

Tema 1: Introducción al lenguaje C++

Tecnología de la Programación de Videojuegos 1

Grado en Desarrollo de Videojuegos

Curso 2023-2024

Miguel Gómez-Zamalloa Gil con de **Rubén Rubio Cuéllar**
cambios

Departamento de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

El lenguaje C++

- ♦ Sucesor del lenguaje de propósito general C
- ♦ C permite programar tanto a alto como a bajo nivel
- ♦ C++ extiende C, e incorpora Programación Orientada a Objetos
- ♦ Incorpora una biblioteca estándar que incluye plantillas (STL) para tipos abstractos de datos (asignatura EDA)
- ♦ Estándares ISO: C++{11, 14, 17, 20, 23, 26}...
- ♦ Compiladores: g++, clang++, Borland C++, Visual C++
- ♦ Entornos de desarrollo (IDEs): Visual Studio, CLion, Eclipse, Xcode, C++Builder, VSCode, MonoDevelop, Netbeans, etc.
- ♦ El estándar no incorpora bibliotecas para interfaz gráfica de usuario (GUI) ni biblioteca gráfica
- ♦ Aplicaciones: videojuegos, sistemas operativos, software edición video/audio/imágenes, bibliotecas gráficas, motores de videojuegos, etc.

Diferencias fundamentales con C#

La sintaxis es muy parecida, pero...

- ♦ Gestión de la memoria: en C++ el programador tiene control total
 - ▶ Memoria de pila vs. heap
 - ▶ Eliminación: C++ no tiene recolector automático de basura
- ♦ C++ es compilado mientras que C# es interpretado
- ♦ Cabeceras y unidades de compilación
- ♦ Entrada/salida
- ♦ Biblioteca estándar (la de C# es más amplia)
- ♦ Bibliotecas externas
- ♦ Etc., etc., etc.

C++ es mucho más potente y peligroso mientras que
C# es más limitado, sencillo y seguro

Ejemplo: programa de consola

hola.cpp

```
// Importamos las cabeceras de las bibliotecas que necesitamos
#include <iostream>           // entrada/salida, define las variables cin y cout (console)
#include <string>             // cadenas de caracteres de la biblioteca estándar (STL)
using namespace std;        /* para nombrar sin cualificar con std:: */

int main(int argc, char* argv[]) {                               // Argumentos: Array de cadenas estilo C
    string str; char ch; int año = 2022;                          // definición de variables
    cout << "¡Hola! \n¿Cómo te llamas? \n";                     // operador de inserción de una cadena
    getline(cin, str);                                           // lectura de una línea en un string
    // cin >> str;                                               // operador de extracción en un string
    cout << "Hola " << str
         << "\n¿Es el año " << 2022 << "? \n";
    cin >> ch;                                                    // operador de extracción de un char
    // cin.get(ch);                                              // lectura de un carácter
    if (ch != 's' && ch != 'S') {                                // condicional
        cout << "¿Qué año es? ";
        cin >> año;
        // operador de extracción de un int
    }
    cout << "El año " << año << ". ¡Gracias!\n";
    return 0;
}
```

Ejemplo: programa con SDL

holaSDL.cpp

```
#include <SDL.h> // Depende de la instalación
#include <iostream>
using namespace std; // Para nombrar sin cualificar con std::

int main(int argc, char* argv[]) {
    SDL_Window* win = nullptr;
    SDL_Renderer* renderer = nullptr;
    SDL_Texture* bitmapTex = nullptr;
    SDL_Surface* bitmapSurface = nullptr;
    int posX = 100, posY = 100, width = 320, height = 240;
    SDL_Init(SDL_INIT EVERYTHING);
    win = SDL_CreateWindow("Hello World", posX, posY, width, height, 0);
    renderer = SDL_CreateRenderer(win, -1, SDL_RENDERER_ACCELERATED);
    bitmapSurface = SDL_LoadBMP("../bmps\\hello_world.bmp");
    bitmapTex = SDL_CreateTextureFromSurface(renderer, bitmapSurface);
    SDL_FreeSurface(bitmapSurface);
    ...
}
```

