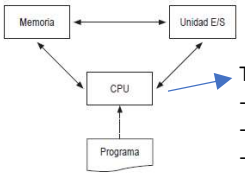
FP

UNIDAD 1: CONCEPTO DE PROGRAMA INFORMÁTICO

Introducción a las computadoras

Computadora: máquina capaz de realizar un elevado número de operaciones en breve espacio de tiempo y con gran precisión.



Lógicas y matemáticas

Tipos de instrucciones:
- Entrada
- Salida
- Cómputo: operac. aritméticas, lógicas y de memoria
- Control

La programación

Proceso de planificar y desarrollar la resolución de un problema mediante una computadora**.** El resultado es un programa que se ejecutará en una computadora para resolver un problema.

Programa

Secuencia de instrucciones codificadas de una forma determinada y registradas en un soporte informático. Al ejecutarlo la computadora realiza una serie de operaciones preestablecidas para resolver el problema.

Computadoras: lenguaje binario (lenguaje máquina) ← Programador: lenguaje alto nivel (C++, Java, Python,..)

Pseudocódigo

Mezcla de lenguaje natural y expresiones matemáticas que permite escribir de modo preciso un programa.

Paso previo a la codificación del programa en lenguaje de programación

Expresa de forma clara, precisa, sin ambigüedad y usando lenguaje natural (al igual que las notaciones gráficas)

Reglas generales:

- Empieza y termina con **Inicio** y **Fin**
- 1 instrucción -> 1 línea
- Palabras reservadas: **Inicio, Fin, Si, Entonces, Si no, Fin_si, Mientras, Fin_mientras, Leer, Escribir, Según**,.
- Usar ***indentación*** para mostrar dependencias entre instrucciones
- Cada estructura control tiene Comienzo y Fin
- Usar mayúsculas solo para nombre elegidos por programador (por ejemplo: nombre variables)

Estructuras de control

Secuencia

Grupo de acciones que se ejecutan una detrás de otra en el orden escrito, sin omitir nada

Condiciones

Según se cumpla una condición se ejecuta uno u otro conjunto de instrucciones

Simple

Si se cumple realiza un conjunto de acciones, si no se cumple no realiza ninguna acción

```
...
   Si condición
      Entonces
         Acción 1
         Acción 2
         ...
         Acción n
   Fin_si
...
```

Doble

Si se cumple realiza un conjunto de acciones, si no se cumple realiza otro conjunto de acciones

```
Inicio
   Leer A y B
   Si A > B
      Entonces
         Mostrar "El mayor es: " A
      Si no
         Mostrar "El mayor es: " B
   Fin_si
Fin
```

Múltiple

Según estructura de control ejecuta uno de los grupos de acciones u otra cuando no coincida ninguno.

```
...
   Según expresión
      = 1 Secuencia de acciones 1
      = 2 Secuencia de acciones 2
      ...
      = n Secuencia de acciones n
      En otro caso
         Secuencia de acciones
   Fin_según
...
```

Repeticiones

Ejecuta repetidamente grupo de acciones dependiendo de una condición (condición de salida).

Mientras

- Repite mientras sea cierta
- Se evalúa al comienzo
- Puede no realizarse nunca

```
...
   Mientras condición
      Acción 1
      Acción 2
      ...
      Acción n
   Fin_mientras
...
```

Hasta

- Repite hasta que sea cierta
- Se evalúa al final
- Se realiza al menos 1 vez

```
...
   Repetir
      Acción 1
      Acción 2
      ...
      Acción n
   Hasta condición
...
```

Para

- Número de repeticiones fijo o conocido
- Definir variable Contador, valor inicial, final e incremento
- Se evalúa al comienzo
- Se realiza al menos 1 vez

```
...
   Para VAR desde VI hasta VF incremento INC
      Acción 1
      Acción 2
      ...
      Acción n
   Fin_para
...
```


Realizado por Juan del Río – juandrh2.0@gmail.com

Fuente: Lizcano Casas David, et al. 2018. *Fundamentos de la programación*. Madrid: Editorial CEF.

FP

UNIDAD 2: ALGORITMOS Y SISTEMAS DE REPRESENTACIÓN DE UN PROGRAMA – 1 de 2

Algoritmo

Conjunto finito de reglas que crean una serie de operaciones para resolver un tipo específico de problema

Del latín *dixit algorithmus*, del persa al-Jwarizmi

Definición formalizada por el modelo computacional de la máquina de Turing

Reciben una entrada y la transforman en una salida (caja negra)

Para ser ejecutado necesita ser implementado en:
- Lenguaje de programación
- Circuito eléctrico
- Aparato mecánico
- …

Características básicas:
- Ser finito
- Ser definible, sin ambigüedades
- Recibir datos de entrada
- Proporciona resultados de salida
- Ser efectivo: rápido y eficiente

Descripción en 3 niveles:
- Informal en lenguaje natural
- Formal en pseudocódigo o diagramas flujo
- Implementación: en lenguaje de programación específico

Algoritmos **computables (decidibles)**:para todo posible valor de las variables de entrada, el algoritmo se detiene dando como salida la solución correcta al problema.

Diagramas de flujo y ordinogramas

Independientes lenguaje programación
Evita ambigüedades (como el pseudocódigo)

Diagrama de flujo

Representación gráfica de un algoritmo o de una secuencia rutinaria

Utilizan símbolos unidos por flechas.
Utilizado inicialmente en economía, programación y procesos industriales, posteriormente en otras disciplinas

Características:
- Existe punto de inicio
- Existe camino hasta la solución
- Existe por lo menos un camino para cualquier punto del diagrama

Recomendaciones:
- Evitar sumideros infinitos (entradas sin salida)
- Evitar burbujas de generación espontánea (salidas sin entrada)
- Evitar flujos y procesos no etiquetados (puede esconder errores) Por ejemplo: que no pertenezca al diagrama

Ventajas:
- Favorece la compresión
- Permite identificar problemas y posibilidad de mejorarlo: pasos redundantes, flujos de retroalimentación, cuellos botella y puntos de decisión

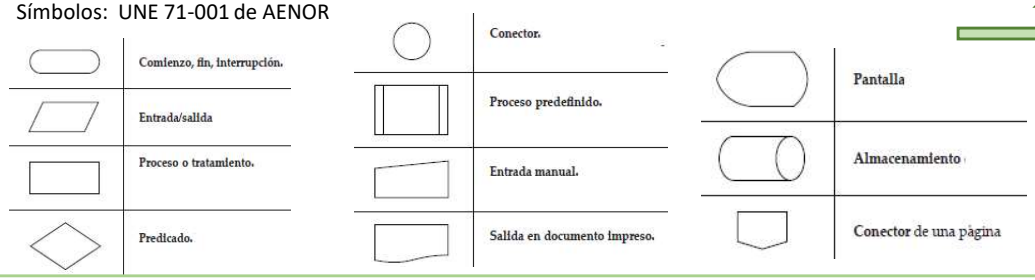
Tipos de formato:
- Vertical
- Horizontal
- Panorámico
- Arquitectónico

Ordinograma

Es un diagrama de flujo en Ciencias de la Computación

Muestra la secuencia lógica y detallada de las operaciones que se necesitan para la realización de un programa

Símbolos: UNE 71-001 de AENOR



Reglas de diseño ordinograma:
- Inicio y fin
- Secuencia: flechas y conectores
- Orden: arriba-abajo, izquierda- derecha
- Un símbolo por acción (salvo que sean simples)
- No usar instrucciones propias de un lenguaje
- A todos los símbolos llega y sale una sola (salvo Inicio, Fin y conectores)
- Etiquetar Entradas y salidas para aclarar enlace
- Todos los símbolos unidos por líneas de flujo
- No se pueden cruzar líneas de flujo
- Al inicio no puede llegarle línea
- Del fin no pude salir ninguna

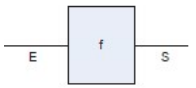
Programación estructurada

Teorema de la estructura, Böhm y Jacopini, 1966 -> *Notes on Structured Programming*, Edsger W. Dijkstra

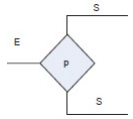
Busca:
- Programación arriba-abajo arborescente fraccionamiento en módulos.
- Facilitar mantenimiento
- Dar estructura fuerte
- Proporcionar capacidad de programar sin errores

Diagrama propio: 1 punto de entrada y salida
Programa propio: el que cumple:
- Es diagrama propio.
-Todo elemento es accesible
- No bucles infinitos

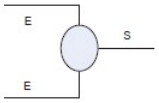
Diagramas



Tratamiento o Procedimiento puro:
- Asignación
- Secuencia
- subprograma
- Parte con 1 entrada y 1 salida



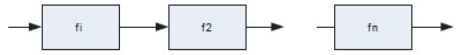
Predicado, condición, exp. booleana o test p:
Según p toma una de las salidas



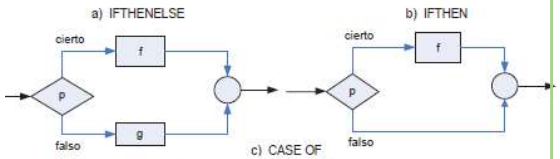
Reagrupamiento:
Unir varias salidas

Diagramas estructurados o privilegiados

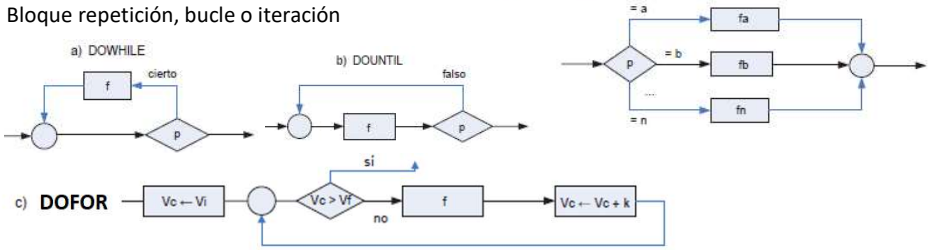
- Bloque secuencial, secuencia o encadenamiento



- Bloque condicional, condición o alternativa



- Bloque repetición, bucle o iteración



Programa estructurado

Dada familia E de diagramas privilegiados, un prog. Propio está estructurado si se puede representar exclusivamente con los diagramas de E.

Programar sin "**GOTO**"

Realizado por Juan del Río – juandrh2.0@gmail.com

Fuente: Lizcano Casas David, et al. 2018. *Fundamentos de la programación*. Madrid: Editorial CEF.

