Resumen Lenguajes de programación

Alex Martínez Ascensión

26 de febrero de 2019

1. Tema 1: Fundamentos de programación

El abaratamiento del hardware en los 70 hizo que se desarrollara una programación estructurada, legible por el programador.

La base de los ordenadores se asienta en la máquina de von Neumann, que consiste en:

- Memoria: almacena instrucciones y datos en palabras binarias. Una dirección de memoria corresponde al número de la posición que almacena esa información. El acceso a la memoria es independiente de si se almacena una instrucción o un dato.
- Unidad Central de Proceso (CPU)

  1) Unidad de control: lee la memoria y genera señales para realizar tareas. Las palabras de memoria que le la unidad de control son las instrucciones.

  2) Unidad aritmética: recibe señales de la unidad de control, los datos de la memoria y realiza operaciones aritméticas.
- Unidad de entrada/salida: permite introducir datos en la máquina, y comunica los resultados.

La memoria de la máquina de von Neumann tiene 1024 palabras, de 40 bits. Cada bit puede ser 0 o 1. La codificación es:

- Si la palabra es un dato, se representa como un entero con signo.
- Si es una instrucción, representa dos instrucciones, codificadas en los primeros 20 bits (más significativos) y los últimos 20 bits.

El tipo de instrucciones permitidas son:

- Operaciones aritméticas: incluye la operación y la ubicación de los operandos.
- Movimiento de datos: incluye las posiciones de memoria base y destino.
- Control de flujo: permite saltar en posiciones del programa. Las instrucciones de un programa se ejecutan secuencialmente. Para saltar entre líneas se emplea la sentencia *goto*:

  1) El *goto* incondicional salta a la posición X siempre.

  2) El *goto* condicional salta si se cumple que el número a comparar sea mayor o igual que 0.

La máquina tarda más en mover una instrucción de memoria a CPU que en ejecutar la instrucción, luego lo primero es el paso limitante en un programa.

En las primeras máquinas de von Neumann se hacia referencia a posiciones de memoria específicas, es decir, existía un direccionamiento absoluto. Esto es un problema, porque si tenemos instrucciones codificadas en lineas con direccionamiento absoluto, y creamos una nueva instrucción, todas las instrucciones por debajo de esta tienen un nuevo número, y los *goto* no sirven. Este problema tardó en resolverse. Asimismo, si se sustraía una instrucción, la manera "sencilla"de solventarlo era rellenar la línea de instrucción con una instrucción "no operación".

1.1. Lenguaje ensamblador

En los 50 el ensamblador fue un avance, pues asignaba a cada operación de un código máquina un nombre. De este modo, era más legible para el programador entender el código máquina. Es decir, la instrucción 0001011010100010 pasa a ser `LOAD J`, de modo que ensamblador toma `LOAD J` y lo transforma a código máquina.

El problema del ensamblador deriva en que el código máquina era único de cada máquina, luego cada ensamblador era diseñado específicamente para cada máquina.

1.2. FORTRAN

FORmula TRANslation fue diseñado por IBM y permitia al programador especificar procedimientos numéricos concisamente, y pasarlo a código máquina. FORTRAN fue diseñado para que su compilación fuera óptima para la máquina IBM 704, en 1954. Esta máquina implementaba una sentencia de bifurcación con tres opciones: `IF (expresion) N1, N2, N3` donde `N1, N2, N3` son posiciones de memoria según si la comparación de la expresión era mayor, igual o menor.

La sentencia iterativa de FORTRAN I era `DO N1 variable = primer, ultimo`. De modo que se repetía las instrucciones donde `N1` es la etiqueta de la última sentencia del fragmento de código a repetir.

FORTRAN I no tipaba, sino que la primera letra de la variable indicaba el tipo: I, J, K, L, M, N indicaba entero, y el resto eran reales. Las variables podían tener hasta 6 caracteres.

FORTRAN II (1958) permitía la compilación de subrutinas de manera separada. En los 60 llegó FORTRAN IV, en el 78 FORTRAN 77 y, en los 90, FORTRAN 90. Todas estas modalidades permitían la compilación en cualquier máquina siempre que existiera un compilador adecuado.

En FORTRAN 90 la palabra `program` incluye las instrucciones del programa principal. Es decir, se inicializa con `program XXXX` y se termina con `end program`. `implicit none` al inicio del programa obliga al programador a declarar todos los nombres de variables. La variables se declaran con tipo al inicio del programa: `real::x`, y si proviene de un archivo externo (variable externa, función) se hace `real, external::x`.

1.3. ALGOL

ALGOL 58 fue diseñado en los 50 con la premisa de que su sintaxis ha de ser tan próxima a la notación matemática como fuera posible. También tenía que ser independiente de la máquina, como pasaba con FORTRAN I en la época. A diferencia de FORTRAN I (1) las variables podían tener cualquier longitud y (2) las variables vectoriales podían albergar cualquier número de dimensiones (FORTRAN I sólo 3) (3) el límite inferior de los vectores podía ser mayor que 1; y (4) se podían emplear sentencias de selección anidadas (en FORTRAN no).

En ALGOL 60 se incluyeron unas cuantas mejoras:

- Se introdujo el concepto de bloque de código, donde se declaraban variables locales al bloque.
- Se permitieron dos formas de pasar parámetros a los subprogramas: (1) paso por valor: se hace una copia del parámetro, y si la copia es modificada en el subprograma, el original se mantiene como al inicio; y (2) paso por referencia: el subprograma trabaja con el parámetro original, siendo afectado por cambios posteriores.
- Se permitieron procedimientos recursivos.
- Se podía declarar un tamaño de array variable, de modo que el tamaño del array venia dado por el valor de la variable en un momento.

Aunque ALGOL 60 no triunfó, algunas de sus mejoras sí lo hicieron.

1.4. LISP

A raíz del desarrollo de la inteligencia artificial, se desarrolló el paradigma de la programación funcional. En el desarrollo de LIPS se implementaron las listas: secuencias con elementos que podían ser de varios tipos (bool, números, arrays, símbolos, caracteres, u otras listas). Operaciones de listas incluían extracción, transformación o eliminación de elementos, así como búsqueda o selección de elementos según condiciones.

Todo este conocimiento de listas (desarrollado en 1956) se plasmó en el lenguaje IPL, y en 1958 se desarrolló LISP, usado ampliamente en IA.

1.5. COBOL

COBOL fue un lenguaje para negocios, y se desarrolló con una premisa: tenia que ser lo más cercano posible al inglés. COBOL incorporó conceptos como la sentencia `define`, empleada en macros como lenguaje de alto nivel. También soportaba estructuras de datos jerárquicas.

1.6. Prolog

PROgramming LOGic fue desarrollado en los 70. El concepto de programación lógica se basa en el empleo de hechos y reglas para representar la información y usar deducción para responder preguntas. Las reglas son relaciones del tipo `if and`, y los hechos son un tipo especial de regla que se verifica sin satisfacer ninguna condición.

1.7. SIMULA 67

SIMULA 67 es una extensión del ALGOL 60, e incluye el concepto de orientación a objetos, incluyendo conceptos como clase, objeto o herencia [AUNQUE Smalltalk fue el primer lenguaje en introducir la POO como tal]. SIMULA 67 introdujo también el concepto de puntero y gestión dinámica de memoria.

1.8. Pascal

La gran limitación de los lenguajes hasta la década de los 60 era que el control de flujo venía principalmente dado por sentencias `goto`. En los 70 se desarrolló la programación estructurada, que buscaban acabar con el problema del código espagueti.

Pascal fue basado en ALGOL 60, y facilitaba la programación estructurada, de modo que el programa podía ser leído de arriba a abajo sin depender en las clásicas sentencias de control de flujo. Por esta razón fue ampliamente usado hasta los 90.

Similar a FORTRAN 90, el programa principal va enclaustrado en la palabra reservada `program`, terminando en `end.` La primera parte del programa empieza con `var` y es donde se declaraban las variables, que termina en `end`. El segundo `begin` incluye ya el cuerpo del programa.

Asimismo las palabras `function` y `end` permiten la definición de una función, tanto dentro como fuera del programa. Si la función se escribe dentro del programa, va dentro de la parte de `var`.

A diferencia de FORTRAN, las asignaciones se hacen con `:=`, en lugar de `=`. Asimismo, las sentencias de ejecución siempre terminan en punto y coma.

1.9. Ada

Ada fue diseñado con un gasto de dinero bastante considerable por el Departamento de Defensa de EEUU, para acabar en agua de borrajas. Ada incluyó 4 principales características:

- Permite declarar paquetes con objetos, tipos de datos y procedimientos.
- Permite el tratamiento de excepciones.
- Las unidades de programa pueden ser genéricas. Es decir, a la hora de instanciar la unidad pueden existir características de los parámetros que ya están precisadas.
- Permite la ejecución de unidades de programa especiales, llamadas tareas.

1.10. C++

Al contrario que FORTRAN 90 y Pascal, C++ distingue mayúsculas de minúsculas en el código, y tienen una estructura más flexibles, pues pueden declararse en cualquier punto del programa, siempre que se declaren antes de ser usadas.

1.11. Paradigmas de programación

- Programación imperativa. Recordamos de la máquina de von Neumann que el programa y los datos están en la memoria. La CPU ejecuta las ordenes con los datos, todo almacenado en la memoria. Por tanto, las variables que representan las celdas de memoria juegan un papel clave.