Ejemplo: programa con SDL

- holaSDL.cpp -

```
...
if (win == nullptr || renderer == nullptr || bitmapTex == nullptr)
    cout << "Error\n";
else {
    SDL_Event e;
    bool exit = false;
    while (!exit) {
        if (SDL_PollEvent(&e) && e.type == SDL_QUIT) {
            exit = true;
        }
        SDL_RenderClear(renderer);
        SDL_RenderCopy(renderer, bitmapTex, nullptr, nullptr);
        SDL_RenderPresent(renderer);
    }
    SDL_DestroyTexture(bitmapTex);
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(win);
}
SDL_Quit();
return 0;
} // main
```

Estructura de los programas

- ✦ En C++ coexiste la **orientación a objetos** con la **programación estructurada** clásica de C
- ✦ Un programa es por tanto un conjunto de: **clases**, **funciones** y **declaraciones** (tipos, variables globales, etc.), posiblemente organizadas en módulos
- ✦ La función especial **main** designa el comienzo del programa. Toma una de estas dos formas:
 - ▶ **int main()** { body }
 - ▶ **int main(int argc, char* argv[])** { body }
 - ❖ **argc**: nº de argumentos pasados al programa
 - ❖ **argv**: array de cadenas al estilo de C (char*)

Tipos básicos

- ♦ Dos familias de tipos: los tipos básicos (`char`, `int`, etc.) y los tipos compuestos (arrays, structs, clases, etc.)
 - ▶ en.cppreference.com/w/cpp/language/type
- ♦ Tipos básicos:
 - ▶ Caracteres: `char`, `unsigned char`, `wchar_t`, `char16_t`, `char32_t`, etc.
 - ▶ Enteros: `int`, `unsigned int`, `short int`, `long int`, `long long int`, etc.
 - ▶ Coma flotante: `float`, `double`, `long double`
 - ▶ Booleanos: `bool`
 - ▶ Otros: `nullptr_t` y `void`
- ♦ Los tamaños dependen de la máquina. Se pueden obtener mediante `sizeof(tipo)`
 - ▶ Ver en.cppreference.com/w/cpp/language/types

Definición de variables

Sintaxis: **tipo** listaVariables;

♦ Ejemplo: **unsigned int** i, j;

Inicialización:

- ♦ ¡Las variables **NO** se inicializan por defecto!
- ♦ Formas de inicialización:
 - ▶ A la C: **tipo** variable = expr;
 - ▶ Constructor: **tipo** variable(exprs);
 - ▶ Uniforme o "cruda" (raw): **tipo** variable{expr};

Inferencia de tipos (auto y decltype):

- ♦ No abusar (no usar hasta que os diga)

```
int k = 0;  
auto i = k;  
decltype(k) j;
```

Constantes:

- ♦ **const** tipo variable = expr; // *Obligatorio inicializarla*
- ♦ Desde C++11: **constexpr** tipo variable = expr; // *También*

Instrucciones

La mayor parte de la sintaxis es igual en C++ y C#:

- ◆ Asignaciones
- ◆ Operadores aritméticos y relacionales (+, -, *, /, %, !, ||, &&)
- ◆ Incrementos/decrementos (++i, i++, --i, i--)
- ◆ Condicionales (excepto **switch** con strings)
- ◆ Bucles (**while**, **for** y **do-while**). El for-each es distinto

Algunas diferencias:

- ◆ Las asignaciones devuelven el valor asignado → **int** i = j = 0;
- ◆ Manejo de referencias y punteros → **int*** k = &i;
- ◆ Operadores (llamadas notación infija) → cout << "hola " << s;

Entrada/salida básica

- ♦ La E/S se realiza a través de flujos (*streams*)
 - ▶ Binarios: se manejan directamente a nivel de bytes
 - ▶ De texto: se manejan a nivel de texto
- ♦ Flujos para la E/S por consola:
 - ▶ **cin**: flujo de texto de entrada conectado con el teclado
 - ▶ **cout**: flujo de texto de salida conectado con la consola
 - ▶ Ambos definidos en la cabecera **iostream**



Entrada/salida básica

La E/S se realiza principalmente mediante el uso de los operadores `>>` y `<<` (inserción y extracción)

♦ Extracción:



* = espacios, tabuladores y saltos de línea

- ▶ Lee del flujo (saltando espacios*) y escribe en la variable
- ▶ Si no puede, se activa el flag de fallo (fail) pero el programa continúa

♦ Inserción:



- ▶ Evalúa la expresión y escribe el texto resultante en el flujo de salida

```
string nombre; int edad;
```

```
cout << "Nombre: ";
```

```
cin >> nombre;
```

```
//getline(cin,nombre);
```

```
cout << "Edad: ";
```

```
cin >> edad;
```

```
cout << nombre << ", le quedan " << 65-edad << " para jubilarse\n";
```

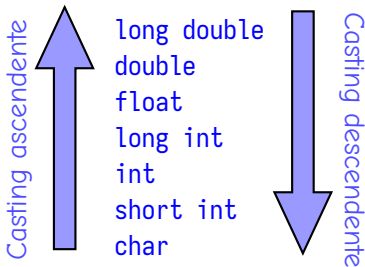


Castings

Casting ascendente

- ♦ Se hace implícitamente al operar

```
short int i = 3;  
int j = 2;  
double a = 1.5, b;  
b = a + i * j;
```



Casting descendente

- ♦ Posible pérdida de precisión

```
int i = 1234; // 4 bytes en memoria  
char c = i; // 1 byte en memoria, pérdida de información  
long j = {2.5}; // error de compilación, solo con corchetes
```

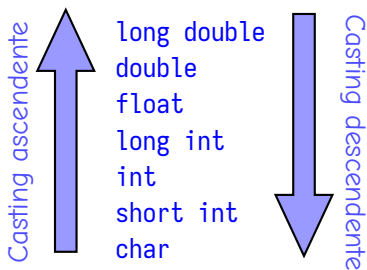
Castings

Castings explícitos:

- ♦ Sintaxis: `tipo(expr)`
- ♦ Ejemplos:

```
int a = 97;  
char b = char(a);
```

```
int a = 3, b = 2;  
cout << a / b;  
cout << double(a) / b;
```



```
// Grados Fahrenheit a Celsius: °C = (5/9) x (°F - 32)
```

```
double f;  
cin >> f;  
cout << (5/9) * (f - 32); // ¡Escribe siempre 0!  
cout << (5.0/9) * (f - 32); // Correcto
```

Tipos compuestos

Dos familias de tipos: los **tipos básicos** (char, int, etc.) y los **tipos compuestos** → en.cppreference.com/w/cpp/language/type

Tipos compuestos:

- ✦ Enumerados: **enum nombre** { valor₁, valor₂, ... } [variables];
- ✦ Referencias: **tipo&**
- ✦ Punteros: **tipo***
- ✦ Arrays (estáticos): **tipo nombre[constanteNúmeroElementos]**
- ✦ Estructuras: **struct nombre** { tipo₁ campo₁; tipo₂ campo₂; ...};
- ✦ Clases: **class nombre** { atributos y métodos };
- ✦ Funciones: son también tipos, se pueden tratar como datos

Renombrando tipos (**typedef** y **using**):

```
typedef unsigned int uint;      (alias)      using uint = unsigned int;
```

Estructuras

- ♦ Sintaxis igual que en C#:

```
struct nombre { tipo1 campo1; ...; tipoN campoN; };
```

- ♦ Ejemplo:

```
struct Fecha {  
    int dia;  
    int mes;  
    int año;  
};  
  
void escribirFecha(Fecha fecha) {  
    cout << fecha.dia << "/" << fecha.mes  
        << "/" << fecha.año;  
}
```

- ♦ Inicialización:

```
Fecha f1 = {11, 9, 2023};    // Campos en orden  
Fecha f2{11, 9, 2023};      // = es prescindible  
Fecha f3;                   // Campos sin inicializar  
Fecha f4 = {.dia = 11, .mes = 9, .año = 2023}; // C++20
```


Estructuras

- ♦ También se pueden definir constructores (e incluso métodos)

```
struct Fecha {  
    int dia; int mes; int año;  
    Fecha(int m, int d) {  
        dia = d; mes = m; año = 2023;  
    }  
};
```