UNIDAD 2: ALGORITMOS Y SISTEMAS DE REPRESENTACIÓN DE UN PROGRAMA – 2 de 2

Programación estructurada (cont.)

Teoremas de programación estructurada

Teorema de estructura (existencia): todo lo que se puede programar se pude poner bajo la forma de estructurada (usando diagramas BLOCK, IFTHENELSE, DOWHILE y/o DOUNTIL)

Corolario de arriba abajo: todo programa propio es equivalente a un diagrama de una de las siguientes formas: BLOCK (f,g), IFTHENELSE (p,f,g), DOWHILE (p,f) y/o DOUNTIL (f,g)

Donde: p = predicado del programa original y f y g = programas reducidos o subprogramas

Teorema de corrección o validación: validación por pasos sucesivos examinado cada nodo y probando localmente.

Teorema de descomposición: la descomosición de una función k obedece las reglas:

Descomposición k = BLOCK(f,g):
- ***f*** definida intervalo donde está ***k***
- Si dos datos diferentes, ***k*** resultados diferentes -> ***f*** lo mismo
- Una vez elegida ***f***, ***g*** queda determinada

Descomposición k = IFTHENELSE(p,f,g):
- ***p*** definida intervalo donde está ***k***
- Una vez elegida ***p***, ***f*** y ***g*** quedan determinadas

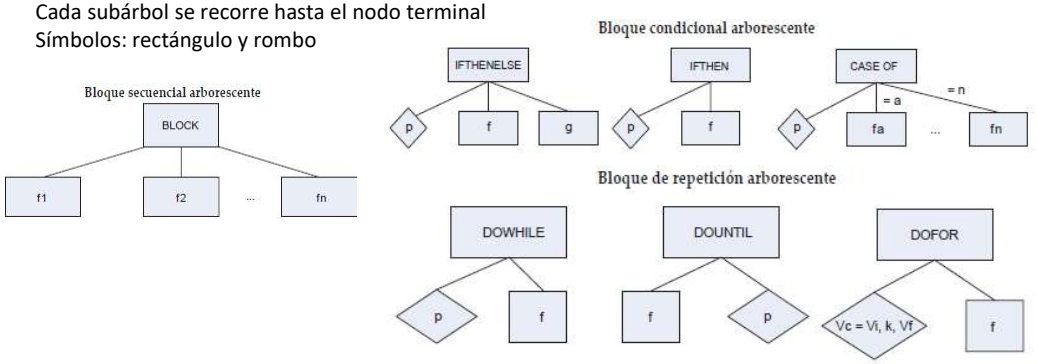
Descomposición k = DOWHILE (p,f):
- Para todos los estados de entrada de ***f*** , que producen los mismos resultados, k debe consistir en no hacer nada (función identidad)
- Si se cumple lo anterior, la descomposición siempre es posible
- ***f*** es una función que ejecuta ***p*** a todos los datos
- Una vez elegida ***f***, ***p*** queda determinada

Si se cumple T. Estructura -> se puede realizar descomposición hasta el final

Si se cumple T. Corrección -> se puede validar a medida que avanza el trabajo

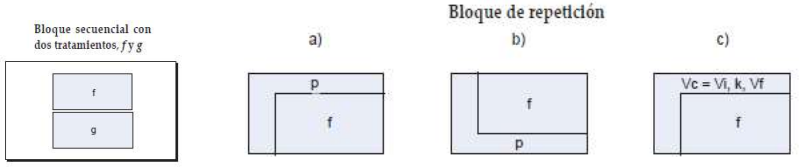
Diagramas estructurados arborescentes

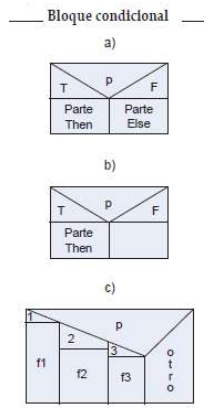
Diagramas estructurados -> representación arborescente basada en Método Tabourier.
La raíz del árbol y cada una de sus ramas son diagramas estructurados.
Lectura de arriba->abajo y izquierda->derecha
Cada subárbol se recorre hasta el nodo terminal
Símbolos: rectángulo y rombo



Diagramas estructurados metodología Nassi-Shneiderman o de Chapin

No facilitan la tarea del programador, pide mayor comprensión
Beneficios: mejora en depuración, documentación y mantenimiento
Tamaño reducido, no usan flechas





Ventajas diagramas estructurados respecto a los ordinogramas clásicos

- Además ayudan a ver mejor la estructura y enlace entre módulos.
- Programas cortos, si es grande se aísla en otros subprogramas

Arborescentes son fáciles de seguir y actualizar (al contrario que los NS/Chapin)
NS-Chapin:
- Ocupan la mitad
- Se puede usar convención para resaltar funciones (arriba e izquierda)

Ordinogramas clásicos: mayor claridad en E/S

Realizado por Juan del Río – juandrh2.0@gmail.com

Fuente: Lizcano Casas David, et al. 2018. *Fundamentos de la programación*. Madrid: Editorial CEF.

FP

UNIDAD 3: TRATAMIENTO INFORMÁTICO DE UN PROBLEMA

Introducción

Ciclo de vida de un programa: describe que ocurre desde que se decide desarrollar un programa hasta que deja de utilizarse.

ETAPAS

Definición de los requisitos del problema

Describir el problema + identificar necesidades (objetivos)
¿Se puede resolver informáticamente?

Pasos:

- Identificar necesidades del cliente → ¿de qué trata el problema y qué se desea obtener?
- Realizar un estudio técnico del sistema → Rendimiento, fiabilidad, facilidad mantenimiento y posibilidad de producción
- Realizar un estudio económico del sistema → Evaluación costes y beneficios previstos
- Establecer restricciones de tiempo y coste
- Evaluar viabilidad del sistema → Análisis viabilidad (económico, técnico y legal)
  Análisis de riesgos:
  - de proyecto: (presupuestarios, agenda, personal)
  - Técnicos (problemas diseño, implantación, mantenimiento..)
  - De negocio (producto no interesa, no se sabe comercializar…)
- Asignar funciones al sistema
- Definir el sistema → Hardware, software, bases de datos, personas…

Crear un modelo de la arquitectura que describa relaciones entre los elementos (interfaz de usuario, entradas, función y control del sistema, salidas, mantenimiento y autocomprobación).

+ Diagrama de contexto de arquitectura: limites de información entre sistema y entorno
+ Diagrama de flujo de arquitectura: subsistemas principales y líneas de flujo de información principales

Resultado:

- Documento "**Especificación del sistema**" -> base para la ingeniería de hardware, software, bases
- Si es viable -> "**Plan de software**" : Objetivos generales, requisitos de software, de bases de datos y arquitectura.

Análisis de requisitos de software

Proceso de descubrimiento, refinamiento, modelado y especificación que lleva a cabo ingeniero de software (analista)

Resultado:

- Se definen flujos de información, estructuras primarias de datos, características funcionales, requerimientos de rendimiento y restricciones impuestas por el cliente
- Criterios globales de validación de los requisitos

Unidades de tratamiento (UT): datos de entrada al proceso, funcionalidad y resultados de salida

Documento "**Especificación de requerimientos de software** (o especificaciones funcionales)":

- Introducción: describe fines y objetivos del software
- Descripción de la información
- Descripción funcional (incluye diagrama de soporte)
- Criterios de validación
- Bibliografía y apéndices: tablas de datos, descripción algoritmos, diagramas, gráficos y otros…

Diseño

Proceso de aplicar distintas técnicas y principios con el propósito de definir un dispositivo, proceso o sistema con los suficientes detalles como para permitir su realización física.

Parte central de la ingeniería de software → dedicar tiempo suficiente
Objetivo: producir pseudocódigo del programa + definir estructuras de datos

Diseño general: estructura funcional y modular del sistema, interfaces entre módulos y estructuras datos
Diseño arquitectónico modular.

Diseño detallado: algoritmos de cada unidad de tratamiento o proceso y del módulo principal.
Se produce el pseudocódigo de todo el programa.

- Procesos complejos: diseñar sistema de pasos o diagrama y después algoritmo estructurado
- Procesos simples: describir pseudocódigo estructurado

Resultado: **Documento de diseño detallado** o **Documento de diseño final,** incluye especificación de las pruebas a realizar.

Codificación

Creación de un programa informático utilizando un lenguaje de programación, tomando como punto de partida el pseudocódigo resultante del diseño detallado.

Resultado: código fuente del programa

Estilo compresible:

- Bien documentado: identificadores significativos, comentarios en cada módulo , descripción estructuras de datos e historia del módulo
- Usar sangrado y guías de estilo oficiales
- Declarar datos en lugares de acceso rápido y los más cerca del segmento de código
- Sentencias simples y fáciles de seguir: evitar condicionales complicados, negativos, usar paréntesis,…
- Especial atención a las entradas y salidas: validar datos de entrada, usar indicativos de fin de datos..

Pruebas

Preparación de conjuntos de datos de entrada y sus correspondientes salidas para verificar que el programa funciona correctamente.

Elemento crítico para la calidad del software
Grupo de **pruebas para cada fase** del ciclo de vida:

- Def. requisitos del problema -> comprobar todo el sistema (fuera de los límites de la ingeniería de software)
- Análisis -> pruebas de validación (satisface todas las expectativas del cliente)
- Diseño -> probar cada módulo y ver que funcionan globalmente
- Codificación -> prueba de unidad de cada módulo : de caja blanca y caja negra:
  - Interfaz del modulo, entradas/salidas..
  - estructuras datos locales
  - valores límite
  - estructuras de control: comprobar que todo se ejecuta por lo menos una vez
  - tratamiento de errores: se ejecutan y son adecuados

Pruebas:

- De caja negra: sobre el interfaz del software: acepta bien las entradas y produce salida correcta. No lógica interna.
- De caja blanca: examen de los detalles procedimentales. Comprobar caminos lógicos.

Implementación del software: codificación + validación mediante pruebas
TDD: Test-driven Development: realizar primero las pruebas y luego construir módulos

Mantenimiento

Gestión y realización de cambios en el programa para mantenerlo actualizado mientras dure su ciclo de vida.

Correctivo: errores que surgen tras la implantación
Perfectivo: ampliación del programa
Adaptativo: a características del entorno externo: técnicas (otros sistema operativo) o de otro tipo

Documentación disponible y adecuada
Esquemas de acciones ante errores o modificación software

Realizado por Juan del Río – juandrh2.0@gmail.com

Fuente: Lizcano Casas David, et al. 2018. *Fundamentos de la programación*. Madrid: Editorial CEF.