  En la programación imperativa se realiza (1) asignación de variables mediante sentencias de asignación, (2) repetición de acciones mediante iteración o recursividad [menos eficiente este último]. Un programa consiste en (1) la declaración de un estado compuesto por variables, y por un punto de control; (2) la especificación de la secuencia de acciones que modifican el estado mediante asignaciones en las que se evalúan expresiones y se cambian los valores de variables, y sentencias de control de flujo.

  Por lo general los lenguajes con paradigmas puramente imperativos suelen ser eficientes a la hora de traducirse a código máquina, pero son menos expresivos.

- Programación funcional. Se realiza la evaluación de expresiones construidas a partir de llamadas a funciones. Las funciones cumplen la propiedad de transparecia referencial: las funciones no tienen efectos laterales en el programa, y solo dependen de los argumentos de entrada.

  Como tal, un lenguaje puramente funcional no emplea variables ni sentencias de asignación, sino que usan aplicaciones funcionales, expresiones condicionales y recursión. Un lenguaje funcional siempre suele implementar (1) funciones primitivas (2) formas funcionales que permiten crear funciones más complejas, (3) operaciones para la aplicación de la función y (4) estructuras para representar datos.

  Por lo general, los lenguajes funcionales permiten una programación a más alto nivel, ya que no hay que preocuparse de gestionar variables, pero disminuye la eficiencia de la ejecución.

- Programación lógica: se basan en el cálculo preicativo. Este paradigma suele permitir el desarrollo en paralelo, y ha sido empleado para el procesamiento de lenguaje natural y el desarrollo de sistemas expertos.

- Orientación a objetos: La orientación a objetos tiene aplicaciones interesantes en a reutilización de software. El diseño orientado a objetos se basa en (1) la abstracción de usar un objeto sin conocer sus detalles internos, (2) la ocultación de información, es decir, que no todos los atributos del objeto sean accesibles al usuario y (3) la modularidad, es decir, la propiedad de descomponer un sistema en módulos que interactúan entre sí.

  Un lenguaje OO debe soportar tres conceptos. (1) Clase: descripción de un tipo de objeto, que tiene variables internas, y operaciones aplicadas a los objetos de esa clase, denominados métodos. (2) Herencia: permite compartir datos y operaciones entre clases, así como definir clases derivadas de una clase base, sin tener que redefinir los atributos de la clase. (3) Ligadura dinámica: al invocar un método de un objeto, el sistema examina la clase y aplica la operación adecuada. Esa examinación se hace a la hora de invocar la operación. Por el contrario, en la ligadura estática se realiza la elección a la hora de la compilación.

1.12. Métodos de implementación

Recordamos, dos piezas clave de un ordenador son su memoria interna (almacena datos y órdenes) y el procesador (realiza las operaciones). Los programas escritos en lenguajes de alto nivel son denominados como código fuente, mientras que los de bajo nivel son código máquina. Generalmente, el código fuente se basa en un sistema operativo para implementar las órdenes a código máquina.

Existen varios modos de implementación:

- Interpretación pura: el intérprete le las instrucciones del programa fuente con los datos de entrada, y ejecuta directamente dichas sentencias. Este método es ventajoso porque los errores de ejecución se relacionan directamente a la línea del código fuente. Sin embargo, el tiempo de ejecución se reduce considerablemente.
- Compilación: el código fuente se traduce a instrucciones en código máquina por el compilador, que genera el programa objeto. La compilación se hace en dos pasos: (1) un analizador léxico agrupa caracteres en unidades léxicas (operadores, símbolos, palabras especiales), que son reconocidas por el analizador sintáctico, que genera árbol de análisis con la estructura sintáctica del programa. Al contrario que la interpretación, la compilación es mucho más rápida, pero sus errores no son tan fáciles de rastrear.
- Sistema híbrido: este sistema primero traduce el código fuente a un lenguaje intermedio, que permite una interpretación más sencilla, pasándolo por el analizador léxico, el analizador sintáctico y el generador de código intermedio. Este tipo de implementación es, por ejemplo, Java, que requiere su máquina virtual para ejecutar el código final.

2. Tema 2: Comenzando a programar en C++

`#include <iostream>` permite realizar operaciones de salida y entrada en consola. Ejemplos:

- `std::cout << "Texto"` Genera el texto cargándolo en el buffer de salida en consola.
- `std::endl` Finaliza la línea en consola.

`std::cout` mantiene el texto en un buffer y lo lanza cuando se cumple que (1) se alcanza la capacidad máxima de `std::cout`, (2) antes de ejecutarse una operación de lectura de datos o (3) se indica expresamente con `std::endl` o `std::flush`. `std::flush` se diferencia de `std::endl` en que no añade un salto de línea.

`#include <iomanip>` es una directiva que permite manipular la manera en la que se enseñan los enteros o floats en consola.

- `std::setprecision(X)` genera números con X cifras significativas (antes y después del punto, vamos), realizando redondeo cuando sea necesario.
- `std::scientific` transforma la salida en notación científica: 32.456 ->3.2456+e001
- `std::fixed` transforma la salida en formato fijo.
- `std::setprecision(X)` en estos dos casos indica el número de dígitos detras del punto decimal.
- `std::hex` y `std::oct` transforman los números enteros en números en representación hexadecimal u octal. `std::showbase` añade `0` o `0x` a los octales o hexadecimales respectivamente.
- `std::boolalpha` enseña datos booleanos como booleanos en consola, mientras que `std::noboolalpha` transforma `true` en 1 y `false` en 0.
- Si queremos rellenar una palabra con caracteres, `std::setw(X)` nos rellena por la izquierda el restante de espacios en blanco. `std::setfill('Y')` nos cambia el símbolo de relleno, y `std::left` cambia el relleno por la derecha.

Lineas de ejemplo de lo anterior:

- `std::cout << std::scientific << std::setprecision(3) << 193.235` ->1.932e+001
- `std::cout << std::hex << std::showbase << 11` ->0xB
- `std::cout << std::noboolalpha << false` ->0
- `std::cout << std::setw(8) << std::setfill('*') << "hola"` ->****hola

Además del flujo de salida `std::cout` que hemos mencionado, existen otros flujos, como `std::cin` que es el flujo de entrada desde teclado; `std::cerr` que sirve para notificar errores, y `std::clog`, que es un buffer de almacenamiento similar a `cerr`, sólo que el primero se vuelca inmediatamente, lo cual disminuye la eficiencia computacional.

3. Temas 3 y 4: declaración de variables

Una variable es una abstracción de una celda o celdas donde se almacenan datos. Cada variable tiene unos atributos, como (1) el nombre o identificador, con reglas según el lenguaje, y con prohibición de usar palabras reservadas; (2) dirección de la primera posición en la memoria; (3) el valor de la variable o (4) el tipo de dato.

Las constantes son variables cuyo valor no puede ser cambiado a lo largo del programa, una vez declaradas. Las constantes se declaran con la palabra `const` así: `const double pi = 3.14;`

La declaración de varias variables puede hacerse en una línea o varias:

```
int a, b, c;
```

```
int a = 1, b = 2, c = 3;
```

El ámbito de una variable es la parte del programa en la que existe, como el bloque de código principal, o un bloque de código secundario. Existen variables locales, que sólo existen en un ámbito determinado, y fuera de él no; variable globales, cuyo ámbito es todo el código; y variables en memoria dinámica, que no tienen un nombre, sólo la dirección de memoria y el valor/tipo. También existen los parámetros formales, que son las variables que aparecen en la definición de la función.

Si la variable puede referenciarse dentro de una parte del programa, entonces es visible. Una variable A oculta a una variable B cuando al llamar a la variable en un ámbito el valor es el de A.

En C++, una variable global y una local pueden tener el mismo nombre. Para diferenciarlas, [si la variable es `x`] se emplea el operador ámbito (`::`) y se llama a la variable `::x`

3.1. Tipos de datos

El tipo de la variable define qué valores y qué operaciones puede tener la variable. Un tipo condiciona el número de posiciones que ocupará la variable en memoria. Distinguimos 3 categorías.

- Nivel de máquina: tipos primitivos [entero, coma flotante, carácter]
- Nivel de lenguaje: tipos estructurados [arrays, estructuras, listas]
- Nivel de programador: tipos enumerados y clases.

Los tipos primitivos más comunes son:

- Entero: los soporta directamente el hardware. El bit más significativo es el signo. Existen varios, como `int`, `long int` u otros.

- Real en coma flotante: es un coeficiente multiplicado por diez elevado a un exponente entero. Según la precisión puede ser `float` o `double`. `double` es mejor porque el hardware está optimizado para trabajar con ellos.
- Boolean: sólo pueden tomar el valor `true` o `false`. Se puede representar con los enteros 0 y 1, pero como ocupa menos memoria que un entero, es más eficiente.
- Caracter: C++ soporta Latin-1, que tiene 256 caracteres, mejor que ASCII (128). Para símbolos extendidos se usa unicode (16 bits = 65536).
- void: en C++, `void` se usa como el tipo de retorno de funciones que no devuelven nada; como tipo de punteros que apuntan a objetos con tipo desconocido.

Para conocer el mayor y menor número (límite) de un tipo, se emplea la cabecera `limits`, con la función `numeric_limits`; de la siguiente manera:

```
#include <limits>

std::numeric_limits<double>::min(); std::numeric_limits<int>::max();
```

Asimismo, `std::numeric_limits<T>::epsilon()` devuelve el *epsilon* del número, es decir, el número positivo más pequeño tal que 1 + *epsilon* - 1 >0.

