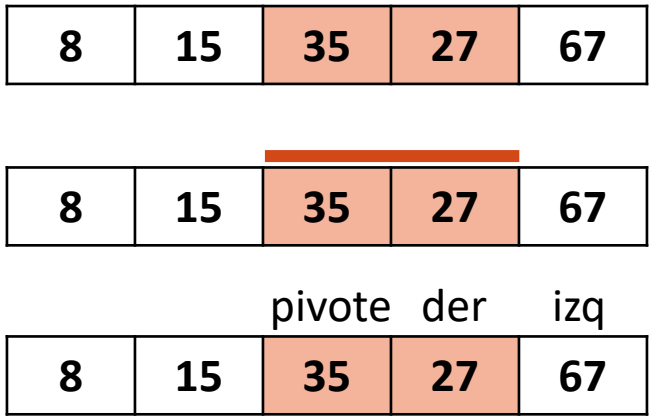

```
Mientras izq <= der:
        Si elem(izq) > elem(piv) > elem(der):
                intercambiar elem(izq) y elem(der)
                izq++ y der--
        Sino:
                si elem(pivote) >= elem(izq):
                        izq++
                si elem(pivote) <= elem(der):
                        der--
Intercambiar elem(pivote) y elem(der)
```

Procesar lado izquierdo pivote

Procesar lado derecho pivote: no hay

1. Algoritmos de ordenación

- **Quicksort:**

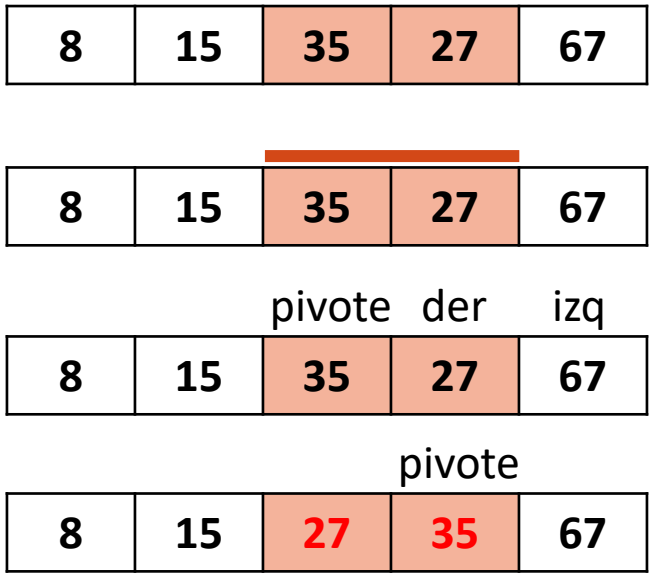


Mientras izq <= der:
    Si elem(izq) > elem(piv) > elem(der):
        intercambiar elem(izq) y elem(der)
        izq++ y der--
    Sino:
        si elem(pivote) >= elem(izq):
            izq++
        si elem(pivote) <= elem(der):
            der--
Intercambiar elem(pivote) y elem(der)

¿izq <= der? Si:
    ¿ (27 > 35) && (35 > 27) ? No:
    ¿ (35 >= 27) ? Si: izquierda++
    ¿ (35 <= 27) ? No

1. Algoritmos de ordenación

- **Quicksort:**

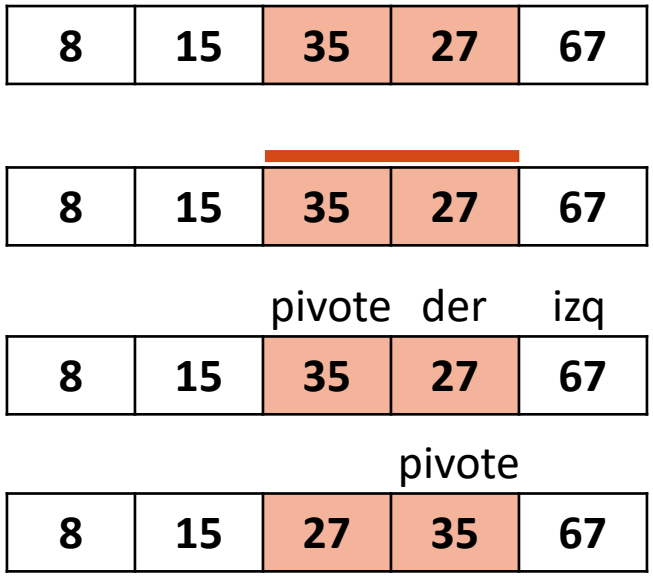


Mientras izq <= der:
    Si elem(izq) > elem(piv) > elem(der):
        intercambiar elem(izq) y elem(der)
        izq++ y der--
    Sino:
        si elem(pivote) >= elem(izq):
            izq++
        si elem(pivote) <= elem(der):
            der--
Intercambiar elem(pivote) y elem(der)

¿izq <= der? No:
    intercambiar elem(piv) y elem(der)
Procesar lado izquierdo pivote
Procesar lado derecho pivote

1. Algoritmos de ordenación

- **Quicksort:**



Mientras izq <= der:
    Si elem(izq) > elem(piv) > elem(der):
        intercambiar elem(izq) y elem(der)
        izq++ y der--
    Sino:
        si elem(pivote) >= elem(izq):
            izq++
        si elem(pivote) <= elem(der):
            der--
Intercambiar elem(pivote) y elem(der)

Procesar lado izquierdo pivote:
solo 1 elem, ordenado
Procesar lado derecho pivote: no hay

1. Algoritmos de ordenación

- **Quicksort:**

8	15	27	35	67

```
Mientras izq <= der:
    Si elem(izq) > elem(piv) > elem(der):
        intercambiar elem(izq) y elem(der)
        izq++ y der--
    Sino:
        si elem(pivote) >= elem(izq):
            izq++
        si elem(pivote) <= elem(der):
            der--
Intercambiar elem(pivote) y elem(der)
```

Finalizan las llamadas recursivas:
elementos ordenados

1. Algoritmos de ordenación

- **Quicksort:**

```
void quicksort(int elem[], int izquierda, int derecha) {
        if (izquierda >= derecha) return;
        int pivote = izquierda, izq = izquierda+1, der = derecha, aux;
        while (izq <= der){
                if (elem[izq] > elem[pivote] && elem[pivote] > elem[der]) {
                        aux = elem[izq]; elem[izq] = elem[der];elem[der] = aux;
                        izq++;                  der--;
                } else {
                        if (elem[pivote] >= elem[izq]) izq++;
                        if (elem[pivote] <= elem[der]) der--;
                }
        }
        aux = elem[pivote];     elem[pivote] = elem[der];    elem[der] = aux;
        pivote = der;
        quicksort(elem, izquierda, pivote-1);
        quicksort(elem, pivote+1, derecha);
}
```

Índice de contenidos

2. Algoritmos de búsqueda

- Permiten buscar elementos dentro de una estructura de datos, bien una lista o array
- Dan solución al problema de si existe o no un elemento determinado en un conjunto finito de elementos, es decir, si el elemento en cuestión pertenece o no a dicho conjunto
- Además, son capaces de indicarnos su localización dentro del conjunto
- Según el tipo, pueden agruparse en:
  - Algoritmos de búsqueda secuencial
  - Algoritmos de búsqueda binaria o dicotómica

2. Algoritmos de búsqueda

- Búsqueda secuencial:
  - Elementos que no están ordenados y no pueden ser ordenados previamente
  - Para ello se recorre todos los elementos de forma secuencial hasta el final
- Búsqueda binaria o dicotómica:
  - Se parte de elementos que están previamente ordenados
  - Reduce el tiempo de búsqueda exponencialmente
  - Algoritmo:
    1. Se compara el elemento central con el deseado
    2. Si es el elemento paro de buscar, sino:
       - Si el elemento es menor que el central, se toma el vector que va del inicio hasta el valor anterior
       - Si el elemento es mayor que el central, se toma el vector que va desde el siguiente hasta el final
    3. Se repite el paso 1 hasta que se encuentra el valor o no queda un intervalo divisible

2. Algoritmos de búsqueda

- Búsqueda binaria o dicotómica:

8	15	27	35	67

Si valor = elem(central): encontrado
Sino:
    si num elem > 1:
        si valor > elem(central)
            analizar elem(central+1) - elem(final)
        si valor < elem(central)
            analizar elem(inicio) - elem(central-1)
    sino:
        no existe

Buscar 67

¿67 == 27?: no
num elem > 1: Si
    ¿67 > elem(central)?: si
        analizar elem(3)-elem(4)

2. Algoritmos de búsqueda

- Búsqueda binaria o dicotómica:

8	15	27	35	67

Si valor = elem(central): encontrado
Sino:
    si num elem > 1:
        si valor > elem(central)
            analizar elem(central+1) - elem(final)
        si valor < elem(central)
            analizar elem(inicio) - elem(central-1)
    sino:
        no existe

Buscar 67

¿67 == 35?: no
num elem > 1: Si
    ¿67 > elem(central)?: si
        analizar elem(4)-elem(4)

2. Algoritmos de búsqueda

- Búsqueda binaria o dicotómica:

8	15	27	35	67

Buscar 67

¿67 == 67?: si

encontrado

Si valor = elem(central): encontrado
Sino:
si num elem > 1:
si valor > elem(central)
analizar elem(central+1) - elem(final)
si valor < elem(central)
analizar elem(inicio) - elem(central-1)
sino:
no existe

2. Algoritmos de búsqueda

- Búsqueda binaria o dicotómica:

8	15	27	35	67

Si valor = elem(central): encontrado
Sino:
    si num elem > 1:
        si valor > elem(central)
            analizar elem(central+1) - elem(final)
        si valor < elem(central)
            analizar elem(inicio) - elem(central-1)
    sino:
        no existe

¿Y si fuera buscar 68?

¿68 == 67?: no
num elem > 1: no
    no existe

2. Algoritmos de búsqueda

- Búsqueda binaria o dicotómica:

```
int busquedaBinaria(int  elementos[], int nelem, int dato) {
        int centro,inicio=0,final= nelem-1;
        while (inicio<=final) {
                centro=((final - inicio)/2) + inicio; // Parte entera
                if(elementos[centro]==dato)
                        return centro;
                else if(dato < elementos[centro])
                        final = centro-1;
                else inicio = centro+1;
        }
        return -1;
}
```

Índice de contenidos

3. Ejercicios

1. Escribe un programa que inicialmente solicite un número, y después solicite tantos números como el valor anteriormente introducido. Estos valores serán almacenados en un vector que posteriormente ha de ser ordenado según el método de la burbuja
2. ¿Sería posible mejorar el método de ordenación de la burbuja? ¿Cómo?
3. Implementa la búsqueda binaria de forma recursiva

Universidad Rey Juan Carlos
Escuela Técnica Superior Ingeniería de Telecomunicación

Programación de Sistemas de Navegación

1.5. Lenguaje C

Argumentos, Depuración, Archivos fuente y cabecera, Compilar y enlazar, Makefile

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

Índice de contenidos

1. Función main

- Recordemos la función main
- La forma general es la siguiente:

```
int main (int argc, char *argv[]) { cuerpo }
```

- Donde:
  - **argc** : Valor no negativo que indica el número de argumentos enviados al programa desde el entorno donde se ejecuta
  - **argv** : Puntero al primer elemento de los argumentos pasados al programa desde el entorno de ejecución (argv[0] hasta argv[argc-1]). Se garantiza que el valor argv[argc] es un puntero nulo
  - **cuerpo** : El cuerpo de la función main
- Los parámetros argc y argv sirven para pasar argumentos al programa desde el sistema operativo cuando se invoca

1. Argumentos main (argc y argv)

- Siempre que se llama al programa, **argc** como mínimo vale 1 y **argv** contiene un dato en la posición 0, ya que el nombre del programa se toma como el primer argumento. Los argumentos **se separan con espacios** y son almacenados como cadenas de caracteres (**strings**)

```
./programa argumento1 argumento2 argumento3
```

- Desde el main, podemos acceder a los argumentos con los que se ha invocado el programa simplemente accediendo a cada una de las posiciones del array **argv**

```
#include <stdio.h>
int main(int argc, char *argv[]) {
        for(int i = 0; i < argc; i++)
                printf("Parámetro %d: %s\n", i, argv[i]);
        return 0;
}
```

Índice de contenidos

2. Depuración con GDB

- **GDB** Debugger es el depurador estándar para el compilador GNU
- Sirve para trazar y modificar la ejecución de un programa, y permite que el usuario controle y altere los valores de las variables internas del programa
- Para depurar un programa en C con **gdb**, aquél debe haberse compilado con la opción **-g** (o bien la más específica, **-ggdb**) de gcc. Esta opción añade una tabla de símbolos ampliada que podrá ser utilizada por gdb

```
gcc -ggdb -o ejecutable archivo.c
```

- **gdb** es un entorno interactivo que admite órdenes o comandos del usuario

2. Depuración con GDB

- Para depurar el binario (ejecutable con errores) se debe iniciar **gdb** con:

```
gdb ejecutable [core]
```

  donde el archivo **core** es opcional (los core corresponden a volcados de memoria realizados por el sistema operativo cuando falla un programa: “core dumped”)
- **gdb** iniciará su ejecución presentando un “prompt” a la espera de órdenes
- Al iniciar el depurador acompañando un **core**, **gdb** indica cómo finalizó el programa. Si no se añade este archivo (o no se generó), habrá que ejecutar el binario con la orden run, la cual conducirá a la orden que generó el error, así como el error producido y la dirección de memoria que falló

2. Depuración con GDB

- Los comandos más útiles de **gdb** son:
  - **run** : ejecuta el binario analizado por **gdb**. Los argumentos aceptados por el binario pueden colocarse a continuación del comando run
  - **help / info** : comando para ayuda / información acerca del programa depurado
  - **backtrace** : muestra el árbol de llamadas a funciones previas al error
  - **list [i ,f]** : muestra las líneas de código fuente de la número i a la número f. Se pueden omitir uno o los dos parámetros
  - **print var** : muestra el valor de la variable var, si ésta tiene un tipo válido
  - **whatis var** : indica el tipo de var
  - **continue** : reanuda la ejecución tras un breakpoint
  - **step** : ejecuta instrucción a instrucción el código que sigue a un breakpoint. Si encuentra una función, ejecuta sus instrucciones de una en una
  - **next** : ejecución paso a paso, ejecutando de modo atómico las funciones

2. Depuración con GDB

- Los comandos más útiles de **gdb** son:
  - **call nomFunc(args)** : invoca a la función nomFunc con los argumentos incluido
  - **finish** : termina la ejecución de la función actual mostrando su valor de retorno
  - **return valor** : termina la función actual devolviendo valor
  - **ptype tipo** : muestra la estructura completa de tipos construidos (structs, ...)
  - **set variable var = valor** : asigna el valor valor a la variable var
  - **quit** : finaliza la sesión de **gdb**

2. Depuración con GDB

- **Breakpoints**: es posible colocar breakpoints (puntos de ruptura) en distintas zonas del código fuente utilizando **gdb**. Las órdenes aceptadas son:
  - **break numLinea** : detiene la ejecución en el número de línea que se indique
  - **break nomFunc** : detiene la ejecución en la llamada a nombreFuncion
  - **break fichero:numLinea / break fichero:nomFunc** : variante para indicar el nombre del fichero (proyectos con varios archivos)
  - **break numLinea if expr / break nomFunc if expr** : detiene la ejecución cuando la condición expr vale 0 (cierto)
  - **info breakpoints** : muestra los puntos de ruptura distribuidos por el código
  - **delete numBreak** : elimina el breakpoint numBreak
  - **disable numBreak** : desactiva el punto de ruptura sin eliminarlo

2. Ejemplo GDB

```
alumno@ubuntu:~/Escritorio/practicas$ gcc -Wall -o ejecutable -ggdb main.c
alumno@ubuntu:~/Escritorio/practicas$ gdb ejecutable
[... TEXTO...]
Leyendo símbolos desde ejecutable...hecho.

(gdb) list
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(int argc,char* argv[]){
5       for (int i=0; i < 1000; i++)
6           printf("Posicion %d\n",i);
7
8       return 0;
9   }
```

2. Ejemplo GDB

```
(gdb) run
Starting program: /home/alumno/Escritorio/practicas/ejecutable
Posicion 0
Posicion 1
Posicion 2
...
Posicion 999
[Inferior 1 (process 28509) exited normally]

(gdb) break 6
Punto de interrupción 1 at 0x555555554662: file main.c, line 6.

(gdb) run
Starting program: /home/alumno/Escritorio/practicas/ejecutable

Breakpoint 1, main (argc=1, argv=0x7fffffffe638) at main.c:6
6           printf("Posicion %d\n",i);
```

2. Ejemplo GDB

```
(gdb) continue
Continuando.
Posicion 0
Breakpoint 2, main (argc=1, argv=0x7fffffffe638) at main.c:6
6           printf("Posicion %d\n",i);

(gdb) next
Posicion 1
5       for (int i=0; i < 1000; i++)

(gdb) print i
$1 = 1

(gdb) whatis i
type = int

(gdb) set variable i = 800
```

2. Ejemplo GDB

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000555555554662 in main at main.c:6
    breakpoint already hit 7 times

(gdb) delete 1

(gdb) continue
Continuando.
Posicion 801
Posicion 802
...
Posicion 999
[Inferior 1 (process 28518) exited normally]
```

Índice de contenidos

3. Archivos fuente .c y cabecera .h

- Motivación:
  - Dividir el programa en componentes de menor tamaño reduce la carga de mantenimiento
  - Al declarar una función se puede hacer de dos formas:
    - Declaración directa
    - Prototipo de función
  - Esta división provoca que un cambio en la función sumaVector, haya que propagarlo en todos los archivos fuente donde se declaró

Declaración directa

```
int sumaVector(int v[], int n){
        int suma = 0;
        for (int i = 0; i < n; i++)
                suma += v[i];
        return suma;
}
```

Prototipo de función

```
int sumaVector (int v[], int n);

float mediaVector(int v[], float n){
        return sumaVector(v,n)/n;
}
```

sumaVector está declarado en otro archivo fuente .c

3. Archivos fuente .c y cabecera .h

- Solución:
  - Generar archivos llamados **cabeceras (.h) que contienen los prototipos** de **las funciones** que **se declaran en el archivo fuente (.c)** con mismo nombre
  - Esta solución hace que un cambio en la función del archivo fuente, haya que llevarlo a su archivo de cabecera, pero evita tener que modificar todos los archivos fuente que anteriormente tenían el prototipo de la función
  - Ahora hay que **incluir el archivo cabecera**, en aquellos archivos fuente que utilizan la función **(#include "archivocabecera.h")**
  - Es posible incluir el archivo cabecera que contiene las correspondientes definiciones en su archivo fuente para que se compruebe si la declaración y su definición son consistentes

3. Archivos fuente .c y cabecera .h

- Las cabeceras #ifndef / #define / #endif son utilizadas para evitar múltiples importaciones. Así, solo se importa una vez aunque haya múltiples archivos fuentes que lo hagan

Archivo suma.h

```
#ifndef SUMA_H
#define SUMA_H

int sumaVector(int v[], int n);

 #endif
```

Archivo main.c

```
#include <stdio.h>
#include "suma.h"

void main(void){
        int v[] = {1,2,3,4};
        printf("Media: %f", sumaVector(v,4)/4.0);
        return;
}
```

Archivo suma.c

```
#include "suma.h"

int sumaVector(int v[], int n){
        int suma = 0;
        for (int i = 0; i < n; i++)
                suma += v[i];
        return suma;
}
```

Índice de contenidos

4. Compilar y enlazar archivos

- Hasta ahora la compilación se ha realizado en 1 paso:

```
gcc main.c   -o salida
```

- Como ya vimos, la forma simple de compilar varios archivos y enlazarlos es llamando al compilador con todos ellos:

```
gcc main.c utilidades.c   -o salida
```

- Sin embargo, la compilación se divide en dos pasos: uno de **compilación** y otro de **enlazado**
- El paso de compilación crea código ensamblador y lo convierte a código objeto. Para realizar este paso debes indicar al compilador que detenga el proceso antes de enlazar. Esto se hace con la opción **-c**:

```
gcc -c main.c
```

4. Compilar y enlazar archivos

- Usualmente un programa C se compone de varios ficheros .c. Para compilarlo, bastará con compilar cada fichero .c por separado, obteniendo como resultado un fichero .o para cada uno
- Ahora para generar el archivo ejecutable, basta con ejecutar el paso de enlazado. Para ello, se llama al comando gcc seguido de la lista de ficheros .o y con la opción -o damos nombre al programa ejecutable:

```
gcc main.o utilidades.o   -o salida
```

- Es interesante la opción **-Wall**, con ella, se mostrarán advertencias (Warnings) sobre malos usos de determinas estructuras, **se recomienda siempre activar esta opción**, pues ayuda a depurar errores:

```
gcc -Wall main.c utilidades.c   -o salida
```

- El código compila limpiamente cuando lo hace sin warnings

Índice de contenidos

5. Make

- La utilidad **make** permite almacenar las reglas de compilación y enlazado necesarias para generar los archivos de un proyecto, de modo que el proceso pueda ser automatizado
- Optimiza el tiempo de compilación puesto que **sólo recompila y enlaza aquellos archivos que hayan cambiado**
- Para ello, make utiliza un archivo de texto de make, Makefile, que contendrá un conjunto de reglas que indican a make cómo crear cada uno de los componentes del proyecto

5. Make

- Cuando se invoca la orden make, la utilidad busca un archivo llamado GNUmakefile, makefile o Makefile, en ese orden. Por convención, **en Linux se utiliza Makefile**. Una regla de un archivo Makefile consiste en:
  - El objetivo que make creará
  - Una lista de dependencias (archivos) necesarias para crear el objetivo
  - Una lista de comandos a ejecutar para crear el objetivo utilizando las dependencias especificadas. Se pueden incluir también órdenes de shell
- Sintaxis de las reglas de Makefile:

```
objetivo : dependencia [ dependencia [...]]
        comando
        comando
        [...]
```

- **OJO**: delante de cada comando debe haber siempre una **tabulación**

5. Make

- El objetivo por defecto es **all**, ejecutado al invocar "make" sin argumentos
- **Variables**: make permite crear y utilizar variables similares a las de shell (tipo cadena de texto), pero con una sintaxis particular (tipo UNIX).

Declaración:
VARIABLE = texto

Uso:
$(VARIABLE)

- Definidas **al principio del makefile**, y por convención **nombradas en mayúsculas**. Ayudan al mantenimiento de los makefiles
- En los makefiles se permiten comentarios, precedidos del símbolo **#**

5. Ejemplo Makefile (I)

```
# Directorio de archivos fuente
SRCDIR = ./

# "all" es el objetivo por defecto. Sus dependencias se comprueban cuando ejecute la orden "make"
en el directorio actual. Puede haber más de una dependencia.
all : ejecutable

#### Reglas secundarias para compilar cada una de las dependencias.
ejecutable : main.o utilidades.o
	gcc -ggdb -Wall main.o utilidades.o -o ejecutable

main.o: $(SRCDIR)main.c
	gcc -c $(SRCDIR)main.c

utilidades.o: $(SRCDIR)utilidades.c
	gcc -c $(SRCDIR)utilidades.c

# "limpiar" es un objetivo falso. Sirve para borrar los archivos .o (objeto).
limpiar :
	rm *.o
```

5. Ejemplo Makefile (II)

```
# Programas estándar que se van a utilizar: compilador y sus opciones.
CC = gcc
CFLAGS = -ggdb -Wall

# Directorio de archivos fuente
SRCDIR = ./
# Archivos binarios a generar
BINDIR= ./ejecutable

# Archivos fuente
SRC = $(SRCDIR)main.c $(SRCDIR)utilidades.c

# Archivos objeto
OBJ = main.o utilidades.o

...
```

```
...

# "all" es el objetivo por defecto. Sus dependencias se comprueban cuando ejecute la orden "make" en el directorio actual. Puede haber más de una dependencia.
all : $(BINDIR)

#### Reglas secundarias para compilar cada una de las dependencias.
$(BINDIR) : $(OBJ)
        $(CC) $(OBJ) $(CFLAGS) -o $(BINDIR)
$(OBJ) : $(SRC)
        $(CC) -c $(SRC)

# "limpiar" es un objetivo falso. Sirve para borrar los archivos .o (objeto).
limpiar :
        rm *.o
```

Índice de contenidos

6. Ejercicios

1. Guarda el siguiente código en un archivo debug.c y compila utilizando la opción -Wall y corrige los warnings si los hubiera:

```
#include <stdio.h>

int main(void) {
        int i;
        for (i; i < 10; i++)
                printf("[%d]",i);
}
```

2. Crea un programa que pueda ser llamado con dos argumentos, el primero será el nombre del usuario y el segundo su edad, y que además de saludarle y le diga su año de nacimiento. El programa debe asegurarse de que en la llamada se introducen al menos los dos parámetros que se necesitan

   Pista: mira las funciones que proporciona la librería **stdlib.h**

6. Ejercicios

3. Genera tres ficheros (**main.c, vector.c, vector.h**) de manera que el archivo main.c contenga únicamente la función main y haga uso de las 4 funciones declaradas en vector.c, para las cuales, dado un vector de enteros y el número de elementos que contiene:
    - Devuelva la suma de sus elementos
    - Devuelva el valor promedio de sus elementos
    - Muestre por pantalla el vector
    - Ordene el vector aplicando el método de la burbuja
4. Compila y enlaza los anteriores archivos de manera que se genere un programa funcional utilizando el comando gcc
5. Genera un archivo Makefile que compile el programa anterior y contenga las opciones de depuración y muestre warnings si los hubiera

Universidad Rey Juan Carlos
Escuela Técnica Superior Ingeniería de Telecomunicación

Programación de Sistemas de Navegación

1.6. Lenguaje C

Manejo de ficheros

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

Índice de contenidos

1. Manejo de ficheros

- Los ficheros, en contraposición con las estructuras de datos vistas hasta ahora (variables simples, vectores, registros, etc.), son **estructuras de datos almacenadas en memoria secundaria**
- El formato de declaración de un fichero es el siguiente:

```
FILE* nombre_fichero;
```

- En C, es en la operación de apertura cuando se puede decidir si el fichero va a ser de texto o binario:
  - **Texto**: sirven para almacenar caracteres
  - **Binario**: sirven para almacenar cualquier tipo de dato

1. Ficheros: apertura y cierre

- Pasos para la utilización de ficheros:
  - Para utilizar las instrucciones de manejo de ficheros es necesario **incluir**:

```
#include <stdio.h>
```

  - Antes de usar un fichero es necesario realizar una **operación de apertura**

```
descriptor_fichero = fopen (nombre_fichero, modo);
```

  - Si se desea **almacenar** datos en él hay que realizar una operación de **escritura** y si se quiere **obtener datos** de él es necesario hacer una operación de **lectura**
  - Cuando ya no se quiera utilizar el fichero se realiza una **operación de cierre** del mismo para liberar parte de la memoria principal que pueda estar ocupando

```
fclose (descriptor _fichero);
```

1. Ficheros: apertura y cierre

- El **nombre_fichero** será una cadena de caracteres que contenga el nombre (y en su caso la ruta de acceso) del fichero tal y como aparece para el sistema operativo
- El **modo** es una cadena de caracteres que indica el tipo del fichero (texto o binario) y el uso que se va ha hacer de él. Los modos disponibles son:
  - **r** : abre un fichero para lectura. Si el fichero no existe devuelve error
  - **w** : abre un fichero para escritura. Si el fichero no existe se crea, si el fichero existe se destruye y se crea uno nuevo
  - **a** : abre un fichero para añadir datos al final del mismo. Si no existe se crea
  - **+** : símbolo utilizado para abrir el fichero para lectura y escritura
  - **b** : el fichero es de tipo binario
  - **t** : el fichero es de tipo texto. Si no se pone ni **b** ni **t** el fichero es de texto. Los modos anteriores se combinan para conseguir abrir el fichero en el modo adecuado

1. Ficheros: apertura y cierre

- La forma habitual de utilizar la instrucción **fopen** es dentro de una sentencia condicional que permita conocer si se ha producido o no error en la apertura:

```
FILE* fichero;
if ((fichero = fopen("nombres.dat", "r") ) == NULL) {
        printf ( " Error al abrir el fichero\n ");
}

[… Tratamiento del fichero …]

fclose(fichero);
```

1. Ficheros: apertura y cierre

- Existen muchas y variadas operaciones para leer y escribir en un fichero
- Como regla general, es aconsejable utilizarlas por parejas, es decir, si se escribe con fwrite se debe leer con fread
- Entre las funciones tenemos:
  - **fread-fwrite**: hace lectura por bloques, sirve para leer y escribir en ficheros que no sean de texto
  - **fgetc-fputc**: lectura por caracteres. Al final **getc** devuelve un EOF
  - **fgets-fputs**: lectura de cadenas (strings)
  - **fscanf-fprintf**: similares a las instrucciones de lectura por teclado, solo que es necesario indicar como primer argumento el fichero

1. Copiar un fichero en otro

```
#include <stdio.h>
int main(void){
	FILE *fin, *fout;
	char *f1 = "fichero1.txt", *f2 = "fichero2.txt";
	if ((fin = fopen(f1,"r")) == NULL){
		printf("Error al abrir el fichero\n");
		return 1;
	}
	fout = fopen(f2,"w");
	char c = fgetc(fin), x;
	while (c != EOF) {
		x = fputc(c,fout);
		if (x != c) printf("Error de escritura\n");
		c = fgetc(fin);
	}
	fclose(fin);
	fclose(fout);
	return 0;
}
```

```
#include <stdio.h>
int main(void){
	FILE *fin, *fout;
	char *f1 = "fichero1.txt", *f2 = "fichero2.txt";
	if ((fin = fopen(f1,"r")) == NULL){
		printf("Error al abrir el fichero\n");
		return 1;
	}
	fout = fopen(f2,"w");
	int tamanyo = 256;
	char linea[tamanyo];
	while (fgets(linea,tamanyo, fin)){
		fputs(linea,fout);
	}

	fclose(fin);
	fclose(fout);
	return 0;
}
```

¿ fprintf(fout,"%s",linea); ?

1. ¿fscanf? ¡Cuidado!

- Hay que tener cuidado con **fscanf**, ya que al igual que en la entrada por teclado, lee el tipo de dato que se le indique, de modo que:

```
int tamanyo = 256;
char string1[tamanyo], string2[tamanyo];
while (fscanf(fin, "%s", string1) != EOF){
        fprintf(fout, "%s", string1);
}
```

- No es lo mismo que:

```
int tamanyo = 256;
char string1[tamanyo], string2[tamanyo];
while (fscanf(fin, "%s %s", string1, string2) != EOF){
        fprintf(fout, "%s %s", string1, string2);
}
```

Índice de contenidos

2. Ejercicios

- Teniendo un fichero **cantidades.txt** como el que sigue:

```
3452
6542
7468
4567
9789
6457
```

1. Escribe un programa que lea cada elemento, y lo almacene en una estructura en formato **int**. Esta estructura debe ser añadida a una lista que debe contener todos los elementos del fichero
2. Ahora **inserta** 2 nuevos elementos a la lista, **elimina** los 2 primeros y **actualiza** el fichero **cantidades.txt** con la nueva lista

Universidad Rey Juan Carlos
Escuela Técnica Superior Ingeniería de Telecomunicación

Programación de Sistemas de Navegación

1.7. Lenguaje C

Sockets

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

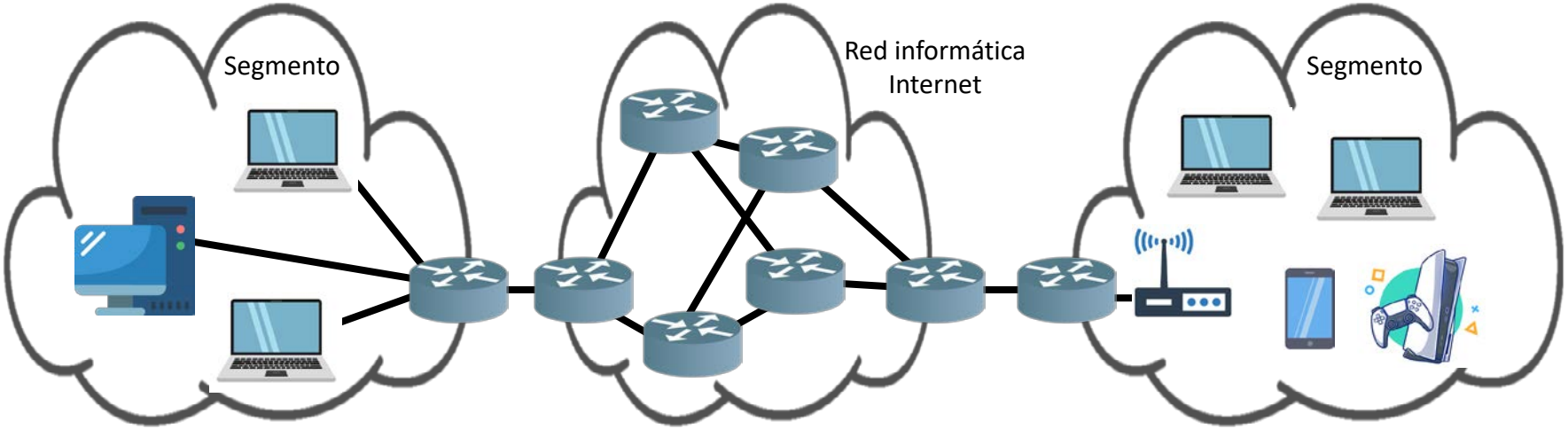
EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

Índice de contenidos

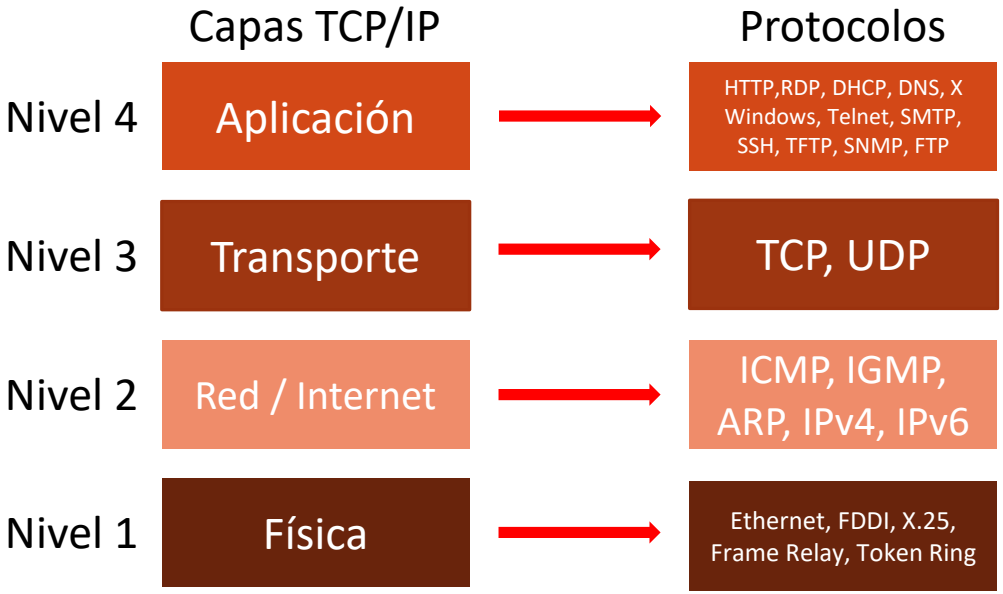
1. Introducción: redes

- Una **red informática** es un tipo de red de telecomunicaciones que permite que los nodos intercambien datos
- Una computadora conectada a la red se llama **host**
- Los hosts están interconectados en la misma **red física** (llamada **segmento**) a través de **enlaces** (alámbricos o inalámbricos)
- Se pueden conectar diferentes segmentos de red con dispositivos llamados **routers**



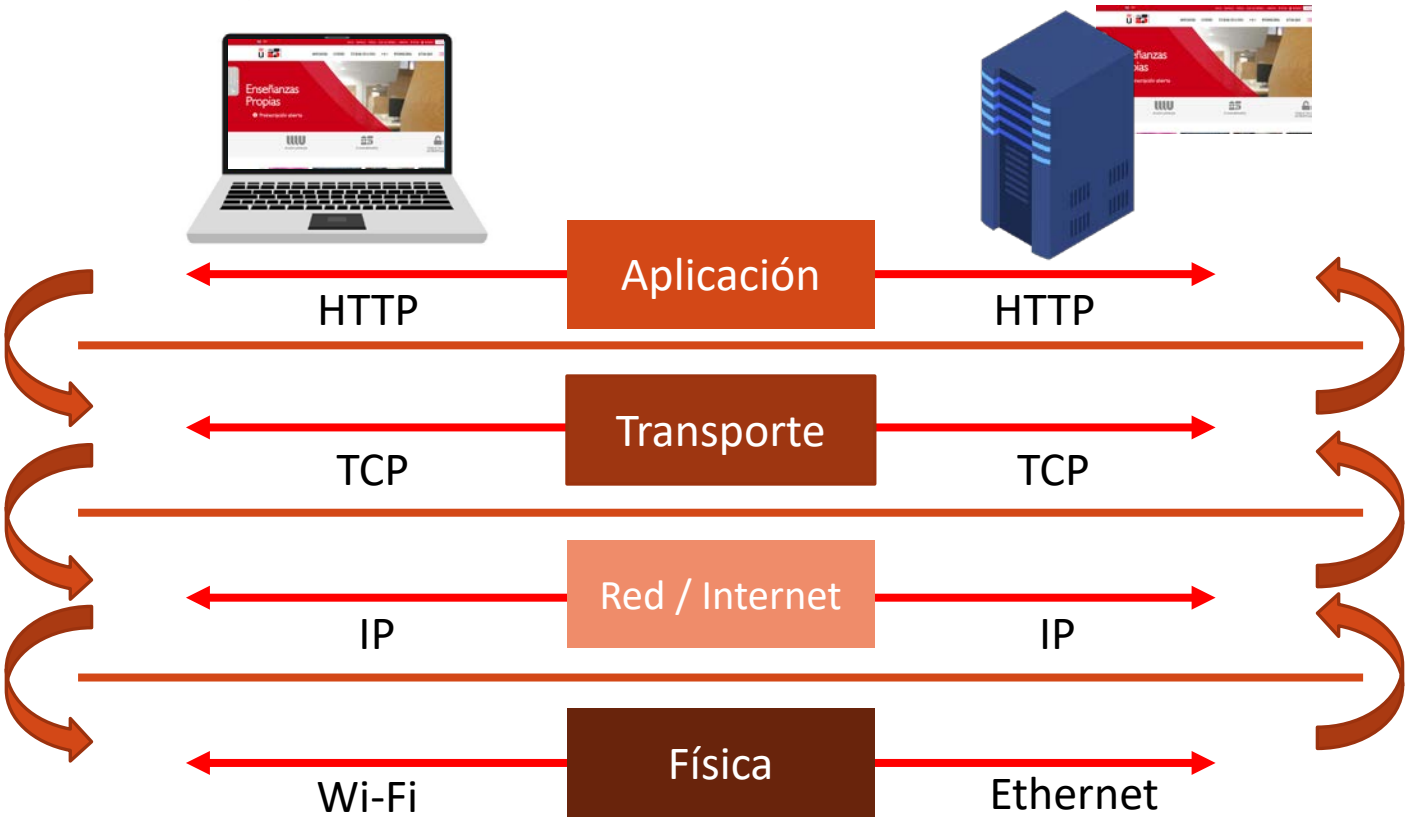
1. Introducción: redes

- Para lograr este objetivo, utilizamos el principio de divide y vencerás
- La arquitectura de comunicaciones se divide en capas en la llamada pila de protocolos
- Cada entidad de un nivel determinado de la pila de protocolos está enfocada en un problema concreto, y proporciona un servicio a la parte de un nivel superior



1. Introducción: redes

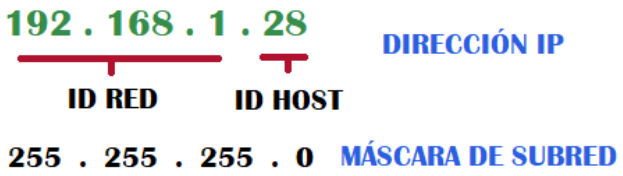
- Cada entidad se comunica con una entidad par (es decir, del mismo nivel) usando un determinado protocolo



1. Introducción: redes

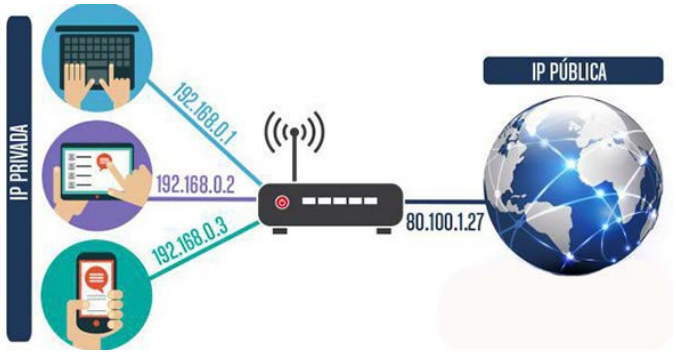
- Para identificar cada host dentro de una red, se utiliza:
  - **Dirección MAC:** conocida también como **dirección física**, es única para cada dispositivo. Está compuesta por 6 bloques de dos caracteres hexadecimales (8 bits) y se corresponde al identificador del equipo a **nivel físico**

  - **Dirección IP:** es una etiqueta numérica, que identifica a un dispositivo (computadora, laptop, teléfono) dentro de un segmento. En **IPv4**, está compuesta por 32 bits, generalmente representado por 4 partes separadas por puntos, las cuales son cada una un número decimal de 8 bits. Esta dirección se engloba dentro del **nivel de red** del modelo TCP/IP. Además se utiliza una **máscara**, la cuál permite saber cuál es el **identificador del host y de la red** dentro del segmento



1. Introducción: redes

- Las direcciones IP pueden ser públicas o privadas:
  - **Pública**: es visible en todo Internet. Cuando accedemos a Internet desde nuestro ordenador obtenemos una dirección IP pública suministrada por nuestro proveedor de conexión a Internet. Esa dirección IP es nuestra dirección IP de salida a Internet en ese momento (puede ser fija o variable)
  - **Privada**: es una dirección fija que se asigna a cada dispositivo conectado a una red privada o doméstica, es decir, la dirección IP que el router asigna a cada ordenador, smartphone, TV…



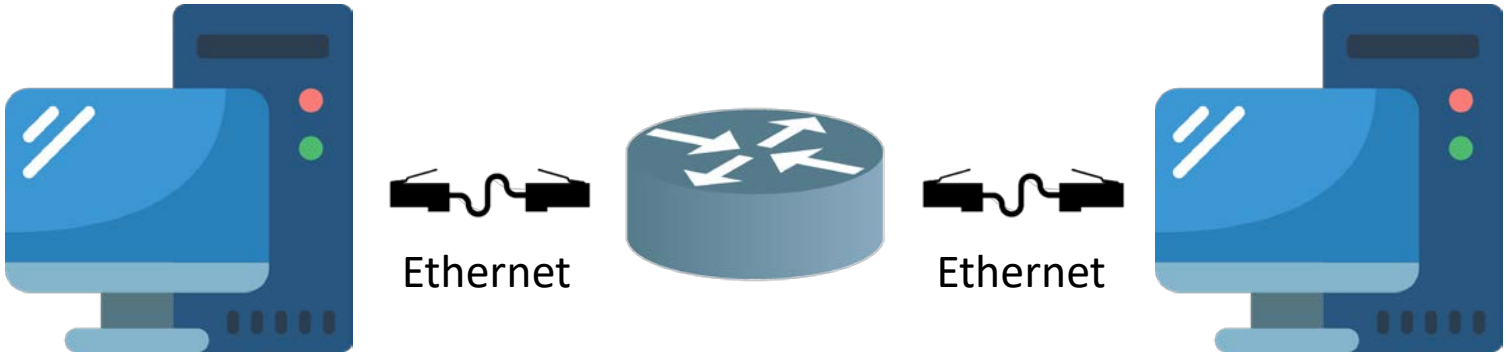
  - Cada dispositivo conectado a un **router** tiene su propia dirección IP privada, mientras comparten la misma IP pública

1. Introducción: redes

- Direcciones IPv4 especiales:
  - **Loopback**: Es una dirección especial de 127.x.x.x reservada para la interfaz virtual interna (loopback). Normalmente **127.0.0.1** (**localhost**) se utiliza para hacer tests dentro del mismo dispositivo
  - **0.0.0.0**: significa “cualquier dirección IPv4”:
    - Como dirección de host, si un servidor escucha en 0.0.0.0, se esperan solicitudes en todas las interfaces de red
    - En el contexto del enrutamiento, generalmente se usa como la ruta predeterminada en las tablas del router
  - **Broadcast**: Todos los bits de host a 1 significa “todos los hosts en esta red“:
    - Se utiliza para enviar un paquete IP a todos los hosts de esa red
    - Si no se conoce el ID de la red, se usa la dirección de transmisión **255.255.255.255** dentro de la red local

1. Introducción: redes

- Para que dos aplicaciones puedan intercambiar información entre sí a través de una conexión red, se necesita:
  - Un protocolo de comunicación común a nivel de red y a nivel de transporte
  - Una dirección del protocolo de red que identifique a cada uno de los ordenadores
  - Un número de puerto que identifique la aplicación dentro del ordenador



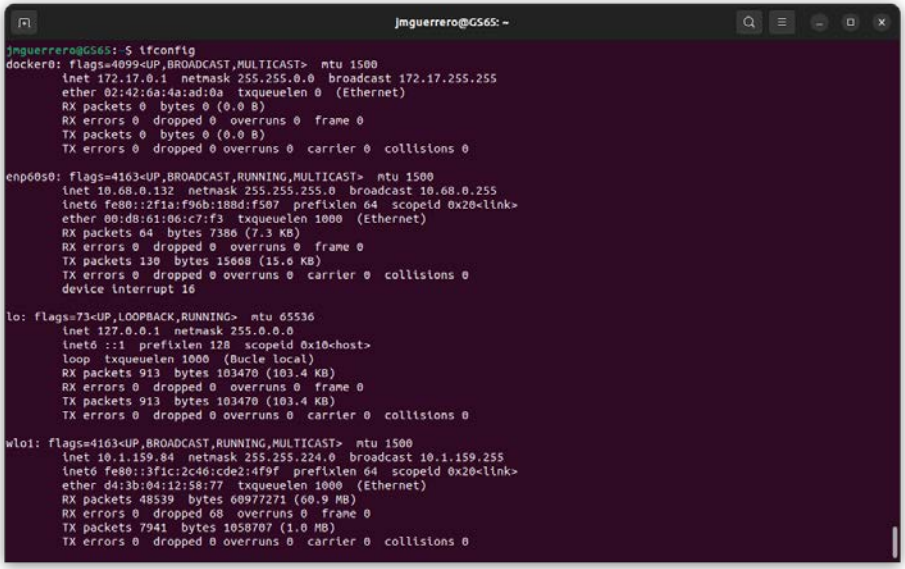
191.168.0.10
:80 http
:8701 app

TCP/IP

191.168.0.20

1. Introducción: redes

- Para saber cuál es la dirección de nuestra máquina:
  - Saber cuál adaptador usamos (Ethernet, Wi-Fi, Loopback, etc.)
  - Ejecutar comando **ifconfig** muestra todas las interfaces de red y sus parámetros
  - La opción **-a** fuerza a **ifconfig** a incluir la información de las interfaces inactivas
  - Los campos MTU y Metric informan sobre los valores actuales de la MTU (unidad máxima de transferencia) y de la métrica para una interfaz dada
  - Las líneas RX y TX dan idea de los paquetes recibidos o transmitidos sin errores, del número de errores ocurridos, de cuántos paquetes han sido descartados, y cuántos han sido perdidos por desbordamiento



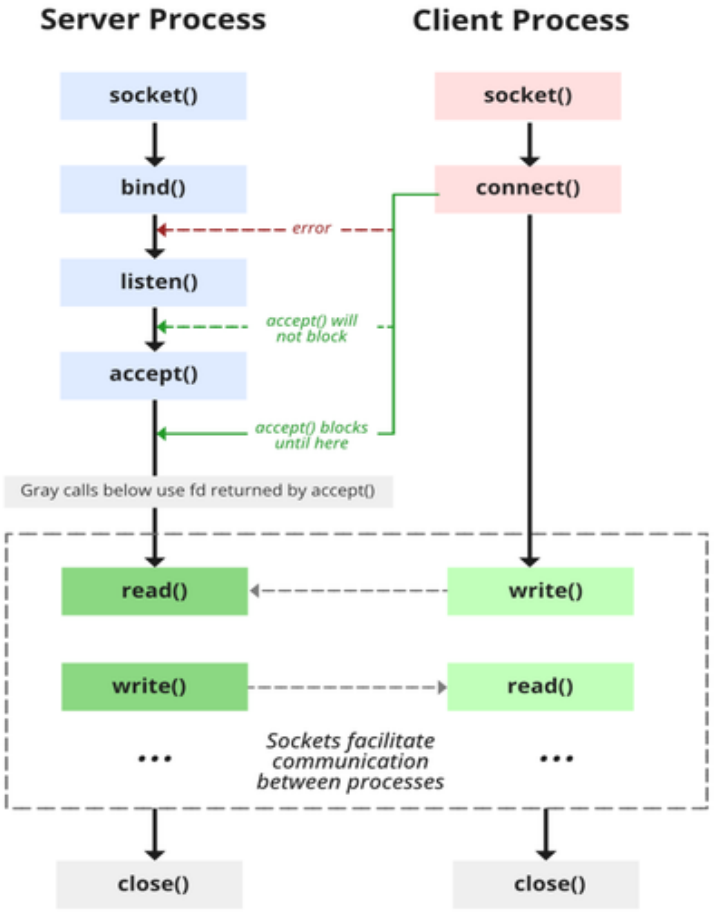
1. Introducción: sockets

- Socket:
  - Concepto abstracto que permite intercambiar información entre dos ordenadores
  - Se define mediante los parámetros:
    - Protocolo de red y de transporte
    - Dirección de red
    - Número de puerto
  - Básicamente, es un conector que recibe peticiones de conexión o solicita las mismas

1. Introducción: sockets

- Socket:
  - Implementan una arquitectura cliente/servidor:
  - Una aplicación, la aplicación **servidor**, permanece a la espera de que otras aplicaciones deseen sus servicios:
    - Utilizando un determinado protocolo de red y de transporte
    - En una determinada dirección de red
    - En un determinado número de puerto
  - Otra aplicación, la aplicación **cliente**, solicita los servicios de la aplicación servidor:
    - Utilizando el mismo protocolo de red y de transporte
    - Una dirección de red
    - Un número de puerto

1. Introducción: sockets



Índice de contenidos

2. Creación socket: apertura

- El primer paso para que cualquier aplicación cliente o servidor pueda comunicarse es crear un socket
- Se realiza con la función **socket()**
- Sus parámetros son:
  - Dominio
  - Tipo
  - Protocolo
- El valor devuelto es un entero:
  - >= 0 si el socket es creado correctamente
  - < 0 si se produce un error en la creación

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

2. Creación socket: apertura

- Dominio: Indica el dominio de comunicación que se desea utilizar:
  - AF_INET: Protocolos de Internet versión 4
  - AF_INET6: Protocolos de Internet versión 6
  - AF_IPX: Protocolos IPX (Novell)
  - AF_APPLETALK: Protocolos Appletalk
  - AF_UNIX o AF_LOCAL: Comunicación local. Optimización de PF_INET para aplicaciones de uso local (conexión entre aplicaciones del propio ordenador)
- Tipo: Tipo de comunicación deseada
  - SOCK_STREAM: Conexión bidireccional confiable con el flujo de datos ordenados
  - SOCK_DGRAM: Mensajes no confiables, sin conexión, con una longitud máxima fija
  - SOCK_SEQPACKET: Conexión bidireccional confiable con el flujo de datos ordenados y datagramas de longitud máxima fija
- No todos los tipos de comunicación deben estar implementados en todos los dominios

2. Creación socket: apertura

- Protocolo: Selección del protocolo particular que utilizará el conector que deseamos crear
  - Generalmente un dominio y tipo solo admite un protocolo particular, por lo que solo puede tomar el valor 0
  - Dominio AF_INET y tipo SOCK_STREAM:
    - Protocolo de transporte TCP
  - Dominio AF_INET y tipo SOCK_DGRAM:
    - Protocolo de transporte UDP

```
#include <sys/types.h>
#include <sys/socket.h>
...
int s;
...
if ( (s=socket(AF_INET, SOCK_STREAM, 0) ) < 0 )
{
        perror("socket");
        exit(0);
}
```

2. Creación socket: cierre

- Es el último paso cuando se quiere cerrar una conexión
- El modo de cierre es independiente del dominio, tipo o protocolo que utilice el socket
- Se realiza con la función **close()**
- Sus parámetros son:
  - Descriptor del socket a cerrar
- El valor devuelto es un entero:
  - 0 si cierra el socket
  - < 0 si se produce un error

```
#include <unistd>
...
int s;
...
if ( close(s) < 0 )
{
        perror("socket");
        exit(0);
}
```

Índice de contenidos

3. Asociar socket a un puerto

- Todo socket que actúe como servidor debe asociarse a un puerto
- El cliente necesita conocer en que puerto se encuentra disponible el servicio para dirigirse al mismo
  - Puerto 21 TCP: FTP (Transferencia de archivos)
  - Puerto 22 TCP: SSH (Conexión segura)
  - Puerto 25 TCP: SMTP (Correo electrónico)
  - Puerto 80 TCP: HTTP (Web)
  - Puerto 443 TCP: HTTPS (Web segura)
  - Puerto 53 UDP: DNS (Servidor de nombres)
  - Puerto 123 UDP: NTP (Sincronización de la hora)

3. Asociar socket a un puerto

- Se realiza con la función **bind()**

```
#include <sys/types.h>
#include <sys/socket.h>
...
int bind(int sockfd, struct sockaddr *addr,int addrlen);
```

- **struct sockaddr** no se utiliza, se utiliza en su lugar **struct sockaddr_in**:
- Hay que incluir la cabecera:

```
#include <netinet/in.h>
```

- **sockfd**: Socket creado con anterioridad
- **addr**: Puntero a la estructura **sockaddr_in** convertida a **sockaddr** mediante una conversión forzada de tipo (cast)
- **addrlen**: Tamaño de la estructura apuntada por el puntero **addr**
- Devuelve:
  - 0 si sucede correctamente
  - < 0 si ocurre un error

3. Asociar socket a un puerto

- La estructura **sockaddr_in** tiene los campos:

```
struct sockaddr_in
{
        int sin_family;
        unsigned short int sin_port;
        struct in_addr sin_addr;
};
```

```
struct in_addr
{
        unsigned long int s_addr;
};
```

- Donde:
  - **sin_family**: Protocolo de comunicación del socket, debe corresponder con el que se uso para crear el socket
  - **sin_port**: Puerto al que se asocia el socket
  - **sin_addr.s_addr**: Dirección de red en la que se pone a la escucha el socket (127.0.0.1, 147.156.1.1, etc.)
    - Si se desea que escuche todas las direcciones de red del ordenador se utiliza la constante INADDR_ANY

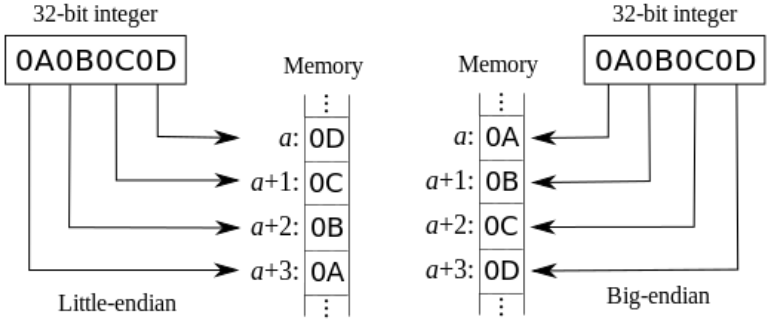
3. Asociar socket a un puerto

- Ejemplo:

```
...
int s;
struct sockaddr_in dir;
...
dir.sin_family=AF_INET;
dir.sin_addr.s_addr=htonl(INADDR_ANY);
dir.sin_port=htons(puerto);

if ( bind(s, (struct sockaddr *)&dir, sizeof(struct sockaddr_in)) != 0 )
{
        perror("bind");
        exit(0);
}
```

3. Asociar socket a un puerto

- Funciones de conversión:
  - Los microprocesadores que incorporan los ordenadores poseen dos formas diferentes de representar los números, primero el byte más significativo y luego el menos significativo (Big-Endian) o al revés (Little-Endian)



  - Dentro de un microprocesador, si cada ordenador manda los datos como los representa internamente, el ordenador que recibe dichos datos puede interpretarlos correctamente o incorrectamente, dependiendo de si su formato de representación coincide con el del emisor o no

3. Asociar socket a un puerto

- Funciones de conversión:
  - Para solucionar este problema, se definió un formato conocido como Orden de los Bytes en la Red (Network Byte Order), que establece un formato común para los datos que se envían por la red, como son el número de puerto, etc.
  - Todos los datos, antes de ser enviados a las funciones que manejan las conexiones en la red, deben ser convertidos a este formato de red:
    - **htonl**: convierte el entero de 32 bits dado por hostlong desde el orden de bytes del hosts al orden de bytes de la red
    - **htons**: convierte el entero de 16 bits dado por hostshort desde el orden de bytes del hosts al orden de bytes de la red
    - **inet_addr**: convierte una **cadena** (string) que contiene una dirección decimal de puntos IPv4 en una dirección adecuada para la estructura de **in_addr**
    - **inet_ntop**: convierte la estructura de dirección de red **in_addr** en una **cadena** (string) en formato decimal de puntos IPv4

3. Asociar socket a un puerto

- Cuando el socket se ha cerrado, el servidor tiene un tiempo de espera asociado antes de permitir reutilizar la dirección local
- Para controlar el uso del socket, y permitir un uso casi inmediato del mismo, existe la función **setsockopt()**

```
#include <sys/socket.h>
...
int setsockopt(int socket, int level, int option_name,
        const void *option_value, socklen_t option_len);
```

- El valor devuelto es un entero:
  - 0 si se ha podido realizar con éxito
  - < 0 si se produce un error

3. Asociar socket a un puerto

- Sus parámetros son:
  - **socket** : Descriptor del socket creado con anterioridad
  - **level**: Especifica el nivel de protocolo. Para establecer opciones a nivel de socket, se utiliza SOL_SOCKET
  - **option_name** y **option_value**: Acción que se lleva a cabo sobre el socket. En nuestro caso utilizaremos SO_REUSEADDR, la cuál si **option_value** toma un valor positivo, indica que se activa y por lo tanto podemos reutilizar la dirección local nuevamente con la función bind()
  - **option_len**: es el tamaño del tipo **option_value**

```
const int enable = 1;
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)) < 0)
                perror("setsockopt(SO_REUSEADDR) failed");
```

Índice de contenidos

4. Diferencias entre TCP y UDP

- TCP y UDP son protocolos de transmisión de datos diferentes
- A través del protocolo IP se determina la manera en que serán transmitidos los datos de un mensaje entre un dispositivo y otro
- Los protocolos TCP y UDP son usados para codificar y enviar el mensaje a la dirección determinada por el protocolo IP

4. Diferencias entre TCP y UDP

- El protocolo TCP (Transfer Control Protocol) es el más habitual por tratarse de un protocolo de transporte 'orientado a conexión'
- TCP está diseñado no solo para transmitir una determinada información entre un dispositivo y otro, sino también para verificar la correcta recepción de la información transmitida es decir, es un protocolo para manejar conexiones de extremo a extremo
- Es el complemento ideal para el protocolo IP, ya que los datagramas del protocolo IP no están diseñados para establecer un sistema recíproco de verificación entre los dispositivos que intercambian la información

4. Diferencias entre TCP y UDP

- El protocolo UDP (User Diagram Protocol) funciona de manera similar al protocolo TCP, pero no es un protocolo de transporte orientado a conexión
- UDP no verifica la recepción de los datos transmitidos entre un dispositivo y otro, por lo que el sistema de verificación de la recepción de los datos debe implementarse en las capas superiores
- La principal ventaja del protocolo UDP consiste en su velocidad:
  - Permite una velocidad de transferencia superior a la del protocolo TCP
  - Es el más utilizado por los servicios de transmisión de voz o vídeo en streaming, donde la velocidad de la transmisión es más importante que una posible pérdida de datos puntual

4. Diferencias entre TCP y UDP

	TCP	UDP
Fiabilidad	Alta	Más baja
Velocidad	Más baja	Alta
Método de transferencia	Los paquetes se envían en una secuencia	Los paquetes se envían en un flujo
Deteción y corrección de errores	Sí	No
Control de congestión	Sí	No
Acuse de recibo	Sí	Solo el checksum

Índice de contenidos

5. TCP: escucha de un puerto

- Un socket TCP asociado a una dirección y puerto, es puesto en escucha con la función **listen()**
- Sus parámetros son:
  - Descriptor del socket a poner a la escucha
  - Tamaño de la cola de aceptación de peticiones
- Devuelve:
  - 0 si se ejecuta de forma correcta
  - < 0 si se produce un error