FP

UNIDAD 4: LA PROGRAMACIÓN Y LOS LENGUAJES INFORMÁTICOS 1/3

La programación

Proceso de creación de un programa informático en el cual se diseña una estructura, se codifica en un lenguaje de programación, se prueba, se depura y se mantiene el código

Pasos:
- Reconocer necesidad de crear un programa
- Especificar requisitos
- Realizar análisis de los requisitos
- Diseñar la arquitectura
- Implementar: codificar y probar
- Implantar: instalar, poner en funcionamiento

Factores de calidad:
- Corrección: hace lo que debe hacer, antes especificar bien y al final compararlo.
- Claridad: para facilitar posterior desarrollo y mantenimiento
- Eficiencia: hacer lo correcto optimizando recursos: Tiempo, memoria necesaria, espacio disco, tráfico red…
- Portabilidad: capacidad de ejecutarse en cualquier plataforma

Si se quiere cierta **garantía de calidad**: usar algún modelo de desarrollo (estructurado, orientado a objetos)

Paradigmas de programación

Modelo básico de construcción de programas informáticos. Estilos que definen la estructura interna y que será en función del tipo de problema. Conjunto de modelos conceptuales que, juntos, modelan el proceso de construcción de un programa y determinan, al final, su estructura.

Ejemplos:
- Procedimental
- Basado en reglas
- Programación lógica
- Programación funcional
- Programación heurística
- Otros: paralela, orientado a objetos, flujo de datos

- Controlan la manera de pensar de cómo solucionar el problema
- Definen modo en el que la solución se formaliza
- Determinan la forma correcta de los programas

Un **lenguaje de programación** refleja bien un paradigma -> soporta el paradigma. Suele identificarse con él.
Modelo de construcción/paradigma ≠ metodología de desarrollo de programas (pasos ciclo de vida del prog.)

Tipos:
- Los que soportan técnicas de **programación de bajo nivel**: por ejem. utilizar ficheros vs bases datos
- Los que soportan métodos de **diseño de algoritmos**: por ejem. Divide y vencerás, programación dinámica
- Los que soportan soluciones de **programación de alto nivel**: como los basados en procedimientos y datos, basados en objetos, en funciones, hechos y reglas…

Paradigmas que soportan soluciones de programación de alto nivel

Categorías:
- Solución **procedimental** u operacional: describe paso a paso el modo de construir la solución
- Solución **declarativa**: señala las características de las solución sin describir cómo procesarla. El lenguaje debe proporcionar cómo realizarlo.
- Solución **demostrativa**: variante del procedimental, aunque con resultados muy diferentes. Especifica la solución describiendo ejemplos y permitiendo que el sistema generalice la solución para otros casos.

Paradigmas procedimentales

Basados en secuencia computacional que se ejecuta etapa a etapa para resolver el sistema. Dificultad en saber si la solución es correcta. Técnicas de verificación para probar la corrección de la implementación.

Tipos:
- Los que actúan modificando repetidamente la representación de sus datos (**efecto de lado o colateral**)
  Variables estrechamente ligadas a las direcciones de memoria. El contenido se actualiza repetidamente. Al terminar la variables representan el resultado.
  Tipos: - Imperativo
  - Orientado a objetos
- Los que actúan creando continuamente nuevos datos (**sin efecto de lado**). Paradigmas funcionales.

La secuencia de ejecución puede ser en serie o paralelo (asíncrono: cooperación procesos o síncrono: procesos simples aplicados simultáneamente a muchos objetos)

Paradigmas declarativos

Construidos mediante hechos, reglas, restricciones, ecuaciones, transformaciones y otras propiedades derivadas del conjunto de valores que configuran la solución.

Variables para almacenar valores intermedios no para actualizar estados de información
En principio eliminan la necesidad de probar la solución.

Tipos:
- Funcional, lógico y el de transformación (pseudodeclarativos): no tienen secuencia de control ni efecto de lado, soluciones son producidas como construcciones más que como especificaciones.
- Otros no necesitan secuencias de control porque utilizan otros mecanismos de control.
  - Basados en formularios y en flujo de datos: usan restricciones en forma de ecuaciones permitidas y dependencia entre fórmulas.
  - Basados en restricciones: en el conjunto de ecuaciones

Paradigmas demostrativos

No se especifican procedimientos, se dan soluciones para que el programa generalice.

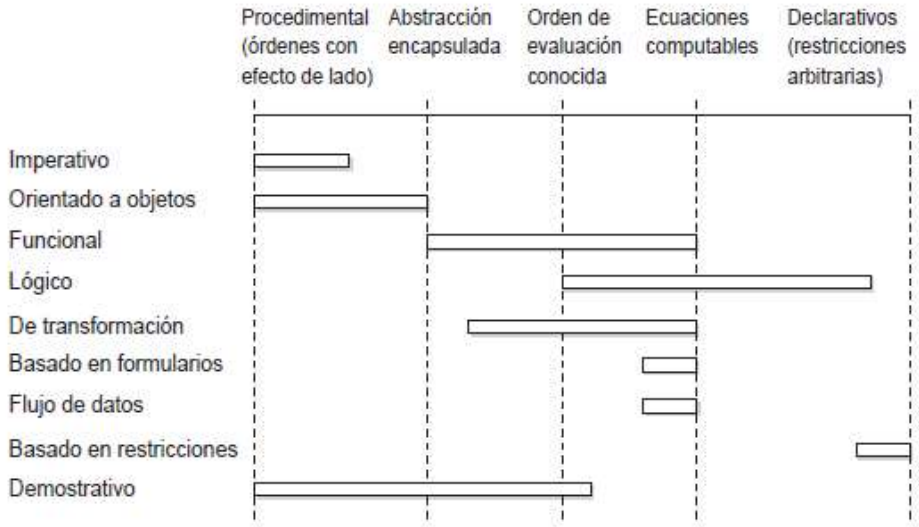
- Generalización usando inferencia: intenta determinar en qué son similares datos u objetos, y desde ahí generalizar similaridades. Se suele emplear serie de juegos de ensayo y otros para validar.
- Programación asistida: observación de acciones ejecutadas y si son similares a pasadas intenta inferir las futuras

Objeciones:
- Si no se comprueban exhaustivamente puede que no generalicen bien.
- La capacidad de inferencia es muy limitada y el usuario debe guiarlo, por lo general.

Satisfactorio en casos muy concretos, con gran conocimiento semántico. **Difícil conocer** cuando es correcto, el algoritmo queda en una representación interna.

Es una **solución bottom-up** (de abajo a arriba): pasa de lo particular a lo general, al contrario que la mayoría de los paradigmas.

Paradigmas según el grado de relevancia de la secuencia de control



Realizado por Juan del Río – juandrh2.0@gmail.com

Fuente: Lizcano Casas David, et al. 2018. *Fundamentos de la programación*. Madrid: Editorial CEF.

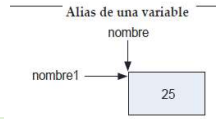
UNIDAD 4: LA PROGRAMACIÓN Y LOS LENGUAJES INFORMÁTICOS 2/3

Paradigma imperativo

Representa un modelo abstracto de computadora: memoria donde se almacenan datos que son transformados en unos resultados.

Arquitectura Von Neumann: Existe un programa en memoria que se va ejecutando secuencialmente.
Datos en memoria -> cálculos -> actualiza memoria
En su forma pura, solo soporta sentencias de modificación de memoria y bifurcaciones, condicionales o no. Los parámetros de un procedimiento son alias de zonas de memoria (se puede alterar valor). Los procedimientos pueden no retornar valor. Se puede alterar también la memoria con referencias a variables globales.
Esencia: cálculo iterativo de valores de nivel inferior y asignación a posiciones de memoria.

Características fundamentales:
- Concepto **celda de memoria** (variable) para almacenar valores: principal componente de la arquitectura. Tienen nombres, sobre ellas se producen efectos de lado y definiciones de alias.
- **Operaciones de asignación**: cada valor calculado debe almacenarse.
- **Repetición:** ejecuta lo complejo a base de repetición de una secuencia de pasos fundamentales



Es un **Paradigma Algorítmico** -> Algoritmo + estructuras de datos = Programa
Concepto de algoritmo -> Es privativo del tipo de programación procedimental

Tipos (según predomine uno u otro):
- **Orientados a expresiones**: más simples y jerárquicas, útiles para formar más complejas. Pierden propiedades matemáticas por tener efecto de lado → `x= ++y - b * c;`
- **Orientados a sentencias:** más influenciado por las características de la máquina

Sentencia condicional:

```
if x > y then v := true else v := false;
```

Expresión condicional:

```
v := x > y
```

Lenguajes de programación

Conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones. Controla comportamiento físico lógico de una computadora.

Componentes:
- Léxico: palabras
- Sintaxis: reglas de cómo realizar construcciones. Por ejemplo, en C/C++:
  sentencia = variable + signo igual + expresión + punto y coma
- Semántica: reglas que permiten determinar significado de cualquier construcción. Por ejemplo:
  nombre=nombre+2 → asigna a una variable el contenido incrementado en 2

Tipos según nivel de abstracción: lenguaje máquina, de bajo nivel y de alto nivel

Lenguajes código máquina

1ª generación

- Escritos con cadenas binarias directamente legibles por la máquina
- Estructura adaptada a los circuitos de la máquina, programación tediosa.
- Instrucciones se relacionan directamente con los registros y circuitos del procesador
- Datos se referencian por medio de direcciones de memoria + instrucciones realizan operaciones simples
- Ligados a la CPU no transferibles (baja portabilidad).
- Difícil programar, programas largos y laboriosos de corregir y depurar

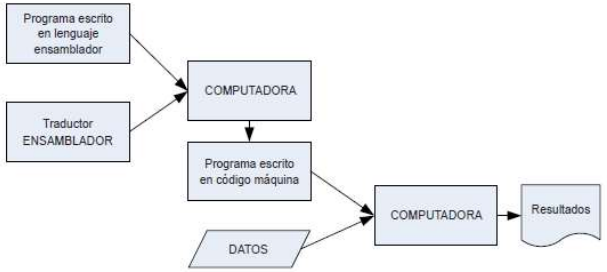
Lenguajes de bajo nivel

2ª generación

- Simbólicos: usan códigos de tipo mnemotécnico, facilita la escritura y depuración, no acorta los programas
- Se acercan al funcionamiento de la computadora
- Ensamblador (lenguaje por excelencia de bajo nivel y el primero utilizado):
  - Similar al lenguaje máquina (depende del procesador). No portable
  - Diferencias respecto a lenguaje máquina:
    - En lugar de direcciones binarias usa identificadores (suma, total, y…)
    - Códigos operación con mnemotécnicos (STO = guardar dato…, MOV =mover dato...)
    - Permite uso de comentarios entre las líneas.