A la hora de inicializar tipos básicos, existe un "cero" para todo tipo, de modo que, si la variable inicializada es GLOBAL, entonces presentará un valor de 0 por defecto. Si la variable es local, entonces tomará el valor que tenga la dirección de memoria. El valor cero es 0 en tipos numéricos, y el valor cero de la tabla de caracteres para `char`, de modo que el carácter diferirá según el valor cero de la tabla que se use.

En los tipos definidos por el programador / lenguaje hay varias opciones:

- Asignar un nombre a un tipo ya existente con `typedef`. Por ejemplo, `typedet in ArrayInt10[10];` genera el tipo `ArrayInt10`, que es un array con 10 elementos.
- Tipos enumerados: está construido por constantes que toman valores específicos según el usuario o por defecto (0, 1, 2, ...). Para construir una variable enumerada en C++ se emplea la palabra `enum`:

  ```
  enum nombre_var {nombre1 = valor1, nombre2 = valor2, ...};
  enum color{blanco, azul, verde, rojo};
  ```

- Estructura [C++/Java/C] o record [Pascal]: es una colección de una o varias variables de diversos tipos, y agrupadas bajo un nombre común. Un ejemplo de estructura es el siguiente:

  ```
  struct Estrella {char tipo; double distancia; int brillo;};
  Estrella e1 = {'a', 1.63, 4};
  ```

- Uniones: está construido por varias variables, pero sólo se permite usar una de ellas al declararla. Por ejemplo: `union ID {int i; char nombre;}`.
- Arrays: son un grupo de variables con el mismo tipo, de modo que es preciso declarar el tipo de los elementos, así como el nombre o el número de elementos (tamaño). El array puede ser unidimensional, en cuyo caso se inicializa así `int arr1D[3] = {1, 2, 3};`; o multidimensional (2D aquí), que se inicializa del siguiente modo: `int arr2D[3][2] = {{1,2},{3,4},{5,6}};`. Para acceder a los elementos se hace: `arr1D[x]` o `arr2D[x][y]`

Otro tipo especial es la cadena de caracteres, que se considera como un array de caracteres. Por ejemplo: `char[] hola = "Hola!";` Una cadena de caracteres siempre termina en el caracter nulo `'\0'`. Importante: las cadenas de caracteres van en dobles comillas, y los caracteres a secas en simples.

Se pueden hacer estructuras con arrays, y arrays de estructuras:

```
struct \begin{Estrella {char nombre[30]; char tipo; double coordenadas[3]};
Estrella e[] = {{'abds', 'A', 1.2, 2.2, 2.1},{'dajw', 'B', 3, 4, 8.9}};
```

Un tipo de librería estandar interesante es `std::string`, que a diferencia del array de caracteres convencional (1) no requiere un tamaño previo y (2) incluye el string nulo, que es el que se crea al inicializar la variable. Así, un ejemplo sería `std::string frase = "Hola!";`

Si seguimos la estructura `std::string nombre(n,c)` nos generará una cadena del caracter `c` repetida `n` veces.

Los flujos también tienen sus propios tipos: `std::cout`, `std::cerr` y `std::clog` pertenecen al tipo `std::ostream`; y `std::cin` pertenece al tipo `std::istream`. Por otra parte, los tipos `std::ofstream` y `std::ifstream` permiten la entrada y salida a ficheros.

3.2. Punteros

Un tipo de variable especial es el puntero, cuyo objetivo es obtener el valor de una dirección de memoria. El puntero puede tener un valor concreto o un valor nulo. Existen dos operadores principales en punteros: el operador *dirección-de*, que devuelve la dirección de memoria de una variable; e *indirección*, que indica el valor asociado a la dirección de memoria.

Cuando se declara un puntero, se declara el tipo de la variable a la que apunta. Supongamos el puntero `p`, pues para definir el tipo de la variable a la que apunta, se declara así: `int *p`. Es importante definir el tipo del puntero, porque el valor que tenga que almacenar igual tendrá más de una posición. Es decir, a un puntero `int` no le podemos asignar un valor `float` porque el espacio asignado no es suficiente.

Otro operador asignado a un puntero es `&`. `&` asigna a un puntero la dirección de una variable. Veamos un ejemplo para entenderlo mejor

```
int main(){
    int *p;
        int v = 2;

        std::cout  << p << " " << *p  << std::endl;
        std::cout  << &v << " " << v  << std::endl;

        p = &v;

        std::cout  << p << " " << *p  << std::endl;
        std::cout  << &v << " " << v  << std::endl;

        *p = 9;

        std::cout  << p << " " << *p  << std::endl;
        std::cout  << &v << " " << v  << std::endl;
        return 0;
}
```

En la primera linea `int *p` lo único que hace es crear el puntero, de modo que se asigna una dirección de memoria a `p`, y `*p` adquiere el valor del que haya en la dirección de memoria `p`. Ahora declaramos `v`, en una dirección de memoria `&v` y con un valor de 2. La primeras dos lineas de impresión retornan 0x401bdb 1528349827 // 0x61ff18 2. En la siguiente línea, `p = &v` indicamos que la dirección de memoria de `p` sea la de `v`, de modo que ahora p tendrá el mismo valor y dirección que `v`. El print lo demuestra: 0x61ff18 2 // 0x61ff18 2. Como ambas variables están puenteadas, si cambiamos el valor de `*p`, entonces cambiamos el de `v`: 0x61ff18 9 // 0x61ff18 9.

IMPORTANTE: si en la primera línea queremos asignar directamente `*p = 9` nos va a dar un fallo. Para hacer eso tenemos que declarar la variable en memoria dinámica.

3.3. Variables en memoria dinámica

Por último, las variables en memoria dinámica son otro tipo especial de variables, ya que estas no dejan de existir cuando el bloque de código en el que se encuentran termina; de modo que son eliminadas por el usuario o al terminar el programa. Las variables en memoria dinámica, además, no tienen nombre asociado, luego la referencia ha da almacenarse en un puntero.

Para declarar una variable en memoria dinámica se emplea la palabra `new`. Un ejemplo de inicialización es el siguiente:

```
int *p; p = new int; *p = 3
```

De este modo, `p` es un valor en memoria (`0x781928`) y `*p` tiene un valor de 3.

Para borrar la variable dinámica tenemos que borrar el puntero. Para ello escribimos `delete p`

El mal manejo de estas variables ocasiona dos errores típicos:

- Punteros a variables eliminadas, cuando el usuario ha borrado la variable dinámica. También pasa si se referencia una variable local que no está en su ámbito.
- Variables dinámicas perdidas: si el puntero que apunta a la variable dinámica se pierde (por asignarse otra vez), no se puede borrar ni usar la variable; de modo que consume memoria hasta que el programa termina, ocasionando fugas de memoria.

3.4. Jibberjabber de lenguajes

- Nombre de variables: En C++ puede empezar po letra o guion bajo y seguir con letras, números o guión bajo. En FORTRAN I la inicial de la variable implicaba su tipo. C++, C y Java distinguen mayúsculas y minúsculas.
- Longitud de variables: FORTRAN 90 y C: 31 caracteres; C++, Ada y Java: no hay límite.
- Inicialización: C++/Java lo permiten, Pascal o Modula-2 no.
- Lugar de declaración: C obliga al inicio, C++/Java no.
- Constantes: en Pascal el valor ha de ser simple, en Modula-2/FORTRAN 90 puede ser resultado de expresión simple (constantes, operadores y otras constantes = ligaduras estáticas); y en Ada/C++/Java se permiten variables no constantes [el valor de la constante dependerá del del resto en el momento de la declaración = ligadura dinámica].
- Bloques: en C/C++/Java se delimitan con , en Pascal se delimita con `begin/end`
- Booleanos: ALGOL 60 introdujo este tipo por primera vez. C no tiene el tipo, y recurre a expresiones numéricas.
- Arrays (declaración): FORTRAN 90 emplea paréntesis y Pascal/C/C++/Java corchetes.
- Arrays (límite inferior): C/C++/Java asocian un límite inferior de 0; u FORTRAN 77 / FORTRAN 90 en 1.
- Arrays (almacenamiento): en FORTRAN se hace por columnas, y en el resto por filas.
- Arrays (inicialización): FORTRAN 77/C/C++/Java lo permiten, Pascal/Modula-2 no.
- Arrays (comprobación): Pascal/Ada/Java comprueban en la ejecución que no hay índices fuera de rango; C/C++/FORTRAN no se hace la comprobación (pero va más rápido).
- Variables en memoria dinámica: se eliminan automáticamente en Java [recolector], pero no en C/C++.

4. Temas 5 y 6: Asignaciones y expresiones

Una sentencia de asignación especifica una expresión y el destino donde guardar esa expresión. En C++ una asignación se hace con el símbolo =,es decir, `variable = expresion`. También se pueden hacer varias asignaciones del estilo `v1 = v2 = v3 = ... = expr` . Existen cuatro tipos principales de expresiones:

- Aritméticas: se especifica un cálculo aritmético con operadores y/o llamadas a funciones.
- Relacionales: se establece una comparación a través de un operador relacional (==, >=, <=, <, >, ...).
- Booleanas: son expresiones compuestas por operadores booleanos (OR, NOT, AND) y otras expresiones relacionales.
- Expresiones condicionales: consisten de un operador ternario `?:` que toma tres expresiones: `e1 ? e2 : e3`. Si se cumple `e1` entonces se realiza e2, y si no se cumple, se hace e3. Sería un equivalente del típico `IF e1 THEN e2 ELSE e3`

4.1. Operadores

Existen varios tipos de operadores según varios criterios.