```
...
int s;
...
if ( listen(s, 5) != 0 )
{
        perror("listen");
        exit(0);
}
```

5. TCP: Servidor - aceptar petición

- Acepta las peticiones de conexión con **accept()**
- Sus parámetros son:
  - Socket asociado a la dirección y puerto y puesto a la escucha
  - Estructura que contendrá la dirección y puerto del cliente del que se acepta la conexión
  - Puntero al tamaño de la estructura
- Devuelve:
  - >= 0 Identificador del socket donde se ha aceptado la conexión
    - Al aceptar una conexión se crea un nuevo socket que es el que atiende la conexión
    - El socket original que escuchaba la dirección y puerto permanece inalterado después de aceptar la conexión
  - < 0 Error en la aceptación de la conexión

5. TCP: Servidor - aceptar petición

```
...
int sock, sock_conectado, tam;
struct sockaddr_in dir;
...
tam = sizeof( struct sockaddr_in );
if ( ( sock_conectado = accept(sock, (struct sockaddr *)&dir, &tam) ) < 0 )
{
        perror("accept");
        exit(0);
}
```

5. TCP: Cliente - conexión

- Solicita la conexión mediante **connect()**
- Sus parámetros son:
  - Descriptor del socket que se utiliza para la conexión
  - Estructura con la dirección y puerto al que se desea conectarse
  - Longitud de la estructura anterior
- Valor devuelto:
  - 0 si la conexión se realiza de forma correcta
  - < 0 si se produce un error

5. TCP: Cliente - conexión

```
...
int sock;
struct sockaddr_in dir;
...
dir.sin_family=AF_INET;
dir.sin_addr.s_addr = inet_addr("147.156.1.1");
dir.sin_port=htons(13);

if ( connect(sock, (struct sockaddr *)&dir, sizeof(struct sockaddr_in) ) < 0 )
{
        if (errno==ECONNREFUSED) /* Servicio no disponible */
                ...
        else /* Otro error */
        {
                perror("connect");
                exit(0);
        }
}
```

5. TCP: Recepción y envío

- Se suele realizar con las funciones **read()** (para leer datos) y **write()** (para escribir datos)
- Sus parámetros son:
  - Descriptor del socket conectado
  - Buffer donde almacenar los datos a leer o que contiene los datos a escribir
  - Tamaño máximo de los datos a leer o tamaño de los datos a escribir
- Valor devuelto:
  - < 0 si sucede un error
  - >= 0 con el número de datos leídos o escritos
  - Generalmente el valor 0 en lectura indica que se ha cerrado la conexión

```
...
#define TAM 100
...
int sock, n;
char buffer[TAM];
...
if ( ( n = read(sock, buffer, TAM) ) < 0 )
{
        perror("read");
        exit(0);
}
```

Índice de contenidos

6. UDP: Recepción de datos

- Se realiza con las funciones **recvfrom()** o **recv()**
- Sus parámetros son:
  - Descriptor del socket
  - Buffer donde almacenar los datos a leer
  - Tamaño máximo de los datos a leer
  - Opciones de envío, generalmente valor 0 (ninguna opción)
  - Puntero a la estructura de datos que contendrá la dirección y puerto que envió los datos (solo en **recvfrom()**)
  - Puntero a la longitud de la estructura de datos anterior (solo en **recvfrom()**)
- Valor devuelto:
  - < 0 si sucede un error
  - >= 0 con el número de datos leídos o escritos

6. UDP: Envío de datos

- Se realiza con las función **sendto()**
- Sus parámetros son:
  - Descriptor del socket
  - Buffer con los datos a enviar
  - Tamaño de los datos a enviar
  - Opciones de envío, generalmente valor 0 (ninguna opción)
  - Puntero a la estructura de datos que contiene la dirección y puerto donde enviar los datos
  - Longitud de la estructura de datos anterior
- Valor devuelto:
  - < 0 si sucede un error
  - >= 0 número de bytes escritos

6. UDP: Envío de datos

```
#define TAM 100
...
int sock;
struct sockaddr_in dir;
char buffer[TAM];
...
dir.sin_family=AF_INET;
dir.sin_addr.s_addr = inet_addr("147.156.1.1");
dir.sin_port=htons(13);

if ( sendto(sock, buffer, strlen(buffer), (struct sockaddr *)&dir, sizeof(struct sockaddr_in)) < 0 )
{
        perror("connect");
        exit(0);
}
```

Índice de contenidos

7. Comprobar estado de un socket

- Se realiza con las función **select()**
- Sus parámetros son:
  - Entero con el valor, incrementado en una unidad, del descriptor del socket más alto de cualquiera de los conjuntos que se desea comprobar
  - Conjunto de sockets a comprobar para lectura
    - En los sockets de tipo SOCK_STREAM también se modifica si llega una petición de conexión a un socket
  - Conjunto de sockets a comprobar para escritura
  - Conjunto de sockets a comprobar para excepciones
  - Limite de tiempo antes de que termine la espera de comprobación
- Valor devuelto:
  - < 0 si sucede un error
  - = 0 Ningún socket ha cambiado su estado
  - > 0 Número de sockets cuyo estado ha cambiado
- Si algún conjunto no desea comprobarse puede ponerse NULL
- Si desea que el tiempo de comprobación sea infinito se pone NULL

7. Comprobar estado de un socket

- Los sockets a comprobar se introducen en el conjunto que se declara como:

  fd_set conjunto;

- Para borrar el conjunto, insertar sockets, etc., se utilizan las funciones:
  - FD_ZERO(&conjunto): Inicializa el conjunto
  - FD_SET(socket, &conjunto): Añade el socket al conjunto
  - FD_CLR(socket, &conjunto): Borra el socket del conjunto
  - FD_ISSET(socket, &conjunto): Comprueba si el socket se encuentra en el conjunto

7. Comprobar estado de un socket

- El limite de tiempo se comprueba con una estructura:

```
struct timeval {
        unsigned long int tv_sec;
        unsigned long int tv_usec;
};
```

- La función **select()**:
  - Deja en la estructura el tiempo que restaba hasta la finalización del tiempo de espera indicado
  - Modifica los conjuntos dejando en ellos tan solo los sockets modificados
    - La función FD_ISSET() permite ver cuales existen en el conjunto devuelto

7. Comprobar estado de un socket

```
...
int sock,n;
fd_set conjunto;
struct timeval timeout;
...
FD_ZERO(&conjunto);
FD_SET(sock, &conjunto);
timeout.tv_sec=1;
timeout.tv_usec=0;
if ( (n = select(sock+1, &conjunto, NULL, NULL, &timeout) ) < 0 )
{
        perror("select");
        exit(0);
}
```

Índice de contenidos

8. Ejercicios

Partiendo del ejemplo: https://github.com/jmguerreroh/socket_c

1. Modifica el **servidor TCP** para que en la llamada al programa se permita seleccionar el puerto de escucha:

```
./server 8081
```

2. Modifica el **cliente TCP** para que en la llamada al programa se permita seleccionar la dirección IP y el puerto del servidor al que se desea conectar:

```
./client 127.0.0.1 8081
```

3. Prueba el **cliente** y el **servidor TCP** ejecutando cada uno en equipos diferentes
4. Modifica el **cliente** y el **servidor TCP** para que detecten cuando una conexión ha sido cerrada
5. Haz que el **servidor TCP** genere un nuevo socket y se quede a la escucha de una nueva conexión cuando se cierre su conexión con el cliente
6. Dentro del **servidor TCP** haz que se registre el número de conexiones que ha tenido y las muestre por pantalla

Universidad
Rey Juan Carlos
Escuela Técnica Superior
Ingeniería de Telecomunicación

Programación de Sistemas de Navegación

2.1. Lenguaje C++

Diferencias C y C++, POO, Clases, Sobrecarga, this, inline, friend, const, static, espacio de nombres, compilar, make

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

Índice de contenidos

Diferencias entre C y C++

- C es un lenguaje estructurado:
  - El código es **secuencial**
  - El conjunto de sentencias o instrucciones se ejecutan una por una siguiendo un orden fijado
  - Se basan en el "divide y vencerás". Cuando una tarea (o procedimiento) es "demasiado" compleja como para ser descrita, se divide dicha tarea en subtareas más sencillas de acometer por separado
  - Esta aproximación es buena a la hora de resolver problemas de gran complejidad. Sin embargo, adolece de varias deficiencias:
    - No permite una reusabilidad clara, y se cae en el síndrome de "reinventar la rueda" en cada proyecto que se acomete
    - No es adecuada para programación orientada a "eventos"

Diferencias entre C y C++

- C++ es un lenguaje de Programación Orientado a Objetos (POO):
  - El código **no es secuencial**
  - Dividido en **clases**, con propiedades: **atributos** y **métodos**
  - Tratan a los datos y a las operaciones que los manipulan como un "todo" al que llamaremos "objeto"
  - Por tanto, un objeto es una entidad autocontenida con una identidad propia y unas características que lo definen
  - C++ es un lenguaje de Programación Orientado a Objetos y soporta los cuatro pilares básicos de la POO (esto se verá en detalle en los temas siguientes):
    - Encapsulación
    - Ocultación de los datos
    - Herencia
    - Polimorfismo

Diferencias entre C y C++

- C++ incluye toda la funcionalidad de C, y añade nuevas mejoras:
  - Nuevos tipos de datos: **bool** (valores booleanos) y **string** (cadenas de texto)
  - La palabra reservada **const** hace que una variable sea de solo lectura
  - Hacer las **funciones inline**, para mejorar el rendimiento en ciertos casos
  - Los ficheros estándar a incluir ya no tienen extensión .h, aunque se conservan los antiguos por compatibilidad. Aquellos ficheros que sean heredados del C llevan una c delante

```
#include <cstdio>    // Fichero stdio.h de C
#include <stream>   // Fichero de C++
```

  - Cuando declaremos estructuras, clases, enumeraciones o uniones no hace falta poner la palabra struct, class... en su definición, sino sólo el nombre que hayamos dado a la estructura, clase...

```
struct punto {
      int x;
      int y;
};
```

```
punto a; // No es necesario struct punto a, como en C
a.x = 4;
a.y = 5;
```

Diferencias entre C y C++

Lenguaje C++

- Programación Orientada a Objetos
- Encapsulación
- Ocultación de datos
- Herencia
- Polimorfismo
- Sobrecarga

Lenguaje C

- Programación estructurada
- Variables y constantes
- Condicionales
- Sentencias repetitivas
- Arrays
- Funciones

De C a C++

- Todo lo aprendido para C, es aplicable a C++:
  - Variables simples: char, short, int, long, float, double, signed, unsigned
  - Operaciones aritméticas: + - * / %
  - Operaciones bit a bit: & | ^ ~
  - Operadores de comparación: < > <= >= != ==
  - Saltos condicionales: if, if - else, if - else if - else
  - Caminos alternativos: switch, goto
  - Comprobaciones multicondicionales: && ||
  - Bucles: while, do-while, for
  - Funciones, retorno y parámetros
  - Tuplas struct
  - Definición de tipos con typedef
  - Tablas [ ]
  - Punteros: & . -> [ ] malloc free NULL void*
- Los fundamentos y la sintaxis son rigurosamente los mismos. **Un programa escrito en C puede compilarse en C++**

De C a C++

- Hay algunas diferencias entre C y C++ que modifican la escritura pero no el diseño del programa:
- **Entrada / Salida**:
  - Las funciones **printf**() y **scanf**(), aunque se pueden seguir utilizando, son reemplazadas con mejoras por los objetos **cout** y **cin**
  - Estos objetos necesitan como en C, ser incluidos utilizando la librería correspondiente (#include <iostream>), modificando su uso

```
#include <iostream>

int main() {
        int number;
        std::cout << "Enter a number:" << std::endl;
        std::cin >> number;
        std::cout << "The number is: " << number << std::endl;
        return 0;
}
```

Con \n la salida será la misma, pero no aparecerá en tiempo real

De C a C++: nuevos tipos

- **Booleano:**
  - Solo pueden tener dos valores: **true** (1) o **false** (0)
  - Un tipo booleano se declara como una variable de otro tipo, pero con la palabra reservada **bool**

```
bool existe = true;
```

- **String:**
  - Son objetos que representan una secuencia de caracteres
  - El tipo string se encuentra dentro del espacio de nombres **std** de la librería **string**, aunque también la podemos encontrar en **std** de la librería **iostream**:

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

  - El tipo string se declara con la palabra reservada **string** y la cadena de caracteres que se le debe asignar ha de ir entre comillas dobles (" ")

```
string nombre = "Juan";
```

  - Los string pueden ser accedidos como un vector (nombre[0]) y utilizados con los operadores: **> < >= <= == !=**

De C a C++: funciones

- **Paso por referencia (modificar la variable parámetro)**:
  - Es posible indicar que un parámetro será pasado por referencia sin necesidad de hacerlo en la llamada
  - En C utilizábamos la siguiente forma:

```
int poner_nota (int* nota){
        *nota = 5; // Se modifica el original
}
```

```
int nota_juan = 0;
poner_nota(&nota_juan);
printf("%d",nota_juan);
```

  - En C++ es posible hacerlo así:

```
int poner_nota (int& nota){
        nota = 5; // Se modifica el original
}
```

```
int nota_juan = 0;
poner_nota(nota_juan);
std::cout << nota_juan;
```

  - Si el parámetro es constante (**const int&** nota) no puede modificarse, el compilador daría error (también válido para punteros)

De C a C++: funciones

- **Ventajas del paso por referencia**:
  - La sintaxis es más sencilla:
    - Es similar al paso por valor
    - No hay que realizar indirecciones
  - Permite modificar la variable pasada como parámetro
  - Es más eficiente que el paso por valor para estructuras de datos:
    - No se pasa una copia sino un alias
    - Si no queremos que se pueda modificar se puede utilizar **const**
  - En C++ el paso por referencia es el recomendado para pasar parámetros:
    - Solamente para tipos básicos que no haya que modificar se utiliza el paso por valor
    - El paso de dirección se reduce fundamentalmente a arrays

De C a C++: funciones

- **Valores por defecto en las funciones**:
  - C++ permite dar valores por defecto a los parámetros de una función
  - Los parámetros por defecto pueden ser omitidos en la llamada a una función
  - Deben estar siempre al final en la lista de parámetros

```
// Declaración ejemplo función
void dibuja_rectangulo(const rectangulo& rect, float escala=1.0) {
        ..........
}


// Llamada a la función
dibuja_rectangulo(rect1, 2.5);      // escala 2.5
dibuja_rectangulo(rect2);           // escala 1.0
```

De C a C++: funciones

- **Sobrecarga**:
  - En C++ una función se puede definir tantas veces como se quiera (sobrecarga)
  - Cada definición debe distinguirse por sus parámetros, en número y en tipo:

```
multiplicar_matriz_escalar(matriz& m, int escalar);
multiplicar_matriz_escalar(matriz& m, double escalar);
multiplicar_matriz_escalar(matriz& m, complejo escalar);
multiplicar_matriz_escalar(const matriz& m, int escalar, matriz& resultado);
```

  - El compilador utiliza una u otra según el número y el tipo de los parámetros
  - En caso de ambigüedad (una llamada puede coincidir con varias funciones) la compilación fallará

De C a C++: forzar tipos

- En C++ se puede forzar una transformación de tipos:
  - Permite que el programador redefina nuevos forzamientos para tipos no estándar

```
float c;
int a;
float b;
.............
c=float(a)/b; // Se pasa el tipo a float
c=(float)a/b; // Lo mismo (más utilizado)
```

- Existen conversiones implícitas
  - Las añade automáticamente el compilador cuando se realizan ciertas operaciones o se pasan parámetros no predefinidos

De C a C++: memoria dinámica

- Donde mayor potencia desarrollan los punteros es cuando se unen al concepto de memoria dinámica
- C++ dispone de dos operadores para acceder a la memoria dinámica, son "**new**" y "**delete**" (en vez de funciones malloc y free de una librería)
- **Toda la memoria que se reserve** durante el programa **hay que liberarla** antes de salir del programa. No seguir esta regla puede tener consecuencias fatales
- **¡Y mucho cuidado!**

  Como ya sabemos, si se pierde un puntero a una variable reservada dinámicamente, dicha variable NO se puede liberar

```
int* a;
struct stPunto { float x,y; }; stPunto* d;

a = new int;
d = new stPunto;

*a = 10;
d -> x = 12;
d -> y = 15;

delete a;
delete d;
```

De C a C++: memoria dinámica

- El operador **new** sirve para reservar memoria dinámica:

  new <tipo de dato> [(<inicialización>)] // Tipo básico, objeto
  new <tipo de dato> **[** <número de elementos> **]** // Arrays dinámicos

  - La inicialización, si aparece, se usará para asignar valores iniciales a la memoria reservada con **new**, pero no puede ser usada con arrays
  - Si la reserva de memoria no tuvo éxito, **new** devuelve un puntero nulo
  - Llama al constructor de un objeto para inicializarlo

- El operador **delete** libera la memoria dinámica reservada con **new**:

  delete <nombre del puntero> // Tipo básico, objeto
  delete[] < nombre del puntero > // Arrays dinámicos

De C a C++: memoria dinámica

- Es importante liberar siempre la memoria reservada con new

```
char *c;
float **f;

// Cadena de 122 caracteres
c = new char[123];
// Matriz: array de 10 punteros a float, cada elemento será un array de 10 float
f = new float*[10];

for(int n = 0; n < 10; n++) f[n] = new float[10];
f[0][0] = 10.32;
c[0] = 'a';

for(int n = 0; n < 10; n++) delete[] f[n];
delete[] f; delete[] c;
```

Índice de contenidos

Programación Orientada a Objetos

Programación tradicional

- Descomposición funcional:
  - Se identifican las funciones que se necesitan para resolver el problema
  - El problema se resuelve con un conjunto de llamadas a estas funciones
- Control de flujo:
  - A partir de unos datos de entrada hay que conseguir unos datos de salida
  - Identifica las estructuras de datos que se necesitan
  - El problema se resuelve haciendo transformación de estos datos

Programación Orientada a Objetos

- Se identifican los objetos abstractos que representan el dominio del problema
- Se identifican las operaciones abstractas que hay que realizar con estos objetos
- La solución a estos problemas es una llamada a estos objetos

CONCLUSIÓN

CÓDIGO MÁS ROBUSTO, FLEXIBLE Y ANTE TODO REUTILIZABLE

Objetos y clases: definición

- **Clase**:
  - Una clase es un tipo de datos que usamos para representar cualquier cosa (real o abstracta) y que definimos de forma que combine:
    - Un conjunto de variables (atributos, o variables miembro)
    - Un conjunto de funciones (métodos, o funciones miembro)
  - Las variables de una clase pueden ser otras clases

  - Ejemplo de declaración de clase:

```
class Gato {
        public:
                unsigned int Edad;
                unsigned int Peso;
                void Maullar();
};
```

Objetos y clases: definición

- **Clase**:
  - Una clase en sí misma no reserva memoria para un 'Gato', ni se puede usar directamente
  - Una clase define cómo es un 'Gato', qué datos contiene y qué cosas puede hacer (Maullar)
  - No usaremos las clases, sino objetos de esa clase
  - Un sistema real estará formado por varios tipos de objetos de distintas clases que interactúan entre sí, colaborando y pasándose información de unos a otros

Objetos y clases: definición

- **Objeto**:
  - Un objeto es una instanciación de una clase determinada, una variable del tipo definido por la clase:

```
Gato Silvestre;
int contador;
```

  - Sobre la instancia de un gato sí podemos actuar, y se accede a sus variables (atributos) y funciones (métodos) de la misma forma que los campos de una estructura:

```
Silvestre.Edad = 3;
Silvestre.Maullar();
```

  - Si utilizamos un puntero:

```
Gato* punteroGato = new Gato();
punteroGato -> Edad = 3; // Equivalente a (*punteroGato).Edad = 3;
punteroGato -> Maullar();
```

Objetos y clases: definición

- **Objeto**:
  - Esa es la diferencia: Las clases son el 'concepto' o definición, y los objetos son las materializaciones de dichas definiciones
  - Al manipular un objeto, estamos atados a la clase a la que dicho objeto pertenece:

```
Silvestre.Ladrar(); // Esto no puede funcionar
```

Objetos y clases: ejemplo

- **Clase:**

```
#include <iostream>

class Gato {
        public:
                unsigned int Edad;
                unsigned int Peso;
                void Maullar();
};
```

- **Objeto**:

```
int main() {
        Gato Silvestre;
        Silvestre.Edad = 5;
        std::cout << "Silvestre tiene " << Silvestre.Edad << " años";
        return 0;
}
```

Clases: características

- **Encapsulamiento**:
  - Una clase combina datos de diferentes tipos junto con las funciones que los procesan
- **Abstracción**:
  - Una clase permite implementar tipos abstractos de datos (generalización)
- Los **datos y funciones** miembro de una clase pueden ser:
  - Privados (**private**): Sólo pueden ser utilizados en el interior de la clase
  - Públicos (**public**): Pueden ser utilizados desde fuera de la clase
  - Protegidos (**protected**): Sólo pueden ser utilizados en el interior de la clase o por clases derivadas (Herencia)

Clases: características

- Atención: todos los miembros de una clase son privados por defecto

```
#include <iostream>

class Gato {
        unsigned int Edad;
        unsigned int Peso;
    public:
        void Maullar();
};

int main() {
    Gato Silvestre;
    Silvestre.Edad = 5;   // Incorrecto
    Silvestre.Maullar(); // Correcto
    return 0;
}
```

Clases: acceso a datos

- Como regla general, las **variables miembros** de las clases han de ser **privadas**
- El acceso desde el **exterior** se consigue proporcionando **métodos públicos**, que serán los que manipulen dichas variables
- De esta forma separamos los detalles de cómo están guardados los datos de cómo se usan dichos datos (podemos cambiar dicha representación sin que los programas que la usan tengan que ser cambiados a su vez)
- Los **métodos** que devuelven el valor de un campo privado se conocen como **accesores** (por convenio con prefijo **get**), y los que los modifican **mutadores** (con prefijo **set**)

Clases: acceso a datos

- Ejemplo:

```
class Gato {
        public:
                void Maullar();

                unsigned int getEdad();
                unsigned int getPeso();

                void setEdad(unsigned int edad);
                void setPeso(unsigned int peso);

        private:
                unsigned int Edad;
                unsigned int Peso;
};
```

Clases: métodos

- Hasta ahora hemos visto cómo se declaran las clases, pero no cómo se implementan
- Los métodos de las clases se implementan comenzando por el tipo que devuelve el método, el nombre de la clase, seguido de dos ':' y el nombre de la función:

```
tipo clase::funcion(parametros)
{
        // Implementación
}
```

Clases: métodos de una clase

- Métodos especiales:
  - **Constructor**: se ejecuta cuando se crea el objeto. Atendiendo a su naturaleza, puede ser:
    - Constructor por defecto
    - Constructor parametrizado
    - Constructor de copia
  - **Destructor**: se ejecuta cuando el objeto desaparece
  - **Operadores de asignación**

  - Si no indicamos constructor o destructor, el compilador incluirá uno por defecto que no toma parámetros y que no hace nada

Constructores y destructores

- **Constructor:**
  - El constructor es una función con el mismo nombre que la clase

```
clase::clase(parametros);
```

- **Destructor:**
  - El destructor tiene el mismo nombre que la clase pero con el símbolo ~ delante
  - No acepta argumentos
  - No devuelven un valor (o void)
  - Se pueden declarar como virtual

```
clase::~clase();
```

- Usaremos el **constructor** para llevar a cabo las inicializaciones que el objeto requiera al construirse, y el **destructor** para lo contrario (liberar memoria, por ejemplo) cuando el objeto se destruya

Ejemplo

```
class Gato {
        public:
                Gato();     // Constructor
                ~Gato();   // Destructor
                void Maullar();
                unsigned int getEdad();
                unsigned int getPeso();
                void setEdad(unsigned int edad);
                void setPeso(unsigned int peso);
        private:
                unsigned int Edad;
                unsigned int Peso;
};
```

```
Gato::Gato() {  // Constructor
        Edad = 3;
        Peso = 2;
}

Gato::~Gato() { // Destructor
        cout << "Adios al gato\n";
}

void Gato::Maullar() {
        …
}
```

Ejemplo

- Podemos tener distintos constructores:

```
class Gato {
	public:
		Gato();
		Gato(unsigned int edad);
		Gato(unsigned int edad, unsigned int peso);
		~Gato();
		void Maullar();
		unsigned int getEdad();
		unsigned int getPeso();
		void setEdad(unsigned int edad);
		void setPeso(unsigned int peso);
	private:
		unsigned int Edad;
		unsigned int Peso;
};
```

```
Gato::Gato() {
	Edad = 3;
	Peso = 2;
}
Gato::~Gato() {
	cout<<"Adios al gato\n";
}
Gato::Gato(unsigned int edad) {
	Edad = edad;
}
Gato::Gato(unsigned int edad, unsigned int peso) {
	Edad = edad;
	Peso = peso;
}
```

Ejemplo

- Así, en la instanciación

```
Gato Centella;
```

lo primero que se ejecuta es el constructor que no toma parámetros, e inicializa la edad y el peso de 'Centella' con 3 y 2 respectivamente

- Y en

```
Gato Pipe(4,5);
```

se ejecuta el constructor que toma dos parámetros e inicializa peso y edad con 4 y 5 respectivamente

Ejemplo

- El resto de funciones se definirían como sigue:

```
void Gato::Maullar() {
        cout << "MIAUUU\n";
}
unsigned int Gato::getEdad() {
        return Edad;
}
unsigned int Gato::getPeso() {
        return Peso;
}
void Gato::setEdad(unsigned int edad) {
        Edad = edad;
}
void Gato::setPeso(unsigned int peso) {
        Peso = peso;
}
```

Clases: variables que son otras clases

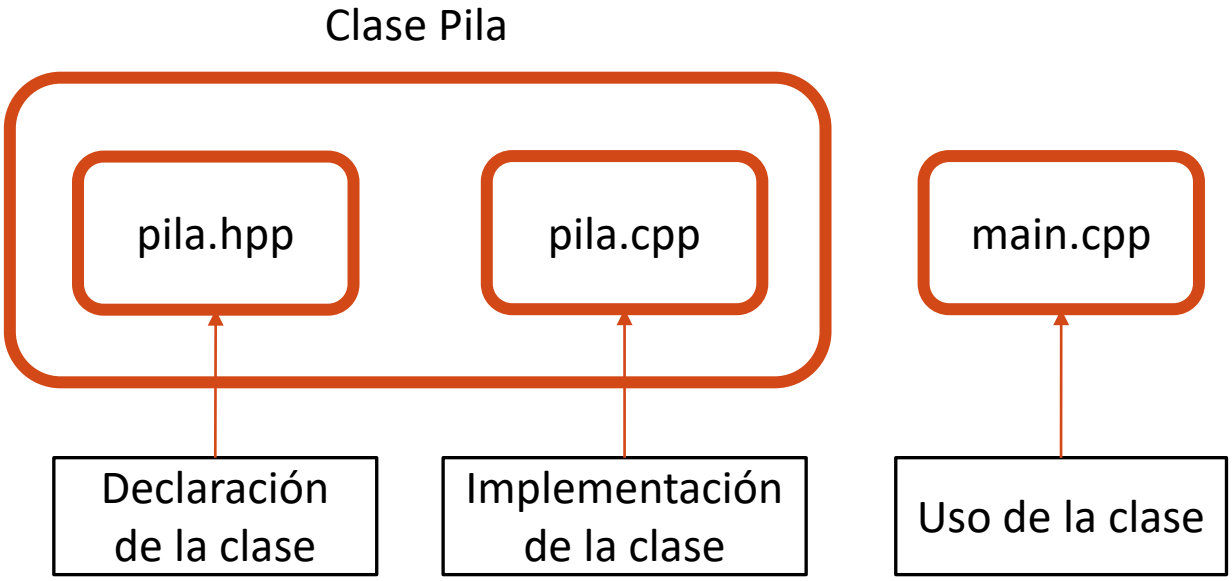
- Nada impide construir una clase utilizando otras clases definidas previamente

```
class Punto {
        public:
                void setX(int x);
                void setY(int y);
                int getX();
                int getY();
        private:
                int X;
                int Y;
};
```

```
class Rectangulo {
        public:
                Rectangulo (Punto arriba_izquierda, Punto abajo_derecha);
                ~Rectangulo();
        private:
                Punto Arriba_Izquierda;
                Punto Abajo_Derecha;
};
```

Índice de contenidos

Ejemplo: Clase Pila



Ejemplo: Clase Pila

```
class CPila {
        private:
                int tam; // tamaño de la pila
                char* ppila; // puntero a la pila
                int top; // última posición ocupada + 1
        public:
                CPila(const int t=20); // constructor
                CPila(const CPila& pila); // constructor copia
                ~CPila(); // destructor
                // operador de asignación:
                CPila& operator=(const CPila& pila);
                void poner_en_pila(char c); // introduce un carácter en la pila
                char sacar_de_pila(); // extrae un carácter de la pila
                bool vacia(); // comprueba si la pila está vacía
                bool llena(); // comprueba si la pila está llena
                // Función externa, amiga de la clase:
                friend bool pilas_iguales(const CPila& s1, const CPila& s2);
};
```

Índice de contenidos

Sobrecarga de los operadores

- Los operadores del lenguaje (+ - * / ++ ...) se pueden redefinir para tratar tipos no estándar
- Ejemplo:
  - Definimos un nuevo tipo denominado complejo y queremos usar expresiones del tipo a+b donde a y b son complejos

```
complejo operator+(const complejo& oper1, const complejo& oper2) {
        complejo resultado;
        resultado.real = oper1.real + oper2.real;
        resultado.imaginaria = oper1.imaginaria + oper2.imaginaria;
        return resultado;
}
```

- Los operadores **. : # .* :: ?:** no se pueden sobrecargar

Índice de contenidos

El puntero this

- C++ dispone de un puntero predefinido llamado '**this**'
- Es solo accesible a los métodos miembro de las clases
- Apunta al objeto desde el cual se llamó a la función (método)

```
CPila& CPila::operator=(const CPila& pila) {
        int i;
        if (tam!=pila.tam) // equivale a "if (this->tam!=pila.tam)"
                cout << "No se pueden asignar pilas de tamaño diferente";
        else {
                top = pila.top; // equivale a "this->top = pila.top;"
                for(i=0;i<top;i++) ppila[i] = pila.ppila[i];
        }
        return *this;
}
```

Índice de contenidos

Funciones inline

- El modificador "**inline**" hace que el compilador optimice la velocidad de ejecución de funciones pequeñas repitiendo el código en cada llamada
- Es equivalente a definir la función dentro del cuerpo de la clase