Desventajas:
- Repertorio reducido de instrucciones
- Formato rígido de instrucciones
- Baja portabilidad y dependencia del hardware

Ventajas por el acceso directo a registros memoria:
- Uso óptimo de recursos hardware
- Código muy eficiente



Lenguajes de alto nivel

3ª generación. Próximos al lenguaje humano.

- Cada instrucción suele corresponder a varias acciones de la computadora.
- Características: **independencia** de la arquitectura física y **portabilidad** (ejecutable en diferente arquitecturas)
- Necesita traducción más compleja
- Reglas lexicográficas, sintácticas y semánticas rígidas, sin ambigüedades.

Traductor de lenguaje de alto nivel:

Compilación: proceso de traducir un programa a código objeto (normalmente código máquina)
Puede requerir una fase previa de preprocesado (directivas que no son del lenguaje)

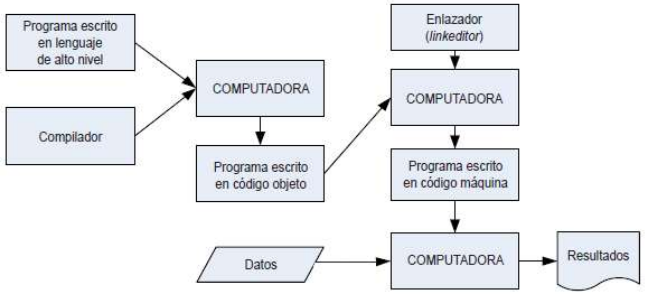
- **Análisis léxico** : verifica toda sentencia si pertenece al lenguaje léxico, carácter a carácter, indica el token de cad elemento reconocido o el error
- **Análisis sintáctico (parser):** los tokens son procesados en estructuras (árboles) y son analizados sintácticamente (análisis de la estructura)
- **Análisis semántico**: validez semántica de las sentencias aceptadas por el analizador sintáctico. Comprueba:
  - ¿todos los identificadores declarados?
  - ¿subexpresión a la izquierda del = es semánticamente válida?
  - ¿ al tipo de identificador valor se le puede aplicar operador multiplicación?
- **Generador de código objeto**
- **Optimización del código objeto:** velocidad ejecución, tamaño programa y necesidades memoria. Otras optimizaciones:
  - Precálculo expresiones constantes
  - Reutilización expresiones comunes
  - Cambio del orden de algunas expresiones…..

Realizado por Juan del Río – juandrh2.0@gmail.com

Fuente: Lizcano Casas David, et al. 2018. *Fundamentos de la programación*. Madrid: Editorial CEF.

La creación de un programa ejecutable conlleva dos pasos:
- Compilación
- Enlazado (linker): enlaza todo código objeto + código objeto de las librerías del sistema

Se pueden hacer por separado, de forma que se puede tener código de varios lenguajes distintos y enlazarlos juntos para tener un solo ejecutable.



Interpretación: El intérprete convierte cada instrucción del código fuente en lenguaje máquina conforme va siendo necesario durante el procesamiento de los datos.
- El código máquina no se graba.
- Es más lento.
- Es más flexible
- No garantiza la privacidad del código fuente

Lenguaje	Principal área de aplicación	Compilado/interpretado
Ada	Lenguaje de sistemas y tiempo real	Lenguaje compilado
ASP	Desarrollo de sitios web dinámicos	Lenguaje interpretado
ASP.NET	Lenguaje ASP con tecnología .NET	Lenguaje interpretado
BASIC	Programación para fines educativos	Lenguaje interpretado
BeanShell	Lenguaje de *scripts*	Lenguaje interpretado
C	Programación de sistema	Lenguaje compilado
C++	Programación de sistema orientado a objeto	Lenguaje compilado
COBOL	Administración	Lenguaje compilado
Fortran	Cálculo	Lenguaje compilado
Java	Programación orientada a objetos e internet	Lenguaje interpretado
JavaScript	Programación orientada a internet	Lenguaje interpretado
JSP	Programación de sitios web dinámicos	Lenguaje interpretado
HTML	Programación orientada a internet	Lenguaje interpretado
MATLAB	Cálculos matemáticos	Lenguaje interpretado
Lisp	Inteligencia artificial	Lenguaje intermediario
Pascal	Educación	Lenguaje compilado
PHP	Desarrollo de sitios web dinámicos	Lenguaje interpretado
Perl	Procesamiento de cadenas de caracteres	Lenguaje interpretado
Python	Desarrollo de sitios web dinámicos	Lenguaje interpretado
Ruby	Desarrollo de sitios web dinámicos	Lenguaje interpretado

Historia de los lenguajes de programación

La teoría formal es anterior al desarrollo práctico.

- Cálculo lambda (años 30): uno de los primeros, intención de modelar la computación
- Años 40: primeros ensambladores y primer lenguaje alto nivel (conocido en 1972)
- 1954-1957: Fortran
- 1958 ALGOL 58-60
- 1960 COBOL: tratamiento de datos a gran escala y LISP destinado a Inteligencia Artificial
- Forth, PAL y PL/I
- 1965. Basic: para enseñanza
- Años 60: Simula: germen programación orientada a objetos y corrutina
- 1970 Pascal: enseñanza y C -> desarrollo S.O. UNIX
- Prolog: lógica de predicados, destinado a Inteligencia Artificial
- SQL : manejo bases de datos, dBase y otros..
- Años 80: C++
- Java: totalmente orientado a objetos

Desde inicios de internet surgieron lenguajes estáticos. Algunos son:

- HTML: lenguaje representación de documentos: Desarrollado por W3C
- JavaScript: lenguaje interpretado.
  Se crea estándar DOM (Document Object Model) para html y xml
- PHP: creación sitios web. 1995. Interpretado en lado servidor
- JSP: Java Server Pages. Lado servidor.
- Python. 1990 lenguaje más limpio de programar. Multiparadigma.
- Ruby. Alto nivel. Interpretado.

Fuente: Lizcano Casas David, et al. 2018. *Fundamentos de la programación*. Madrid: Editorial CEF.

FP

UNIDAD 5: EL LENGUAJE IMPERATIVO C++

Introducción

Bjarne Stroustrup -Laboratorios Bell (AT&T)

- Independiente del sist. Operativo y arquitectura
- Orientado a objetos multipropósito basado en C, que añade:
  - Mejora de C
  - Soporte para abstracción de datos
  - Soporte programación orientada a objetos

Historia de C++

- 1967 - lenguaje BCPL - Martin Richards
- 1970 - lenguaje B - Ken Thompson (UNIX)
- 1973 - lenguaje C - Dennis Ritchie (reescribe UNIX)
- Se estandariza por ANSII
- 1981 – C con clases
- 1984 – C++
- 1985 –traductor C++ a C. *The C++ Programming Language.* Libre acceso a AT&T C++ Translator 1.0
- 1986 – 1989 – se añaden mejoras
- 1991 – definición más rigurosa.*The Annotated C++ Reference Manual*
- 1997 – se publica la 3ªed. *The C++ Programming Language*
- 1998 - se aprueba el estándar internacional ISO/ANSI
- 2011 y 2014 – nuevas revisiones

Características generales de C++

- **Núcleo de pequeño** tamaño: 75 palabras
- **Rápida** ejecución de los programas: relativo bajo nivel + optimización de los compiladores
- Muy **flexible**: código compacto y eficente
- **Transportable**: salvo desarrollado para una arquitectura específica o usado librerías propias
- **Estructurado**: sentencias control de flujo + permite definir datos estructurados
- Posibilidad de programación **modular**
- Posibilidad de usar **punteros y memoria dinámica**
- **Case-sensitive:** diferencia mayúsculas y minúsculas
- Formato libre: respetar homogeneidad y dotar de claridad

Para programadores de C:
- Permite reutilización de código
- Crear nuevos tipos rápidamente
- Gestión de memoria más transparente y fácil
- Menos errores: sintaxis más estricta y comprobación de errores
- Mecanismo ocultación de información para ser más robusto

Elementos básicos de C++

Sentencia: cualquier orden o instrucción válida. Todas van seguidas de **;**
- Se ejecutan secuencialmente
- Hay sentencias especiales como las de Flujo de Control

Declaraciones: de variables, tipo de datos y funciones

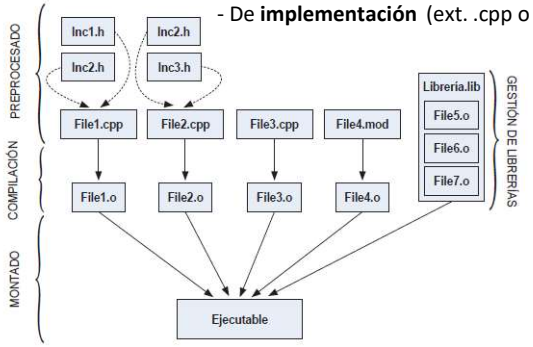
Bloque: conjunto de declaraciones o sentencias entre llaves {} (0, 1 o más…)

Funciones: fragmento de programa que, dados unos parámetros de entrada, realiza una determinada acción y devuelve el correspondiente resultado. Obligatorio que exista **main()**

Estructura de un programa C++

Partes lógicas (no es necesario este orden ni agrupación):
- **Directivas de preprocesador:** previo a compilación
- **Tipos de datos**
- Datos globales: visibles a todas las funciones
- **Declaración de funciones o prototipos**
- **Programa principal (main)**: se ejecuta al inicio
- **Definición de funciones**

- El código se puede agrupar en diferentes **ficheros fuente** -> se **copilan** los ficheros -> se **enlazan** (link) junto con las librerías que proporciona el compilador → **programa final**
- Suele haber dos ficheros por módulo:
  - De **cabecera** (ext. .h): declaraciones de los tipos, variables y funciones
  - De **implementación** (ext. .cpp o .cc) definición de funciones



Palabras reservadas de C++

Cuadro 1. Conjunto de palabras reservadas

asm	auto	bool	break	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export
extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	operator	private
protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast
struct	switch	template	this	throw
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
volatile	wchar_t	while		

Cuadro 2. Palabras reservadas como representaciones alternativas de algunos operadores

and	and_eq	bitand	bitor	compl	not
not_eq	or	or_eq	xor	xor_eq	

Comentarios en C++

Necesarios para aclarar el código y aumentar legibilidad. Ignorados por el compilador.