- ♦ De hecho es a (casi) todos los efectos igual que una clase
- ♦ Inicialización:
 - ▶ Llamada al constructor: Fecha f(9, 11); o Fecha f = Fecha(9, 11);
 - ▶ Uniforme: Fecha f{9, 11}; o Fecha f = {9, 11};
 - ▶ Si hay constructor sin argumentos o ninguno: Fecha f;

Estructuras: pila vs. heap

Un struct puede estar en la **pila** o en el **heap**:

♦ En la pila:

```
Fecha f(4,1,2019);           // Constructor  
escribirFecha(f);
```

♦ En el heap (mediante punteros):

```
Fecha* f1 = new Fecha(1,1,2019); // Constructor  
Fecha* f2 = new Fecha{2,1,2019}; // Sintaxis uniforme  
escribirFecha(*f1); // Indirección, manda la Fecha, no el puntero  
delete f1; // ¡Hay que borrar la memoria dinámica creada!  
delete f2;
```

Arrays estáticos

- ♦ Los arrays de C# y Java son **dinámicos**
- ♦ C++ permite usar tanto arrays dinámicos como estáticos

Arrays estáticos:

tipo nombre[constanteNúmeroElementos];

- ♦ Se almacenan en la pila
 - ♦ El tamaño es **constante** y debe conocerse en tiempo de compilación
 - ♦ El tamaño no es parte del tipo
 - ♦ No se comprueba si el índice es válido
 - ♦ Inicialización:
- } Responsabilidad del programador

```
int datos[5] = {0,1,2,3,4};  
int datos[5] {0,1,2,3,4};  
int datos[] {0,1,2,3,4}; // Las tres definiciones son equivalentes  
// Cuidado: int datos[]; es un array de 0 elementos  
int datos[5] {}; // {0,0,0,0,0};  
int datos[5] = {1,2}; // {1,2,0,0,0};
```

Arrays estáticos

- ♦ Los identificadores de tipo array son punteros constantes al primer elemento del array
 - ▶ Los arrays NO se pueden asignar

```
int unos[] {1,1,1};  
int dos[3];  
dos[0] = unos[0] + 1;  
dos = unos; // ERROR de compilación (dos es constante)
```

- ▶ El paso de parámetros es por referencia (pasa el puntero)

- ♦ Tipo contenedor **array** de la STL:

```
int myarray[3] = {10,20,30};  
for (int i = 0; i < 3; ++i)  
    ++myarray[i];  
for (int elem : myarray)  
    cout << elem << '\n';
```

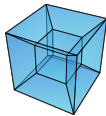


```
#include <array>  
array<int,3> myarray {10,20,30};  
for (int i = 0; i < myarray.size(); ++i)  
    ++myarray[i];  
for (int elem : myarray)  
    cout << elem << '\n';
```

Arrays multidimensionales

♦ Sintaxis:

```
tipo nombre[tamaño1][tamaño2]...[tamañoN];
```



♦ Ejemplo:

```
constexpr int NUMFILAS = 50;  
constexpr int NUMCOLS = 100;  
using Matriz = double[NUMFILAS][NUMCOLS];
```

```
int main() {  
    Matriz matriz;  
    for (int i = 0; i < NUMFILAS; i++)  
        for (int j = 0; j < NUMCOLS; j++)  
            matriz[i][j] = 0;  
    matriz[2][98] = 1;  
    cout << matriz[2][98];  
}
```

	0	1	2	3	...	98	99
0	0	0	0	0		0	0
1	0	0	0	0		0	0
2	0	0	0	0		1	0
48	0	0	0	0		0	0
49	0	0	0	0		0	0

Arrays multidimensionales

♦ Inicialización:

- ▶ Secuencias anidadas

```
int cuads[5][2] = {{1,1}, {2,4}, {3,9}, {4,16}, {5,25}};
```

- ▶ Si hay más celdas que valores proporcionados, el resto se inicializa al valor por defecto (0 para **int**, etc.)