- Número de operandos
  1) Operadores unarios: admiten un operando. Ej. `NOT`, + y - de signo de números.
  2) y 3) Operadores binarios (dos operandos) y ternarios (tres operandos).
- Posición de notación
  1) Infija: el operador va entre los operandos (a+b).
  2) Prefija: el operador va antes (+ab).
  3) Postfija: el operador va después (ab+).

En C++ se puede hacer una asignación con operadores aritméticos, es decir, la expresión `var = var ? expr` se convierte en `var ?= expr`. Los operadores aritméticos están optimizados para el compilador.

Otro tipo de operador es el de incremento o decremento ++ / --. Aunque hacen lo mismo, el orden de asignación cambia según se haga a la izquierda (prefijo) o a la derecha (postfijo) de la variable.

- `x = i++ -> x = i; i = i + 1;`
- `x = ++i -> i = i + 1; x = i;`

4.2. Asociatividad y precedencia

Precedencia: A grandes rasgos (de más a menos)

```
[. => [] ++ --] [! - + * new] [* / \%] [+ -] [<< >>] [< > <= >=] [== !=] [&&] [||] [= ?=]
      1             2          3      4       5         6          7     8     9    10
```

- 1/2: Operaciones y funciones propias del lenguaje (indexación de arrays, negación, operadores unarios, desreferencia de punteros...).
- 3/4: Operadores aritméticos al uso.
- 5: Inserción y extracción (I/O).

- 6/7: Igual que / menor o igual que / ...
- 8/9: AND / OR.
- 10: Asignación y asignación con operadores aritméticos.

Asociatividad: existen operaciones asociativas hacia la izquierda [a+b+c = ((a+b) + c)] o hacia la derecha [a**b**c = a**(b**c)].

4.3. Sistema de tipos

Sobrecarga de operadores: un operador está sobrecargado cuando, por la razón que sea, tiene una función adicional a la propia (básica) del sistema, o, más generalmente, cuando tiene funciones diferentes según al tipo al que esté asociado. Por ejemplo, el operador `*` entre dos enteros difiere de entre un entero y un array.

Conversiones de tipo: pueden ser explícitas, si el programador las define directamente en el programa (hacer un cast), o implícitas [coerciones] si las hace directamente el compilador. La coerción sucede porque la máquina sólo puede operar entre unos tipos de datos específicos. Existen reglas para hacer coerciones (entre tipos básicos, por ejemplo, `double > float > int > char`, o si una función está definida como `int`, el tipo del return será convertido a `int` si se puede). El problema obvio de las coerciones es que el compilador pierde eficiencia.

En C++, por compatibilidad con C, permite trabajar con enteros como si fueran booleanos. Así, 0 es `false`, y cualquier entero positivo es `true`. Así, `true + true` es 2, por ejemplo.

Debido a los problemas que puede originar trabajar con diversos tipos, los lenguajes implementan la verificación de tipos, que puede ser estática o dinámica. En la verificación estática se hace la comprobación una vez, al traducirse el programa a código máquina; y en la dinámica se hace durante la ejecución en el programa introduciendo el compilador código extra. Ya que esto último hace que la verificación dinámica sea más lenta, los lenguajes suelen implementar verificación dinámica.

En línea con esto último, un lenguaje es fuertemente tipado si todas sus expresiones pueden ser verificadas estáticamente.

4.4. Operadores aritméticos, relacionales y lógicos

Recordamos la jerarquía entre operadores aritméticos es `++/-- > - (unario) > /*\% > +-`; entre relacionales y lógicos es `! > [< <= > >=] > [== !=] > && > ||`. Entre los operadores aritméticos y los relacionales, los aritméticos tienen preferencia.

Cuando se aplica / a dos enteros se devuelve el cociente entero.

En cuanto a la asociatividad, `||` y `&&` son asociativos por la izquierda. Así, con `||`, si una de las expresiones por la izquiera es `true`, entonces toda la expresión es `true`, y no hace falta seguir evaluando. Para `&&`, si por la izquierda una expresión es `false`, entonces no hace falta seguir evaluando, y toda la expresión es `false`.

4.5. Operadores $\ll$ y $\gg$

En C++ estos operadores están sobrecargados. Por defecto, realizan acciones de desplazamiento a la derecha (>>) o a la izquierda (<<) de palabras de bits. Por ejemplo:

```
int a = 9; // 0b0000000000000101 = 8 + 1 = 9
int b = 9 >> 1; // 0b0000000000010100 = 32 + 4 = 36
int c = 9 << 2; // 0b0000000000000010 = 2
```

Sin embargo, para las cabeceras `std::ostream` y `std::istream` el significado de los operadores es el de poner en (<<) y obtener de (>>). El operador >> ya se ha explicado con `std::cout`; ahora explicamos cómo usar <<.

<< va junto con `std::cin` para incluir una entrada de teclado en una variable. Para ello se asigna el contenido de `std::in` a una variable:

```
std::string nombre; std::cin >> nombre;
```

De este modo, el flujo de entrada se vuelva en la variable `nombre`. El flujo de entrada se hace en los siguientes pasos (1) los caracteres del teclado se len y almacenan en el flujo de entrada `st::cin`; (2) cuando hay un retorno, se vuelca el flujo a la variable. En el volcado, se desechan los salto de línea, espacios o tabuladores (con `std::noskipws` sí se tienen en cuenta) y si se encuentra uno de ellos, se almacena en la variable el flujo de entrada hasta ese punto.

Así, por ejemplo:

```
std::string var; // Introducimos "Hola mundo!"
std::cin >> var; // var = "Hola"
```

La frase `" mundo!"` queda reservada en el flujo de entrada hasta que se designe otra variable que lo soporte. En ese momento, se eliminará el espacio y se incluirá `"mundo!"` en la segunda variable.

También podemos almacenar las entradas en un array; como por ejemplo:

```
int a[3];
std::cin >> a[0] >> a[1] >> a[2];
```

4.6. Operaciones matemáticas

La cabecera `cmath` proporciona nuevas funciones como `acos(x), atan(x), cos(x), cosh(x), log(x), pow(x,y)`, que sobrecargan las operaciones al uso (`+-*/`) para acomodarlas a estas funciones.

Asimismo, la librería estándar contiene la clase `complex`, que permite trabajar con complejos. Estos complejos pueden tener a su vez diferentes tipos. Por ejemplo: `std::complex<float> num(2.3, 1)` sería $2{,}3 + i$. Existen varias funciones aplicadas al tipo `std::complex`:

- `x.real() / x.imag()` devuelven la parte real e imaginaria del número.
- `conj(x)` devuelve el conjugado.
- `norm(x) / abs(x) / arg(x)` devuelven la norma (cuadrado del módulo), el módulo y el ángulo de `x`.

4.7. Operando con strings

El tipo `std::string` permite varias operaciones propias.

- Concatenación: el operador + está sobrecargado para que, con dos `string` los junte. También se puede concatenar poniendo un espacio entre strings: `std::string a = "hola" "mundo"`
- Concatenación y asignación (+=): A una variable le concatena otra variable.
- Comparación: la comparación se hace con los operadores < y > y usa un orden lexicográfico.
- Tamaño: `x.size()` devuelve la longitud del string.
- Modificación: `assign / append` son equivalentes `a = / +`. `x.replace(p,n,y)` replaza `n` caracteres en la posición `p` con la variable `y`. `x.replace(p, y)` inserta en la posición `p` la variable `y`. `x.erase(p,n)` elimina `n` caracteres en la posición `p`.

- Búsqueda: `x.find(y)` busca la primera coincidencia de `y` en `x`, y `x.find(y, p)` busca a partir de la posición `p`. `rfind` hace lo mismo pero de derecha a izquierda.
- Comparación: `compare` hace una comparación de string o de substrings. La forma más simple es `x.compare(y)` que compara los strings enteros y devuelve un entero positivo si `y > x`. La forma más comple es `x.compare(px, nx, y, py,` que compara el substring de longitud `nx` en posición `px` de `x` con el homólogo de `y`.

Además de `string`, existe la cabecera `sstream` con el objeto `std::stringstream` que permite almacenar strings que vengan de consola u otras fuentes. Par introducir strings en un objeto `std::stringsream` se emplea la operación <<. Un ejemplo:

```
std::stringstream msg;
msg << "Hola, esto es " << "un " << "mensaje.";
std::cout << msg.str() << std::endl;
```

4.8. Operando con punteros

En el anterior tema se había hablado de punteros. Ahora es lo mismo con más historia. Antes hablábamos de los operadores & y *. El operador & (dirección de) proporciona la dirección de memoria en la que se almacena la variable; y el operador * (indirección) proporciona el dato almacenado en la posición de memoria.

Ponemos este ejemplo de uso:

```
int i = 30;
int *p;
p = &i; // Asigna al punto la direccion de memoria de i.
```

Si ahora decimos que `*p = 20`, entonces, como `p` apunta a `&i`, la dirección de memoria es la misma, luego el valor de `i` es 25.

Pongamos otro ejemplo:

```
int i1 = 1, i2 = 2, p1, p2;
p1 = &i1;
p2 = &i2;
p1 = p2 // Caso 1
*p1 = *p2 // Caso 2
```

En este caso, tenemos que `i1` y `i2` son dos variables con dos direcciones de memoria diferentes; luego `p1` y `p2` apuntan a direcciones diferentes. Haciendo `p1 = p2`, el puntero `p1` apunta a `&i2`, luego el valor es 2, y la dirección de memoria es la misma que la de `p2`.

Si por el contrario escribimos `*p1 = *p2`, el valor de `*p1` es 2, pero la dirección de memoria es la de `&i1`. Así, el valor de `i1` ahora es 2.