```
inline CPila::CPila(const int t) {
        ppila = new char[t];
        tam = t;
        top = 0;
}
```

Funciones y clases friend

- Funciones amigas:
  - Permiten que una función "global", externa a la clase, pueda acceder a los miembros de una clase como si fuera un método de la misma
  - Ejemplo, función que compara dos pilas:
    - Debe poder acceder a los miembros de la clase, pero estos son privados

```
bool pilas_iguales (const CPila& s1, const CPila& s2) {
        if (s1.top != s2.top) return false;
        for (int i=0; i < s1.top; i++)
                if (s1.ppila[i] != s2.ppila[i]) return false;
        return true;
}
```

    - Para solucionarlo se puede declarar amiga de la clase pila

```
class CPila {
        …
        public:
                friend bool pilas_iguales(const CPila& s1, const CPila& s2);
};
```

Funciones y clases friend

- Clases amigas:
  - Permite que una clase pueda acceder a los miembros de otra (clases sin ninguna relación de herencia)

```
class item {
        private:
                int data;
        public:
                // ...
                friend class ConjuntoItems;
};

class ConjuntoItems {
        ..........................
        // puede acceder a miembros data de la clase item;
};
```

Índice de contenidos

Métodos y miembros const

- Una función de una clase (método) se puede declarar constante (const)
- Esto significa que si la función modifica el objeto para el que es invocada el compilador dará un mensaje de error

```
class CPila {
        private:
                ....
                int top; // última posición ocupada + 1
        public:
                .......
                bool vacia() const; // no puede modificar los atributos
                bool llena() const; // no puede modificar los atributos
};
```

Métodos y miembros const

- Los miembros de una clase pueden ser constantes
- Se pueden inicializar únicamente cuando se crean
- La inicialización tiene una sintaxis especial

```
class CPila {
        private:
                ..............
                const int tam; // última posición ocupada + 1
        public:
                CPila(const t=20);// constructor
                ..............
};
CPila::CPila(const int t) : tam(t) {
        ppila = new char[t];
        top = 0;
}
```

Métodos y miembros static

- C++ permite definir tanto datos como métodos comunes a todos los objetos de una clase (abstractos)
- Una variable estática tiene duración estática (existe mientras existe el programa) y se inicializa a 0 a menos que se especifique lo contrario. Dicha variable se comparte entre todas las instancias de la clase
- Un método estático solo puede acceder a otros datos y métodos que también sean estáticos

Métodos y miembros static

- Ejemplo: Contador de instancias de CPila

```
class CPila {
        private:
                ....................
                static int contador;
        public:
                ....................
};
```

```
CPila::CPila(const int t) {
        ppila = new char[t];
        top = 0;
        contador++;
}

CPila::~CPila() {
        delete [] ppila;
        contador--;
}

// Inicialización
int CPila::contador = 0;
```

Métodos y miembros static

- Ejemplo: El contador de la clase CPila es privado, por lo tanto no puede ser accedida desde el exterior de la clase. Si queremos preguntar cuantas pilas hay, sin necesidad de disponer de un objeto, hay que crear un método estático

```
class CPila {
private:
        .................
        static int contador;
public:
        .................
        int numero_pilas() {return contador;}
        // requiere que haya objeto
};
```

```
class CPila {
private:
        .................
        static int contador;
public:
        .................
        static int numero_pilas() {return contador;}
        // No requiere que haya objeto
};
```

```
cout << "Número de pilas: " << CPila::numero_pilas() << endl;
```

Índice de contenidos

Espacios de nombres

- El C clásico adolece de algunos problemas relativos al manejo de identificadores
- Cuando el programa alcanzaba un gran tamaño, empezaban a presentarse problemas de colisión de los nombres asignados a los objetos
- Esto se puede resolver con C++
  - C++ es muy adecuado para la programación de grandes sistemas
  - Permite separar los nombres de clases, variables y funciones en espacios de nombres

Espacios de nombres

- Los espacios de nombres son posibles porque C++ permite dividir el espacio total de los identificadores de los objetos del programa (lo que se denomina el espacio general de nombres) en subespacios distintos e independientes
  - Dispone de una palabra clave específica: **namespace**

- En síntesis, el proceso consiste en declarar un espacio de nombres asignándole un identificador y delimitándolo por un bloque entre llaves
- Dentro de este cuerpo pueden declararse los objetos correspondientes al mismo, después los objetos pueden accederse mediante diversas técnicas. La más directa mediante el operador ::, denominado precisamente por esto “de resolución de ámbito”

Espacios de nombres

- Ejemplo:

```
namespace ALPHA {
        ...
        long double LD;
        float f(float y) { return y; }
}
namespace BETA {
        ...
        long double LD;
        float f(float z) { return z; }
}

ALPHA::LD = 1.1; // Acceso a variable LD del espacio ALPHA
ALPHA::f(1.1);     // Acceso a función f de ALPHA
BETA::LD = 1.0;    // Acceso a variable LD del espacio BETA
BETA::f(1.1);      // Acceso a función f de BETA
BETA::X = 1 ;      // Error: Elemento X no definido en BETA
```

Espacios de nombres

- La biblioteca estándar de plantillas se encuentra localizada en el espacio de nombres std

```
#include <iostream>
int main() {
        cout << "Salida estándar incorrecta." << endl;
        std::cout << "Salida estándar correcta." << std::endl;
}
```

```
#include <iostream>
int main() {
        using namespace std;
        // A partir de aquí si alguna variable no se encuentra
        // se busca en el espacio de nombres std
        cout << "Salida estándar correcta." << endl;
}
```

Es posible y común ponerlo detrás de los #include

Espacios de nombres

- Existen cuatro formas de acceder a los elementos de un subespacio (veremos solamente una):
    1. Una declaración explícita
    2. La declaración using
    3. La directiva using
    4. Búsqueda en el espacio de los argumentos

- Las dos primeras se utilizan para accesos de elementos individuales. La tercera permite un cómodo acceso a todos los elementos de un subespacio. La última es un caso especial, y se refiere a un mecanismo de acceso automático implícito en todas las funciones C++

Índice de contenidos

Compilación en C++

- **g++** permite realizar y controlar las etapas de preproceso, compilación, ensamblado y enlazado necesarias para generar el código objeto de un programa
- La orden **g++** genera por defecto, como resultado de la compilación, un archivo con el nombre **a.out** con el código ejecutable
- Para especificar otro archivo destino se puede utilizar el flag **o**

```
g++ prueba.cpp -o ejecutable
```

  - devuelve, en caso de una compilación sin errores, un archivo ejecutable con el nombre **ejecutable**, resultado de compilar **prueba.c**

```
g++ main.cpp arch1.cpp arch2.cpp  -o salida
```

  - compila **main.cpp**, **arch1.cpp** y **arch2.cpp**, y los enlaza generando **salida**
- **g++** decide qué hacer con cada archivo (preproceso, enlazado, ...) dependiendo de la extensión del archivo origen: .cpp, .o, ...

Archivos fuente .cpp y cabecera .hpp

- Los archivos llamados **cabeceras (.hpp) que contienen los prototipos** de **las funciones y de las clases** que **se declaran en el archivo fuente (.cpp)** con mismo nombre
- Ahora hay que **incluir el archivo cabecera**, en aquellos archivos fuente que utilizan la función **(#include "archivocabecera.hpp"**)
  - Es posible incluir el archivo cabecera que contiene las correspondientes definiciones en su archivo fuente para que se compruebe si la declaración y su definición son consistentes
- Las cabeceras #ifndef / #define / #endif son utilizadas para evitar múltiples importaciones. Así, solo se importa una vez aunque haya múltiples archivos fuentes que lo hagan

Archivos fuente .cpp y cabecera .hpp

- Ejemplo:

Archivo Gato.hpp

```
#ifndef GATO_H
#define GATO_H

class Gato {
        private:
                int Edad;
                int Peso;

        public:
                void Maullar();
};

 #endif
```

Archivo Gato.cpp

```
#include "Gato.hpp"
#include <iostream>
using namespace std;

void Gato::Maullar() {
        cout << "MIAUUUU" << endl;
}
```

Archivo main.cpp

```
#include "Gato.hpp"

int main(void){
        Gato Silvestre;
        Silvestre.Maullar();
        return 0;
}
```

Compilar y enlazar archivos

- Para compilar el siguiente ejemplo podemos hacerlo como en C, de manera que se compilen y enlacen entre ellos:

```
g++ main.cpp Gato.cpp -o salida
```

- Como en C, la compilación se divide en dos pasos: uno de **compilación** y otro de **enlazado**. El paso de compilación crea código ensamblador y lo convierte a código objeto. Esto se hace con la opción **-c**:

```
g++ -c main.cpp
```

- Ahora para generar el archivo ejecutable, basta con ejecutar el paso de enlazado. Para ello, se llama al comando **g++** seguido de la lista de ficheros **.o** y con la opción **-o** damos nombre al ejecutable (**-Wall** nos muestra los warnings):

```
g++ -Wall main.o Gato.o   -o salida
```

Make

- La utilidad **make** permite almacenar las reglas de compilación y enlazado necesarias para generar los archivos de un proyecto, de modo que el proceso pueda ser automatizado
- **make** utiliza un archivo de texto (Makefile) que contendrá un conjunto de reglas que indican a **make** cómo crear cada uno de los componentes del proyecto. Antes del comando tiene que haber una tabulación

```
objetivo : dependencia
[tab] comando
[tab] comando
      [...]
```

- Las variables se nombran en mayúsculas al inicio (VARIABLE = …) y se utilizan poniendo $(VARIABLE):

Declaración:
VARIABLE = texto

Uso:
$(VARIABLE)

Ejemplo Makefile

```
# Directorio de archivos fuente
SRCDIR = ./

# "all" es el objetivo por defecto. Sus dependencias se comprueban cuando ejecute la orden "make"
en el directorio actual. Puede haber más de una dependencia.
all : salida

#### Reglas secundarias para compilar cada una de las dependencias.
salida : main.o Gato.o
	g++ -Wall main.o Gato.o -o salida

main.o: $(SRCDIR)main.cpp
	g++ -c $(SRCDIR)main.cpp

Gato.o: $(SRCDIR)Gato.cpp
	g++ -c $(SRCDIR)Gato.cpp

# "limpiar" es un objetivo falso. Sirve para borrar los archivos .o (objeto).
limpiar :
	rm *.o
```

Índice de contenidos

Ejercicios

1. Si declaramos una clase Gato con una variable privada Edad, y definimos dos objetos de tipo Gato (Silvestre y Tom), ¿puede Silvestre acceder a la variable miembro Edad de Tom?
2. Escribir el código que declare una clase llamada 'Empleado' con los datos miembro 'Edad', 'AnyosDeServicio' y 'Salario'
3. Reescribir la clase Empleado para hacer los datos miembro privados, y proporcionar métodos de acceso públicos para cambiar y obtener el valor de los datos miembro
4. Crea un método "**void imprime()**" que muestre los valores de Empleado
5. Escribir un programa con la clase Empleado que cree dos empleados. Modifica sus valores Edad, AnyosDeServicio y Salario, y utiliza la función anterior para mostrar sus valores

Ejercicios

5. Proporcionar un método "**void miles()**" a la clase Empleado que nos diga cuantos miles de euros gana el empleado, redondeado al mil más cercano. Pista: mira las funciones de la librería <cmath>
6. Cambiar la clase Empleado de forma que se puedan inicializar Edad, AnyosDeServicio y Salario cuando se cree el empleado
7. Crea un método "**void numeroEmpleados()**" que cuente cuántos empleados hay sin necesidad de un objeto de la clase Empleado
8. Replica el código anterior usando punteros
9. Divide el programa en 3 archivos y compila con un Makefile: (CEmpleado.hpp, CEmpleado.cpp, main.cpp)

Universidad Rey Juan Carlos
Escuela Técnica Superior Ingeniería de Telecomunicación

Programación de Sistemas de Navegación

2.2. Lenguaje C++

Constructor, destructor, copia, sobrecarga de operadores

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

Índice de contenidos

Constructores y destructores

- Métodos especiales:
  - **Constructor**: se ejecuta cuando se crea el objeto. Atendiendo a su naturaleza, puede ser:
    - Constructor por defecto
    - Constructor parametrizado
    - Constructor de copia
  - **Destructor**: se ejecuta cuando el objeto desaparece
  - **Operadores de asignación**

  - Si no indicamos constructor o destructor, el compilador incluirá uno por defecto que no toma parámetros y que no hace nada

Constructores y destructores

- **Constructor:**
  - El constructor es una función con el mismo nombre que la clase

```
clase::clase(parametros);
```

- **Destructor:**
  - El destructor tiene el mismo nombre que la clase pero con el símbolo ~ delante
  - No acepta argumentos
  - No devuelven un valor (o void)
  - Se pueden declarar como virtual

```
clase::~clase();
```

- Usaremos el **constructor** para llevar a cabo las inicializaciones que el objeto requiera al construirse, y el **destructor** para lo contrario (liberar memoria, por ejemplo) cuando el objeto se destruya

Índice de contenidos

Constructor copia

- El constructor de copia es usado (entre otras cosas) cada vez que hace falta crear un objeto temporal:

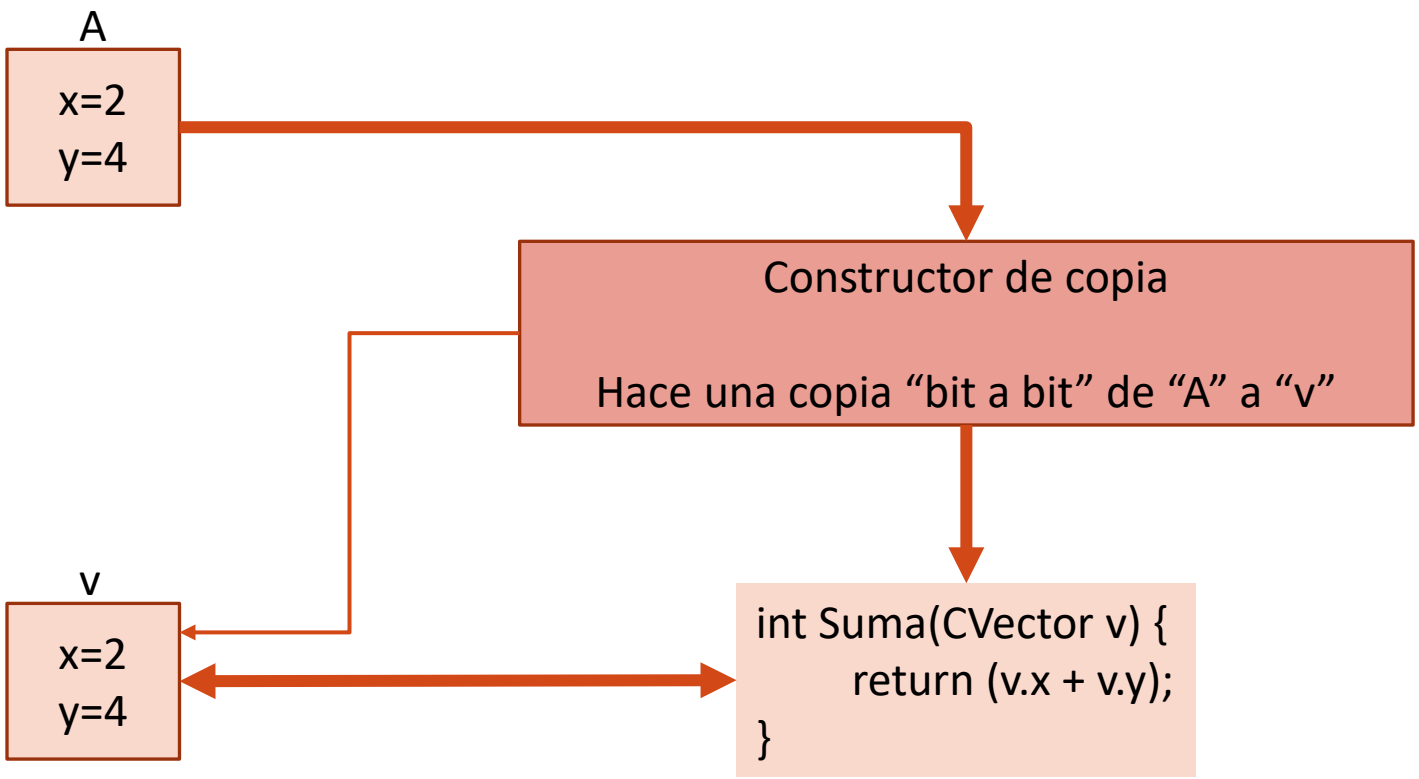
```
class CVector {
        public:
                int x, y;
};
```

```
int Suma(CVector v) {
        return (v.x + v.y);
}
```

- Cada vez que se hace una llamada a "Suma" se crea un objeto temporal "v" que es copia del objeto pasado como parámetro inicialmente

```
CVector A;
A.x=2; A.y=4;
int valor = Suma(A);
// se crea una copia de "A" que se usa dentro
// de la función "Suma", por ser paso por valor
```

Constructor copia



Constructor copia

- Cuando no se especifica otra cosa, el compilador genera un "constructor de copia" genérico que duplica "bit a bit" el objeto que se pasa para obtener una copia idéntica del mismo en otra posición de memoria
- Sin embargo, dicho constructor de copia genérico no funciona "siempre", hay que estudiar en cada caso si necesitamos redefinir dicho constructor por uno hecho por nosotros
- Un ejemplo típico es el de objetos que usan memoria dinámica (aunque pueden darse más casos)

Constructor copia

- Supongamos la siguiente clase:

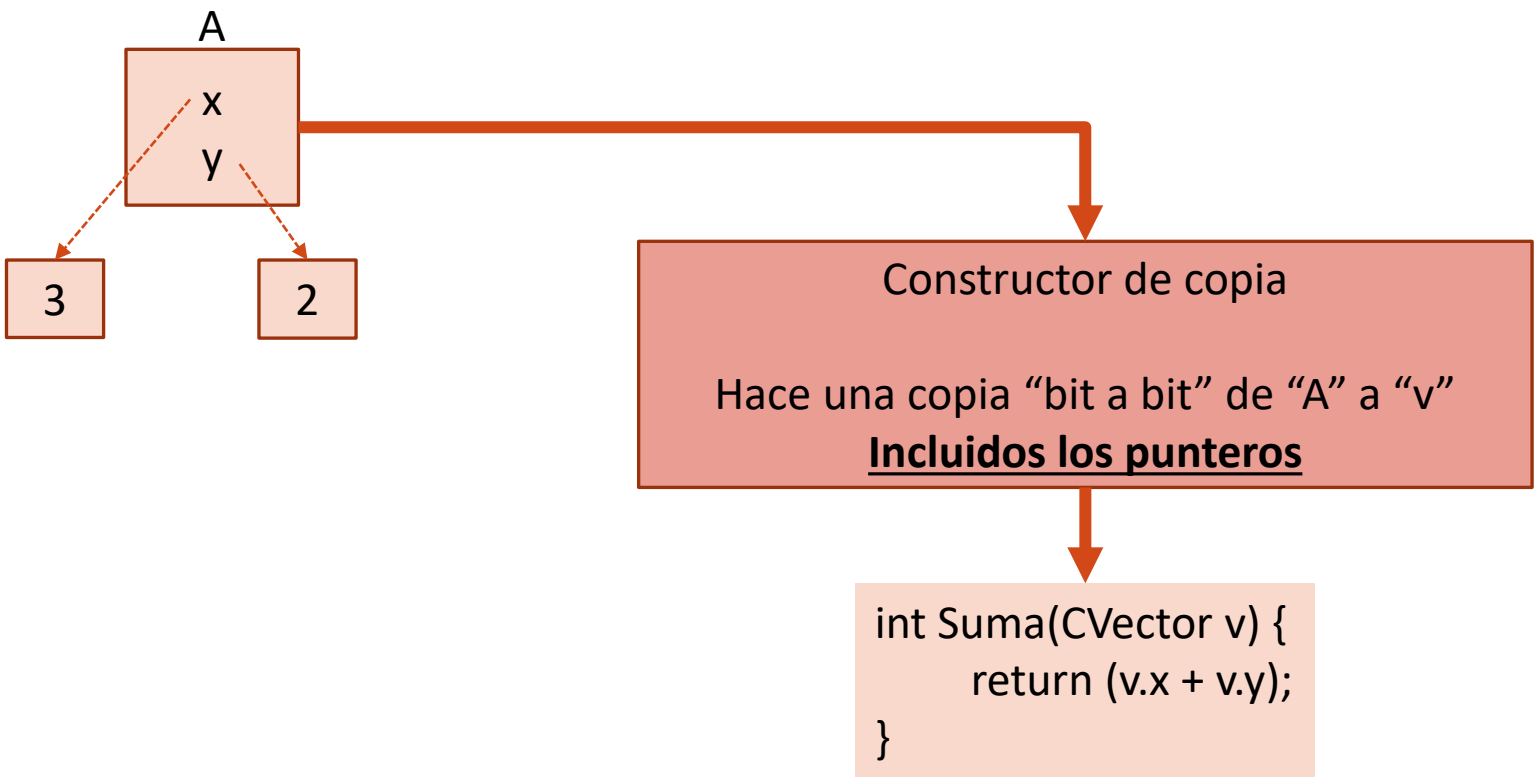
```
class CVector {
        public:
                int *x, *y;

                CVector() {
                        x = new int;
                        y = new int;
                }

                ~CVector() {
                        delete x;
                        delete y;
                }
};
```

Constructor copia

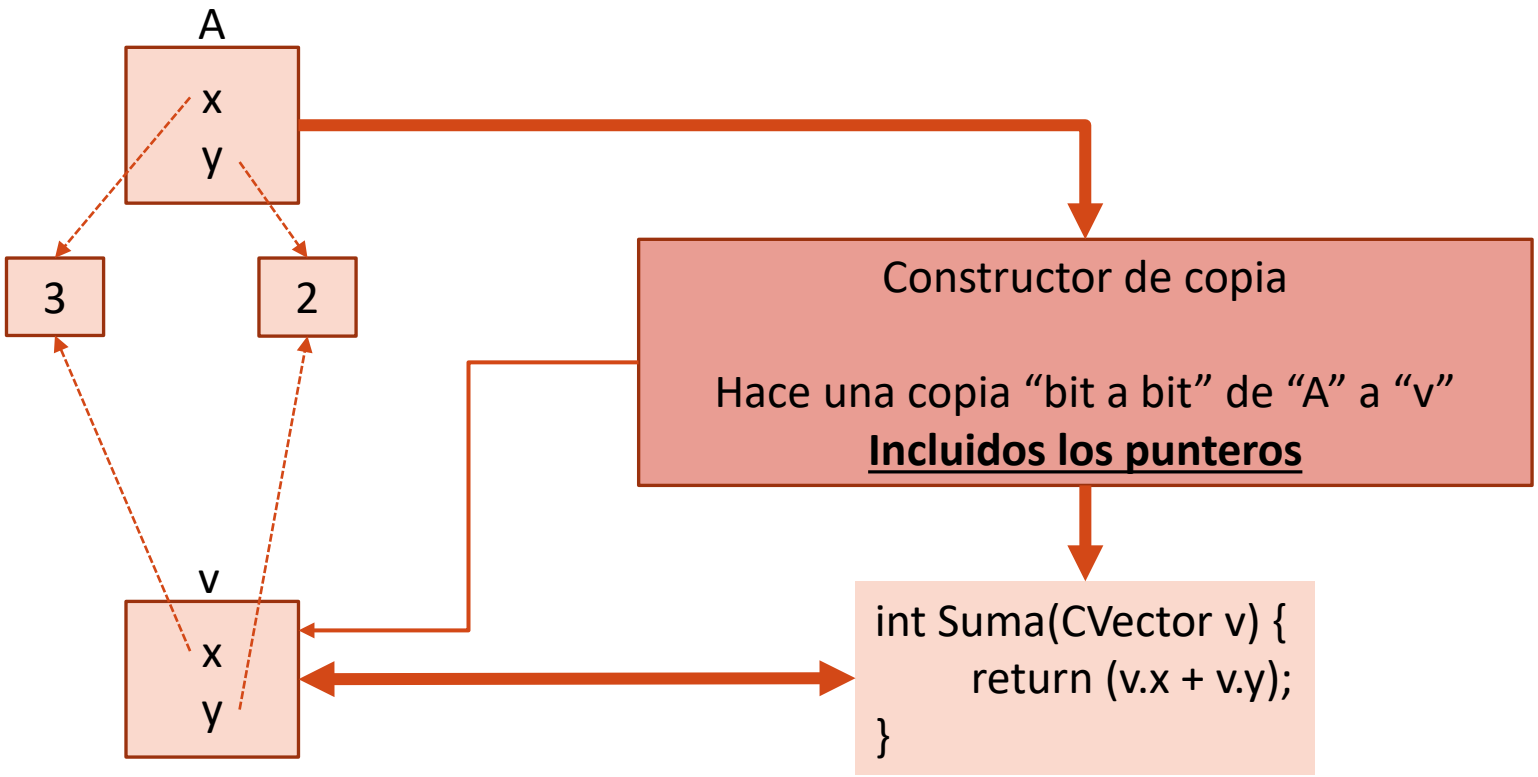
- En este caso una llamada a la función “Suma” tiene efectos catastróficos:

```
CVector A;
*(A.x)=2;
*(A.y)=5;
int x = Suma(A);
```

Constructor copia



Constructor copia



Constructor copia

Al retornar "Suma", se ejecuta el destructor de "v" y BORRA LOS DATOS DEL OBJETO ORIGINAL

```
int Suma(CVector v) {
        return (v.x + v.y);
}
```

Constructor copia

- Como se ve en el ejemplo, destruir un objeto que se hizo copiando otro con el constructor de copia original puede no ser válido en absoluto
- En estos casos, el programador debe proporcionar el constructor "correcto"
- El constructor de copia se distingue del constructor por omisión en que recibe como parámetro un objeto de su mismo tipo por referencia. En el ejemplo:

```
CVector (CVector& objeto);
```

Constructor copia

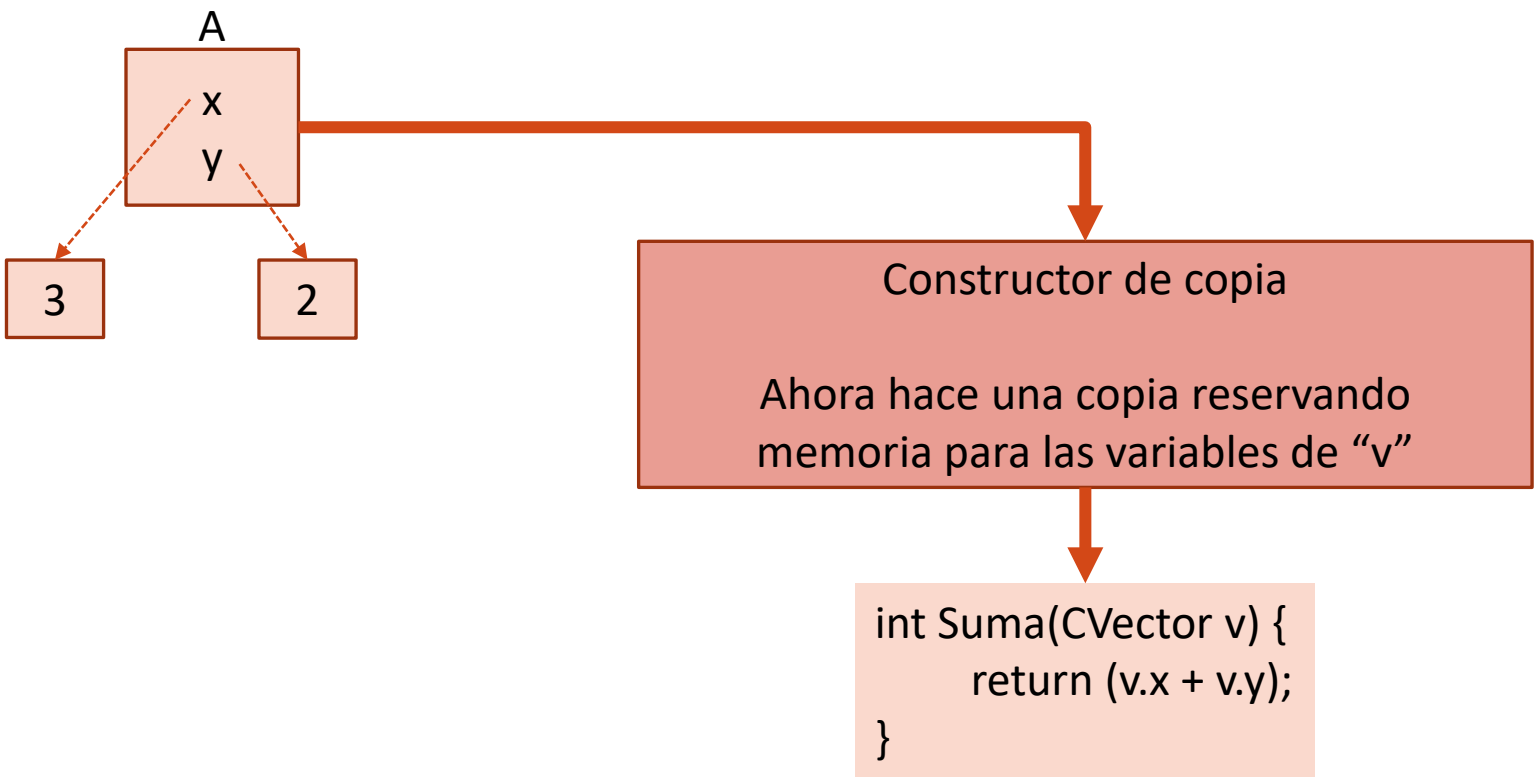
- Supongamos la siguiente clase:

```
class CVector {
        public:
                int *x, *y;
                CVector() { x = new int; y = new int; }
                ~CVector() { delete x;  delete y; }