♦ Se almacenan secuencialmente por filas (varían más rápido los índices de más a la derecha)

	0	1
0	1	1
1	2	4
2	3	9
3	4	16
4	5	25



cuads[0][0]	1
cuads[0][1]	1
cuads[1][0]	2
cuads[1][1]	4
cuads[2][0]	3
cuads[2][1]	9
cuads[3][0]	4
cuads[3][1]	16
cuads[4][0]	5
cuads[4][1]	25

(en memoria)

Cadenas de caracteres

♦ En C++ hay dos alternativas para el manejo de cadenas

1. Cadenas "al estilo de C"
2. El tipo **string**

1. Cadenas al estilo de C - "cstrings"

- ♦ Simplemente son arrays de tipo **char** (con longitud máxima) y la parte ocupada delimitada por el carácter especial **'\0'**
- ♦ No hay encapsulación ni gestión del espacio
- ♦ Funciones de utilidad en **cstring**: **strlen**, **strcat**, **strcpy**, **strcmp**, etc.

```
char hola[8] = "hola";  
char adios[] = {'a', 'd', 'i', 'o', 's', '\0'};
```

h	o	l	a	0	0	0	0
0	1	2	3	4	5	6	7
a	d	i	o	s	0		
0	1	2	3	4	5		

```
hola = adios;    // ERROR de compilación  
adios[5] = '!';  // Quita el \0, muy peligroso  
cout << adios;   // ERROR: ejecución imprevisible
```

```
strcat(hola, adios);  
// ¡No se chequea nada!  
// ¡Escribe sobre otros datos!
```

Cadenas de caracteres

2. El tipo string

- ♦ Análogas a las cadenas de C# y Java
- ♦ Encapsulación en la clase string con gestión automática del espacio
- ♦ Muchas más funciones (métodos) de utilidad
 - ▶ www.cplusplus.com/reference/string/string

```
string hola = "hola";  
string adios = "adios";  
adios.push_back('!');  
hola += adios;  
cout << hola;
```

Conversión entre string y cstring

- ♦ Es muy habitual tener que convertir de string a cstring, por ej. al usar operaciones de C que requieren un cstring
- ♦ La función c_str toma un string y devuelve un cstring

```
string filename = "datos.txt";  
SDL_Surface* imagen = IMG_Load(filename.c_str()); // Carga de imagen en biblioteca C
```


Paso de parámetros

- ◆ **Parámetros de entrada:** A través de ellos se envían datos a la función, para ser utilizados en su implementación.

```
int cuadrado(int num)
{ return num*num; }
```

```
int main()
{ int c = cuadrado(10+3); }
```

- ◆ **Parámetros de salida (&):** A través de ellos se devuelven resultados obtenidos por la ejecución → **out** en C#

```
void divEntera(int D, int d, int& c, int& r)
{ c = D / d; r = D % d; }
```

```
int main() { int c, r;
divEntera(20, 13, c, r); }
```

- ◆ **Parámetros de entrada/salida (&):** A través de ellos se envían datos y se devuelven resultados → **ref** en C#

```
void cuadrado(int& num)
{ num *= num; }
```

```
int main()
{ int c = 13; cuadrado(c); }
```

Paso de parámetros

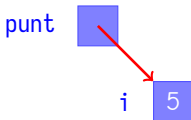
- ♦ Los parámetros por referencia pueden declararse constantes, poniendo **const** antes o después del tipo
- ♦ Apropiado para parámetros de entrada de tipo no básico:
 - Evita la copia del argumento y no permite modificarlo
- ♦ Ejemplo:

```
void escribirFecha(const Fecha& fecha) {  
    cout << fecha.dia  
        << "/" << fecha.mes  
        << "/" << fecha.año;  
    // fecha.dia++;  
    // -> Error de compilación  
}
```

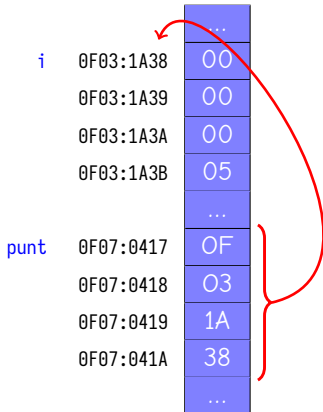
```
int main() {  
    Fecha f{2,1,2019};  
    escribirFecha(f);  
}
```

Punteros

- ♦ Los punteros contienen direcciones de memoria
- ♦ Sintaxis: `tipo* punt;`
 - ▶ La variable `punt` guarda la dirección donde se almacena un dato de tipo `tipo`