Si definimos un puntero por `p`, la siguiente dirección de memoria viene dada por `p+1`, y su valor por `*(p+1)`. Esto sirve para acceder a arrays, ya que dos elementos contiguos están en dos direcciones de memoria contiguas. Así, por ejemplo, si definimos `int v[4] = {1,2,3,4}; int p = &v[0];`, entonces `*(p+2)` es 3.

4.9. Operando con vectores

El vector es un objeto definido por el tipo `std::vector` de la libreria `vector`. Los vectores se definen como `std::vector<tipo> nombre = {1, 2, ...}` o como `std::vector<tipo> nombre(n,d)` donde `n` es el número de elementos, y `d` es el valor del elemento repetido. Los vectores tienen varias operaciones.

- Acceso. El acceso se hace con el operador `[]` o con la función `at`: `v[i]` o `v.at(i)`. A diferencia de `[]`, cuando se llama `at` primero se comprueba que el índice a buscar no esté fuera de rango. Si lo está, se lanza una excepción `std::out_of_range`.
- Acceso por iterador. Un iterador se define como `std::vector<tipo>::iterator nombre_iterador`. Una vez declarado el iterador, se puede asignar a una posición de un vector (`iter = v.begin()` o `iter = v[2]`), y, como con un puntero, se puede acceder al valor de las siguientes posiciones haciendo `*(iter + n)` o `*(iter++)`.
- Tamaño. `v.size()`. Para saber si está vacío, se escribe `v.empty()`. `v.capacity()` te da el número de posiciones que tiene reservadas el vector, estén asignadas o no.
- Asignación. La función `assign` incluye elementos de un array en otro, y elimina los elementos que estaban anteriormente en esa posición. Existen varias maneras de asignar: (1) `v.assign(n,d)` asigna `n` posiciones con el valor `d`; (2) `v.assign(iter1, iter2)` se asigna el rango de valores que tomen los iteradores `iter1` hasta `iter2`; (3) `v.assign(p1, p2)` se asigna el rango de valores que tomen los punteros `p1` y `p2` asignados a un array.
- Inserción y eliminación: `v.push_back(x)` añade el valor `x` al final del array (si hay sitio), y `v.pop_back()` elimina el último término. `v.insert(p, x)` inserta el elemento `x` en la posición `p`; y `v.insert(p, iter1, iter2)` inserta los elementos de un vector desde la posición asociada a `iter1` hasta la asociada a `iter2`. Similarmente, `v.erase(p)` y `v.erase(iter1, iter2)` eliminan las posición asociada a un iterador, o entre dos iteradores.
- Reserva: `v.reserve(n)` reserva `n` espacios posteriores, si los hay. Si no, reubica el vector en una dirección de memoria que tenga los huecos libres necesarios.
- Operaciones relacionales: los operadores `== < <= > >=` están sobrecargados, y se emplea un orden lexicográfico para la comparación.

4.10. Operando con estructuras

Las estructuras también tienen operaciones jugositas.

- Acceso: para realizar acceso a un elemento de una estructura se emplea el operador punto `.`. Por ejemplo, si desarrollamos `struct Persona{char nombre[30]; int edad};` y creamos la estructura `Persona Paco {"Francisco",` entonces podemos saber la edad de `Paco` haciendo `Paco.edad`. Si tenemos una estructura anidada tipo `struct strGrande{strPeque p1, strPeque p2}`, podemos acceder a las variables de las subestructuras concatenando puntos: `nombrestrGrande.p1.variable`.
- Asignaciones: si dos estructuras tienen el mismo tipo de estructura, entonces haciendo `nombre_estr_1 = nombre_estr_2` realizamos la asignación.
- Punteros a estructuras: al igual que otros objetos, las estructuras aceptan punteros. Por ejemplo, `Persona *p; p = &Paco` nos daría un puntero hacia la estructura `Paco`. Así, podemos acceder a la edad de `Paco` haciendo `(*p).edad` [ponemos paréntesis porque `.` precede a `*`] o, con notación propia, `p -> edad`.

4.11. Jibberjabber de lenguajes

- Asignaciones: en C++/C/Java/FORTRAN/Basic con =; en Ada/Modula-2/Pascal con `:=`.
- Expresiones condicionales / asignaciones con operadores aritméticos /incremento y decremento: C/C++/Java las tienen.
- Mezcla de tipos: Ada sólo permite mezclar enteros con coma flotante; el resto son mejores.
- En Java no hay coerciones; en C y C++ sí.

5. Temas 7 y 8: control de flujo

Los lenguajes de programación tienen sentencias de control de flujo para establecer decisiones y realizar diversas tareas durante la ejecución. Vamos a mirar las sentencias más importantes.

5.1. Sentencias de selección

C++ tiene dos sentencias de selección, if y switch.

La forma de la sentencia if es:

```
if (expresion)
        codigo_clausula_then
else
        codigo_clausula_else
```

En este caso, `if` y `else` son palabras reservadas del lenguaje, y los códigos de clausulas pueden ir entre llaves o sin ellas (sólo si es una línea). La expresión puede ser booleana o aritmética. Si es aritmética, se toma *false* si el valor es 0, y *true* si es diferente de cero.

Se pueden generar estructuras más complejas anidando varias sentencias `if`. La sentencia `if` más próxima a una sentencia `else` es la que la asocie. Una estructura típica es la escalera `if-else-if`:

```
if (expr1)
   codigo1
else if (expr2)
   codigo2
...
else
   codigo_else
```

Por otro lado está la sentencia `switch`, que permite seleccionar un caso según una constante que surja.

```
switch (expresion de seleccion) {
        case c1:
                codigo1
        case c2:
                codigo2
        ...
        default:
                codigon
                }
```

5.2. Sentencias iterativas

También llamadas bucles, permiten la ejecución ininterrumpida de un código, hasta que se satisfaga una condición.

La sentencia más general es el bucle `for`, que tiene la siguiente forma:

```
for (inicializacion; condicion; iteracion)
        cuerpo del bucle
```

Similar al `if`, el cuerpo ha de ir entre llaves y, si es una sentencia, puede ir sin ellas. La cabeza del bucle tiene 3 elementos: (1) inicialización de la variable: puede ser local al bucle o no, y puede ser una o más variables; (2) condición aritmética o Booleana que, mientras sea *true* se ejecuta; (3) iteración donde se actualiza la variable de inicialización.

Pese a que el `for` usual tiene los tres elementos en la cabecera, pueden creare bucles con cabeceras incompletas. Por ejemplo:

```
for ( ; fin ; ){}
for (; ; ){}
```

El primer bucle dura siempre que la variable `fin` sea `true`; y el segundo corre hasta que se termine por dentro con un `break`.

También existen los bucles `while` [bucle lógico pre-condición] y `do-while` [bucle lógico post-condición], que tienen la forma siguiente:

```
while (condicion)
        cuerpo while
```

y

```
do
 cuerpo
while (condicion);
```

En el bucle `while` se evalúa primero la condición y, si es cierta, la ejecuta. En el bucle `do-while`, por el contrario, primero ejecuta el cuerpo y luego evalúa la condición. Entonces, si la condición es falsa desde el principio, `do while` ejecuta una vez el cuerpo, y `do-while` no lo ejecuta nunca.

En cualquiera de los bucles podemos hacer dos tipos de ejecuciones inmediatas:

- La sentencia `break` finaliza automáticamente la ejecución del bucle.
- La sentencia `continue` se salta el contenido del bucle y pasa a la siguiente iteración.

5.3. Excepciones

Las excepciones son, como otros objetos, objetos con tipo propio, que paran la ejecución del programa. Para capturar un error, es decir, para cuando surja un error en un bloque, que el programa no finalice directamente, se emplea la cláusula `try-catch`.

```
try{
Bloque de codigo}
catch (tipo excepcion1 arg){
Bloque de catch}
catch (tipo excepcion2 arg){
Bloque de catch}
...
catch (tipo excepcionn arg){
Bloque de catch}
```

Obviamente, el tipo de cada error debe ser diferente por cada catch. Si el tipo de excepción que surge no corresponde con ninguno del tipo de catch que hay, se pueden hacer dos cosas:

- Se emplea un código general de `catch`, que consiste en tres puntos entre paréntesis:

  ```
  catch (...) {
  Codigo
  }
  ```

- Si no hay nada más, el programa termina. Se invoca automáticamente una función de la librería estandar llamada `terminate()`, que llama a `abort()` y el programa termina.

En un `catch` podemos mostrar el mensaje original empleando `nombre.what()`:

```
catch(std::invalid_argument errorx){
std::cout << "El error es este: \n" << errorx.what() << std::endl;}
```

Además de atrapar excepciones las podemos generar con la sentencia `throw`:

```
throw nombre_excepcion;
throw std::domain_error ("Error: tu dominio esta fuera de rango.")
```

Si queremos hacer algo con la excepción, necesitamos ponerla dentro de un bloque `try`.