Formas:

- De bloque
- De línea

```
/* esto es un comentario de bloque
   que puede ocupar varias líneas */
// esto es un comentario de línea
int i;    // otro comentario de línea
```

No pueden anidarse.
Resulta útil aportar información de cada función para el programador y para otros.
Se pueden evitar si se usan nombres significativos (*contador* en lugar de *c*)

Realizado por Juan del Río – juandrh2.0@gmail.com

Fuente: Lizcano Casas David, et al. 2018. *Fundamentos de la programación*. Madrid: Editorial CEF.

FP

UNIDAD 6: EXPRESIONES EN C++ (1 de 3)

Expresión Operandos (constantes, variables, etc.) + operadores (aritméticos, lógicos, relacionales, etc…)

Constantes Representan valores que no cambian a lo largo de la ejecución del programa

Tipos:
- Enteras: si tamaño es 16 bits -> (-32,768, 32767) 2^{16} . Notación octal (0-7 dígitos): prefijo -> 0 (0274,01,-012,..)
  Notación hexadecimal (0-9 dígitos + A-F): prefijo -> 0x (0x5D4, -0x1,+0x7D9,..)
  Modificadores de precisión U (unsigned), L (long) . Pueden colocarse en cualquier orden.
- Reales: Notación normal : 5235.256, -9.455,… Notación científica: potencias de 10 (1,526e23, .56E-25…)
  Tipo Double por defecto. Modificador f (float) y L (long double)
- Lógicas: true (cualquier valor distinto 0), false (valor 0)
- Carácter (comillas simples ‘ ’). Representa carácter de un conjunto de caracteres (por ejemplo: ASCII).
  Se almacenan como entero. Carácter nulo =NULL

Usar \ para caracteres especiales

'\n': cambio de línea.	'\0': carácter nulo o NULL.
'\r': retorno de carro.	'\"': comillas dobles.
'\t': tabulador horizontal.	'\'': comilla simple.
'\v': tabulador vertical.	'\\': barra inversa.
'\f': salto de página.	'\%': tanto por ciento.
'\b': retroceso de un espacio.	'\xdd': carácter cuyo código sea el número indicado en hexadecimal.
'\a': alarma o campana.	'\ddd': carácter cuyo código sea el número indicado en octal.

- Cadenas de caracteres: "\t¡Hola, mundo!\n" . Siempre terminan con carácter nulo (‘\0’) de forma automática

Variables Tipo de identificador. Reciben un nombre que define el programador. Necesario declararlas previamente especificando el tipo de datos.

Nombre: empieza por letra o _ , seguido por letras, números o _. Cualquier longitud pero el número de caracteres significativos depende del compilador. Diferencia mayúsculas de minúsculas.

Declaración -> tipo *nombre_variable*;
Inicialización -> tipo *nombre_variable = valor*; tipo *nombre_variable (valor)*;

Identificador constante -> **const** tipo *nombre_variable = valor*; (no se podrá alterar el valor)

Tipos de datos Clases:
- Básicos: un valor sencillo
- Avanzados, elaborados o complejos: varios datos o estructuras de datos
- Definibles por el programador

Tipos básicos

Entero: int Modificadores: short, long, unsigned, signed -> tamaño short ≤ tamaño int ≤ tamaño long
Volatile : indica al compilador que no realice optimización. Suele ser usado cuando hay módulos en otro lenguaje que puede en hacer uso de esa variable.

Real: float Modificador double = long float, long double (1.5e2222L)

Carácter: char char digito= '1', car= 0;

Lógico: bool

Vacio / nulo void

Punteros: * Dato que apunta a un dato. Dirección de memoria. tipo * var;
`* = operador de indirección`

Referencias: & Alias de una variable. Deben inicializarse y no pueden cambiarse. Usados en funciones.

Declaraciones en bloques:
- pueden aparecer en cualquier parte del bloque; incluso, tras otras sentencias. Recomendable al principio
- alcance hasta el final del bloque

Conversiones de tipos:
Permite la mezcla de diferentes tipos en operaciones. Se convierte el dato de menor precisión en el de mayor.
Si dos operandos del mismo tipo, el resultado también lo será (8/5 = 1).
Se recomienda usar las conversiones manuales de tipos.

Reglas para tipos compatibles:
1. `char`, `unsigned char` y `short` se convierten a `int` siempre para operar.
2. `unsigned short` se convierte a `unsigned int` siempre.
3. `float` se convierte a `double` siempre.
4. si un operando es `long double`, el otro se convierte a `long double`.
5. si un operando es `double`, el otro se convierte a `double`.
6. si un operando es `float`, el otro se convierte a `float`.
7. si un operando es `unsigned long`, el otro se convierte a `unsigned long`.
8. si un operando es `long`, el otro se convierte a `long`.
9. si un operando es `unsigned`, el otro se convierte a `unsigned`.

Operadores

Operador: Elemento del lenguaje que permite indicar qué operación se va a realizar con unos determinados datos.
Operando: Datos sobre los que actúa el operador.
Expresión: forma de representar y manejar datos de manera simple y eficaz. Puede ser una constante, una variable o varias interconectadas por uno o más operadores.

Tipos:
- Unarios o monarios: sólo un operando. Prefijo o sufijo -742, x ++ , siceof (x)…
- Binarios: dos operandos. Infijo. a/b, 3< 4 , 4+6 …
- Ternarios: tiene 3 operandos: a ? f : g

Precedencia y asociatividad:
Asociatividad de izquierda a derecha -> a + b + c → a +b -> +c

::	Modificador de visibilidad o alcance
->	Referencia al miembro de una estructura mediante un puntero
.	Referencia al miembro de una estructura
()	Llamada a una función
[]	Referencia a un vector
	Asociatividad de derecha a izquierda
*	Referencia a un puntero
&	Dirección
-	Menos unario (número negativo)
+	Más unario (número positivo)
!	No, negación lógica (*NOT*)
~	Complemento a 1
++	Autoincremento
--	Autodecremento
(*tipo*)	Conversión forzada de tipos (*casting*)
sizeof *exp*	Tamaño en memoria de la expresión *exp*
sizeof (*tipo*)	Tamaño en memoria del tipo indicado
new	Solicitud de memoria dinámica
delete	Liberación de memoria dinámica

Realizado por Juan del Río – juandrh2.0@gmail.com

Fuente: Lizcano Casas David, et al. 2018. *Fundamentos de la programación*. Madrid: Editorial CEF.

UNIDAD 6: EXPRESIONES EN C++ (2 de 3)

.* ->*	Referencia a un puntero a un miembro de una estructura Referencia a un puntero a un miembro de una estructura a través de un puntero a dicha estructura
* / %	Producto División Módulo o resto de la división
+ -	Suma Resta
<< >>	Desplazamiento de bits a la izquierda Desplazamiento de bits a la derecha
< > <= >=	Menor que Mayor que Menor o igual que Mayor o igual que
== !=	Comparador de igualdad (es igual) Comparador de desigualdad (es distinto)
&	Y (*AND*) de bits
^	O exclusivo (*XOR*) de bits
|	O (*OR*) de bits
&&	Y (*AND*) lógico
||	O (*OR*) lógico
`cond? exp1: exp2`	*Asociatividad de derecha a izquierda* Condicional. Si se cumple la condición *cond*, devuelve *exp1*; en caso contrario, *exp2*
	Asociatividad de derecha a izquierda
= += -= *= /= %= >>=	Asignación Suma los dos operandos y asigna el resultado al primero Resta el primer operando menos el segundo y asigna el resultado al primero Multiplica los operandos y asigna el resultado al primero Divide el primer operando entre el segundo y asigna el resultado al primero Extrae el módulo de dividir el primer operando entre el segundo y asigna el resultado al primero Desplaza hacia la derecha el primer operando tantos bits como indique el segundo operando, asignando el resultado al primer operando
<<= &= ^= |=	Desplaza el primer operando a la izquierda tantos bits como indique el segundo operando, asignando el resultado al primer operando Realiza un *AND* bit a bit entre los dos operandos y asigna el resultado al primero Realiza un *XOR* bit a bit entre ambos operandos y asigna el resultado al primero Realiza un *OR* bit a bit entre los dos operandos y asigna el resultado al primero
,	Operador coma. Ejecuta el operando de la izquierda, ejecuta el de la derecha y devuelve el resultado de esta última ejecución

Se puede alterar el orden con paréntesis.

Operadores aritméticos: + , - , * , / , % — Operandos: números o caracteres
- a%b, a y b enteros y b ǂ 0
- a/b , b ǂ 0

Relacionales <, >, <=, >=, ==, != → Resultado = 1 o 0 (verdadero o falso)

Lógicos: &&, ||, ! → Realiza **evaluación en cortocircuito o perezosa**: si al evaluar una parte puede asegurar el valor completo, no continuará evaluando el resto

Operadores y expresiones de asignación = , +=, -=, *=, /= ,%=

a = b -> si a y b compatibles -> tipo de b se convierte al tipo de a. Puede ocasionar pérdida de información.
-> si a y b no compatibles -> error

Se permiten asignaciones múltiples: a=b=c= expresión → a= (b= (c= expresión))

Operadores y expresiones sobre bits ~ , &, ^, | , <<, >>

Complemento a uno ~ — 1 -> 0 y 0 -> 1

Operadores lógicos de bits: &, ^ (o exclusivo) , | — Comparación de derecha a izquierda

Operadores de desplazamiento << , >> — Dos operandos: entero a desplazar y nº de desplazamientos. Por un lado se pierden bits y por el otro se rellenan con ceros

En C++ nuevo uso = manejo de canales (streams) en entrada/salida

Operadores de asignación &=, |=, ^=, <<=, >>= — Asociatividad de **derecha** a izquierda

Operadores y expresiones manejo de punteros

Operador de dirección & — Proporciona dirección de memoria de un valor. Puntero = &valor

Operador de indirección * — Proporciona el valor de la posición de memoria almacenada en un puntero. Valor = *Puntero

Fuente: Lizcano Casas David, et al. 2018. *Fundamentos de la programación*. Madrid: Editorial CEF.