                CVector (CVector& original) {
                        x = new int;
                        y = new int;
                        *x = *(original.x);
                        *y = *(original.y);
                }
};
```

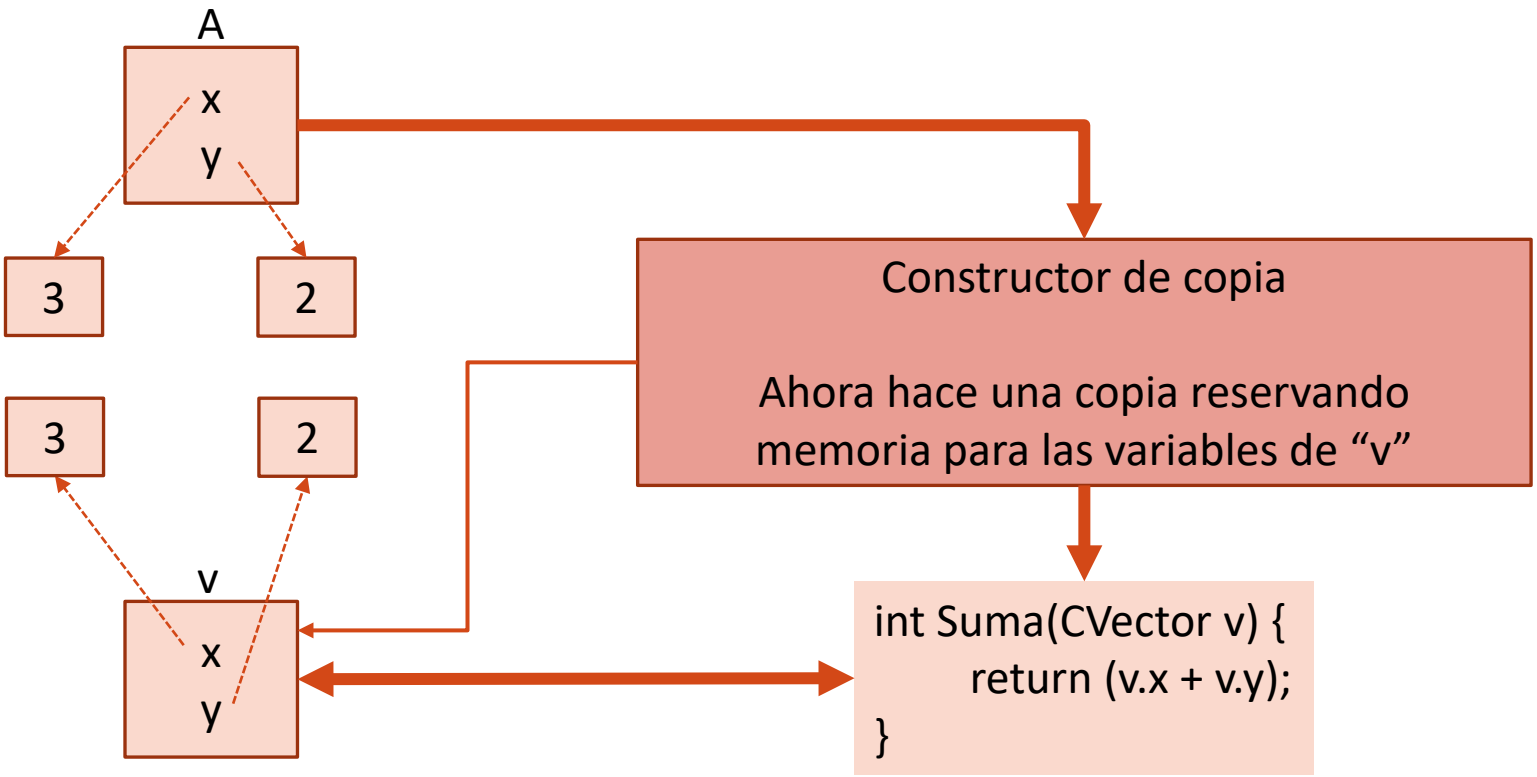
Constructor copia

- El constructor de copia ahora funciona correctamente, copia la información del objeto original en la copia teniendo en cuenta que hay que reservar memoria para los dos valores de tipo int
- De esta manera, el destructor borrará la memoria asignada para el objeto copia, NO la memoria del original

Constructor copia



Constructor copia



Constructor copia

Al retornar "Suma", se ejecuta el destructor de "v" y BORRA SUS DATOS RESPETANDO LOS DEL OBJETO ORIGINAL

```
int Suma(CVector v) {
        return (v.x + v.y);
}
```

Constructor copia

- Por último, recordar que el constructor de copia también sirve para inicializar objetos en tiempo de declaración:

```
CVector A;
A.x = 7;
A.y = 9;
CVector B(A);  // Se copia en B el objeto A
```

Índice de contenidos

Sobrecarga de los operadores

- C++ permite cambiar la funcionalidad actual de algunos operadores dentro de su clase
- La sobrecarga de operadores es el método por el cual podemos cambiar la función de algunos operadores específicos para realizar una tarea diferente

```
tipo_devuelto Nombre_Clase :: operator op (argumentos) {
        // Nueva funcionalidad
}
```

- Hay varios tipos:
  - Sobrecarga de un **operador unario** (aquellos que poseen solo una parte, o1++) con y sin funciones miembro (**friend**)
  - Sobrecarga de un **operador binario** (aquellos que poseen dos partes, izquierda y derecha, o1+o2) con y sin funciones miembro (**friend**)

Sobrecarga de los operadores

- Hay algunos criterios que hay que tener en cuenta:
  - En el caso de una función que no sea **static**, el operador binario debe tener solo un argumento y el unario no debe tener argumentos
  - En el caso de una función **friend**, el operador binario debe tener dos argumentos y el unario solo uno
  - Todo miembro de la clase debe ser público (**public**) si se implementa una sobrecarga del operador
  - Los operadores **= () [] ->** no se pueden sobrecargar cuando se declaran como funciones **friend**
  - Los operadores **. : # .* :: ?:** no se pueden sobrecargar
- Los **argumentos** en la sobrecarga del operador se pasan solo por **referencia**, no funcionará si los argumentos se pasan por valor, porque se pasa una copia del objeto

Sobrecarga: operador unario

- En la función de operador unario, no se deben pasar argumentos. Funciona solo con objetos de una clase. Es una sobrecarga de un operador que opera en un solo operando
- Ejemplo sobrecargar (-) :

```
int main()  {
        // Creamos un rectángulo con
        // base 3 y altura 2
        Rectangulo rect(3, 2);

        // Queremos usar el operado (-)
        // para restar en uno su base y altura
        -rect;
        return 0;
}
```

Sobrecarga: operador unario

```
class Rectangulo {
	public:
		// Miembros del objeto
		int base, altura;

		// Constructor parametrizado
		 Rectangulo(int b, int a)  {
			this -> base = b;
			this -> altura = a;
		}

		// Sobrecarga del operador (-) para decrementar el rectangulo
		void operator-() {
			base--;
			altura--;
			cout << "\nDecremento base y altura: " << base << " | " << altura;
		}
};
```

Sobrecarga: operador unario

- En el ejemplo anterior, muestra que no se pasa ningún argumento y no se devuelve ningún valor (return) porque el operador unario funciona en un solo operando. El operador (-) cambia la funcionalidad a su función miembro
- Hay que tener en cuenta que algo así como **rect2 = -rect** no funcionará, porque **operator-()** no devuelve ningún valor

Sobrecarga: operador binario

- Para la sobrecarga de un operador binario, debe haber un argumento ya que se opera con dos operandos
- Si tenemos en cuenta el ejemplo anterior y se quiere realizar la sobrecarga del operador (+), sería:

```
class Rectangulo {
        public:
                int base, altura;
                Rectangulo() {base = 0; altura = 0;} // Constructor por defecto
                // Sobrecarga del operador (+) para sumar dos rectangulos
                Rectangulo operator+(const Rectangulo& r) {
                        Rectangulo resultado;
                        resultado.base = this -> base + r.base; // igual a base + r.base
                        resultado.altura = this -> altura + r.altura; // igual a altura + r.altura
                        return resultado;
                }
};
```

Sobrecarga: operador binario

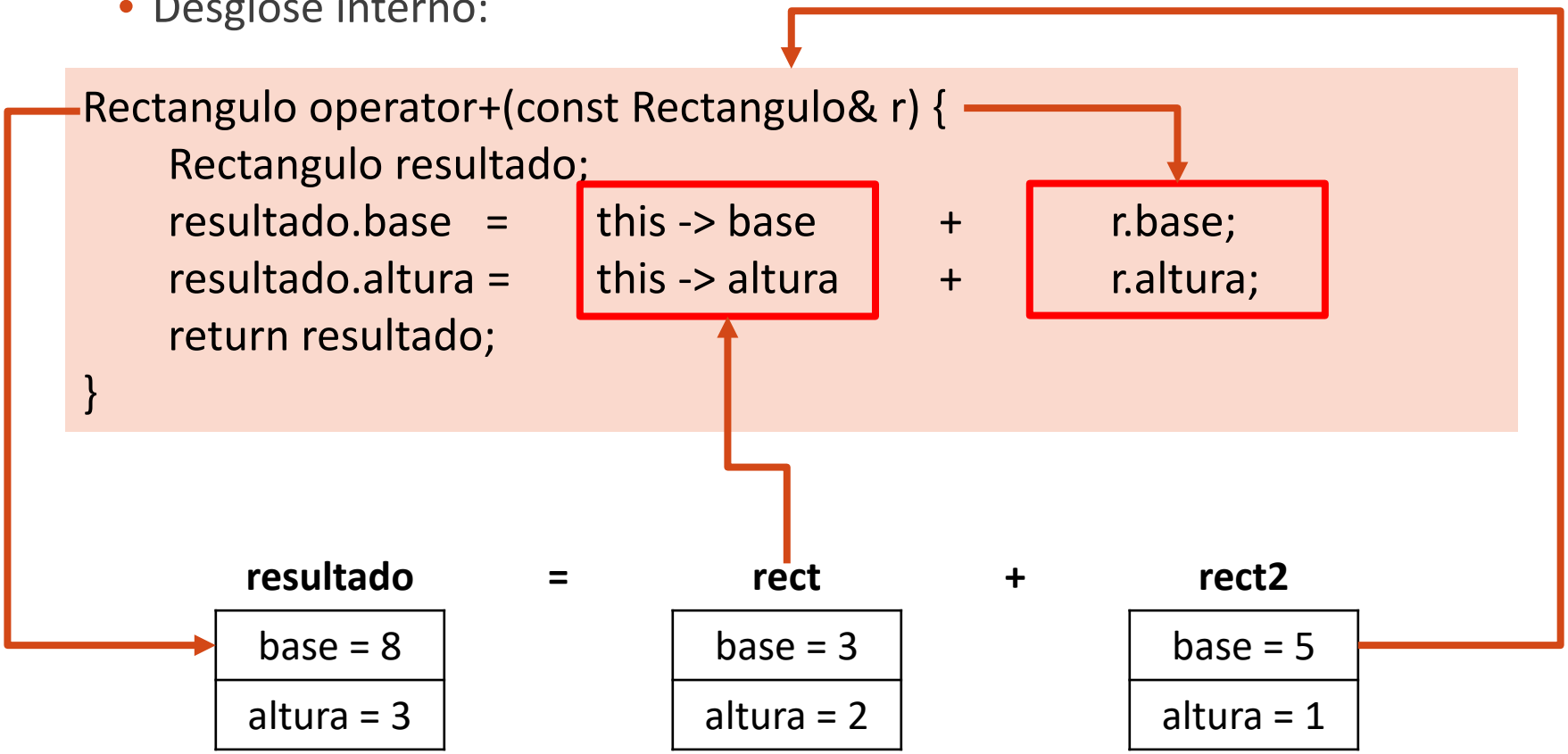
- Ahora se podría hacer la suma de dos rectángulos:

```
int main()  {
        // Creamos dos rectángulos uno con base 3
        // y altura 2; y otro con base 5 y altura 1
        Rectangulo rect(3, 2);
        Rectangulo rect2(5, 1);

        // Nuevo rectángulo
        Rectangulo resultado;

        // Sumamos los dos rectángulos
        resultado = rect + rect2;
        return 0;
}
```

Sobrecarga: operador binario

- Desglose interno:

```
Rectangulo operator+(const Rectangulo& r) {
    Rectangulo resultado;
    resultado.base   =  this -> base   +  r.base;
    resultado.altura =  this -> altura +  r.altura;
    return resultado;
}
```



Sobrecarga: función friend

- En este enfoque, la función de sobrecarga del operador debe preceder con la palabra clave **friend**
- Hay que tener en cuenta que la función de operador **friend** toma **dos parámetros en un operador binario**, y **uno en un operador unario**. Todo el trabajo y la implementación serían lo mismo que la función de operador binario, excepto que esta función se implementará fuera del alcance de la clase

```
class Rectangulo {
        public:
                ...
                friend Rectangulo operator+(const Rectangulo& r1, const Rectangulo& r2);
};
Rectangulo operator+(const Rectangulo& r1, const Rectangulo& r2) {
        Rectangulo resultado;
        resultado.base = r1.base + r2.base;
        resultado.altura = r1.altura + r2.altura;
        return resultado;
}
```

Resumen: sobrecarga unarios

- Sin funciones miembro:

Alumno.hpp

```
class Alumno {
        private: int edad = 0;
        public:
                friend const Alumno& operator+(const Alumno& a);
                friend const Alumno operator-(const Alumno& a);
                friend const Alumno operator~(const Alumno & a);
                friend Alumno* operator&(Alumno& a);
                friend Alumno operator!(const Alumno& a);
                // Sin argumento const:
                friend const Alumno& operator++(Alumno& a); // Pre:
                friend const Alumno operator++(Alumno& a, int); // Post:
                friend const Alumno& operator--(Alumno& a); // Pre:
                friend const Alumno operator--(Alumno& a, int); // Post:
};
```

Resumen: sobrecarga unarios

- Sin funciones miembro:

Alumno.cpp

```
const Alumno operator-(const Alumno& a) {
        cout << "-Alumno\n";
        return Alumno(-a.edad);  // Constructor parametrizado
}
const Alumno& operator++(Alumno & a) { // Pre: ++Alumno
        cout << "++Alumno\n";
        a.edad++;
        return a;
}
const Alumno operator++(Alumno & a, int) { // Post: Alumno++
        cout << "Alumno++\n";
        Alumno a_aux(a.edad);
        a.edad++;
        return a_aux;
}
```

main.cpp

```
int main() {
        Alumno a(20);    a.print();
        Alumno b = -a;   b.print();
        Alumno c = a++;  c.print();
        Alumno d = ++a;  d.print();
        return 0;
}
```

```
20
-Alumno: -20
Alumno++: 20
++Alumno: 22
```

Resumen: sobrecarga unarios

- Con funciones miembro (uso del **this**):

Alumno.hpp

```
class Alumno {
        private: int edad = 0;
        public:
                const Alumno& operator+() const;
                const Alumno operator-() const;
                const Alumno operator~() const;
                Alumno* operator&() const;
                Alumno operator!() const;
                // Sin funciones miembro const:
                const Alumno& operator++(); // Pre:
                const Alumno operator++(int); // Post:
                const Alumno& operator--(); // Pre:
                const Alumno operator--(int); // Post:
};
```

Resumen: sobrecarga unarios

- Con funciones miembro:

Alumno.cpp

```
const Alumno Alumno::operator-() const {
        cout << "-Alumno: ";
        return Alumno(-this->edad);
}
const Alumno& Alumno::operator++() { // Pre: ++Alumno
        cout << "++Alumno: ";
        this -> edad++;
        return *this;
}
const Alumno Alumno::operator++(int) { // Post: Alumno++
        cout << "Alumno++: ";
        Alumno a_aux(this -> edad);
        this -> edad++;
        return a_aux;
}
```

main.cpp

```
int main() {
        Alumno a(20);    a.print();
        Alumno b = -a;   b.print();
        Alumno c = a++;  c.print();
        Alumno d = ++a;  d.print();
        return 0;
}
```

```
20
-Alumno: -20
Alumno++: 20
++Alumno: 22
```

Resumen: sobrecarga binarios

- Sin funciones miembro:

Alumno.hpp

```
class Alumno {
        private: int edad = 0;
        public:
                // Crean un valor nuevo: + - * / % ^ & | << >>
                friend const Alumno operator+(const Alumno& left, const Alumno& right);

                // Modifican y devuelven un valor:  += -= *= /= %= ^= &= |= >>= <<=
                friend Alumno& operator+=(Alumno& left, const Alumno& right);

                // Operadores condicionales (true/false): == != < > <= >= && ||
                friend int operator==(const Alumno& left, const Alumno& right);
};
```

Resumen: sobrecarga binarios

- Sin funciones miembro:

Alumno.cpp

```
const Alumno operator+(const Alumno & left,  const Alumno & right) {
        cout << "+: " ;
        return Alumno(left.edad + right.edad);
}
Alumno& operator+=(Alumno& left, const Alumno& right) {
        cout << "+=: " ;
        if(&left == &right) {/* autoasignación */}
        left.edad += right.edad;
        return left;
}
int operator==(const Alumno& left, const Alumno& right) {
         cout << "==: " ;
        return left.edad == right.edad;
}
```

main.cpp

```
int main() {
        Alumno a(20); a.print();
        Alumno b(23); b.print();
        Alumno c = a + b; c.print();
        b += a;  b.print();
        cout << (a == b) << endl;
        cout << (b == c) << endl;
        return 0;
}
```

```
20
23
+: 43
+=: 43
==: 0
==: 1
```

Resumen: sobrecarga binarios

- Con funciones miembro (uso del **this**):

Alumno.hpp

```
class Alumno {
        private: int edad = 0;
        public:
                // Crean un valor nuevo: + - * / % ^ & | << >>
                const Alumno operator+(const Alumno& right);

                // Modifican y devuelven un valor:  += -= *= /= %= ^= &= |= >>= <<=
                Alumno& operator+=(const Alumno& right);

                // Operadores condicionales (true/false): == != < > <= >= && ||
                int operator==(const Alumno& right);
};
```

Resumen: sobrecarga binarios

- Con funciones miembro:

Alumno.cpp

```
const Alumno Alumno::operator+(const Alumno & right) {
        cout << "+: " ;
        return Alumno(this -> edad + right.edad);
}
Alumno& Alumno::operator+=(const Alumno& right) {
        cout << "+=: " ;
        if(this == &right) {/* autoasignación */}
        this -> edad += right.edad;
        return *this;
}
int Alumno::operator==(const Alumno& right) {
         cout << "==: " ;
        return this -> edad == right.edad;
}
```

Podemos quitar this ->

main.cpp

```
int main() {
        Alumno a(20); a.print();
        Alumno b(23); b.print();
        Alumno c = a + b; c.print();
        b += a;  b.print();
        cout << (a == b) << endl;
        cout << (b == c) << endl;
        return 0;
}
```

```
20
23
+: 43
+=: 43
==: 0
==: 1
```

Índice de contenidos

Ejercicios

1. Crea una clase **Complejo** que permita trabajar con números complejos (parte real y parte imaginaria). Incluye los siguientes métodos: constructores (por defecto y parametrizado), accesores, mutadores, y sobrecarga los siguientes operadores: suma, resta, multiplicación, división y cout (operador <<: ver iostream)

2. Crea una clase **Rectangulo** que modele rectángulos por medio de cuatro **puntos** (vértices). Dispondrá de dos constructores: uno que cree un rectángulo partiendo de sus cuatro vértices y otro que cree un rectángulo partiendo de la base y la altura, de forma que su vértice inferior izquierdo esté en (0,0). La clase también incluirá un método para calcular la superficie, otro que desplace el rectángulo en el plano (± x, ± y) unidades y otro que pinte el rectángulo

Ejercicios

3. Crea las siguientes clases (cada una en su archivo):
   - **Motor**: con métodos para arrancar el motor y apagarlo
   - **Rueda**: con métodos para inflar la rueda y desinflarla
   - **Ventana**: con métodos para abrirla y cerrarla
   - **Puerta**: con una ventana y métodos para abrir / cerrar la puerta y abrir / cerrar la ventana
   - **Coche**: con un motor, cuatro ruedas y dos puertas; con los métodos que te parezcan adecuados para manipular cada uno de sus componentes
4. Compila todos los ejercicios anteriores con un **Makefile**

Universidad Rey Juan Carlos
Escuela Técnica Superior Ingeniería de Telecomunicación

Programación de Sistemas de Navegación

2.3. Lenguaje C++

Herencia, Polimorfismo

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

Índice de contenidos

Herencia

- Mecanismo exclusivo y fundamental de la POO. La herencia no está contemplada en la programación basada en tipos (TAD)
- Es el principal mecanismo que ayuda a fomentar y facilitar la reutilización del software: Las clases como componentes software reutilizables
- Si se necesita una nueva clase de objetos y se detectan suficientes similitudes con otra clase ya desarrollada, se toma esa clase existente como punto de partida para desarrollar la nueva:
  - Se adoptan automáticamente características ya implementadas:
    - Ahorro de tiempo y esfuerzo
  - Se adoptan automáticamente características ya probadas:
    - Menor tiempo de prueba y depuración

Herencia

- Base de la aplicación del mecanismo de herencia:
  - Suficientes similitudes: Todas las características de la clase existente (o la gran mayoría de ellas) resultan adecuadas para la nueva
  - En la nueva clase se ampliará y/o redefinirá el conjunto de características
- Características de las clases que se adoptan: Todos los miembros definidos en las clases
  - Atributos
  - Métodos
- Dependiendo de la forma en que se aplique el mecanismo de herencia, en la nueva clase se puede tener o no acceso a ciertas características heredadas (a pesar de que se adopten todas)

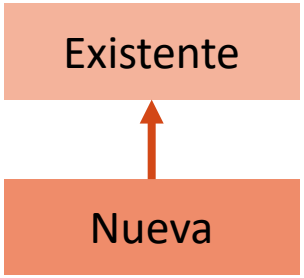
Herencia

- La relación de herencia se establece entre una nueva clase (referida aquí con el nombre **Nueva**) y una clase ya existente (referida aquí con el nombre **Existente**)
- Un poco de terminología:
  - **Existente** se dice que es la clase **base**, la clase madre o la superclase (término genérico de la POO)
  - **Nueva** se dice que es la clase **derivada**, la clase hija o la subclase (término genérico de la POO)
  - También se utiliza el término derivación para referirse a la herencia
  - La clase **Nueva** es la que tiene establecida la relación de herencia con la clase **Existente**; **Existente** no necesita a **Nueva**, pero **Nueva** sí necesita la presencia de **Existente**

Herencia

- La relación de herencia se establece entre una clase **Nueva** y una clase **Existente**
- Sobre la clase que hereda de la existente:
  - **Nueva** hereda todas las características de **Existente**
  - **Nueva** puede definir características adicionales
  - **Nueva** puede redefinir/anular características heredadas de **Existente**
  - El proceso de herencia no afecta de ninguna forma a la superclase **Existente**
- Esto se hace colocando dos puntos (**:**) y el nombre de la superclase entre el nombre de la subclase y la llave de apertura de la estructura class

```
class Subclase : Superclase {
        ...
};
```



Herencia: public

- C++ contempla distintas maneras de aplicar la herencia
- En el modo de herencia **pública**, se usa la palabra reservada **public** precediendo al nombre de la superclase. Los miembros heredados mantienen su modo de acceso:
  - Los que son **públicos** en la superclase **siguen siendo públicos** en la subclase
  - Los que son **privados** en la superclase **siguen siendo privados** en la subclase
- En los métodos de la subclase **NO se tiene acceso** a los **atributos y métodos privados** heredados
- A los objetos de la subclase se les puede seguir pasando mensajes que corresponden a métodos públicos heredados, es el modo más habitual:

```
class Subclase : public Superclase {
        ...
};
```

Herencia: private

- En el modo de herencia **privada** es el **predeterminado** si no se indica explícitamente otro modo, se usa la palabra reservada **private** precediendo al nombre de la superclase
- Todos los miembros heredados se convierten en privados:
  - Los que son **públicos** en la superclase **pasan a ser privados** en la subclase, pero no se ocultan y se pueden acceder en ésta
  - Los que son **privados** en la superclase **siguen siendo privados** en la subclase y se ocultan, resultando **inaccesibles** en ésta
- En los métodos de la subclase no se tiene acceso a los campos privados heredados, pero sí a los campos públicos heredados, aunque se hayan convertido en privados
- A los objetos de la subclase no se les puede pasar mensajes que correspondan a métodos públicos heredados (ahora son privados en la subclase)

```
class Subclase : private Superclase {
        ...
};
```

=

```
class Subclase : Superclase {
        ...
};
```

Herencia: ejemplo

- La siguiente clase Persona representa a una persona cualquiera:

Persona.hpp

```
class Persona {
        private:
                string nif, nombre;
                int edad;
        public:
                Persona(string nif = "", int edad = 0, string nombre = "");
                Persona(const Persona&);
                Persona& operator=(const Persona&);
                ~Persona();
                void setNif(string cad);  void setEdad(int num);  void setNombre(string cad);
                string getNif() const;  int getEdad() const;  string getNombre() const;
                void mostrar() const;
                string nombreCompleto() const;
                void saludo();
                void leerDatos();
};
```

Herencia: ejemplo

- Supongamos ahora que queremos implementar una clase **Alumno**
- Los alumnos también tienen NIF, nombre, apellidos y edad, por lo que los atributos de la clase **Persona** son adecuados para Alumno
- Los métodos de la clase Persona, en principio, parecen todos ellos adecuados para la clase Alumno. Por tanto, los alumnos son personas
- Resulta así adecuado aplicar el mecanismo de herencia y crear la nueva clase **Alumno** a partir de la existente Persona. Se puede hacer mediante derivación pública o privada, que es la por defecto:

```
class Alumno : public Persona {
        ...
};
```

```
class Alumno : private Persona {
        ...
};
```

Alumno.hpp

Herencia: protected

- Pero a veces resulta necesario disponer en la **subclase** de algún medio de **acceso a los miembros privados heredados**
- Como no deseamos convertir en públicos los miembros de la superclase que son privados, la solución consiste en establecer el modo de acceso de los miembros privados de la superclase como **protegido**
- Entonces:
  - Los **miembros protegidos** se definen en una **sección protected** de la estructura class
  - Los **miembros protegidos** se comportan en la clase en la que están definidos como si fueran privados
  - Los métodos de una subclase sí tienen acceso a los miembros protegidos heredados de la superclase
  - En una **herencia pública**, los **miembros protegidos** heredados **siguen siendo protegidos** en la subclase
  - En una **herencia privada**, los **miembros protegidos** heredados **pasan a ser privados** en la subclase, **pero no se ocultan y se pueden acceder** en ésta

Herencia: resumen

- Resultado de la herencia según el tipo de los miembros en la clase base y en la clase derivada

Herencia	Clase base	Clase derivada
Public	Public	Public
	Protected	Protected
	Private	
Protected	Public	Protected
	Protected	Protected
	Private	
Private	Public	Private
	Protected	Private
	Private	

Herencia: resumen

- Tipos de acceso a los miembros según su tipo en la clase base y la herencia realizada:

Tipo de dato de la **clase base**	**Clase derivada** de una clase base **public**	**Clase derivada** de una clase base **private**	**Otras clases** sin relación de herencia con la clase base
Private	No accesible directamente	No accesible directamente	No accesible directamente
Protected	Protected	Private	No accesible directamente
Public	Public	Private	Accesible mediante operador (.) o (->)

Herencia

- **No todos los miembros de la superclase se heredan en la subclase**. Estos métodos, si en la subclase no están definidos, el compilador añade automáticamente unos
- En cuanto a los **constructores y destructores**:
  - Si en la subclase **no** están **definidos**, **se crean automáticamente** un constructor predeterminado y un destructor
  - Si **no** se crea un **constructor de copia** en la subclase, el compilador añade uno tomando como **base el de la superclase**
  - **Cuando se crea** un ejemplar de la subclase, se invocan automáticamente todos los **constructores** de todas las **superclases**, en el **mismo orden** establecido por la jerarquía (de más general a más específico), y finalmente el constructor propio de la subclase
  - **Cuando se destruye** un ejemplar de la subclase, se invocan automáticamente todos los **destructores** de todas las **superclases**, en **orden inverso** al establecido por la jerarquía (de la más específica a la más general), comenzando por el destructor propio de la subclase

Herencia

- En cuanto al **operador de asignación**:
  - Si en la subclase **no** está **definido**, el compilador añade automáticamente uno tomando como **base el de la superclase** y añadiendo además copias bit a bit para los atributos propios de la subclase
- Por regla general, crearemos explícitamente los constructores (por defecto y copia), el destructor y el operador de asignación en la subclase

Herencia: ejemplo

- La clase Alumno quedaría de la siguiente manera:

Alumno.hpp

```
class Alumno : public Persona {
        private:
                int curso;
                string carrera;
        public:
                Alumno(string nif = "", int edad = 0, string nombre = "", string carrera = "", int curso = 1);
                Alumno(const Alumno&);
                Alumno& operator=(const Alumno&);
                ~Alumno();
                void setCurso(int);
                int getCurso() const;
                void setCarrera(string);
                string getCarrera() const;
};
```

Herencia

- En la subclase Alumno se ha ampliado el conjunto de métodos heredados y se han definido nuevos métodos en la subclase
- Frecuentemente, algún método de la superclase no resultará del todo adecuado para la subclase. Entonces, se puede redefinir el método en la subclase sin más que declarar una función miembro con el mismo nombre
- Por ejemplo, supongamos el siguiente método mostrar() de la superclase Persona:

```
void Persona::mostrar() const {
        out << nif << endl;
        cout << nombreCompleto() << endl;
        out << "Edad: " << edad << endl;
}
```

Persona.cpp

- Podemos ver que no resulta del todo adecuado para los objetos de la subclase Alumno, ya que no muestra la información correspondiente al atributo propio curso y carrera

Herencia

- Entonces es posible redefinir el método mostrar() heredado definiendo en la subclase Alumno una función miembro con ese mismo nombre, de modo que será esta función miembro la que se ejecute cuando se pasen mensajes mostrar() a los objetos de la subclase Alumno
- No hay que olvidar que, en el método redefinido, es posible aprovechar el código del método de la superclase en caso de que su código sea una parte válida del total de código necesario

```
void Alumno::mostrar() const {
   Persona::mostrar();
   cout << "Curso: " << curso << endl;
}
```

Alumno.cpp

Herencia múltiple

- Se puede hacer que una clase herede de más de una superclase. Para ello se pueden incluir varias clases en la lista de herencia, cada una con su modo
- Basta con incluir varias clases en la lista de derivación, cada una con su modo de derivación:

```
class SuperClase1 {
        ...
};

class SuperClase2 {
        ...
};

class Subclase : public SuperClase1, public SuperClase2 {
        ...
};
```

Herencia múltiple

- Hay que tener en cuenta el problema que puede surgir, la colisión de identificadores:

```
class SuperClase1 {
	public:
		void mostrar() { cout << "SuperClase1" << endl; }
};

class SuperClase2 {
	public:
		void mostrar() { cout << "SuperClase2" << endl; }
};

class Subclase : public SuperClase1, public SuperClase2 {
	...
};
```

Herencia múltiple

- Solución:
  - Si la Subclase redefine los métodos mostrar() heredados, será su propia versión la que se use con sus objetos
  - Sino, si la Subclase no redefine los métodos heredados, se puede resolver la ambigüedad con el operador de resolución de ámbito:

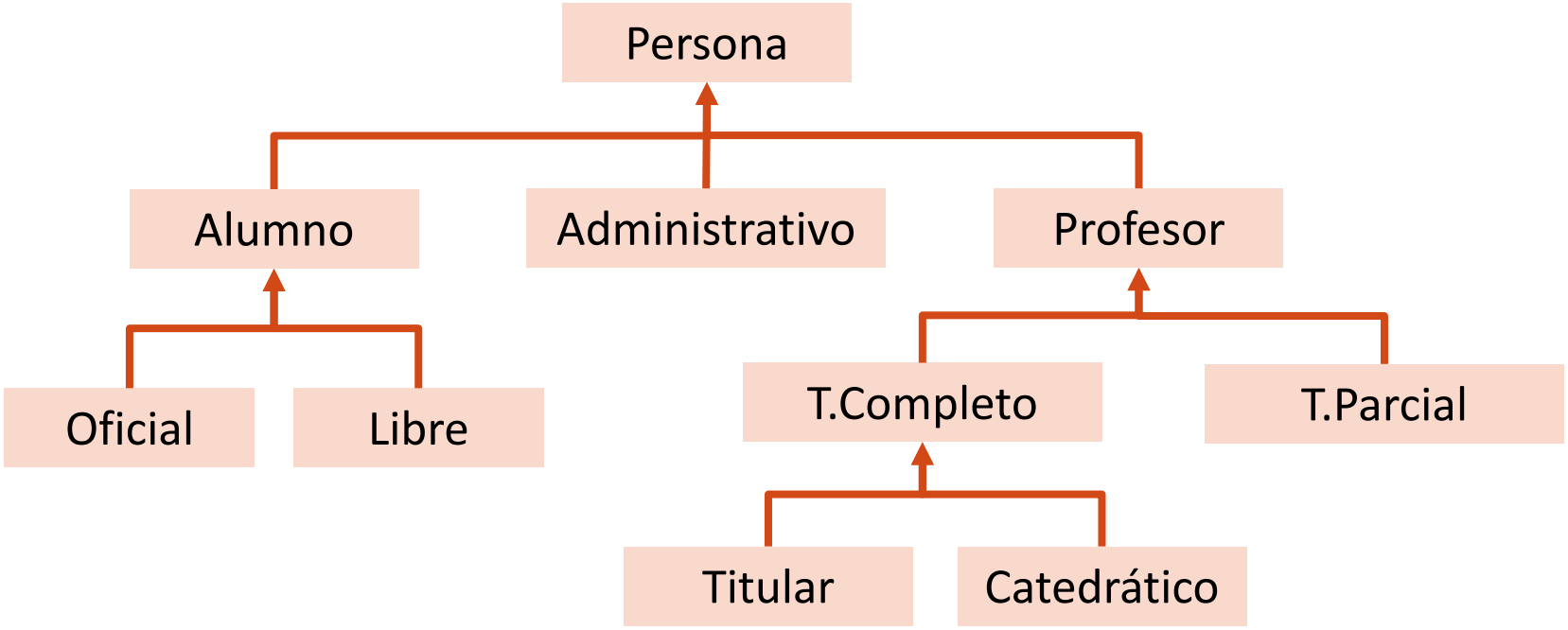
```
Subclase obj1;
...
obj1.SuperClase1::mostrar();
obj1.SuperClase2::mostrar();
```

  - Y sino, si el compilador no puede resolver la ambigüedad notificará un error

```
obj1.mostrar();
// main.cpp:25:10: error: request for member 'mostrar' is ambiguous
```

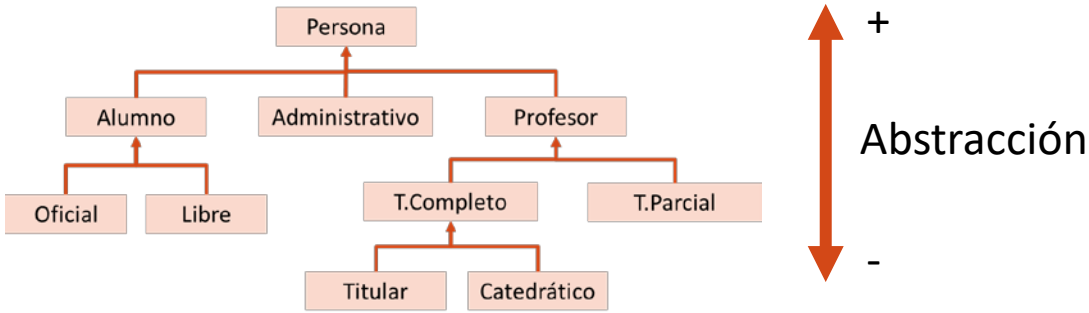
Herencia: jerarquía

- A medida que se establecen relaciones de herencia entre las clases, implícitamente se va construyendo la jerarquía de clases de la aplicación que se está desarrollando



Herencia: jerarquía

- Cuanto más arriba en la jerarquía, más abstractas son las clases (menor nivel de detalle)
- A medida que se desciende por la jerarquía aumenta el nivel de detalle (disminuye la abstracción)
- Así, una Persona es una entidad mucho más abstracta y general que un Titular
  - Cada clase de la jerarquía debe implementar todas las características que son comunes a todas sus subclases
  - Cada subclase debe contemplar únicamente las peculiaridades que la distinguen de su superclase



Índice de contenidos

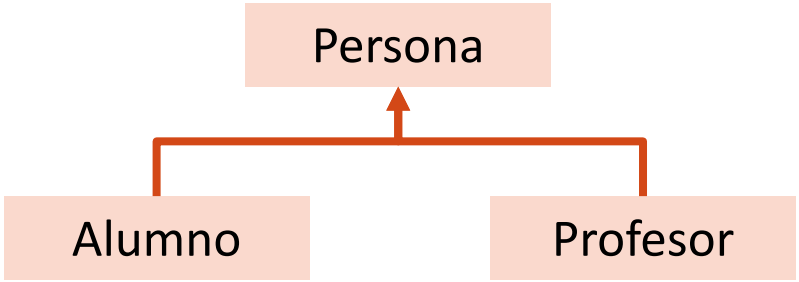
Polimorfismo: funciones virtuales

- Hay que tener presente que un Alumno es un Persona, pero no tiene por qué ser al revés
- La idea del **polimorfismo** es la de tratar de la misma manera objetos distintos. Si por ejemplo tenemos un vector de personas, a la hora de acceder a cada una de ellas se comportarían según lo que sea, alumno o profesor
- Funciona sólo con **Punteros** y **Referencias**
- Esto se consigue con lo que se llama ligadura dinámica, a través de las **funciones virtuales**
  - Nos permiten indicar que si dicha función está redefinida, en ejecución se mira el tipo del objeto y se invoca al método correspondiente
  - Si una clase no lo redefine, se sube en la jerarquía hasta encontrar la definición
- Para definir una función como virtual, basta con poner delante de la misma la palabra **virtual**

Polimorfismo: ejemplo

- Supongamos tenemos la siguiente jerarquía, donde cada Persona tiene una función distinta:

```
class Persona{
	public:
		virtual void funcion(){
			cout << "No hago nada" << endl;
		}
};
```




```
class Alumno : public Persona{
	public:
		void funcion(){
			cout << "Estudio" << endl;
		}
};
```

```
class Profesor : public Persona{
	public:
		void funcion(){
			cout << "Doy clases" << endl;
		}
};
```

Polimorfismo: ejemplo

- Ahora podemos crear un vector de personas, donde cada una de ellas será de un tipo en concreto:

```
int main() {
	Persona *v_personas[3];
	v_personas[0] = new Persona();
	v_personas[1] = new Alumno();
	v_personas[2] = new Profesor();
	for(int i = 0; i < 3; i++) {
		v_personas[i] -> funcion();
	}
	for (int i = 0; i < 3; i++) {
		delete v_personas[i];
	}
	return 0;
}
```

Polimorfismo: ejemplo

- En este punto, si decimos a cada uno de los elementos que nos diga su función, el resultado sería el siguiente:

```
No hago nada
Estudio
Doy clases
```

- En cambio, si no se hubiera declarado la función como virtual en la clase Persona, dado que el vector es de Personas, el resultado habría sido:

```
No hago nada
No hago nada
No hago nada
```

Índice de contenidos

Ejercicios

1. Crea la clase **Vehiculo**, la cual contiene como atributos: marca, modelo y velocidad; y como métodos: sus **accesores**, **mutadores**, **constructores** (por defecto y parametrizado) y **destructor**, y además, el método **mostrar()** que se encargue de mostrar su información por pantalla, otro que incremente su velocidad mediante la sobrecarga del **operador ++** y de comparar si un vehículo es más rápido que otro mediante la sobrecarga del **operador >=**

2. Crea las clases **Moto**, **Coche**, **Autobus** y **Camion** que hereden de **Vehiculo**, los cuales contendrán además de los atributos anteriores, el número de ruedas (2,4,6,8) y su límite de velocidad; y como métodos, además de los anteriores contendrá: **mostrar()** que permite mostrar su información actualizada y **potenciarueda()**, que divide la potencia del vehículo entre el número de ruedas y muestra el resultado por pantalla

Ejercicios

3. Crea un archivo con la función **main()** que se encargue de crear un vector de 10 **Vehiculos**. Cada uno de ellos será un puntero a un tipo de vehículo distinto (**Moto**, **Coche**, **Autobus**, **Camion**), los cuales tendrán diferentes valores

4. Recorre el vector mostrando la información de cada uno de ellos, y prueba las funciones creadas, tanto para el tipo **Vehiculo** como para cada uno de los tipos de vehículo (**Moto**, **Coche**, **Autobus**, **Camion**)

Universidad Rey Juan Carlos
Escuela Técnica Superior Ingeniería de Telecomunicación

Programación de Sistemas de Navegación

2.4. Lenguaje C++

Templates, Librería STL

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

Índice de contenidos

Templates

- A medida que aumenta la complejidad de nuestros programas y sobre todo, de los problemas a los que nos enfrentamos, descubrimos que tenemos que repetir una y otra vez las mismas estructuras
  - A menudo tenemos que implementar arrays dinámicos para diferentes tipos de objetos, listas dinámicas, pilas, colas, árboles, etc.
- El problema es que el código es siempre similar, y hasta ahora estamos obligados a reescribir ciertas funciones que dependen del tipo o de la clase del objeto que se almacena
- Para evitar esto, existen las **plantillas (templates)**, las cuales nos permiten parametrizar estas clases para adaptarlas a cualquier tipo de dato

Templates: funciones

- Las **plantillas de funciones** son funciones especiales que pueden funcionar con tipos genéricos. Esto nos permite crear una plantilla de función cuya funcionalidad se puede adaptar a **más de un tipo o clase** sin repetir el código completo para cada tipo
- En C++ esto se puede lograr usando parámetros de **plantilla**. Un parámetro de plantilla es un tipo especial de parámetro que se puede usar para **pasar un tipo como argumento**: al igual que los parámetros de función regulares se pueden usar para pasar valores a una función, los parámetros de plantilla permiten pasar también tipos a una función. Estas plantillas de funciones pueden usar estos parámetros como si fueran cualquier otro tipo regular
- El formato para declarar plantillas de funciones con parámetros de tipo es:

template <identificador de clase**>** declaracion_funcion;

Templates: funciones

- Por ejemplo, para crear una función de plantilla que devuelva el menor de dos objetos que podríamos usar:

```
template <class Tipo> Tipo GetMin (Tipo a, Tipo b) {
        Tipo resultado = ( a < b ? a : b );
        return resultado;
}
```

```
if (a < b) { resultado = a; }
else { resultado = b; }
```

- Aquí hemos creado una función de plantilla con **Tipo** como parámetro de plantilla. Este parámetro de plantilla representa un tipo que aún no se ha especificado, pero que se puede usar en la función de plantilla como si fuera un tipo normal
- Como se puede ver, la plantilla de función **GetMin** devuelve el menor de dos parámetros de este tipo aún indefinido

Templates: funciones

- Para usar esta plantilla de función, usamos el siguiente formato para la llamada a la función:

```
nombre_funcion <Tipo> (parametros);
```

- Y por ejemplo, para llamar a **GetMin** y comparar dos valores de tipo entero (int) y long, podemos escribir:

```
int x = 3, y = 4, z;
z = GetMin <int> (x,y);
```

```
long x = 3, y = 4, z;
z = GetMin <long> (x,y);
```

- Cuando el compilador encuentra esta llamada a una función de plantilla, utiliza la plantilla para generar automáticamente una función que reemplaza cada aspecto de **Tipo** por el tipo pasado como el parámetro de plantilla real (int o long en este caso) y luego lo llama
- Este proceso es realizado automáticamente por el compilador y es invisible para el programador

Templates: funciones

- Al utilizar el tipo genérico **Tipo** como parámetro para **GetMin**, el compilador puede descubrir qué tipo de datos debe instanciar sin tener que especificarlo explícitamente entre paréntesis angulares (como hemos hecho antes de especificar <int> y <long>)

```
int x = 3, y = 4, z;
z = GetMin (x,y);
```

- Dado que tanto x como y son de tipo int, y el compilador puede descubrir automáticamente que el parámetro de plantilla solo puede ser int. Este método implícito produce exactamente el mismo resultado
- En este caso, llamamos a nuestra plantilla de función **GetMin()** sin especificar explícitamente el tipo entre **<>**. El compilador determina automáticamente qué tipo se necesita en cada llamada

Templates: funciones

- Debido a que nuestra función de plantilla incluye solo un parámetro de plantilla (clase **Tipo**) y la propia plantilla de función acepta dos parámetros, ambos de este tipo **Tipo**, no podemos llamar a nuestra plantilla de función con dos objetos de diferentes tipos como argumentos:

```
int x = 3, y;
long z = 4;
y = GetMin(x,z);
```

- Esto no es correcto, ya que nuestra plantilla de función **GetMin** espera dos argumentos del mismo tipo, y en esta llamada usamos objetos de dos tipos diferentes

Templates: funciones

- Pero también podemos definir plantillas de funciones que acepten más de un parámetro de tipo, simplemente especificando más parámetros de plantilla entre **<>**. Por ejemplo:

```
template <class T, class U> T GetMin (T a, U b) {
        return ( a < b ? a : b);
}
```

- En este caso, nuestra plantilla de función **GetMin()** acepta dos parámetros de diferentes tipos y devuelve un objeto del mismo tipo que el primer parámetro (**T**). Ahora ya podríamos llamar a **GetMin()** con:

```
int x = 3, y;
long z = 4;
y = GetMin <int, long> (x, z); // y = GetMin(x, z);
```

Templates: clases

- También tenemos la posibilidad de escribir plantillas de clase, para que una clase pueda tener miembros que usen parámetros de plantilla como tipos
- Por ejemplo, para crear una clase que sea una lista de 10 elementos de tipo **T**:

```
template <class T> class miLista10 {
        private:
                T elementos[10];
        public:
                miLista10 (T value){
                        for (int i=0; i<10; i++)
                                elementos[i]=value;
                }
};
```

Templates: clases

- La clase que acabamos de definir sirve para almacenar 10 elementos de cualquier tipo válido. Por ejemplo, si quisiéramos declarar un objeto de esta clase para almacenar valores enteros de tipo int con los valores que sean inicializados a 32, escribiríamos:

```
miLista10 <int> lista (32);
```

- Pero esta misma clase también se usaría para crear un objeto para almacenar cualquier otro tipo:

```
miLista10 <double> lista (32.0);
```

- O incluso:

```
Persona p;
miLista10 <Persona> lista (p);
```

Templates: clases

- En el ejemplo anterior, la única función miembro en la plantilla de clase se ha definido en línea dentro de la declaración de clase misma
- En caso de que definamos un miembro de función fuera de la declaración de la plantilla de clase, siempre debemos preceder esa definición con el prefijo: **template <...>**

```
template <class T> class miLista10 {
        private:
                T elementos[10];
        public:
                miLista10 (T value){
                        for (int i=0; i<10; i++)
                                elementos[i]=value;
                }
                T GetMin();
};
```

```
template <class T> T miLista10 <T> :: GetMin() {
        T min = elementos[0];
        for (int i=0; i<10; i++)
                if (elementos[i] < min)
                        min = elementos[i];
        return min;
}
```

En el caso de Persona, hay que sobrecargar los operadores para que trabaje adecuadamente

Templates: clases

- En el caso de Persona, o de cualquier otra clase no estándar, hay que **sobrecargar los operadores** para que trabajen adecuadamente
- En el caso de definir una función fuera de la declaración de la plantilla, no se pueden separar en distintos archivos. Tanto la clase con sus atributos y métodos como la declaración de sus funciones, tienen que ir en el **mismo archivo**

Índice de contenidos

Librería STL

- La STL (Standard Template Library) proporciona una colección de estructuras de datos contenedoras y algoritmos genéricos que se pueden utilizar con éstas
- Una estructura de datos se dice que es contenedora si puede contener instancias de otras estructuras de datos
- Es una biblioteca de clases de contenedores, algoritmos e iteradores.
- Es una biblioteca generalizada y, por lo tanto, sus componentes están parametrizados (templates)
- STL tiene cuatro componentes:
  - Algoritmos
  - Contenedores
  - Las funciones
  - Iteradores

Librería STL

- **Algoritmos**: permiten efectuar operaciones predefinidas sobre las estructuras de datos de la STL así como sobre aquellas estructuras de datos definidas por el usuario
- **Contenedores**: estructuras de datos que contienen, y permiten manipular, otras estructuras de datos
- **Iteradores**: conectan las estructuras de datos con los algoritmos genéricos
- La STL tiene una serie de características muy deseables:
  - Es muy eficiente
  - Es adaptable
  - Permite evitar errores al proporcionar los algoritmos ya predefinidos

Librería STL

- La STL dispone de las siguientes estructuras:

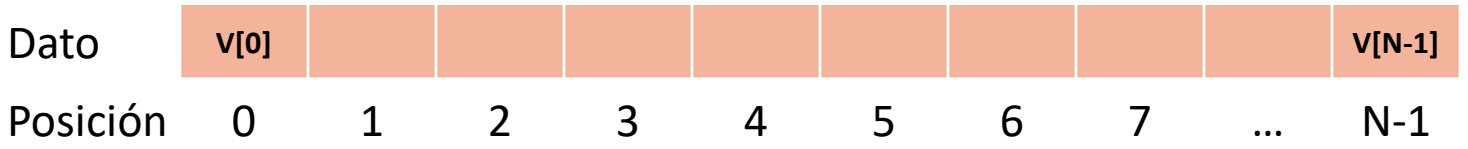
Contenedores lineales	Contenedores asociativos	Contenedores adaptados
Vector *vector<T>*	**Conjunto** *set<T, Compare>*	**Pila** *stack<Container>*
Lista *list<T>*	**Multiconjunto** *multiset<T, Compare>*	**Cola** *queue<Container>*
Doble cola *deque<T>*	**Mapa** *map<Key, T, Compare>*	**Cola de prioridad** *priority_queue<Container,Compare>*
C++11 **Array** *array<T>*	**Multimapa** *multimap<Key, T, Compare>*	
C++11 **Doble cola** *Forward_list<T>*		

Librería STL

- Los contenedores se dividen en tres categorías:
  - **Contenedores lineales: Almacenan** los objetos de forma **secuencial** permitiendo el **acceso** a los mismos de forma **secuencial y/o aleatoria** en función de la naturaleza del contenedor
  - **Contenedores asociativos:** En este caso cada objeto almacenado en el contenedor tiene asociada una **clave**. Mediante la clave los objetos se pueden almacenar en el contenedor, o recuperar del mismo, de forma rápida
  - **Contenedores adaptados:** Permiten cambiar un contenedor en un nuevo contenedor modificando la interface (métodos públicos y datos miembro) del primero. En la mayor parte de los casos, el nuevo contenedor únicamente requerirá un **subconjunto** de las capacidades que proporciona el contenedor original

Librería STL: vectores

- Al igual que en C, un vector es un grupo de elementos finito del mismo tipo, indexados por una clave entera
- Los vectores pueden ser considerados como una generalización del array de C++. Los vectores son útiles cuando el número de elementos a manejar es conocido de forma aproximada
- Con los vectores podemos hacer las mismas operaciones que con los arrays de C++, pero además pueden crecer dinámicamente



Librería STL: cadenas/strings

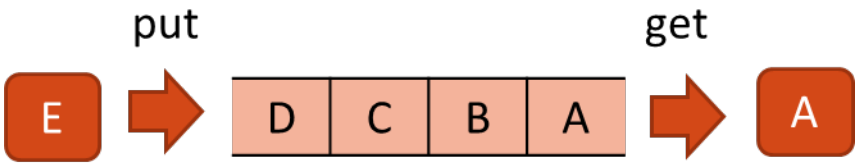
- Como ya sabemos, una cadena se puede considerar como un vector de caracteres
- Dado que en una cadena podemos desear implementar operaciones como buscar una subcadena, este tipo de datos se considera a parte de los vectores
- Como ya vimos, la biblioteca de funciones básicas de manejo de vectores de caracteres de C/C++ se utiliza con **#include <cstring>** (string.h en C)
- Y entre otras, están disponibles:
  - **strcpy** (copia una cadena en otra)
  - **strcmp** (compara dos cadenas)
  - **strcspn** (busca una cadena en otra)
  - **strdup** (duplica una cadena)
  - **strnset** (copia los n primeros caracteres de una cadena)
  - **sprintf** (copia una cadena formateada)
  - etc...

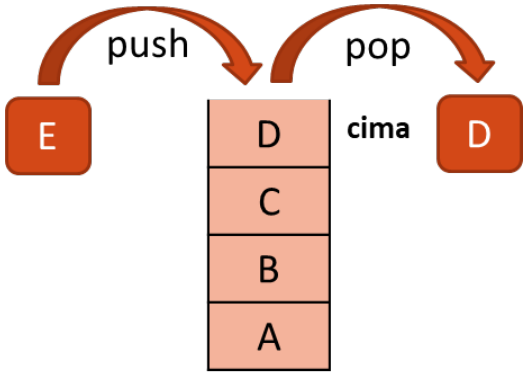
Librería STL: listas

- Una lista es una estructura de datos donde el número de elementos no está limitado
- Como en un vector, la lista contiene elementos del mismo tipo. Las listas no están indexadas, i.e., hay que acceder a los elementos de uno en uno
- Casi todas las operaciones básicas son muy costosas, pero a cambio la inserción y borrado son muy eficientes

Librería STL: Pilas y Colas

- Las pilas y colas se pueden considerar como formas especializadas de la cola doble
- Los elementos de una pila siguen el protocolo LIFO (el primero que sale es el último que entra). Los elementos solo pueden ser añadidos o eliminados del tope de la pila
- Una cola sigue el protocolo FIFO (el primero que sale es el primero que entra). Los elementos se insertan al final de la cola, y se extraen del frente



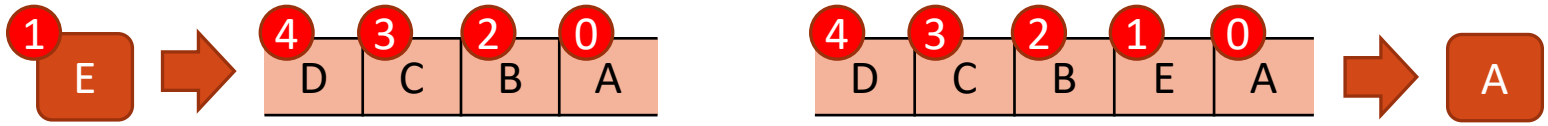


Librería STL: colas de doble fin

- Como una lista, una cola de doble fin (deque, double-ended queue) tiene tamaño arbitrario, según los elementos se inserten o se borren
- La cola de doble fin se optimiza para la inserción/eliminación de elementos del extremo de la estructura (en tiempo constante)
- Como en un vector, una cola doble es una estructura indexada, permitiendo un rápido acceso a los elementos

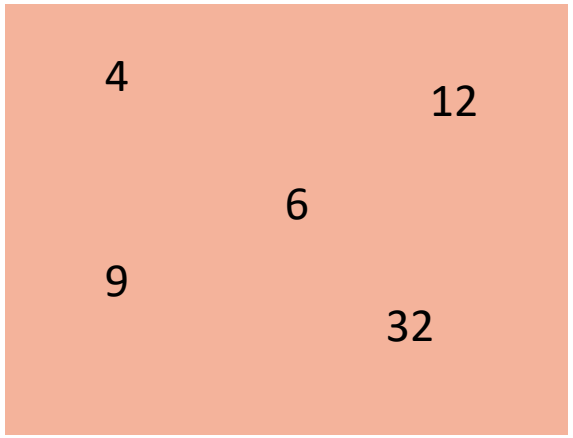
Librería STL: colas de prioridad

- Una cola de prioridad se optimiza para inserción de nuevos elementos asociados a una prioridad, y para desencolar los elementos con mayor prioridad
- Las colas de prioridad se pueden interpretar como conjuntos en los que el elemento de más prioridad está disponible al instante, pero el resto de elementos están guardados aleatoriamente
- Si dos elementos tienen la misma prioridad, se desencolarán siguiendo el orden de cola
- Ejemplo (0 mayor prioridad):



Librería STL: conjuntos

- Un conjunto es una colección simple de valores únicos
- Aunque los conjuntos no establecen un orden en los elementos, la biblioteca STL sí lo hace, para tener un acceso más rápido a estos
- Un multiconjunto (multiset, bag) es un conjunto que puede tener elementos repetidos



Librería STL: mapa

- Un mapa (diccionario) es, como un vector, una colección de elementos indexados
- Pero, al contrario que un vector, los valores índice no han de ser enteros. Una mapa es entonces una colección de ASOCIACIONES de pares <clave,valor>
  - Un ejemplo del mundo real es un diccionario, donde las claves representan las palabras, y los valores el significado

Clave	Valor
1011	Juan
1012	Pepe

- Un multimapa es similar a un mapa, con la excepción de que permite repeticiones de clave

Librería STL: algoritmos genéricos

- Habitualmente **el código se reescribe** para utilizarlo en otros programas, pero los **algoritmos** comunes (algoritmos de búsqueda, algoritmos de ordenación,...) deben **adaptarse** a cada problema:
  - **Problema**: Tarea repetitiva que debe ser eliminada
  - **Solución**: Generar algoritmos genéricos que puedan ser utilizados en una amplia variedad de situaciones
- Así, existe una gran cantidad de **librerías de código reutilizable**. Los ejemplos más comunes son las librerías de funciones matemáticas (**math**) y las de manipulación de cadenas de caracteres (**string**)
- Los algoritmos genéricos separaran los algoritmos en sí, de la representación de datos que manipulan

Librería STL: algoritmos genéricos

- Para incorporar estos algoritmos, es necesario incluirlos:

```
#include <algorithm>
```

- Para usar este tipo de algoritmos, vamos a ver un ejemplo con la función **std::find**. Esta función devuelve un iterador al primer elemento en el rango [*primero*, *último*) cuyo dato es igual a *valor*. Si no se encuentra dicho elemento, la función devuelve el *último*

```
template<class It, class T>  It find (It primero, It ultimo, const T& valor) {
        while (primero != ultimo) {
                if (*primero == valor) return primero;
                ++primero;
        }
        return ultimo;
}
```

Librería STL: algoritmos genéricos

- Esta función podemos usarla para buscar un valor determinado almacenado en un vector de C++ convencional:

```
int datos[100];
int* donde = find(datos, datos+100,7);
```

- Internamente el algoritmo de búsqueda finalmente quedaría como:

```
int* find (int* primero, int* ultimo, const int& valor) {
        while (primero != ultimo) {
                if (*primero == valor) return primero;
                ++primero;
        }
        return ultimo;
}
```

Librería STL: iteradores

- Para muchos contenedores, una de las operaciones más comunes es **recorrer** (iterar) los **elementos** de la colección
- Un iterador es un **objeto que puede retornar una referencia** a cada uno de los elementos de un contenedor
- Los iteradores son la base de construcción de los algoritmos genéricos
- La dificultad de este hecho es proporcionar al programador los medios para construir con facilidad bucles, **sin violar la estructura interna** de la colección

Librería STL: iteradores

- La **solución** propuesta por la biblioteca estándar de plantillas es un mecanismo conocido como **ITERADOR**
- Los iteradores son una **generalización de los punteros** que permiten al programador trabajar con diferentes estructuras de datos (contenedores) de una forma uniforme (desempeñan el mismo papel que los punteros de C++)
- Como en los punteros, los iteradores se modifican fundamentalmente con el operador ++. Si se aplica esta operación a un iterador que denota el valor final de una secuencia, dicho iterador pasa a valer “más allá del final”

Librería STL: iteradores

- Un iterador debería soportar todas las operaciones que puedan realizarse con un puntero. Estas operaciones se resumen en la tabla siguiente para los iteradores *'x'* e *'y'* y el número entero *'n'*:

x++	*x + n*	*x - y*	*x > y*	**x*
++x	*x – n*	*x == y*	*x <= y*	*x = y*
x--	*x += n*	*x != y*	*x >= y*	
--x	*x -= n*	*x < y*	*x[n]*	

- Un iterador *j* es "**alcanzable**" mediante un iterador *i* si después de una secuencia finita de operaciones ++ el iterador *i* pasa a ser igual que *j*
- Los rangos se pueden usar para describir todo el contenido de un contenedor construyendo un iterador al elemento inicial y un iterador de "final de contenedor"
  - Se supone (aunque no se verifica) que el segundo iterador es alcanzable desde el primero

Librería STL: iteradores

- A la hora de incrementar o decrementar los iteradores, se pueden utilizar dos tipos de operaciones: Preincremento y Postincremento
- **Preincremento**: la variable incrementa su valor antes de usarse

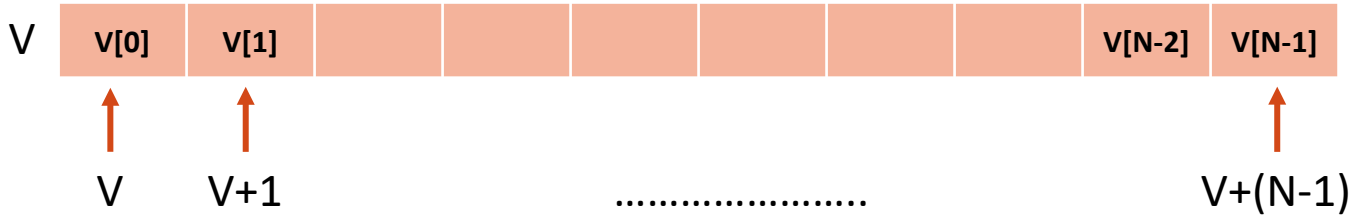
```
int a = 0;
int b = ++a;

El resultado sería a = 1 y b = 1
```

- **Postincremento**: la variable se usa antes de incrementar su valor