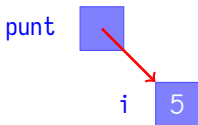
- ♦ ¿Para qué sirven los punteros?
 - ▶ Para compartir datos
 - ❖ Estructuras de datos eficientes
 - ❖ Paso de parámetros
 - ▶ Para gestionar datos dinámicos



Punteros - los operadores & y *

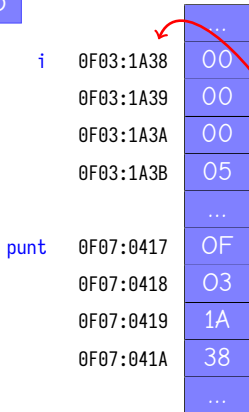
- El operador & (obtener dirección de memoria de)

```
int i = 5;  
int* punt = nullptr;  
punt = &i;
```



- El operador * (indirección)

```
int i = 5;  
int* punt = nullptr; // nulo -> 0  
punt = &i;  
cout << *punt << endl;  
*punt = 10;  
cout << i << " " << punt  
      << " " << *punt;
```



- No confundir con el * del tipo

Punteros - direcciones válidas

- ♦ Un puntero obtiene una dirección válida:
 - ▶ Al asignarle una dirección con el operador &
 - ▶ Al asignarle el valor **nullptr** (también **NULL** o incluso 0)
 - ▶ Al asignarle otro puntero válido (de mismo tipo base)

```
int i;  
int* q;      // q no tiene aún una dirección válida  
int* p = &i; // p toma una dirección válida  
q = nullptr; // ahora q ya tiene una dirección válida  
q = p;      // otra dirección válida para q
```

- ♦ Usar punteros no válidos es uno de los grandes peligros de C y C++

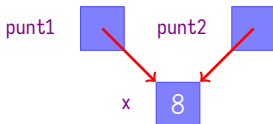
```
int* p; *p = 12;
```

- ▶ Si fuese nulo se obtendría un error de ejecución (es malo, pero...)
- ▶ Pero podría ser mucho peor si apunta a algo no válido:
 - ❖ Podríamos inadvertidamente sin querer otros datos de nuestro programa (y de otros si el sistema operativo no lo impidiera)

Punteros - copia y comparación

Compartición

```
int x = 5;
int* punt1 = nullptr; // punt1 no apunta a nada
int* punt2 = &x; // punt2 apunta a la variable x
punt1 = punt2; // ambos apuntan a la variable x
*punt1 = 8; // *punt1, *punt2 y x son la misma variable
```



Comparación de punteros

- ◆ Los operadores `=` y `!=` nos permiten saber si dos punteros apuntan al mismo dato

```
int* punt1, *punt2; // Ojo a la declaración de múltiples punteros
if (punt1 == punt2)
    cout << "Apuntan al mismo dato" << endl;
else
    cout << "No apuntan al mismo dato" << endl;
if (punt1 != nullptr) cout << "Puntero no nulo (apunta a algo)" << endl;
```

Punteros - paso de parámetros en C

- ✦ En C no existe el paso de parámetros por referencia
- ✦ El paso de parámetros por referencia en C se debe hacer por tanto a bajo nivel por medio de punteros

C

```
void cuadrado(int* num) {  
    *num = *num * *num;  
}  
  
int main() {  
    int c = 13;  
    cuadrado(&c);  
}
```

Obtiene la dirección de memoria de la variable C

C++

```
void cuadrado(int& num) {  
    num = num * num;  
}  
  
int main() {  
    int c = 13;  
    cuadrado(c);  
}
```

¡Ojo! No se pone & como en C#

Punteros - paso de parámetros en C

- ✦ En C no existe el paso de parámetros por referencia
- ✦ El paso de parámetros por referencia en C se debe hacer por tanto a bajo nivel por medio de punteros

C

C++

```
void cuadrado(int* num) {
```

*n

```
}
```

```
int main() {  
    int c = 13;  
    cuadrado(&c);  
}
```

SDL está programada en C y por tanto hay que usar punteros al llamar a sus funciones

```
void cuadrado(int& num) {
```

```
int main() {  
    int c = 13;  
    cuadrado(c);  
}
```

Obtiene la dirección de memoria de la variable C

¡Ojo! No se pone & como en C#

Memoria dinámica

Pila
Heap
Datos globales
Código del programa
S.O.