Los tipos más comunes de excepción son los siguientes:

- `std::domain_error`
- `std::invalid_argument`: el argumento no es válido para la función
- `std::length_error`: error al crear un objeto muy largo
- `std::out_of_range`: un argumento de una función está fuera de rango.
- `std::overflow_error`: Overflow aritmético
- `std::range_error`: Error en un range interno
- `std::underflow_error`
- `std::bad_alloc`: sucede cuando, al crear una nueva variable en memoria dinámica con `new`, no queda memoria en el almacenamiento dinámico.+

5.4. Entrada por teclado

`std::cin` es el método de introducción de datos por consola. Sin embargo, teníamos un problema cuando no sabíamos *a priori* el número de veces que íbamos a necesitar la entrada por teclado. Ahora que sabemos usar bucles, podemos fácilmente hacerlo con un `while`:

```
while (std::cin >> variable_entrada){
Bloque de codigo}
```

`std:cin` incluye conversión a booleano porque pertenece al tipo `std::istream`. Este bucle de código va a seguir activo hasta que:

- Se produzca un error con el teclado (de hardware o lo que sea).
- Encuentra un carácter de fin de fichero. En este caso el flujo de entrada acaba en false y termina.
- Encuentra un whitespace, en cuyo caso termina el ciclo, pero puede seguir otro nuevo.
- Encuentre un carácter que no pertenezca al tipo asociado a la variable que recibe los caracteres.

Si por ejemplo se introdujera "1 2 3 4 return*z* `variable_entrada` fuera `int` por cada ciclo introduciría un número, ya que cuando encuentra un espacio. Cuando procesa el return, el programa se queda aún a la espera de un eof para salir del bucle.

Existen varias funciones que se pueden llamar en el entorno `std::cin`.

- `std::cin.eof()` Devuelve `true` si se encuentra un fin de fichero.
- `std::cin.fail()` Devuelve `true` si ha encontrado un fallo en el flujo de entrada.

- `std::cin.bad()` Devuelve `true` si se ha producido un error fatal en el flujo de entrada.
- `std::cin.good()` Devuelve `True` si NO se ha producido un error.
- `std::cin.clear()` Libera el flujo a un estado que le permite seguir trabajando.
- `std::cin.ignore(n,c)` Descarta caracteres del flujo de entrada hasta que el número es `n` o encuentra un caracter `c`.
- `std::cin.get()` Devuelve el primer caracter del flujo de entrada, extrayéndolo del flujo (considera los whitespace también).
- `std::cin.peek()` Devuelve el primer carácter del flujo de entrada, sin extraerlo del flujo (considera los whitespace también).

Si, por la razón que sea, `std::cin` es falso, es necesario usar `std::cin.clear()`; y si tiene dentro caracteres, hay que usar `std:cin.ignore()`. Para quitar todos los caracteres, se hace así: `std::cin.ignore(std::numeric_limits<std::stre`

Los resultados de `std::cin.get()` o `peek()` se pueden almacenar en una variable `char`.

5.5. Entrada y salida por fichero

La entrada y salida de fichero se hace con la cabecera `fstream`, concretamente con los tipos `std::ifstream` y `std::ofstream`. Por tanto, se declara la variable archivo con uno de esos tipos:

```
std::ifstream archivo_entrada
```

Una vez se ha declarado la variable, hay que introducir o leer texto del fichero asociado. Para ello hay 3 modos:

- `std::ios::in` Lee el texto
- `std::ios::out | std::ios::trunc` Escribe borrando el contenido del fichero antes.
- `std::ios::out | std::ios::app` Añade contenido al ya existente.

La manera completa de leer/escribir es entonces

```
std::ofstream archivo_out;
archivo_out.open("Ruta/de/archivo.txt", std::ios::out | std::ios::app)
```

O, todo en una linea:

```
std::ofstream archivo_out("Ruta/del/archivo.txt", std::ios::out | std::ios::app)
```

Las operaciones de lectura y escritura se hacen con << >>, y el archivo se cierra con la función `close()`.

La lectura puede ser de dos modos principales:

- Carácter a carácter o palabra a palabra. Para ello se emplea la siguiente estructura:

```
char c;
std::string s;
while (!f.eof()){
        f >> std::noskipws >> c;
        f >> s;
}
```

  La diferencia principal es que si la carga de texto es a un `char` se carga letra a letra, y si `std::string` se carga palabra por palabra. `std::noskipws` incluye los whitespaces en el texto.

- Línea a línea: Se almacena toda la línea en un string, se hace con la función `getline()`.

```
std::string s;
while (getline(f, s)){
f >> s;
}
```

IMPORTANTE: Si se guarda el nombre de archivo en una variable, esta debe ser array de chars. Para transformar un string en array de chars se emplea `nombre.c_str()`.

5.6. Jibberjabber de lenguajes

- Originalmente las sentencias `goto` eran las únicas de control de flujo. ALGOL 60 fue el primero en introducir las sentencias actuales.
- Los bloques de código se delimitan con llaves en C++/C/Java y con begin/end en Pascal.
- FORTRAN tenian una formula primitiva de case, ALGOL-W la generalizó, Pascal tiene la sentencia `case` (con begin/end), en Ada la selección tiene que ser un tipo enumerado y tiene la opción de **others**; FORTRAN 90 tiene una sentencia case/when/others, y Java/C++/C tienen la sentencia switch que sabemos.
- Pascal tiene una sentencia `for` con la forma `for v := i to f do` para subidas, y `for v := i downto f do` para descendentes. Solo sube/baja de 1 en 1. Si el bucle termina bien, la variable no se almacena; pero si falla, se queda el último valor. Ada es similar a Pascal.
- Expresión de control de bucle: en C++ puede ser aritmético o booleano, en C aritmético, y en Java booleano.
- En FORTRAN 90 no había bucles controlados por expresión booleana. En Ada, el equivalente del `while` es el `repeat-until`.
- El equivalente de `break` en Modula-2 es `exit`.
- C++/Java/Ada tienen control de excepciones; C/FORTRAN no.

6. Temas 9 y 10: Subprogramas

Un subprograma es un fragmento de código definido aparte del programa principal al que se le asigna un nombre y puede aceptar parámetros. Los subprogramas se clasifican en funciones, si retornan un valor, o procedimiento, si no lo retornan.

Toda función que se digne debe tener [en C++] (1) nombre (2) tipo de retorno (3) parámetros formales (4) algoritmo. Cuando se usa la función para algo se la llama o invoca. En ese momento se evalúan los parámetros. El paso de los parámetros puede ser por valor o por referencia:

- Por valor: el valor de la variable se copia a otra dirección de memoria y se trabaja con él. Cualquier cambio en la variable interna no afectará a la externa.
- Por referencia: el valor de la variable está asociado a su dirección de memoria, de mode que si la variable se modifica dentro de la función, fuera de ella también. En C++ los arrays siempre se pasan por referencia.

En el caso en el que la variable se pase por referencia, hay dos modos de pasar los argumentos:

- Usando punteros: el parametro formal es un puntero al tipo de la variable y se pasa como parámetro la dirección de memoria.

```
void masuno(int * a){
*a += 1;
return;}

int main(){
int s = 9;
masuno(&s);
return 0;}
```

- Usando parámetros de referencia: se declara la función con un parámetro que incluye el operador posición. En el resto de partes se emplea la variable a secas.

```
void masuno(int & a){
a += 1;
return;}

int main(){
int s = 9;
masuno(s);
std::cout << s ;
return 0;}
```

En C++ el tipo de la función es obligatorio. Si la función no retorna nada, se tiene que emplear el tipo `void`:

```
void f(){}
```

En ese caso, las variables deberían pasarse por referencia si se quieren cambiar.

Con respecto a la localidad de las variables, las variables son locales a la función y, dentro de ésta, siguen la misma jerarquía de bloques (se pueden ocultar). Las funciones también se pueden sobrecargar:

```
int A(int n, int m, int p){
 return n + m + p;}
int A(int n, int m){
 return n+m;}
```

En C++ las variables dentro de una función pueden llevar la palabra reseravda `static`. En ese caso, la variable se almacena de forma global, es decir, se mantiene una vez llamada la función, de modo que si se llama a la función de nuevo, conserva el valor de cuando se corrió la última función.

Se puede llamar a `A(2,3,4)` o a `A(2,3)` sabiendo que el número de argumentos o sus tipos son diferentes.

6.1. Punteros

Punteros colgantes: cuando se hace un return, se genera una copia de la variable (como si se pasara por valor, cuando puede hacerse), y la variable retornada aparece en otra dirección; de modo que la variable interna es inaccesible.

Por ejemplo, este código

```
#include <iostream>

int A(){
        int i = 1;
        std::cout << "i dentro de la funcion " << i << ' ' << &i << std::endl;
        return i;
}


int* B(){
        int j = 1;
        std::cout << "j dentro de la funcion " << j << ' ' << &j << std::endl;
        return &j;
        }

int main(){
        int i = A();
        std::cout << "i fuera de la funcion " << i << ' ' << &i << std::endl;

        int * j = B();
        std::cout << "j fuera de la funcion " << j << ' ' << &j << std::endl;
        return 0;
}
```

El output es

```
i dentro de la funcion 1 0x61feec
i fuera de la funcion 1 0x61ff1c
j dentro de la funcion 1 0x61feec
j fuera de la funcion 0 0x61ff18
```

Vemos que en `A`, `i` toma una dirección de memoria diferente a cuando se hace el return. En `B` pasa igual, pero además, la variable queda colgada porque estamos pasando una referencia: en lugar de pasar `0x61feec`, el return pasa `0x61ff18`, otra dirección distinta, que tiene otro valor asociado, por supuesto.