```
int a = 0;
int b = a++;

El resultado sería a = 1 y b = 0
```

Librería STL: iteradores

- Así, de forma abstracta, un **iterador** se entiende como un **puntero** a elementos de la colección
- Cada contenedor de la **STL** contiene sus **iteradores** correspondientes, de forma que los algoritmos genéricos de la STL pueden llamarse con estos iteradores
- Un iterador puede utilizarse para especificar una posición en la colección de datos, mientras que dos iteradores pueden especificar un rango de datos (similar a cómo lo hacen un par de punteros)



Librería STL: iteradores

- Los iteradores se dividen en varias categorías de forma que en cada una de ellas únicamente se proporcionan las operaciones que pueden implementarse de forma eficiente
- Las categorías de iteradores de la STL son las siguientes:
  - **forward iterators**: Iteradores que pueden avanzar al elemento siguiente
  - **bidirectional iterators**: Iteradores que pueden avanzar al elemento siguiente o retroceder al elemento
  - **random access iterators**: Iteradores que pueden avanzar o retroceder más de una posición de una vez

Librería STL: iteradores

- Al igual que puede haber punteros nulos, un iterador puede tener un valor no válido
- En C++, cuando dos punteros especifican un rango de datos, se considera que el puntero que especifica el final no se encuentra en el rango de datos (suele ser “el siguiente al último”)
- De la misma forma, un iterador que denote el final de una estructura hará referencia a una posición fuera del rango válido
  - Por ello es ilegal desreferenciar un iterador que se use para especificar el final de un rango
  - Al iterador que no denota una posición válida (como los iteradores indicativos de posición fuera de rango) se le llama **iterador nulo**

Librería STL: iteradores

- Los tres **requisitos** que un **iterador** debe cumplir son (protocolo de iterador):
  - Un iterador puede ser **comparado** por igualdad con otro (son iguales si apuntan a la misma posición, y distintos en otro caso)
  - Un iterador puede ser **desreferenciado** usando el operador (*), para obtener el valor del objeto apuntado (dependiendo del tipo de iterador y contenedor, el iterador también se puede usar para cambiar el valor del elemento al que apunta)
  - Un iterador puede ser **incrementado**, accediendo al siguiente elemento de la secuencia, usando el operador de incremento (++)

Índice de contenidos

Anexo

- Las distintas funciones pueden encontrarse en:
  - Algoritmos genéricos:

    http://www.cplusplus.com/reference/algorithm/
  - String:

    http://www.cplusplus.com/reference/string/string/?kw=string

Índice de contenidos

Ejercicios

1. Crea la función que ordena un vector de números (algoritmo de la burbuja) de manera que pueda utilizarse con distintos tipos de números (int, float, double, long). Para ello, convierte la función conocida utilizando un template

2. Crea la función genérica "**elevar (a, b)**", de manera que dados dos números (a y b) en distintos formatos (int, long, double, float), devuelva el número $\mathbf{a^b}$ en el formato del número **a**

3. Crea una clase genérica **Vector**, que almacene un vector de elementos numéricos. Los elementos de este vector podrán ser de tipo int, long, float, double. Crea un constructor parametrizado que dados dos números (**a** y **b**), cree un vector de tamaño **a** cuyos valores iniciales sean **b**

Ejercicios

4. En la clase **Vector**, proporciona los métodos:
   - **print():** que muestra los valores del vector
   - **media():** que devuelve la media del vector
   - **max():** que devuelve el valor máximo del vector

5. Crea una plantilla **Matriz** para un clase que permita trabajar con una matriz. Para ello, es necesario guardar el tamaño que va a tener la matriz (alto y ancho), así como permitir que se puedan guardar distintos tipos de números (int, double, long,…)

6. En la clase anterior, crea un constructor parametrizado que dados dos números (**a** y **b**), cree una matriz de **a** filas, **b** columnas y cuyos valores iniciales sean **0**

Ejercicios

7. En la clase Matriz, crea otro constructor parametrizado que dados tres números (**a**, **b** y **c**), cree una matriz de **a** filas, **b** columnas y cuyos valores iniciales sean **c**

8. En la clase Matriz, proporciona los métodos:
   - **print():** que muestre los valores de la matriz
   - **suma(A,B)** y **resta(A,B)**, de manera que dadas dos matrices (A y B), devuelva una matriz con la suma o la resta de ambas respectivamente. Valora que A y B puedan contener valores numéricos de distinto tipo

Universidad Rey Juan Carlos
Escuela Técnica Superior Ingeniería de Telecomunicación

Programación de Sistemas de Navegación

2.5. Lenguaje C++

Librería STL: vector, lista

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

Índice de contenidos

Librería STL: vector

- Los vectores de la STL son una generalización del concepto de array unidimensional
- Un vector se entiende como una colección finita de datos del mismo tipo indexada en enteros. El número de elementos se provee como parte de la declaración
  - Ejemplo: double c[6];
- Los índices válidos tienen como rango desde 0 al número de elementos menos 1, y se accede a los elementos a través del operador []

Librería STL: vector

- ¿Por qué no utilizar los arrays de C++?
  - El compilador no controla los valores de los índices de los vectores en tiempo de ejecución
  - No hay una forma "trivial" de saber el número de elementos que contiene
  - No se puede cambiar el tamaño del array dinámicamente
  - Apenas se dispone de operaciones de alto nivel que trabajen con vectores
- ¿Por qué implementar una nueva clase vector?
  - De forma similar a las cadenas, para corregir todas las deficiencias anteriores

Librería STL: vector

- Los **vectores**, a diferencia de las cadenas, son un tipo de datos incompleto, ya que problemas diferentes necesitan vectores de datos diferentes (enteros, dobles, racionales, complejos, cartas ...)
- Esa **heterogeneidad** exigiría implementar un tipo vector de enteros, un tipo vector de dobles, y así sucesivamente, lo cual es muy ineficiente
- La solución a este problema es **parametrizar** el tipo vector, describiéndolo en función de otro tipo

Librería STL: vector

- Para ello necesitamos alguna manera de expresar la **abstracción** de "un vector de tipo T", donde T representa un tipo desconocido
- Esto permitiría definir un **vector genérico**, junto a sus operaciones, siempre que no hagamos referencia directa al tipo T, ya que éste es desconocido
- Posteriormente podríamos crear instancias más específicas de este tipo, sustituyendo el tipo desconocido T por un tipo concreto
- Para ello existen las **plantillas** (**templates**)

Librería STL: vector

- Acceso a la clase:

```
#include <vector>
```

- Declaración e inicialización de vectores:
  - Puesto que es una clase plantilla, las declaraciones de los vectores deben incluir el tipo de componente (este puede ser un tipo primitivo, un puntero, o un tipo definido por el usuario, siempre que dicho tipo tenga un constructor por defecto):

```
vector<int> vec_uno(10);
```

  - El resto de constructores permiten crear vectores indicando otros parámetros además del tamaño:

```
vector<int> vec_dos(5,3); // 5 elementos a 3
vector<int> vec_tres; // vector sin elementos
vector<int> vec_cuatro(vector_dos); // copia de vector_dos
```

Librería STL: vector

- Acceso a elementos:
  - Se accede a los elementos para usar o modificar su valor utilizando el operador **[]**, como si fuera una cadena o un vector de C/C++ ordinario
  - IMPORTANTE: La implementación de la STL no comprueba los límites del vector (otras implementaciones sí lo hacen)

```
cout << vec_demo[1] << endl;
vec_demo[1]=17;
```

  - Funciones:
    - **front**() devuelve el primer elemento del vector
    - **back**() devuelve el último elemento del vector

```
cout << vec_demo.front() <<  vec_demo.back() << endl;
```

Librería STL: vector

- Inserción de elementos:
  - **push_back(T)** inserta el elemento tras la última posición del vector e incrementa su tamaño en uno (debe comprobar si hay suficiente espacio, y si no lo hay, aumentarlo):

```
vec_demo.push_back(3);
```

  - **insert(it,T)** inserta el elemento delante de la posición apuntada por un iterador:

```
vec_demo.insert(iterador,5);
```

  - **swap(vec)** intercambia los valores de dos vectores:

```
vec_demo.swap(vec_swap);
```

Librería STL: vector

- Borrado de elementos:
  - **pop_back()** elimina el último elemento del vector, y reduce su tamaño en uno:

```
vec_demo.pop_back();
```

  - **erase()** puede borrar la posición indicada por el iterador:

```
vec_demo.erase(iterador);
```

  - o borrar los elementos entre dos iteradores:

```
vec_demo.erase(iterador, iterador);
```

Librería STL: vector

- Cambio de tamaño del vector:
  - En general podemos definir dos tipos de tamaño en un vector:
    - Tamaño (size): Número de elementos contenidos en el vector
    - Capacidad (capacity): Número de elementos que puede contener el vector sin que haga falta redimensionarlo


```
cout << "tamaño: " << vec_demo.size() << endl;
cout << "capacidad: " << vec_demo.capacity() << endl;
// Tamaño: 4 y Capacidad: 6
```

  - Eliminar un elemento disminuye el tamaño pero no la capacidad
  - Añadir un elemento más allá de la capacidad implica asignar nueva memoria, y copiar en ella todo el vector

Librería STL: vector

- La capacidad (que luego puede aumentarse si se desea) se indica con reserve:

```
vec_demo.reserve(20);
```

- No olvidar que una relocalización invalida todas las referencias, punteros e iteradores contenidas en el vector que denotaran elementos del vector anteriores al cambio de capacidad
- **empty()** indica si el vector tiene tamaño 0 (suele ser más eficiente que comparar el resultado de **size()** con cero)

Librería STL: vector

- Tanto el tamaño como la capacidad de los vectores se almacenan en variables enteras
- Se proporciona acceso a dichas variables a través de **size()** y **capacity()**
- Se modifican a través de **resize()** y **reserve()**
  - La función **resize()** llama a **reserve()** para asegurarse de que la capacidad es al menos tan grande como el nuevo tamaño, y entonces actualiza la variable
  - Si la capacidad actual es ya mas grande que la que se requiere, no se hace nada más. De lo contrario, se reserva un nuevo bloque de memoria acorde al nuevo tamaño, y se usa el algoritmo genérico **copy()** para copiar los valores del bloque de memoria viejo al nuevo
  - Tras eso, el bloque de memoria usado por los datos viejos se libera

Librería STL: vector

- Iteradores:
  - Como cualquier clase de la librería STL (incluida cadena/string), vector contiene una definición de tipo para el nombre '**iterator**', lo cual nos permite tratar los elementos
  - Esto permite declarar iteradores para cualquier tipo de vector:

```
vector<double>::iterator itr;
for (itr=vec_demo.begin(); itr!=vec_demo.end(); ++itr)
        ...
```

  - Las funciones **begin()** y **end()** proporcionan acceso a iteradores de acceso aleatorio de elementos del vector (inicio y fin)
  - Recordar que los iteradores soportados por estas funciones pueden dejar de ser válidos tras insertar o borrar elementos

Librería STL: vector

- Algoritmos genéricos:
  - Mucha de la funcionalidad de la clase vector no está en sus funciones miembro sino en algoritmos genéricos comunes a muchas estructuras de datos.
  - Vamos a ver algunas
- Búsqueda:
  - La clase vector no proporciona ningún método directo que diga si un elemento concreto está o no dentro del vector, pero podemos usar **find()** o **count()** para este propósito
  - Partiendo de los iteradores de principio y fin:

```
vector<int>::iterator start = vec_demo.begin();
vector<int>::iterator stop = vec_demo.end();
```

Librería STL: vector

- Opción 1: **find()** busca un valor entre dos posiciones

```
if (find(start,stop,32) != stop)
        ... // elemento encontrado
```

- Opción 2: **count()** cuenta las ocurrencias de un valor entre dos posiciones

```
int cont = 0;
count(start, stop, 17, cont);
if (cont != 0)
        ... // elemento en colección
```

Librería STL: vector

- Ordenación:
  - La clase vector no mantiene sus valores ordenados de forma automática, pero se puede usar el algoritmo genérico **sort()** o algún otro para tal fin
  - La forma más simple usa la comparación "menor que" para el tipo de elemento:

```
// Ordenación ascendente simple, utiliza el operador simple ‘<‘
sort(vec_demo.begin(), vec_demo.end());
```

Librería STL: vector

- Ordenación:
  - Una forma alternativa es proporcionar el operador de comparación en forma de función:

```
class CMayor {
        public:
                bool operator() (int a, int b) { return a > b; }
};

CMayor comparaFun; // Ordenación descendente

// Utiliza el operador '()' en el objeto comparaFun
sort(vec_demo.begin(), vec_demo.end(), comparaFun);
```

Librería STL: vector

- Operaciones del tipo vector:

Constructores

vector<T> v;	Constructor por defecto
vector<T> v(int);	Tamaño específico
vector<T> v(int,T);	Tamaño y valor iniciales
vector<T> v(unVector);	Constructor copia

Acceso a elementos

v[i];	Acceso a posición i
v.front();	Primer valor
v.back();	Último valor

Librería STL: vector

- Operaciones del tipo vector:

Inserción

v.push_back(T);	Poner elemento al final
v.insert(iterator,T);	Insertar en el iterador
v.swap(vector<T>);	Intercambiar valores con vector

Borrado

v.pop_back();	Extrae elemento del final
v.erase(iterator);	Borra un elemento
v.erase(iterator,iterator);	Borra un rango de valores

Librería STL: vector

- Operaciones del tipo vector:

Tamaño

v.capacity();	Máximo elementos a almacenar
v.size();	Elementos almacenados
v.resize(unsigned, T);	Cambia el tamaño
v.reserve(unsigned);	Tamaño del buffer
v.empty()	Cierto si vector vacío

Iteradores

vector<T>::iterator itr;	Declarar un nuevo iterador
v.begin();	Comienzo del iterador
v.end();	Iterador del final

Librería STL: vector

- Operaciones del tipo vector:

fill(iterator start, iterator stop, value) Rellena el intervalo con valor inicial
copy(iterator start, iterator stop, iterator destination) Copia una secuencia en otra
max_element(iterator start, iterator stop) Encuentra el elemento mayor
min_element(iterator start, iterator stop) Encuentra el elemento menor
reverse(iterator start, iterator stop) Invierte la aparición de elementos en la estructura

Librería STL: vector

- Operaciones del tipo vector:

count(iterator start, iterator stop, target value, counter) Cuenta los elementos iguales a value, incrementando el contador
count_if(iterator start, iterator stop, unary fun, counter) Cuenta los elementos que satisfacen una función, incrementando el contador
Transform(iterator start, iterator stop, iterator destination, unary) Aplica una función al intervalo, introduciendo los valores modificados en destino
find(iterator start, iterator stop, value) Devuelve un iterador al valor en la colección
find_if(iterator start, iterator stop, unary function) Encuentra un valor para el que la función devuelve cierto

Librería STL: vector

- Operaciones del tipo vector:

replace(iterator start, iterator stop, target value, replacement value) Reemplaza los elementos iguales a target por replacement
replace_if(iterator start, iterator stop, unary fun, replacement value) Reemplaza los elementos que cumplen la función
sort(iterator start, iterator stop) Ordena el intervalo en orden ascendente
for_each(iterator start, iterator stop, function) Ejecuta una función para cada elemento de la colección
iter_swap(iterator, iterator) Intercambia los valores especificados por los dos iteradores

Librería STL: vector

- Algoritmos para vectores ordenados:

iterator merge(iterator s1, iterator e1, iterator s2, iterator e2, iterator dest) Mezcla dos colecciones ordenadas en una tercera
void inplace_merge(iterator start, iterator center, iterator stop) Mezcla dos secuencias adyaccentes ordenadas en una
bool binary_search(iterator start, iterator stop, T value); Busca un elemento en la colección
iterator lower_bound(iterator start, iterator stop, T value); Encuentra el primer elemento menor o igual que value en la colección
iterator upper_bound(iterator start, iterator stop, T value); Encuentra el primer elemento mayor que value en la colección

Librería STL: vector

- Implementación:

```
template <class T> class vector {
        private:
                unsigned int mySize;
                unsigned int myCapacity;
                T* buffer;
        public:
                typedef T* iterator;
                typedef T value_type;
                // constructores
                vector () { buffer=0; resize(0); }
                vector (unsigned int size) { buffer=0; resize(size); }
                vector (unsigned int size, T initial);
                vector (vector& v);
                ~vector () { delete [] buffer; }
……
```

Librería STL: vector

- Implementación:

```
……
        // funciones miembro
        T back () { return buffer[mySize-1]; }
        iterator begin () { return buffer; }
        int capacity () { return myCapacity; }
        bool empty () { return mySize == 0; }
        iterator end () { return begin() + mySize; }
        T front () { return buffer[0]; }
        void pop_back () { mySize--; }
        void push_back(T);
        void reserve (unsigned int newCapacity);
        void resize (unsigned int newSize) { reserve(newSize); mySize=newSize; }
        int size () { return mySize; }
        // operadores
        T& operator[] (unsigned int index) { return buffer[index]; }
};
```

Índice de contenidos

Librería STL: lista

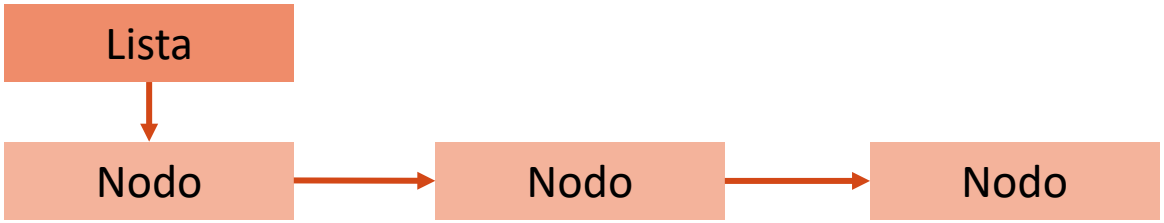
- Ahora estudiaremos el tipo de datos lista, implementando la clase contenedora “list” de forma similar a como está implementada en la STL
- Estudiaremos:
  - La abstracción de tipo de datos lista
  - Programas de ejemplo
  - Una implementación de listas
  - Iteradores de listas
  - Variaciones más comunes

Librería STL: lista

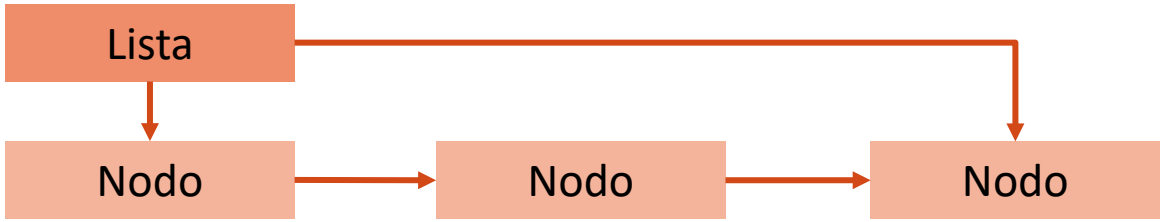
- La lista es una abstracción de datos que adquiere sentido en aquellos problemas en los que se necesita mantener una colección de datos, pero, o bien no se sabe a priori el número de elementos a almacenar, o dicho número varía muy ampliamente
- La idea de la lista es mantener los elementos de la colección en una cadena en la que cada elemento está enlazado tiene un puntero hacia el elemento siguiente
- Si el elemento no apunta a ningún otro, su puntero tiene un valor nulo

Librería STL: lista

- Como ya vimos, la forma más sencilla de lista mantiene un único puntero al primer elemento:

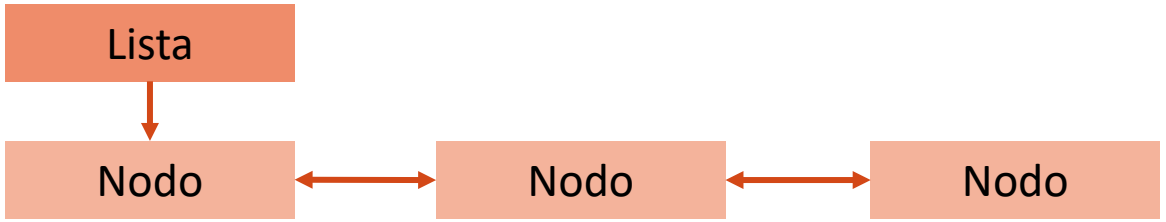


- Hay distintas variaciones de esta versión, como por ejemplo mantener un puntero tanto al primer como al último elemento, y de esta manera permitir inserciones al final y al principio de la lista en tiempo constante:



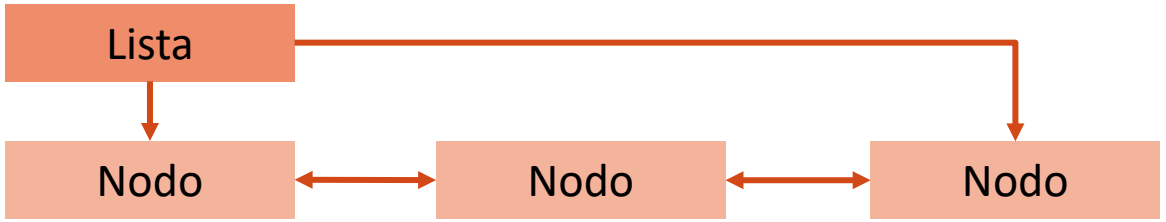
Librería STL: lista

- Otra variante muy común mantiene dos punteros para cada elemento, uno al elemento siguiente y otro al elemento anterior
- Esto permite recorrer la lista en los sentidos, en vez de únicamente hacia adelante. A este tipo de listas se las denomina “doblemente enlazadas” (doubly-linked):



Librería STL: lista

- El tipo de lista que proporciona la STL combina todas estas variantes, de forma que los punteros al principio y final de la lista permiten inserciones y borrados eficientes en ambos extremos, y además los enlaces dobles permiten recorrer la lista (usando iteradores) adelante y atrás:



Librería STL: lista

- Acceso a la clase:

```
#include <list>
```

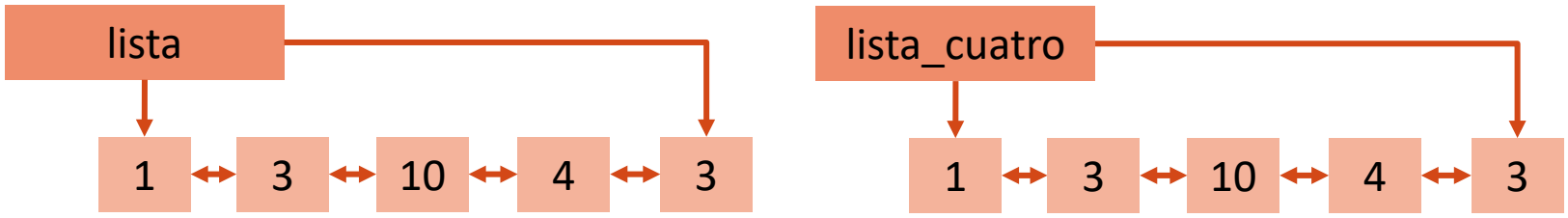
- Constructores por defecto:
  - Se usa una plantilla para indicar el tipo que va a albergar la lista
  - Dicho tipo puede ser un tipo básico, o uno definido por el usuario (y debe proporcionar un constructor por defecto)

```
list<int> lista_uno;
list<Carta*> lista_dos;
list<Carta> lista_tres;
```

Librería STL: lista

- Constructores de copia y asignación:
  - El constructor de copia se puede usar para inicializar una lista con valores sacados de otra lista
  - El operador de asignación lleva a cabo la misma acción
  - En ambos casos se usa el operador de asignación para llevar a cabo la copia de cada elemento

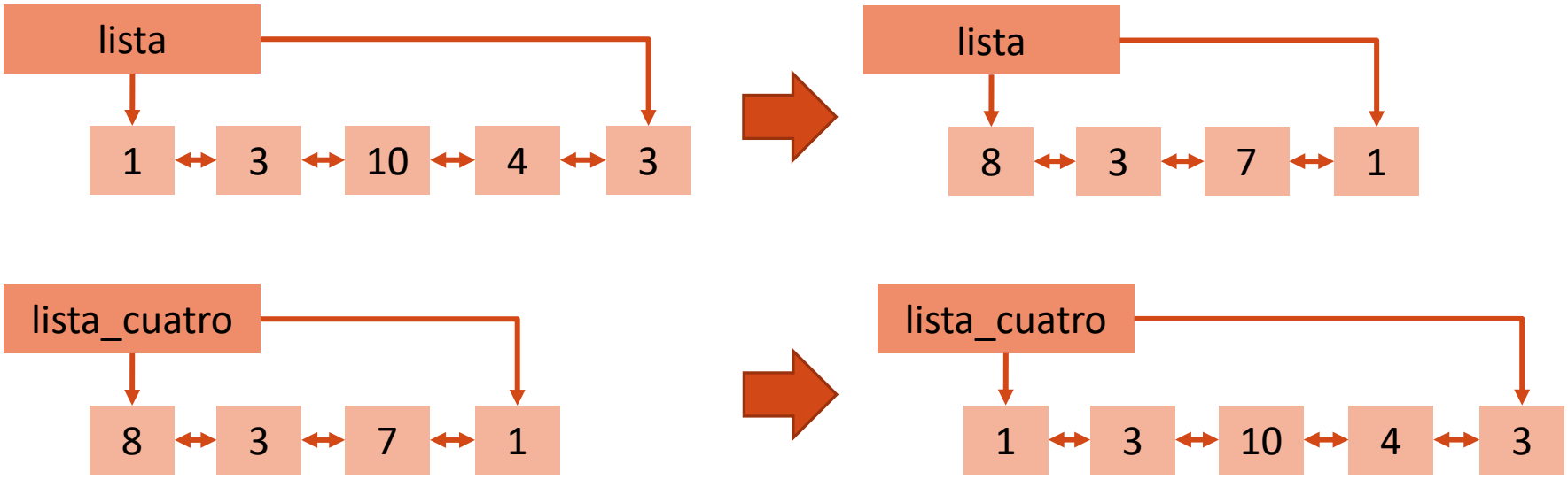
```
list<int> lista_cuatro(lista);
list<Carta> lista_cinco;
lista_cinco = lista_tres;
```



Librería STL: lista

- Intercambio:
  - Es una operación muy eficiente, y preferible a cualquier copia elemento a elemento

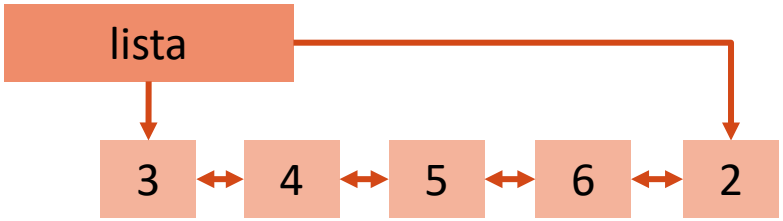
```
lista.swap(lista_cuatro);
```



Librería STL: lista

- Acceso a elementos:
  - **front()** y **back()** devuelven, pero no borran, el primer y último elemento de la lista
  - Para acceder a otros elementos, o bien se emplean iteradores, o bien se eliminan elementos hasta que el que se desee se convierta en el último o el primero

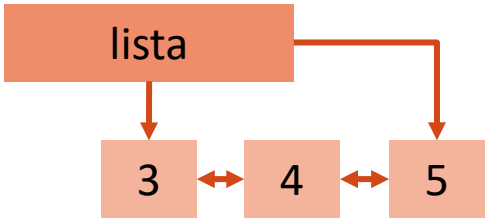
```
int inicio = lista.front(); // inicio = 3
int final = lista.back();   // final = 2
```

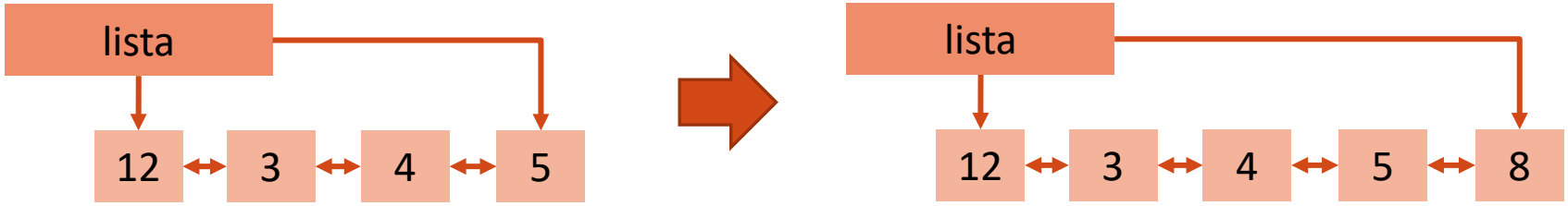


Librería STL: lista

- Inserción:
  - Se puede insertar tanto al principio como al final de la lista de forma muy eficiente