UNIDAD 6: EXPRESIONES EN C++ (3 de 3)

Operadores paréntesis ()	Llamadas a funciones. Operador n-ario
Operadores condicional ? :	expresión1 ? expresión2 : expresión3
Operadores signo +,-	1 operando
Operadores incremento y decremento ++, --	Incrementan / disminuyen en una unidad ++ x -> incrementa antes de usar x X++ -> incrementa después de usar x
Operadores tamaño **sizeof**	Devuelve tamaño en bits
Operadores conversión de tipos	(tipo) expresión ó tipo (expresión)
Operador coma ,	Encadenar varias expresiones, devolviendo el resultado de la última
Operador de acceso a miembros **. ->**	registro.miembro punterofecha->dia
Operador acceso a vectores **[ ]**	Vector[3]
Operador de alcance **::**	Permite modificar la visibilidad de un elemento
Operadores **new** y **delete**	Asignación y liberación dinámica de memoria. New reserva memoria a un tipo de dato. Si no hay suficiente devuelve NULL Aplicar delete sobre un puntero nulo es inofensivo (llama a su destructor)

Fuente: Lizcano Casas David, et al. 2018. *Fundamentos de la programación*. Madrid: Editorial CEF.

FP

UNIDAD 7: SENTENCIAS Y ENTRADA/SALIDA BÁSICA

Sentencias básicas

Asignación Guardar valor en variable. variable= expresión; Sentencias sin utilidad: a +b; C && d;

Llamadas a funciones Puede devolver o no valor. Leer_teclado ();

C++ tiene librería matemática extensa.
- Include <math.h> o <stdlib>

- Valor absoluto: **`int abs (int)`**.
- Seno: **`double sin (double)`**.
- Coseno: **`double cos (double)`**.
- Exponencial (*e* elevado al operando): **`double exp (double)`**.
- Logaritmo neperiano: **`double log (double)`**.
- Potencia (el primer operando elevado al segundo): **`double pow (double, double)`**.
- Raíz cuadrada: **`double sqrt (double)`**.

Sentencia "Return" Se emplea dentro de una función. Fuerza la finalización.
return [expresión]; función tipo void → return no tiene expresión, no devuelve nada

Sentencia vacía ; No hace nada, dota de claridad al código

Sentencias de control de flujo

Orden en el que se ejecutan las sentencias.

Estructura secuencial Se ejecutan en el mismo orden

Estructura condicional

IF

```
if (condición)
   sentencia;
```

```
if (condición)
{
   sentencias;
}
```

```
if (condición)
   sentencia1;
else
   sentencia2;
```

```
if (condición)
{
   sentencias1;
}
else
{
   sentencias2;
}
```

SWITCH

Usar **Break**

```
switch (expresión)
{
   case et1: sentencias1;
   ...
   case etn: sentenciasn;
   [default: sentencias;]
}
```

Sentencias repetitivas

WHILE Cuidado bucles infinitos

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

FOR

```
for (inicialización; condición; actualización)
{
  sentencias;
}
```

```
for (n= 1, total= 0; n < MAX; total += n, n++)
   ;   // sentencia vacía
```

DO-WHILE Se ejecuta al menos una vez

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

Sentencias de salto

Break

Sólo dentro de FOR, WHILE, DO-WHILE, SWITCH
Fuerza salida del bucle. No abusar de su uso.
En FOR existen otras opciones para no usarlo y obtener código mejor estructurado
En SWICTH es habitual usarlo para que no ejecute todas las sentencias.

Continue

Sólo dentro de FOR, WHILE, DO-WHILE
Menos utilizada de Break, fuerza finalización de la iteración en curso. No abusar de su uso.

Entrada / salida básica

Permiten la comunicación entre el programa y el exterior

- No hay palabras reservadas, no forman parte del lenguaje. Se han creado librerías.
- En C++ se define un canal (stream): cin, cout, cerr

Salida

- Se indica con << por ejemplo: cout << "Resultado : z= " << x * y << '\n';
- Clase ostream (incluir iostream.h)
- Manipuladores (incluir iomanip.h)

```
cout << hex << v << endl;
   // Muestra v en hexadecimal y salta de línea
```

Manipulador	Acción producida
dec	base decimal
hex	base hexadecimal
oct	base octal
endl	inserta '\n' y vacía el canal

- cerr – para info sobre errores

Entrada

- Se indica con >> por ejemplo: cin >> x >> y;
- Clase istream

Fuente: Lizcano Casas David, et al. 2018. *Fundamentos de la programación*. Madrid: Editorial CEF.

FP

UNIDAD 8: FUNCIONES 1 de 2

Introducción

Permite dividir el programa en partes más pequeñas, más sencillas de implementar.
Puede recibir datos de entrada, realizar cálculos y devolver un resultado.

Funciones en C++

Declaración

Prototipo de función: indica al compilador el nombre, el tipo de parámetros y el valor devuelto

```
[tipo] nombre_función ([parámetros]);
```

[tipo] : cualquier tipo básico, complejo (struct, unión,…) o void
nombre_función : seguir normas de los identificadores
[parámetros] : son optativos si no tiene poder void, se puede añadir el identificador de cada uno.

Lo mejor es incluirlos en ficheros de cabecera (.h). Esto permite su reutilización de forma más sencilla.
Puede declararse si se define justo antes de su uso. No es lo recomendable.

Definición

```
[tipo] nombre_función ([tipos parámetros]) bloque
```

<- Bloque: contiene el código

```
int max (int a, int b)
{
   if (a > b)
      return a;
   return b;
}
```

→

```
a= max (4, 5);
b= max (5.8, 3);
c= max (a, b);
d= max (max (a * 2, b/2), c * c);
```

Función **main**: necesaria en todo programa. Se ejecuta en primer lugar.

```
void main (int argc, char *argv[])
{
     /* código de la función principal */
}
```

argc: entero, número argumentos
*argv: parámetros proporcionados por el Sistema Operativo

Funciones librería

Funciones predefinidas, para su acceso se utilizan ficheros cabecera (.h) con **#include stdlib** → función **exit**

Funciones en línea

inline :solicita que se expanda el código donde sea llamada.

```
inline [tipo] nombre_función ([parámetros]);
```

- Ahorran tiempo ejecución
- Aumentan tamaño del código
- Total semántica de función (no tienen efectos laterales de las macros del preprocesador)

El compilador puede negarse cuando:

- La función es demasiado grande.
- La función es recursiva.
- La función es llamada antes de su definición.
- Se invoca a la función más de una vez en una misma expresión.
- La función contiene un bucle o un switch.

Una función ideal es aquella que realiza una función sencilla.

Parámetros

Tipos:
- Parámetros formales: aparecen en la definición de una función
- Parámetros actuales: aparecen en la llamada a la función. Deben coincidir en número y tipo con los formales. Salvo que haya parámetros opcionales. El tipo puede ser compatible.

Paso de parámetros:
- **Por valor**: el valor se copia al parámetro formal. Si se modifica en la función, el parámetro actual no se verá afectado, conserva su valor original.

- **Por referencia**: el valor actual se ve afectado. Se maneja la dirección del parámetro actual. Los dos valores se almacenan en la misma dirección de memoria. Solo se pueden enviar identificadores o expresiones de los que se pueda obtener una dirección (punteros, vectores..)

```
void swap (int &x, int &y)
{
   int t;
   t= x;
   x= y;
   y= t;
}
```

```
int a= 3, b= 7;
swap (a, b);
cout << a << b;   // imprime 7 y 3
```

- **Por dirección**: se pasa un puntero del parámetro actual. Sirve para parámetros con tamaños muy grandes.

```
void swap (int *x, int *y)
{
  int t;
  t= *x;
  *x= *y;
  *y= t;
}
```

```
int a= 3, b= 7;
swap (&a, &b);
cout << a << b;    // imprime 7 y 3
```

Si se quiere evitar la modificación del parámetro dentro la función, usar **const.**

```
void funcion (const double *par);
```

- **Por omisión:** fijar valores predefinidos en caso de omisión. Tienen que estar en los últimos parámetros

```
int aleat (int i= 0, int j= 100);
```

Alcance y visibilidad

Zona de código fuente donde puede utilizarse una variable dada.
En C++ desde que se declara hasta el final del bloque.

Variable local: toda variable declarada dentro de un bloque (sólo puede ser utilizada en ese bloque).
Si hay dos variables con el mismo nombre, la del bloque interior oculta a la otra, salvo si se usa **::**

Variable global: toda variable declarada fuera de un bloque (puede ser utilizada dentro de cualquier bloque).
Se recomienda no usarlas por estilo, legibilidad y facilidad de mantenimiento.
Tipos:
- **de fichero**: alcance -> fichero -> definirla como **estática**
- **Del programa:** alcance -> varios ficheros -> usar **extern**

Variables estáticas

Globales estáticas: extern -> varios ficheros , static -> sólo un fichero

Locales estáticas: static: sigue siendo local pero mantiene valor a sucesivas llamadas

Fuente: Lizcano Casas David, et al. 2018. *Fundamentos de la programación*. Madrid: Editorial CEF.

Variables estáticas (continuación)

Modo de almacenamiento	Palabra clave	Duración	Alcance	Lugar de definición
Automático	`[auto]`	Temporal	Local	Dentro de las funciones
Registro	`register`	Temporal	Local	
Estático	`static`	Permanente	Local	
Externo	`[extern]`	Permanente	Global	Fuera de las funciones
Global estático	`static`	Permanente	Fichero	

- **Automático**: modo por defecto. El compilador ubica la variable donde vea oportuno (memoria, pila, registro..

- **Registro**: almacenamiento en registro del procesador. Uso intensivo, acceso muy rápido.

Recursividad

Una función se llama a sí misma.
Suele ser menos eficiente que funciones secuenciales o iterativas, pero son más elegantes.

Componentes:
- El paso recursivo: llamada a si misma
- El caso base: permite detener la secuencia de llamadas, finalizar la función y devolver un valor.

Es necesario asegurarse que alcanzará el caso base o condición de finalización.

Sobrecarga de funciones

Tipo de polimorfismo (un concepto de la orientación a objetos) que consiste en la capacidad de que un elemento adopte diferentes formas

Permite definir distintas funciones con el mismo nombre, distinta implementación y parámetros.

Cada llamada a la función se basará en el número y tipo de argumentos.

Deberían sobrecargarse únicamente las que hagan operaciones semejantes.

```
   // Prototipos:
void print (int);

void print (float);
void print (int, int);
   // Definiciones:
void print (int e)    // imprime un entero
{
   cout << "El entero vale: " << e << endl;
}
void print (float f)    // imprime un real
{
   cout << "El real vale: " << f << endl;
}
void print (int ent, int dec)
{   // imprime dos enteros y los separa con un punto
   cout << ent << "." << dec;
}
   // Llamadas de ejemplo:
print (12);    // llama a print (int)
print (20, 5);    // llama a print (int, int)
print (3.14);    // llama a print (float)
```


Fuente: Lizcano Casas David, et al. 2018. *Fundamentos de la programación*. Madrid: Editorial CEF.