Las funciones en C++ también pueden actuar como punteros. Si empleamos la siguiente construcción:

```
tipo_retorno (*nombre_puntero)(lista_tipo_params); //El paréntesis en nombre puntero es necesario
```

Podemos crear una función que apunte a esa dirección. Esto nos es útil cuando en un sitio ha de emplearse una función genérica, de modo que varias funciones pueden ser llamadas donde está el puntero.

```
int suma(int a, int b){
return a+b;
}
int resta(int a, int b){
return a-b;
}

// Opción 1
int main(){
int (* operacion_binaria)(int, int);
operacion_binaria = suma;
std::cout << operacion_binaria(3, 4);
return 0;}
```

```
// Opción 2
int aplica_operacion(int (* operacion_binaria)(int, int), int x, int y){
std::cout << "El resultado de la operacion es " << (* operacion_binaria)(x, y);
return 0;}

int main(){
aplica_operacion(suma, 3, 4);
aplica_operacion(resta, 3, 4);
return 0;}
```

6.2. Recursividad

Decimos que una función es recursiva cuando dentro de su algoritmo se hace una llamada a si misma. Toda función recursiva necesita un caso base para que pueda "salir", porque si no entraría en bucle.

Existen tres tipos de recursividad principales:

- Lineal: La función f llama, a lo sumo, una vez a f de nuevo. En esta recursividad, cada vez que se llama a la función se tiene que mantener una memoria del valor anterior, de modo que si la función tiene que llamarse muchas veces puede haber un desbordamiento. En el tipico caso del factorial:

  ```
  int fact(int n) {
  if (n == 0)
  return 1 ;
  else
  return n * fact(n-1) ;
  }
  ```

  Si hacemos `fact(4)` tenemos que almacenar en memoria que hay que multiplicar 4 * `fact(3)`, luego 4 * 3 * `fact(2)`, y luego 4 * 3 * 2 * `fact(1)`, lo cual puede ser tedioso.

- De cola: este tipo de recursividad modifica el argumento de llamada de modo que el valor es calculado sin necesidad de recursividad. Por ejemplo:

  ```
  int fact(int n) {
  return tail_fact(n,1) ;
  }

  int tail_fact(int n, int a) {
  if (n == 0)
  return a ;
  else
  return tail_fact(n-1,n*a) ;
  }
  ```

  En este caso, `tail_fact` ya mantiene el cálculo del factorial dentro de los argumentos, luego no hay que recuperar hacia atrás la información.

- Recursividad anidada: sucede cuando hay varias llamadas a la misma función o cuando en alguno de los argumentos de la llamada recursiva se encuantra la misma función. Un ejemplo es la función de Ackermann:

  ```
  int A(int n, int m) {
   if (n == 0) return m+1;
  ```

```
 else if (m == 0) return A(n-1, 1);
 else return A(n-1, A(n, m-1));
}
```

6.3. Excepciones

Las excepciones son como siempre, solo que a la hora de escribir funciones, puede indicarse en la cabecera de la función (no es obligatorio), qué tipos de excepciones pueden lanzarse.

```
tipo_retorno nombre_funcion (lista_params)
 throw (excep1, excep2, ...)
 { cuerpo_de_la_funcion }
```

6.4. Prototipos

Toda función ha de ser declarada, y puede ser definida (declarada + algoritmo) al mismo tiempo o después. Para declarar una función simplemente hay que escribir

```
tipo_retorno nombre_funcion (lista_tipo_params)
```

6.5. Organización del programa en varios ficheros

Muchas veces interesa tener el programa principal con la función main, y el resto de subprogramas en otras funciones. C++ permite guardar los programas en varios ficheros, siempre que se mantenga la siguiente estructura:

- En un fichero .cpp se escriben las funciones que se deseé.
- En un fichero .h (con el mismo nombre que el .cpp) se escriben los prototipos de las funciones
- Tanto el fichero .cpp de las funciones como en el .cpp principal se tiene que hacer referencia al fichero cabecera escribiendo `#include "cabecera.h"`.

Hay que fijarse en que va con comillas. Eso indica que el fichero está en el mismo direcorio que el ".cpp.ºriginal. Si se escribiera `#include <cabecera.h>` se estaría llamando a la cabecera de la librería estandar.

Además, si queremos generar un espacio de nombres (como `std` para la librería estandar) tenemos que añadir tanto en el .h como el .cpp de las funciones el siguiente trozo:

```
namespace nombre_espacio {
 declaraciones y definiciones
}
```

Por ejemplo, si el espacio es `miesp`, entonces en el archivo principal, a la hora de llamar a la función `x` habría que escribir `miesp::x`.

Para que el código corra hay que ejecutar los dos .cpp.

6.6. Jibberjabber de lenguajes

- En Ada/Modula/Pascal se puede distingir un procedimiento de una función con PROCEDURE y FUNCTION, mientras que C/C++/Java no consideran los procedimientos sino que los subprogramas se definen como funciones que pueden no devolver ningún dato (aunque tienen un return implicito).
- FORTRAN77 no soporta recursividad.

7. Temas 11 y 12: Estructuras de datos

Conforme desarrollemos cada estructura la complementaremos con su desarrollo en C++. Las estructuras de datos abstractas se han definido en función de unas operaciones que se aplican a tal estructura. Para que una estructura sea de un tipo abstracto se tiene que cumplir que al menos uno de los operandos o el resultado de la operación ha de ser de ese tipo abstracto.

Muchas de estas estructuras están implementadas en la librería estándar (Standard template library, STL). Las ramas based de la STL son los contenedores, o objetos que alojan otros objetos, algoritmos aplicables sobre objetos, e iteradores, objetos similares a los punteros que permiten recorrer el contenedor como se recorre un array por punteros.

7.1. Listas

Una lista es una secuencia de 0 o más elementos de un tipo. Si su longitud es 0, entonces la lista está vacía. La lista es flexible porque su longitud puede aumentar o disminuir, y se pueden insertar o eliminar elementos. Las operaciones básicas sobre una lista L son: (1) INSERT(x, p, L), (2) LOCATE(x, L) encuentra la primera posición del elemento x, (3) RETRIEVE(p, L) devuelve el elemento en la posición p, (4) DELETE(p, L), (5,6) NEXT/PREVIOUS(p, L), (7) MAKENULL(L) vacía la lista y (8) FIRST(L).

Las listas pueden implementarse de dos formas:

- Usando un array: Se diseña la lista de modo que el elemento i tome la posición $i-1$ del array. Dado que el array tiene un tamaño fijo, hay que crear un array más grande, y registrar el último elemento de la lista. El array es ventajoso en cuanto acceso, porque el i elemento está en la i posición de memoria, y no es necesario recorrer el array. Así, el acceso es rápido. Sin embargo, la inserción y eliminación son lentas, porque hay que desplazar todos los elementos a partir del elemento insertado / eliminado; además de que el tamaño máximo de la lista está restringido al tamaño del array.
- Usando estructuras autorreferenciadas. En este caso, para cada elemento la estructura tiene el valor del elemento (o un puntero al valor) y un puntero a la siguiente dirección. En las listas doblemente enlazadas también hay un puntero al elemento anterior, para así recorrer la lista en ambos sentidos. Las ventajas/inconvenientes de esta estrategia se invierten: el acceso es más costoso porque hay que leer todos los elementos desde la cabecera hasta el deseado pero, por el contrario, eliminar e insertar elementos es mucho más rápido, ya que solo hay que cambiar los punteros asociados.

Las listas están implementadas con el tipo de dato `std::list`, y es una lista doblemente enlazada, con acceso secuencial; es decir, no se puede acceder al elemento i si no se ha recorrido la lista entera hasta ese punto.

La lista es declarada así

```
#include<list>
std::list <tipo> nombre_lista;
```

También se puede inicializar (dos ejemplos):

```
std::list<int> lst_a = {1, 2, 3, 4};
std::list<int> lst_b(2,100);  // (100, 100)
```

También se puede inicializar una lista con los elementos de otra

```
std::list<int> lst2(lst1);
```

Las operaciones más importantes aplicables a listas son:

- `lst.empty()` Devuelve `true` si está vacía
- `lst.size()`
- `lst.push_back(x)` / `lst.push_front(x)` añade un elemento al final o inicio de la lista.
- `lst.pop_back(x)` / `lst.pop_front(x)` elimina el elemento del final o inicio.

Los iteradores en listas se declaran así

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

Una vez el iterador ha sido declarado y asignado a un elemento de la lista (con `lst.begin()` por ejemplo), podremos desplazarnos por el iterador usando `nombre_iterador++` o `nombre_iterador--`, o cualquier otra operación similar. Escribiendo `*nombre_iterador` se accede al elemento asociado en la lista.

- `lst.begin()` / `lst.end()` Devuelve un iterador al inicio/fin de una lista. Para asignarlo hay que hacer `std::list<tipo>::iterator p = lst.begin();`. Importante: para `end()` el iterador marca NO el elemento final de la lista, sino lo siguiente. Para llegar al último elemento hay que escribir `p--`.
- `lst.insert(p, val)` / `lst.insert(p, n, val)` inserta `n` elementos (o 1) con el valor `val` en el sitio del iterador `p`. Si se trata de `insert(p, p1, p2)` inserta una copia de los elementos entre `p1` y `p2`. `p` no cambia de sitio.
- `lst.erase(p)` o `lst.erase(p1, p2)` elimina el elemento del iterador `p` o entre los iteradores `p1` y `p2`.
- `lst.splice(p, lst_dest)` / `lst.splice(p, lst_dest, p0)` / `lst.splice(p, lst_dest, p1, p2)` inserta en `lst` (en `p` como última posición) la lista `lst_dest`. Se puede elegir también la posición `p0` del elemento de selección en `lst_dest` o el inicio y final con `p1` y `p2`.