- ♦ El **heap** (montón)
 - ▶ Enorme zona de memoria donde podemos alojar datos que se crean/destruyen durante la ejecución
- ♦ El Sistema Gestor de Memoria Dinámica (SGMD)
 - ▶ Creación de memoria: el programa le solicita memoria y el SGMD busca una zona y devuelve la dirección (operador **new**)
 - ▶ Destrucción de memoria: el programa le indica al SGMD que un dato ya se puede eliminar y éste lo elimina (operador **delete**)
 - ▶ En otros lenguajes el SGMD incluye un **recolector de basura**
- ♦ Ventajas de la memoria dinámica:
 - ▶ El programa puede ajustar el uso de memoria a sus necesidades
 - ▶ El programa puede ajustar el tiempo de existencia de los datos
 - ▶ Es una zona muy grande (la pila está mucho más limitada)

Memoria dinámica: new y delete

♦ Operador **new**

```
tipo* nombre = new tipo;
```

- ▶ Pide al SGMD un bloque de memoria para un dato del tipo
- ▶ Se devuelve la dirección de memoria donde comienza el bloque

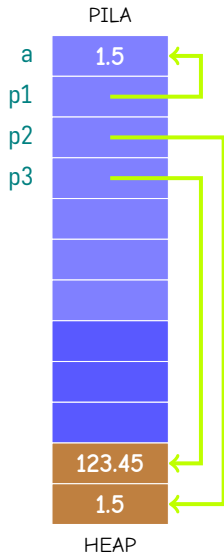
♦ Operador **delete**

```
delete nombre; // nombre ha de ser puntero
```

- ▶ Le pide al SGMD que elimine el bloque apuntado por el puntero
- ▶ Como siempre, eliminar significa “marcar como libre”
- ▶ El puntero deja de contener una dirección válida
 - ♦ Por seguridad se debe poner a nulo, nombre = **nullptr**; (así se puede saber)

Memoria dinámica - ejemplo

```
double a = 1.5;  
double *p1, *p2, *p3; // Ojo a los *  
p1 = &a;  
p2 = new double;  
*p2 = *p1;  
p3 = new double;  
*p3 = 123.45;  
cout << *p1 << endl;  
cout << *p2 << endl;  
cout << *p3 << endl;  
delete p2;  
delete p3;
```



Arrays dinámicos

- ♦ Declaración: `tipo* nombre;`
- ♦ Creación: `nombre = new tipo[expresiónNumérica];`
- ♦ Destrucción: `delete [] nombre;`

```
int n;  
... // La variable n toma valor en ejecución  
  
int* arr = new int[n]; // Array dinámico de n elems de tipo int  
for (int i = 0; i < n; i++)  
    arr[i] = i;  
for (int i = 0; i < n; i++)  
    cout << arr[i] << '\n';  
delete [] arr; // La memoria usada por arr queda libre  
arr = nullptr; // Por seguridad
```

Memoria dinámica - errores comunes

1. Olvido de destrucción de un dato dinámico

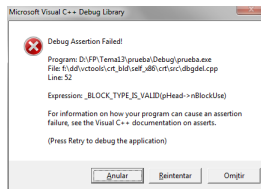
- ▶ C++ no dará indicación del error y el programa parecerá terminar correctamente, pero quedará memoria desperdiciada
- ▶ Veremos cómo hacer que Visual Studio nos informe

```
int main() {  
    int* p = new int(5); // Crea e inic.  
    cout << *p;  
    return 0;  
}
```

2. Destrucción de un dato inexistente

- ▶ En general podría provocar errores de ejecución impredecibles

```
int main() {  
    Fecha* f1 = new Fecha{31,12,1999};  
    Fecha* f2 = f1;  
    delete f1;  
    delete f2; // Error, ¡ya se ha destruido!  
}
```



Memoria dinámica - errores comunes

3. Pérdida de referencia: al menos provocará basura

```
Fecha* f1 = new Fecha{31,12,1999};  
Fecha* f2 = new Fecha{1,1,2000};  
f1 = f2 ; // ¡Referencia perdida!  
delete f1;  
delete f2; // ¡Error, ya se ha destruido!
```