FP

UNIDAD 9: TIPOS AVANZADOS DE DATOS

Vectores y matrices

Vectores

Secuencia de valores del mismo tipo, acceso indicando su posición (o índice)

Declaración -> `tipo nombre [dim];`
- Tipo: puede ser tipo básico o avanzado
- Dim: constante o expresión constante, conocida a priori,

Inicialización -> `int x[5]= {1, 2, 4, 8, 16};`
- Si faltan elementos quedan sin inicializar
- Si hay demás genera error en compilador

Si se declara e inicia al mismo tiempo no es necesario indicar la dimensión

```
char respuestas[]= {'b', 'a', 'c', 'c', 'a', 'b', 'c', 'c', 'a', 'b'};
```

Acceso elementos `v[0] ='a'`

El compilador no comprueba los límites
Responsabilidad del programador no salirse

Matrices

Similar a un vector, pero con más dimensiones, necesarios varios índices, vector de vectores.
Si tiene 2 dimensiones → fila y columna `tipo matriz[dim1]...[dimn];`

```
int mat1[3][4]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
int mat2[3][4]= {{1, 2, 3, 4}, {5, 6}, {9, 10, 11, 12}};
int mat3[3][4]= {{1, 2, 3, 4}, {5, 6, 7, 8}};
```

1	2	3	4
5	6	7	8
9	10	11	

1	2	3	4
5	6		
9	10	11	12

1	2	3	4
5	6	7	8

Se rellena por filas, si es mayor de 2 dimensiones se recomienda usar {} para separar.
Se puede omitir el tamaño de la primera, pero no de los siguientes:

```
char respuestas[][3]= {{'b', 'a', 'c'}, {'c', 'a', 'b'}};
```

Almacenamiento en memoria:
- posiciones consecutivas, empezando con la posición 0.
- Espacio ocupado = Multiplicar tamaño dimensiones y tamaño de cada elemento

Punteros

Variable es una zona de memoria a la que se accede utilizando su nombre.
Posición definida por el compilador.
Una tabla de símbolos almacena nombre y dirección de memoria en la compilación

Puntero = Variable cuyo valor es la dirección de una zona de memoria.
Si representa la dirección de una variable → se dice que el puntero apunta a la variable

Declaración ->

```
float *pf;    // puntero a reales
long int *pl;    // puntero a enteros grandes
short **pps;    // puntero a punteros a enteros pequeños
char (*pc)[10];    // puntero a vectores de 10 caracteres
```

El tamaño de un puntero depende de la máquina y de cómo se organice el espacio de direcciones de memoria

Operador dirección & — Obtiene la dirección de memoria de una variable.
`Puntero= &Numero;`

Operador puntero * — Accede al valor almacenado en una dirección de memoria
`x= *ptr;`

```
float i, x, *ptr;
ptr= &i;
x= *ptr;
```

```
float i, x;
x= i;
```

Memoria dinámica — Hasta que no se inicializa un puntero hace referencia a una posición indeterminada.
Otra forma, reservar una nueva posición de memoria dinámica libre. `new tipo`
Si no encuentra suficiente -> excepción -> std::bac_alloc_ (Null en compiladores antiguos)

Siempre hay que liberar la memoria después: `delete puntero;`
En la tabla de bloques de memoria -> referencia vacía (disponible)
Buena práctica: puntero = NULL ; -> facilita detección de errores y si un puntero tiene dirección válida o no.
Si se intenta liberar un puntero nulo no hará nada.

Puntero a varios elementos de un tipo: `int *q= new int [80];` (puntero a 80 enteros)

Aritmética de punteros

Se puede asignar, comparar y restar, pero no sumar punteros.

Si un puntero apunta a una zona capaz de almacenar varios datos consecutivos se pueden sumar enteros como posiciones de memoria (del tamaño del tipo de dato).

```
*(ptri + 1)= 7;    /* se guarda un 7 en el 2º entero al que apunta */
```

Es habitual usar un segundo puntero para recorrer la memoria. `for (y= 0; y < 200; y++, ptrj++)`

Punteros y vectores

Nombre de un vector es la dirección del primer elemento el contenido es ese elemento:

```
int i, *ptra, a[100];
ptra= a;    /* el puntero apunta al comienzo del vector */
```

Equivalentes: `*(ptra + i)` `a[i]` `ptra[i]` `*(a + i)`

Puntero=variable, nombre de vector= constante
- Se puede hacer: `ptra = a;` `ptra++;`
- No se puede hacer: `a = ptra;` `a++;`

char s[5]; → reserva en memoria 5 espacio para 5 caracteres
char *s; -> puntero a caracteres

```
int array[10];
int *ptr= array;
```

```
int array[10];
int *ptr;
ptr= array;
```

```
int array[10];
int *ptr;
ptr= &array[0];
```

Cadenas

Secuencia de caracteres que termina en '\0' . Es un vector o puntero a caracteres.
`char s[20]; char *s; string s;` string.h

```
char cadena[5]= "Hola";
char *cadena= "Hola";
string cadena ("Hola");
```

```
char cadena [5]= {'H', 'o', 'l', 'a', '\0',};
char cadena [ ]= {'H', 'o', 'l', 'a', '\0',};
```

Añadir

La asignación directa sólo es posible si se ha declarado como puntero (no como vector)
Copia de cadenas: - copiar uno a uno los elementos
- Usando **strcpy**
- Usando **string** en lugar de char

Estructuras (registro)

Agrupación de datos relacionados entre sí, pero que pueden ser de tipos distintos.
Es posible declarar sin nombre, pero en general es poco útil.

```
struct [nombre]
{
   tipo1 campo_1;
   tipo2 campo_2;
   ...
   tipon campo_n;
} [variable];
```

```
Fecha f= {11, 2, 1966};
```

`variable.campo`

```
cout << (*persona).nombre;
cout << persona->nombre;
```

Se puede definir el tamaño de los campos: `tipo campo: nº de bits;`
El tipo de los campos puede ser uno de los siguientes: **bool**, **char**, **unsigned char**, **short**, **unsigned short**, **long**, **unsigned long**, **int** o **unsigned int**.

Una limitación que tienen los campos de bits es que no se puede obtener su dirección. Por ejemplo, el siguiente código sería erróneo: &var.numero.

Uniones

```
union [nombre]
{
   tipo1 campo_1;
   tipo2 campo_2;
   ...
   tipon campo_n;
} [variable];
```

Similar a una estructura. Permiten representar varios campos distintos, pero compartiendo la misma zona de memoria, de forma que en cada instante dado solo uno de los campos tiene información válida.

La memoria que ocupa es la del campo más grande.
Se inicia dando valor del primer campo.

Definición de tipos

`typedef tipo nuevo_tipo;`

Modificadores:
- []: vector
- (): función
- * : puntero

Reglas de prioridad:
- mayor prioridad cuanto + próximos estén al nombre de la variable.
- * menor prioridad y los [] la mayor
- Usar paréntesis vector de

```
int (*f[5])();
```

cinco punteros a funciones enteras

Tipos enumerados

Cuando se necesita usar unos valores entre los que existe alguna relación de orden o, simplemente, por aclarar la escritura del código

```
enum Dias {lunes, martes, miercoles, jueves, viernes, sabado, domingo};
```

Solo admite valores dentro del rango definido. `hoy= jueves;`
Son equivalentes a números enteros (0,1,2..). Lunes=0, martes=1 ... Puede ser arbitraria o se puede asignar un valor inicial

```
enum Pares {cero, ocho= 8, seis= 6, cuatro= 4, dos= 2};
enum Meses {enero= 1, febrero, marzo, abril, mayo, junio, julio,
agosto, septiembre, octubre, noviembre, diciembre};
```


Fuente: Lizcano Casas David, et al. 2018. *Fundamentos de la programación*. Madrid: Editorial CEF.

FP

UNIDAD 10: EL PREPROCESADOR DE C++. LIBRERÍAS 1 de2

Preprocesador de C++

Toda directiva debe comenzar por # en la primera columna del código, cada una en una línea.

Definir nombres simbólicos

permite asignar un nombre a un determinado elemento del lenguaje. Suelen utilizarse para números, caracteres..
Se suelen poner en mayúsculas.

```
#define nombre [valor]      #define PI 3.141592
```

Macros

Son como nombres simbólicos pero con argumentos.
Cuidado: se sustituye literalmente y puede generar resultados imprevistos.

```
#define nombre(argumentos) expresión
```

```
#define cuadrado(x) x * x
#define cuadrado(x) ((x) *(x))
c= cuadrado (a++);  c= ((a++) * (a++));
```

La directiva **undef** → deshace la definición.

Inclusión de ficheros

```
#include <fichero>   -> En el directorio de librerías del compilador
#include "fichero"   -> En el mismo directorio que el archivo o donde se indique, sino lo encuentra va al otro directorio
```

Solo ficheros de cabecera (header)
Si es librería standard de C++ hay que añadir: **using namespace** std;

Compilación condicional

- *#ifdef NOMBRE* Compila las líneas siguientes si NOMBRE está definido
- *#ifndef NOMBRE* No compila si NOMBRE está definido
- *#if expresión* Se evalúa *expresión (admite defined o !defined)*
- *#else, #elif*
- *#endif Termina bloque condicional*

Usos:
- Escribir para varias máquinas o compiladores (por ejemplo si depende del tamaño de los datos o direcciones)
- Protección ante declaraciones circulares
- En desarrollo para código adicional:
  - depuración
  - cálculos de eficiencia….

```
#ifndef _CIRCULO_H_
#define _CIRCULO_H_
   // Código que se tiene que compilar una sola vez
#endif
```

Otras directivas

- **#line** constante [“fichero”] - para definir numero de línea en errores, depuración simbólica o cuando otro programa genera código C++
- **#error** cadena – genera mensaje de error con *cadena* si se compila ese código
- **#pragma** parámetros – depende de la implementación, se suele usar para modificar la configuración de la compilación.