```
lista.push_front(12);
lista.push_back(8);
```

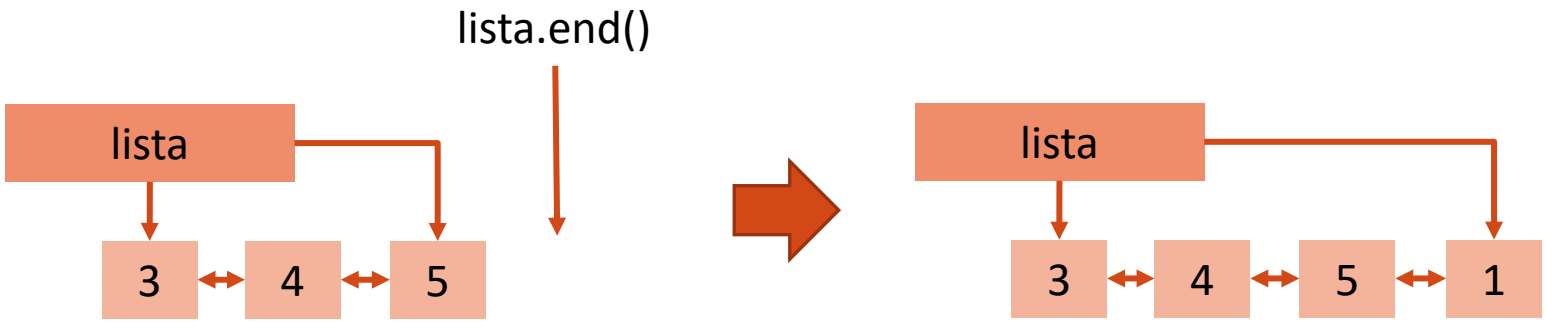




Librería STL: lista

- Inserción:
  - **insert()** inserta un elemento en mitad de la lista, concretamente justo antes del elemento denotado por el iterador

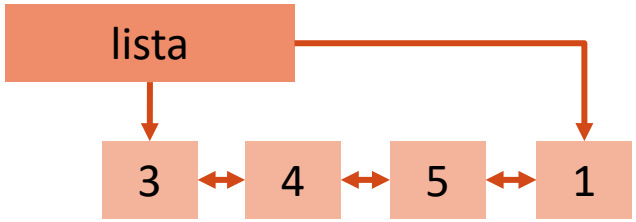
```
lista.insert(lista.end(),1);
```



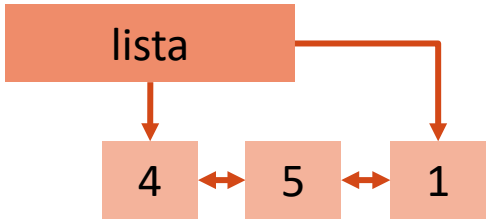
Librería STL: lista

- Borrado:
  - Podemos borrar el primer o el último elemento de la lista

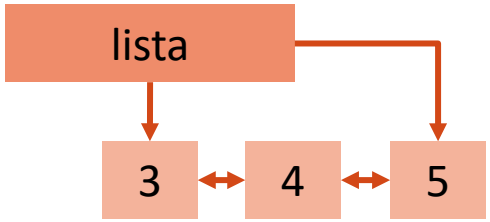
```
lista.pop_front();
lista.pop_back();
```



pop_front()



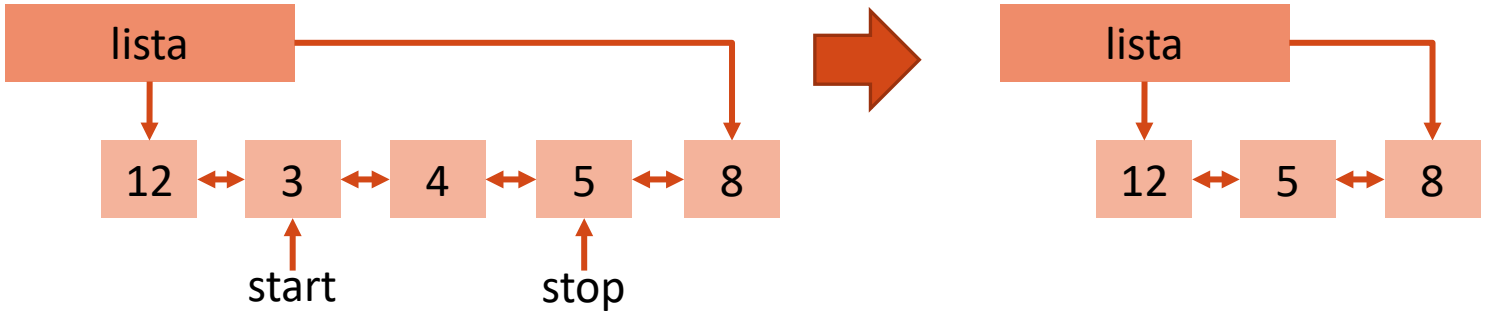
pop_back()



Librería STL: lista

- Borrado:
  - **erase()** borra el elemento apuntado por un iterador. Dicho iterador deja de ser válido una vez el elemento ha sido borrado, pero el resto no se ve afectado
  - Borrar un elemento intermedio es una operación muy eficiente

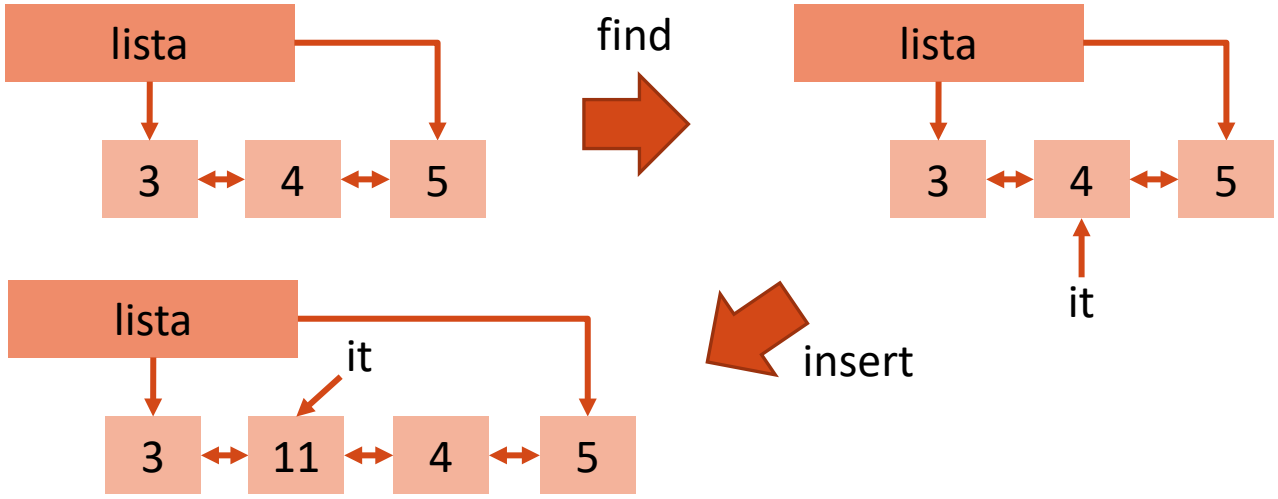
```
lista.erase(itr);
lista.erase(start,stop);
```



Librería STL: lista

- Encontrar un elemento:
  - El algoritmo genérico **find()** busca un elemento concreto de la lista. Si no lo encuentra, devuelve la posición final de la lista

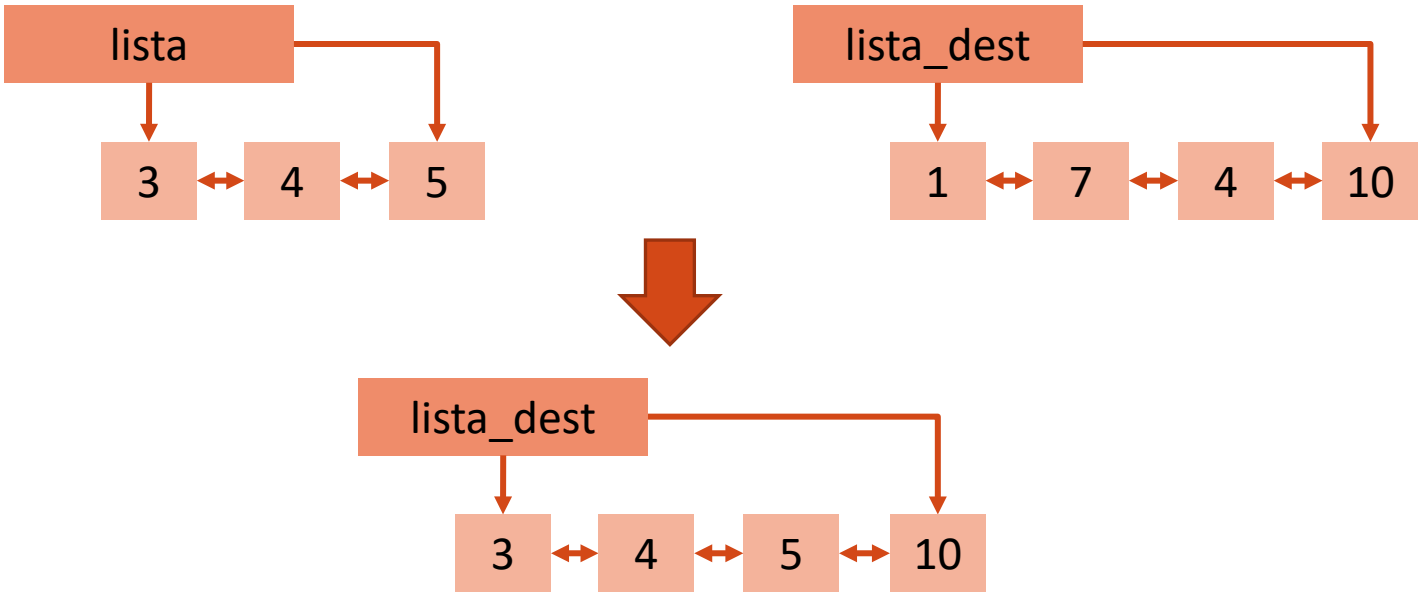
```
lista<int>::iterator itr = find(lista.begin(),lista.end(),4);
itr = lista.insert(itr,11);
```



Librería STL: lista

- Copiar:
  - El algoritmo genérico **copy()** copia un rango de una lista en otra

```
copy(lista.begin(),lista.end(),lista_dest.begin());
```



Librería STL: lista

- Operaciones del tipo lista:

Constructores y asignaciones

list<T> l;	Constructor por defecto
list<T> l(unaLista);	Constructor de copia
l = unaLista;	Asignación
l.swap (unaLista);	Intercambiar valores con otra lista

Acceso a elementos

l.front();	Primer elemento de la lista
l.back();	Último elemento de la lista

Librería STL: lista

- Operaciones del tipo lista:

Inserción

l.push_front(T);	Poner elemento al principio de la lista
l.push_back(T);	Poner elemento al final de la lista
l.insert(iterator,T);	Insertar antes de iterador

Tamaño

l.empty();	TRUE si la lista está vacía
l.size();	Nº de elementos de la lista

Librería STL: lista

- Operaciones del tipo lista:

Borrado

l.pop_front(T);	Extrae el primer elemento
l.pop_back(T);	Extrae el último elemento
l.erase(iterator);	Borra un elemento
l.erase(iter1,iter2);	Borra un rango de elementos
l.remove(T);	Borra las ocurrencias de un valor
l.remove_if(predicado);	Borra elementos condicionalmente

Librería STL: lista

- Operaciones del tipo lista:

Iteradores

list<T>::iterator itr;	Declara un nuevo iterador
l.begin();	Iterador de inicio
l.end();	Iterador final
l.rbegin();	Iterador inicial de marcha atrás
l.rend();	Iterador final de marcha atrás

Librería STL: lista

- Operaciones del tipo lista:

Varios

l.reverse();	Invierte el orden de los elementos
l.sort();	Ordena de forma ascendente
l.sort(comparacion);	Ordena usando la comparación
l.merge(lista);	Fusiona con otra lista ordenada

- Para hacer uso de algunas funciones, hay que sobrecargar correctamente los operadores == <

Índice de contenidos

Anexo

- Las distintas funciones pueden encontrarse en:
  - Vector:

    http://www.cplusplus.com/reference/vector/vector/
  - Lista:

    http://www.cplusplus.com/reference/list/list/

Índice de contenidos

Ejercicios

1. Empresa

Se quiere implementar un sistema de gestión de inventario para una empresa

Por simplicidad, los productos que almacenan se diferencian a través de un identificador numérico

El estado del inventario se implementa mediante dos listas:

- Una lista representa el stock de productos disponibles en oferta (el Almacén). Será una lista de productos
- Otra lista representa los pedidos de productos por parte de los clientes que en ese momento no están disponibles en el almacén (la Demanda). Será una lista de productos

Ejercicios

Para administrar el inventario tendremos dos operaciones:

- pedido_cliente(), que procesa los pedidos de los clientes
- aprovisionamiento(), que procesa la llegada de producto de un suministrador

El funcionamiento de la empresa es el siguiente:

- Si un cliente nos hace un pedido y lo tenemos en almacén se lo damos, sino lo añadimos a la lista de pedidos
- Cuando llega un cargamento de un determinado producto hemos de comprobar si algún cliente nos ha hecho el pedido. Si es así, lo servimos inmediatamente, y si no, lo metemos en el almacén como stock

Universidad Rey Juan Carlos
Escuela Técnica Superior Ingeniería de Telecomunicación

Programación de Sistemas de Navegación

2.6. Lenguaje C++

Librería STL: deque, pila, cola, mapa, multimapa

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

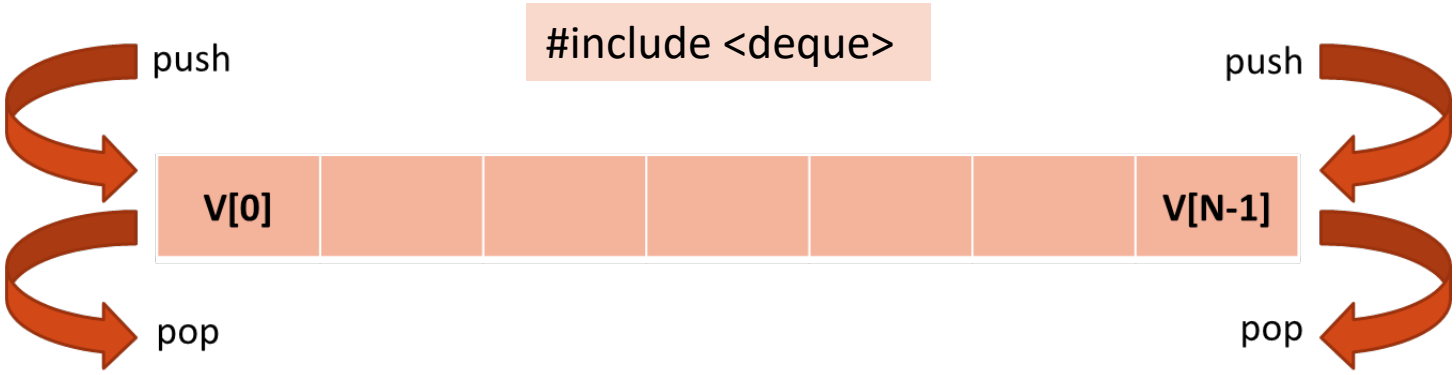
EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

Índice de contenidos

Colas de doble fin: deque

- Una cola de doble fin (deque) es un tipo de secuencia que como los vectores proporciona iteradores de acceso aleatorio
- Además es de tiempo constante en operaciones de inserción y borrado al principio y al final de los datos. La inserción y borrado de elementos en el medio es de tiempo lineal
- Esto es, una deque es una estructura de datos optimizada especialmente para insertar y eliminar elementos en los extremos



Librería STL: deque

- Operaciones del tipo deque:

Constructores

deque<T> d;	Constructor por defecto
deque<T> d(int);	Tamaño específico
deque<T> d(int, T);	Tamaño y valor iniciales
deque<T> q(unaDeque);	Constructor copia
d = unaDeque	Asignar una deque a otra

Tamaño

d.size();	Número de elementos contenidos
d.empty();	True si el vector está vacío

Librería STL: deque

- Operaciones del tipo deque:

Borrado

d.pop_front();	Borrar el primer elemento
d.pop_back();	Borrar el último elemento
d.erase (iterador);	Borrar un elemento concreto
d.erase (iter,iter);	Borrar un rango de elementos

Iteradores

deque<T>::iterator itr	Declarar un nuevo iterador
d.begin();	Iterador de inicio
d.end();	Iterador de fin

Librería STL: deque

- Operaciones del tipo deque:

Acceso e inserción

d[i];	Elemento 'i' de la deque
d.front();	Primer elemento de la colección
d.back();	Último elemento de la colección
d.insert(iterator,T);	Insertar antes de iterador
d.push_front(T);	Insertar valor al inicio de la deque
d.push_back(T);	Insertar valor al final de la deque

Índice de contenidos

Pilas y colas

- En esta sección estudiaremos los tipos de datos abstracto (plantilla) pila y cola
- Los puntos a estudiar son:
  - La abstracción de datos pila y cola
  - Adaptadores
- Como ya sabemos:
  - Una pila es una clase particular de lista en que todas las operaciones de inserción y borrado tienen lugar en un extremo denominado “cabeza” o “tope”. Son listas de tipo LIFO
  - Una cola es una estructura de datos lineal donde las eliminaciones se realizan por uno de sus extremos, denominado frente, y las inserciones por el otro, denominado final. Son listas de tipo FIFO

Pilas y colas: adaptadores

- No se puede construir una secuencia vector, list, deque a partir de otra de ellas sin pérdida de eficiencia
- Sin embargo, los tipos stack y queue son “adaptadores” (adaptors), y se denominan así porque se implementan no como tipos puros, sino que adaptan otros tipos ya definidos, es decir, no son contenedores separados, sino adaptadores de contenedores básicos
- Adaptan la interfaz de dichos contenedores, proporcionan nombres para aquellas operaciones que tengan sentido en el contexto de su uso como stack o queue

Pilas y colas: adaptadores

- Un adaptador de contenedor **solo proporciona una interfaz** restringida a un contenedor, en particular los adaptadores **no proporcionan iteradores**; se pretende que sólo se utilicen a través de sus interfaces especializadas
- Las técnicas que se usan para crear un adaptador de contenedor a partir de un contenedor son útiles, en general, para adaptar de modo no intrusivo la interfaz de una clase a las necesidades de los usuarios

pila

```
#include <stack> // std::stack

std::stack<vector<int>> pila;
```

cola

```
#include <queue>  // std::priority & std::priority_queue

std::queue <list<int>> cola;
std::priority_queue <deque<int>> cola_prioridad;
```

Pilas y colas: adaptadores

- Operaciones con pila (stack):

Constructores

stack<T> pila;	Constructor defecto, pila vacía
stack<T, container<T>> pila;	Constructor defecto usando container
stack<T> pila (container);	Constructor copia
stack<T, container<T>> pila (cont);	Constructor copia

- Container puede ser un contenedor de los vistos en clase. Por defecto es un **deque** (cola de doble fin), pero podemos utilizar **list** o **vector**

Pilas y colas: adaptadores

- Operaciones con pila (stack):

Funciones miembro

empty();	Indica si la pila está vacía
size();	Tamaño de la pila
top();	Referencia al elemento de la cima
push(T);	Insertar un elemento
pop();	Eliminar el elemento de la cima
swap(container);	Intercambia dos pilas

Pilas y colas: adaptadores

- Operaciones con cola (queue):

Constructores

queue<T> cola;	Constructor defecto, cola vacía
queue<T, container<T>> cola;	Constructor defecto usando container
queue<T> cola (container);	Constructor copia
queue<T, container<T>> cola (cont);	Constructor copia

- Container puede ser un contenedor de los vistos en clase. Por defecto es un **deque** (cola de doble fin), pero podemos utilizar **list** o **vector**

Pilas y colas: adaptadores

- Operaciones con cola (queue):

Funciones miembro

empty();	Indica si la cola está vacía
size();	Tamaño de la cola
front();	Referencia al primer elemento (sig.)
back();	Referencia al último elemento
push(T);	Insertar un elemento
pop();	Eliminar el siguiente elemento
swap(container);	Intercambia dos colas

Índice de contenidos

Mapas: asociaciones

- De forma similar a un vector tradicional, un **vector** (o array) **asociativo** es una estructura de datos indexada:
  - Los elementos de la estructura son accesibles gracias a un índice que los identifica
  - A diferencia de los vectores, **el índice** no tiene por qué ser obligatoriamente un entero, sino que **puede ser cualquier tipo de dato ordenado**
  - El ejemplo típico es el de un diccionario, en el que accedemos a definiciones (los datos) mediante palabras (los índices)
- Los mapas (o diccionarios) son tipos concretos de vectores asociativos

```
#include <map>
```

Mapas: definición

- Un **mapa** es una secuencia de **pares (clave,valor)** en el que cada "valor" se identifica (indexa) acorde a su "clave"
- Esto permite recuperar los valores rápidamente a partir de la clave:
  - Como mucho se almacena un valor para cada clave
  - En otras palabras, cada clave de un mapa es única
  - Un mapa proporciona iteradores bidireccionales
- El mapa requiere que exista una operación "<" ("menor que") para sus tipos clave y mantiene sus elementos ordenados para que se puedan realizar iteraciones en orden

Mapas: definición

- Contenedor asociativo:

```
template <class Key, class Value> class Asoc {
        public:
                Value& operator[](const Key&);
                // devuelve una referencia al Value que corresponde con Key
        // ...
};
```

- Una clave de tipo "Key" da nombre a un valor asociado de tipo "Value". Los contenedores asociativos son una generalización de la noción de array asociativo
  - El **map** (mapa) es un vector asociativo tradicional, en el que se asocia un sólo valor a cada **clave única**
  - El **multimap** (multimapa) es un vector asociativo que permite **elementos duplicados** para una clave dada

Mapas: definición

- Un ejemplo son los días de la semana, dado un día (clave) tenemos el número de día en el mes (valor), esto es lo que se conoce como par<clave,valor>:

Clave (key)	Valor (value)
“Lunes”	15
“Martes”	16
…	
“Domingo”	21

```
#include <utility>

pair<key, value> p;
p.first; // Devuelve key
p.second; // Devuelve value
```

```
#include <iostream>
#include <map> // incluye pair
using namespace std;
typedef pair<string, int> dia_semana;

int main() {
        dia_semana lunes("Lunes",15); // Usando typedef
        pair<string, int> martes("Martes",16); // Sin usar typedef

        map<string, int> semana;
        semana.insert(lunes);
        semana.insert(martes);
        map<string, int>::iterator it = semana.begin();
        while (it != semana.end())  {
                cout << "Dia: " << it -> first << " - Numero: " << it -> second << endl;    ++it;
        }
        return 0;
}
```

Mapas: definición

- En la función **insert**, par_key_value es del tipo pair<K, V> que se encuentra en el espacio de nombres std de map. Esta operación devuelve un pair<iterador,bool> que es **true** y el **iterador nulo** si ha podido insertar el elemento, sino, devuelve **false** y el **iterador al elemento existente**
- Su uso sería:

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

int main() {
        map<char,int> m;
        pair<map<char,int>::iterator,bool> in1 = m.insert ( pair<char,int> ('j',32) );
        if (in1.second) cout << "Elemento no existe" << endl;
        else cout << "Elemento ya existe: " << in1.first -> second << endl;

        pair<map<char,int>::iterator,bool> in2 = m.insert ( pair<char,int> ('j',32) );
        if (in2.second) cout << "Elemento no existe" << endl;
        else cout << "Elemento ya existe: " << in2.first -> second << endl;

        return 0;
}
```

Salida:

```
Elemento no existe
Elemento ya existe: 32
```

Librería STL: map y multimap

- Operaciones del tipo map y multimap:

Constructores

map<K, V> m; multimap<K, V> m;	Constructor por defecto
map<K, V> m (itr_begin, itr_end); multimap<K, V> m (itr_begin, itr_end);	Constructor parametrizado utilizando un rango
map<K, V> d(unMap); multimap<K, V> d(unMultimap);	Constructor copia

Iteradores

map<K, V>::iterator itr; multimap<K, V>::iterator itr;	Iterador
m.begin();	Iterador inicio
m.end();	Iterador fin

Librería STL: map y multimap

- Operaciones del tipo map y multimap:

Inserción y borrado

m[key];	Referencia elemento asociado a la 'clave'
m.insert(par_key_value);	Inserta un par (clave,valor)
m.erase(key);	Borra elemento asociado a una clave
m.erase(iterator);	Borra elemento especificado por iterator

Librería STL: map y multimap

- Operaciones del tipo map y multimap:

Funciones y capacidad

m.empty();	TRUE si m está vacío
m.size();	Número de elementos de m
m.count(key);	Número de elementos con clave 'key'
m.find(key);	Busca un valor con clave 'key'
m.lower_bound(key);	Itr con la primera ocurrencia de 'key'
m.upper_bound(key);	Itr con la ultima ocurrencia de 'key'
m.equal_range(key);	Rango de elementos iguales

Índice de contenidos

Anexo

- Las distintas funciones pueden encontrarse en:
  - Deque:
    http://www.cplusplus.com/reference/deque/deque/
  - Pila:
    http://www.cplusplus.com/reference/stack/stack/
  - Cola:
    http://www.cplusplus.com/reference/queue/queue/
  - Mapa:
    http://www.cplusplus.com/reference/map/map/
  - Multimapa:
    http://www.cplusplus.com/reference/map/multimap/

Índice de contenidos

Ejercicios

1. Se quiere crear una estructura para la gestión de usuarios, de manera que cada nombre de usuario está asociado a un ID único. Suponiendo que el ID es el número de su DNI sin letra, se pide:
   - Crear una clase que albergue la estructura necesaria para la gestión de los nombres de usuarios y su ID
   - Proporcionar el método necesario para devolver el nombre del usuario asociado a un ID
   - Mostrar todos los usuarios existentes en el sistema
   - Dado un nombre, devolver todos los IDs asociados a usuarios con dicho nombre

Universidad Rey Juan Carlos
Escuela Técnica Superior Ingeniería de Telecomunicación

Programación de Sistemas de Navegación

2.7. Lenguaje C++

Manejo de excepciones

JOSÉ MIGUEL GUERRERO HERNÁNDEZ

EMAIL: JOSEMIGUEL.GUERRERO@URJC.ES

Índice de contenidos

Índice de contenidos

Manejo de excepciones

- Las excepciones son en realidad errores durante la ejecución
- Si se produce un error y no se implementa un sistema de manejo de excepciones, el programa termina de forma abrupta. En ese caso, es probable que:
  - Si hay ficheros abiertos no se guarde el contenido, ni se cierren
  - Los objetos no serán destruidos, y se producirán fugas de memoria
- En programas pequeños podemos prever las situaciones en que se pueden producir excepciones y evitarlas

Manejo de excepciones

- Las excepciones más habituales son las de peticiones de memoria fallidas
- Cuando solicitamos memoria al sistema, lo habitual es que exista suficiente memoria disponible y no haya ningún problema
- Pero, ¿qué ocurre si dicha memoria no se concede?

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

int main() {
        int* x = new int[100000000];
        return 0;
}
```

```
terminate called after throwing an
instance of 'std::bad_alloc'
  what():  std::bad_alloc
```

Manejo de excepciones

- El manejo de excepciones consiste en transferir la ejecución del programa desde el punto donde se produce la excepción a un manipulador que coincida con el motivo de la excepción
- Esto se consigue "encerrando" el código susceptible a fallos en un bloque **try {}**
- Si alguna de las instrucciones de ese bloque provocan una excepción en el flujo del programa, se dirige al final de ese bloque, buscando un bloque **catch (…) {}** que capture la excepción
- Si no existe el bloque catch, el programa terminaría
  - El bloque **catch(...)** captura cualquier excepción
- El tratamiento que se efectúa tras una excepción, es lo que se implementa dentro del bloque catch

Manejo de excepciones

- Por lo general, un bloque de manejo de excepciones tiene la siguiente forma:

```
try {
        // Código donde se lanza la excepción
}
catch (tipo_excepción parametro) {
        // Qué hacer cuando ocurra la excepción
}

…. // Todas los tipos que se quieran catch() {}
catch (…) {
        // Qué hacer cuando ocurre una excepción
        // que no tenemos prevista que ocurra
}
```

Este bloque puede repetirse tantas veces como se quiera

Manejo de excepciones

- Siguiendo con el ejemplo anterior, en este caso, ahora al producirse la excepción, el programa muestra por pantalla que no tiene la memoria suficiente:

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

int main() {
	try {
		int* x = new int[100000000];
	} catch (...) {
		cout << "No hay memoria suficiente." << endl;
	}
	return 0;
}
```

Manejo de excepciones

- Como hemos visto, catch (…) captura cualquier excepción, pero es conveniente identificar cada una de ellas para poder hacer un tratamiento correcto
- Si se usa una clase escrita por terceras personas o de las que se incluyen con el compilador y desea utilizar el mecanismo try, se debe conocer el tipo de excepción lanzado por dicha clase para así poder escribir el catch correspondiente para el tratamiento de dicho error

Manejo de excepciones

- En el caso anterior, lo correcto sería controlar la excepción **bad_alloc**, que nos indica que ha habido un problema con la reserva de memoria (**new**):

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

int main() {
	try {
		int* x = new int[100000000];
	} catch (bad_alloc&) {
		cout << "No hay memoria suficiente." << endl;
	}
	return 0;
}
```

Manejo de excepciones

- En el ejemplo anterior, la excepción se lanza de forma implícita, ya que es una excepción estándar
- Pero nosotros también podemos incluso forzar a que cuando ocurra un comportamiento no deseado, se lance una excepción de forma explícita
- Para ello se utiliza el comando **throw**

```
try {
        throw 'w'; // El tipo sería char
} catch (char c) {
        cout << "Excepción con valor: " << c << endl;
}
```

Manejo de excepciones

- Las excepciones pueden estar anidadas:
  - Cuando se produce una excepción se busca un manipulador apropiado en el rango del try actual
  - Si no se encuentra se retrocede al anterior, de modo recursivo, hasta encontrarlo. Cuando se encuentra se destruyen todos los objetos locales en el nivel donde se ha localizado el manipulador, y en todos los niveles por los que hemos pasado

```
try {
        try {
                try {
                        throw 'x'; // valor de tipo char
                }
                catch(int i) {}
                catch(float k) {}
        } catch(unsigned int x) {}
} catch(char c) {
        cout << " Excepción con valor: " << c << endl;
}
```

Clase exception

- Los errores generados por las librerías estándar de C++ pueden ser capturados por un catch que tome un parámetro tipo **exception**

```
string s = "Hola";
try {
        cout << s.at(100) << endl;
} catch(exception& e) {
        cout << e.what() << endl;
}
```

La función **at()** de la clase **string**, lanza una excepción cuando se trata de leer o escribir un componente fuera de rango

- Realmente, **exception** es una clase base de la cual se puede heredar y sobrescribir los métodos para el tratamiento de excepciones. La clase **exception** está incluida en la librería **<exception>** y su estructura es la siguiente:

```
class exception {
        public:
                exception() throw() { }
                virtual ~exception() throw();
                virtual const char* what() const throw();
};
```

Clase exception

- Si se desea crear una excepción, basta con crear una nueva clase que sobrescriba los métodos:

```
class division_cero : public exception {
        public:
        const char* what() const throw() {
                return "Error: división por cero";
        }
};
```

- Ahora se podrían lanzar excepciones de este tipo, por ejemplo:

```
double num, den;
cin >> num; cin >> den;
try {
        if (den == 0) throw division_cero();
        cout << num << " / " << den << " = " << num / den << endl;
} catch(exception& e) {
        cout << e.what() << endl;
}
```

Índice de contenidos

Ejercicios

1. Escribe un programa que genere un número aleatorio e indique si el número generado es par o impar. El programa utilizará para ello el lanzamiento de una excepción

2. Escribe un programa que juegue con el usuario a adivinar un número. El ordenador debe generar un número entre 1 y 500, y el usuario tiene que intentar adivinarlo. Para ello, cada vez que el usuario introduce un valor, el ordenador debe decirle al usuario si el número que tiene que adivinar es mayor o menor que el que ha introducido el usuario. Cuando consiga adivinarlo, debe indicárselo e imprimir en pantalla el número de veces que el usuario ha intentado adivinar el número. Si el usuario introduce algo que no es un número, debe indicarlo en pantalla, y contarlo como un intento

Universidad Rey Juan Carlos

Escuela Técnica Superior
Ingeniería de Telecomunicación