7.2. Pilas

La pila es una variación sobre una lista: en este caso los elementos se insertan y eliminan por uno de los extremos de la lista, siguiendo la disciplina LIFO (Last In First Out); es decir, los elementos que más tiempo llevan en la pila son los que salen antes. Las pilas aceptan las siguientes operaciones: (1) MAKENULL(S), (2) TOP(S) devuelve el elemento en la parte superior, (3) POP(S) elimina el elemento superior, (4) PUSH(S, x) inserta el elemento x en la parte superior y (5) EMPTY(S) devuelve *true* si la lista está vacía.

Las pilas se encuentran en la librería `stack` y sus funciones son estas:

- `pila.empty()` / `pila.size()`
- `pila.push(val)` `pila.pop()` introduce o elimina el valor.
- `pila.top()`

7.3. Colas

Similar a las pilas, pero en este caso la disciplina es FIFO (First In First Out). Las operaciones son las misma, solo que ahora se substituye TOP(S) por FRONT(Q), que devuelve el primer elemento; PUSH(S, x) for ENQUEUE(Q, x) para insertar el elemento, y se añade DEQUEUE(Q), que elimina el primer elemento de la cola.

Por lo general, asi se emplean estructuras autorreferenciadas, conviene que se emplee un puntero para el último elemento de la cola. Si se emplea la estrategia del array, conviene implementar el array como si fuera un anillo, de modo que se tienen los punteros `cab` y `end` para el primer y último elementos de la cola y, como sabemos la longitud del array, si se alcanza el último elemento del array, el siguiente elemento de la cola empieza enla posición 0 del array.

Las colas se encuentran en la librería `queue` y sus funciones son estas:

- `cola.empty()` / `cola.size()`
- `cola.push(val)` `cola.pop()` introduce o elimina el valor.
- `cola.front()` / `cola.back()` indica el primer y último elemento.

7.4. Mapas

Un mapa es una función que mapea (jeje) elementos de un tipo *dominio* (d) a un tipo *rango* (r), de modo que $r = M(d)$. Algunos mapas pueden ser implementados como una función (hacer un cálculo, por ejemplo, o contar el número de elementos para una lista) y otros necesitan ser descritos individualmente (como si fuera un diccionario).

Las operaciones sobre mapas son (1) MAKENULL(M) vacía el mapa, (2) ASSIGN(M, d, r) define que $M(d) = r$ tanto si ya está definido antes como si no; y (3) COMPUTE(M, d, r) devuelve *true* y asigna a una variable r el resultado de $M(d)$, y *false* si $M(d)$ no está definido.

Los mapas vienen definidos en la librería `map`, puesto que cada elemento del mapa es una tupla, entonces la declaración de los mapas se hace así

```
std::map<tipo_clave, tipo_valor> nombre_variable;
```

Los mapas también tienen las funciones `map.empty()` / `map.size()`.

Como las listas, los mapas también tienen iteradores:

```
std::map<tipo_clave, tipo_valor>::iterator nombre_iterador;
```

Cada tupla de un mapa está guardado como objeto del tipo `std::pair`, y se accede con el valor del puntero y los atributos `first` y `second`. Es decir, si `p` es un puntero, los accesos son `(*p).first` y `(*p).second`.

Para insertar nuevos elementos escribimos `nombre_mapa[clave] = valor`. Si ya existía una clave, reemplaza el valor. También se puede insertar empleando un objeto de tipo `std::pair`, pero es largo y absurdo. El primer campo del objeto `pair` es un iterador al nuevo elemento, y el segundo es un booleano que, si es `true`, inserta un elemento nuevo; y si es `false`, es que ya se ha insertado.

```
std::pair<std::map<tipo1, tipo2>::iterator, bool> p;

p = m.insert(std::pair<std:string, int>("A", 1));
```

Para acceder al iterador se usa `par.first`, y para el acceso a la variable booleana, se escribe `par.second`. Si denominamos al iterador `p`, entonces el acceso a la clave se hace como `p->first` y `p->second`. También podemos acceder al valor si conocemos la calve escribiendo `map[clave]`. Si la clave no existe, entonces te devuelve un valor vacío (0 si es `int`, por ejemplo)

Para buscar si existe un par con una clave determinada se emplea la función `find`. El resultado de la función es un iterador, luego el resultado ha de almacenarse en una variable iterador.

```
std::map<std::string,int>::iterator p; p = map.find(clave);
```

Si la clave no existe, entonces el iterador apuntará a la posición siguiente del último elemento del mapa.

También existe la función `mapa.count(var)`, que devuelve 1 o 0 según si existe la clave o no. Para eliminar elementos
I

7.5. Árboles

El árbol es un tipo de dato abstracto que mantiene una jerarquía. Una estructura es un árbol si cumple que

- Cualquier nodo tiene 0 o más nodos hijo.
- Tiene un sólo nodo raíz.
- Cualquier nodo, excepto el raíz, sólo tiene un nodo padre. Si varios nodos comparten el mismo padre, son nodos hermanos.
- Por tanto, cualquier nodo (excepto el raíz) es descendiente del nodo raíz.

La altura de un nodo se define como la longitud del camino más largo desde el nodo hasta una hoja. Por otra parte, la altura del árbol es la altura del nodo raíz.

Los árboles admiten las siguientes operaciones: (1) PARENT(n, T) devuelve el padre del nodo n y si el nodo es el raíz, devuelve un nodo nulo; (2) LEFTMOST_CHILD/RIGHTMOST_CHILD(n, T) devuelve el hijo del nodo n que más a la izquierda/derecha esté; (3) LABEL(n, T) devuelve la etiqueta del nodo n; (4) CREATE(i, v, T_1, $\cdots$, T_n) crea un nodo r con la etiqueta v y los árboles hijos $T_1, \cdots, T_n$ en la posición i; (5) ROOT(T) devuelve el nodo raíz del árbol y (6) MAKENULL(T) hace nulo el árbol.

La implementación del árbol se hace con estructuras autorreferenciadas o mediante arrays. Usando arrays se emplean dos arrays: el primero tiene la numeración del nodo padre y el segundo tiene la etiqueta del nodo. Similar al resto de ocasiones, el array es más lento si queremos insertar o eliminar nodos porque hay que alterar la estructura entera del array.

Si se emplean estructuras autorreferenciadas entonces:

- Un puntero apunta al nodo raíz
- Cada nodo tiene (1) un puntero hacia el nodo padre, (2) uno o varios punteros a los nodos hijos y (3) un puntero a la variable almacenada.

8. Temas 13 y 14: Algoritmos

Esta sección se va a desarrollar de manera muy sucinta.

El desarrollo de algoritmos es todo un arte, y hay varias cosas a considerar. Existen varios paradigmas de diseño, aunque los más importantes son 5:

- Fuerza bruta
- Divide y vencerás: consiste en dividir un problema en problemas más simples, y una vez estén resueltos, reconstruir la solución al problema inicial.

- Programación dinámica: similar al divide y vencerás en que se divide el problema en problemas más simples, solo que se implementa la memoización, es decir, guardar soluciones ya calculadas que se necesitan más tarde. En resumen, la programación dinámica suele simplificarse a problemas recursivos con memoización.
- Programación lineal: consiste en optimizar una función lineal de modo que sus variables estén sujetas a restricciones dadas por inecuaciones.
- Búsqueda y enumeración: en este paradigma los problemas suelen modelarse como grafos, de modo que se emplean rutinas de grafos para solucionar el problema.

Para describir los algoritmos suelen usarse o bien diagramas de flujo o pseudocódigo, donde los detalles de un código suelen omitirse, y se reflejan las ideas clave.

Otro concepto clave de la algoritmia es la complejidad. Para evaluar la carga computacional de un algoritmo, se parte de una magnitud que representa el tamaño natural del problema a resolver. Si N es la magnitud que representa el problema (longitud de una lista, por ejemplo) entonces la complejidad de un algoritmo se estima con una función que se acerca asintóticamente, prescindiendo de constantes de proporcionalidad. La complejidad se establece con la notación O. Una función $g(N)$ se dice que es $O(f(N))$ si existen c y N_0 tal que, para todo $N > N_0$ el tiempo de ejecución es menor a $c \cdot f(N)$.

La notación O sirve para hacer comparación de algoritmos. Sabemos que un algoritmo con $O(n^2)$ va a ser peor que $O(\log(n))$ a partir de un N. También hay que considerar que para todo algoritmo debería tenerse el valor O para el mejor caso (ordenar una lista ordenada, por ejemplo), el peor caso, y el caso medio; ya que muchos algoritmos flaquean en unas cosas u otras.

En C++ existe la librería `algoritm`, que integra los algoritmos más comunes aplicables a un vector. Por simplificación, las `p` representan iteradores del vector.

- `count(p1, p2, val)`. Devuelve el número de elementos de una secuencia entre `p1` y `p2` con valor `val`.
- `count_if(p1,p2,f)`. Devuelve el número de elementos que cumplen la función `f`.
- `remove_copy(pBegin, pEnd, pResultBegin, val)`. Devuelve en otro vector, con el iterador `pResultBegin`, la secuencia del vector original sin repeticiones de `val`.
- `replace_copy(pBegin, pEnd, pResultBegin, val, valNuevo)`. Copia la secuencia origen en la secuencia destino sustituyendo el valor `val`.
- `reverse(pBegin, pEnd)`. Invierte el orden de la secuencia entre los iteradores.
- `transform(pBegin, pEnd, pResBegin, f)`. Aplica la transformación asociada a la función y almacena el resultado en la nueva secuencia empezando por `pResBegin`. Devuelve el iterador `pResEnd`. La función de transformación tiene que aceptar el argumento con el tipo que le corresponde, y hacer el return correspondiente.