Librerías

Cadenas de caracteres

```
#include <cstring>   // Versión C++
#include <string.h>  // Versión C
```

char * puntero a conjunto caracteres, debe acabar en ‘\0’

Funciones:
- int strlen (char *str) – longitud sin contar con ‘\0’
- Int strcmp (const char *str1, const char *str2) – compara 2 cadenas -> 0 si son iguales, >0 si el 1er carácter es mayor en str1 que str2, <0 si es menor
- char * strcpy(char * destino, char * origen) – copia una cadena en otra (tamaño destino >= origen)
- char * strcat(char * destino, char * origen) – concatena origen al final de destino

Entrada / salida

stdio.h (C)
iostream (C++) Jerarquía de clases. Se realizan las operaciones a través de canales.
Canal = refleja flujo de datos desde fuente a consumidor (cin,cout,cerr)

Salida: cout (<<)

- ostream &ostream::**put** (char ch) cout.put ('A').put('B'); // Imprime A y B en cout
- ostream &ostream::**write** (char *buff, int max) cout.write ("Solución", 3); // Imprime Sol
- ostream &ostream::**flush ()** – vacía buffer forzando su escritura
- ostream &ostream::**seekp**(long pos) – mueve puntero escritura hasta la posición indicada
- Long &ostream::**tellp**(long pos) -devuelve posición actual del puntero de escritura

Entrada: cin (>>)

- istream &istream::**getc** () -extrae carácter char c= cin.getc()
- istream &istream::**get** (char &ch) -extrae carácter y lo sitúa en ch
- istream &istream ::**get (char *buf, int max, char ch='\n')** – lee hasta max-1 carác. o ch
- istream &istream ::**getline**(**char *buf, int max**) – lee línea hasta max-1 caracteres
- istream &istream ::**read** (**char *buf, int max**) –lee max caracteres
- istream &istream::**seekg** (long pos) –mueve puntero hasta posición indicada
- istream &istream ::**tellg()**– devuelve posición actual del puntero de lectura
- istream &istream ::**putback**(**char ch**) – devuelve último carácter leido
- istream &istream ::**peek()** –devuelve siguiente carácter sin extraerlo del canal.

Ficheros: fstream.h

Modo	Acción
ios::app	Añadir nuevos datos al final del fichero
ios::ate	Abrir y desplazarse hasta el fin del fichero
ios::in	Abrir para extraer (por omisión en ifstream)
ios::out	Abrir para insertar (por omisión en ofstream)
ios::trunc	Sobrescribir el contenido (por omisión, si se indica ios::out y no se indica ios::app ni ios::ate)
ios::nocreate	Si el fichero no existe, error
ios::noreplace	Si el fichero existe, error si se abre para escribir y no está ios::app ni ios::ate
ios::binary	Modo binario

- ifstream()
- ifstream(char *nombre, int modo=ios:in)
- ofstream()
- ofstream(char *nombre, int modo=ios::out)
- fstream()
- fstream(char *nombre, int modo)
- void (char *nombre, int modo))
- void close() - cierra fichero

El **formato** se aplicará según los indicadores:
- skipws
- left
- rigth
- dec, oct, hex (iomanip.h)
- uppercase, endl,…

```
cout << hex << v << endl;
```

Se pueden crear nuevos manipuladores:

```
ostream &upperhex (ostream &os)
{
   os << setiosflags (ios::showbase | ios::uppercase) << hex;
   return os;
}

cout << upperhex << valor << endl;
```

Errores

- int ios::good() – no error→≠ 0
- int ios::bad() - error→≠ 0
- int ios::fail() – falló la última operación
- int ios::eof() – fin de fichero
- int ios::rdstate() – devuelve el estado
- int ios::clear(int=0)– inicializa el estado del canal

UNIDAD 10: EL PREPROCESADOR DE C++. LIBRERÍAS 2 de 2

CSTDLIB

Funciones generales: memoria, números aleatorios, comunicación con el entorno…

- void **exit** (int estado)- cierra programa, ficheros, vacía buffers..
- int **atoi** (const char *str) – cadena caracteres a entero)(INT_MAX o INT_MIN)
- int **atof** (const char *str) – cadena caracteres a real (HUGE_VAL)
- int **abs**(int num) – valor absoluto
- **div_t div**(int numerador, int denominador)
- Int rand() –números pseudoaleatorios

```
div_t res= div (38, 5);
int resultado= res.quot;    // resultado valdrá 7
int resto= res.rem;    // resto valdrá 3
```

CMATH

- double **asin** (double x) - arcoseno
- double **acos** (double x) - arcocoseno
- double **atan** (double x) ó (double y, double x)
- double **ceil** (double x) - redondeo entero no menor que x
- double **floor** (double x) - redondeo entero no superior que x
- double **fabs** (double x) – absoluto de un real
- double **fmod** (double numerador, double denominador)

CCTYPE

Clasificar caracteres, parámetro es int (corresponde al carácter)

- int **isalnum** (int c) **-** alfanumérico
- int **isalpha** (int c) **-** letra
- int **isdigit**(int c) **-** dígito
- int **islower**(int c) / isupper(int c) – minúscula / mayúscula
- int **isspace**(int c) – blanco, tab o nueva línea
- int **ispunt**(int c) – signo puntuación
- int **isxdigit**(int c) – dígito hexadecimal

- int **toupper** (int c) – cambia mayusc →min.
- int **tolower** (int c) - cambia minúsc. → mayusc.

ERRORES

cerrno.h. **errno**= variable global (int) que guarda resultado de la última función ejecutada.
void perror (const char *str) → personalizar mensaje error.
No obstante, C++ dispone de mecanismo de excepciones para manejo errores.

CLIMITS.H Y CFLOAT.H

Definiciones simbólicas para números reales y enteros

CHAR_BIT	Número de bits que ocupa un carácter
INT_MIN	Mínimo entero representable
INT_MAX	Máximo entero representable
UINT_MAX	Máximo entero sin signo representable
FLT_MAX	Máximo número real (float, double y long double) representable

DBL_MAX	Máximo número real
LDBL_MAX	Máximo número real
FLT_MIN	Mínimo número real (
DBL_MIN	Mínimo número real (
LDBL_MIN	Mínimo número real (

STL (Standard Template Library)

Estructuras de datos genéricos (contenedores) y sus operaciones

Librería	Descripción	Funciones
vector	Un vector almacena los elementos de manera estrictamente lineal. Los vectores son útiles para: acceder a cada elemento por su índice, iterar sobre los elementos siguiendo un orden, agregar o eliminar elementos en cualquier posición, etc.	• front: devuelve el primer elemento • back: devuelve el último elemento • size: devuelve el número de elementos que contiene • empty: indica si el vector está vacío • operador [ ]/at: devuelve el elemento de una posición • push_back: inserta un elemento al final • pop_back: elimina el último elemento • insert: inserta un elemento en la posición indicada • erase: elimina el elemento de la posición indicada • swap: intercambia el contenido de dos vectores • clear: borra todo el contenido del vector
deque	Contiene un tipo de datos de lista doblemente enlazada. Este tipo de lista es útil para: acceder a cada elemento por su índice, iterar sobre los elementos siguiendo un orden, agregar o eliminar elementos de cualquier extremo, etc.	• Las mismas funciones que vector • push_front: insertar un elemento al principio • pop_front: elimina el primer elemento
list	Da la posibilidad de manejar datos organizados en forma de lista. Las listas son útiles para: realizar una inserción eficiente de elementos en cualquier parte, movimiento eficiente de elementos dentro de la lista, iterar sobre elementos hacia delante o hacia atrás, etc.	• front: devuelve el primer elemento • back: devuelve el último elemento • size: devuelve el número de elementos que contiene • empty: indica si la lista está vacía • push_back: insertar un elemento al final • pop_back: elimina el último elemento • push_front: inserta un elemento al principio • pop_front: elimina el primer elemento • insert: inserta un elemento en la posición indicada • erase: elimina el elemento de la posición indicada • swap: intercambia el contenido de dos listas • clear: borra todo el contenido • splice: mueve elementos entre listas • remove: elimina elementos con determinado valor • remove_if: elimina elementos que cumplan una condición • unique: elimina valores duplicados • merge: fusiona dos listas ordenadas • sort: ordena la lista • reverse: invierte el orden de los elementos
queue	Contiene los tipos de datos y la funcionalidad para manejar una estructura de datos del tipo cola, en la que los elementos se insertan por un extremo y se extraen por el contrario.	• size: devuelve el número de elementos que contiene • empty: indica si la cola está vacía • push: inserta un elemento • pop: elimina el siguiente elemento • top: devuelve el elemento de la cima • front: devuelve el siguiente elemento • back: devuelve el último elemento (el más nuevo)
priority_queue	Contiene una cola con prioridades en la que siempre se devuelve el elemento que más prioridad tenga, según un criterio de ordenación.	• size: devuelve el número de elementos que contiene • empty: indica si la cola está vacía • push: inserta un elemento • pop: elimina el elemento de la cima • top: devuelve el elemento de la cima
set	Guarda un conjunto de elementos únicos, es decir, no puede haber dos elementos iguales. Los elementos son su propia clave.	• size: devuelve el número de elementos que contiene • empty: indica si el conjunto está vacío • insert: inserta un elemento • erase: elimina un elemento • swap: intercambia el contenido de dos conjuntos • clear: borra todo el contenido • find: busca un elemento • count: cuenta las veces que está un elemento en e conjunto
multiset	Es igual que un conjunto, pero permitiendo más de un elemento igual.	Las mismas que set
map	Guarda un conjunto de elementos únicos, según una clave dada. Cada elemento está compuesto por la clave y el elemento que se quiere guardar.	• Las mismas que set • operador [ ]: devuelve el elemento de una posició
multimap	Al igual que map, pero permite más de un elemento con la misma clave.	Las mismas que map
stack	Contiene los tipos de datos y la funcionalidad para manejar una estructura de datos del tipo pila, en la que los elementos se insertan y se extraen siempre de un mismo extremo de la estructura (la cima).	• size: devuelve el número de elementos que contiene • empty: indica si la pila está vacía • push: inserta un elemento en la cima • pop: elimina el elemento de la cima • top: devuelve el elemento de la cima

bitset	Es un contenedor de valores lógicos, es decir, contiene valores 0 (falso) o 1 (cierto).

- Los operadores que manejan bits (&, |, ^, etc.)
- operador [ ] / test: devuelve el bit de una posición
- set: pone a 1 los bits (todos o el indicado)
- reset: pone a cero los bits (todos o el indicado)
- flip: cambia el valor de un bit al opuesto (uno o todos)
- to_ulong: convierte el conjunto a un entero sin signo
- to_string: convierte el conjunto al tipo string
- count: cuenta el número de unos
- size: devuelve el tamaño
- any: devuelve cierto si hay algún uno
- none: devuelve cierto si no hay ningún uno