4. Acceso a un dato tras su eliminación

- ▶ Provoca accesos a través de punteros no válidos
- ▶ Podría provocar errores de ejecución impredecibles

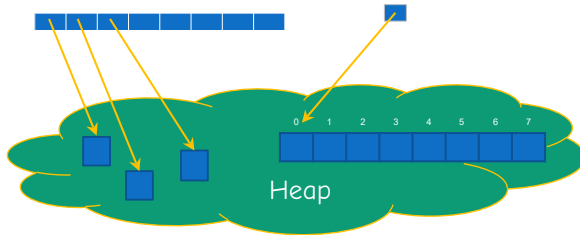
```
Fecha* f1 = new Fecha{31,12,1999};  
Fecha* f2 = f1;  
escribirFecha(*f1);  
delete f1;  
escribirFecha(*f2); // ¡Acceso a puntero no válido!
```

Arrays dinámicos vs. arrays de dinámicos

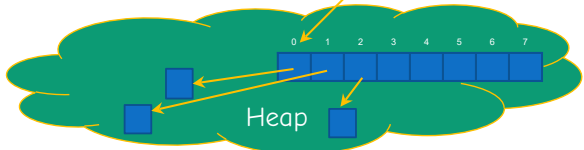
Array de dinámicos



Array dinámico



Array dinámico de datos dinámicos



Más sobre entrada/salida

El método `get()`

- ✦ El operador de inserción (`>>`) lee del flujo (saltando espacios*) y escribe en la variable según su tipo
- ✦ El método `get()` de `istream` lee el siguiente carácter del flujo sea lo que sea (incluyendo espacios). Ejemplo:

```
char c; cin.get(c);
```

La función `getline`

- ✦ La lectura de strings mediante el operador de inserción (`>>`) lee hasta encontrar un espacio*
- ✦ La función `getline` lee del flujo hasta un salto de línea (incluyendo espacios) y guarda el resultado en un string

```
string nombre, apellidos;  
getline(cin, nombre); getline(cin, apellidos);
```


Entrada/salida con ficheros de texto

Ejemplo: copia de un fichero en otro quitando espacios

```
#include <iostream>
#include <fstream>

void quitarEspacios(const string& fichEntrada, const string& fichSalida) {
    ifstream input;
    ofstream output;
    input.open(fichEntrada);
    output.open(fichSalida);
    if (!input.is_open()) cout << "No se encuentra el fichero" << endl;
    else {
        char c;
        input.get(c);
        while (!input.fail()) {
            if (c != ' ') output << c;
            input.get(c);
        }
    }
    input.close(); output.close(); // innecesario al salir de ámbito
}
```

Definición de operadores

- ✦ En C++ se pueden definir funciones de tipo operador que luego pueden llamarse en notación infija

```
istream& operator>>(istream& in, Fecha& f) {  
    char c;  
    in >> f.dia >> c >> f.mes >> c >> f.anyo;  
    return in;  
}  
  
ostream& operator<<(ostream& out, const Fecha& f) {  
    out << f.dia << "/" << f.mes << "/" << f.anyo;  
    return out;  
}  
  
Fecha& operator++(Fecha& f) {  
    f.dia++; // if (...) ...  
}  
  
bool operator<(const Fecha& f1, const Fecha& f2) {  
    return ((f1.anyo < f2.anyo) || ... );  
}
```

```
int main() {  
    Fecha f1, f2;  
    cin >> f1 >> f2;  
    if (f1 < f2) cout << ++f2;  
    else cout << ++f1;  
}
```

Excepciones

- ♦ Las excepciones son señales que se generan para tratar errores (situaciones excepcionales) en ejecución
- ♦ La gestión de excepciones de los lenguajes de prog. es simplemente un mecanismo de flujo control pensado para ello
 - ▶ Por poder, se podría usar para cualquier cosa, aunque no es recomendable
- ♦ Cuando se genera (se lanza o eleva) una excepción se transmite, automáticamente a través de la pila de llamadas a funciones, hasta que es capturada (tratada o interceptada)
- ♦ Las excepciones deben tratarse procurando que la aplicación siga ejecutándose con las menores contrariedades posibles, o terminando la ejecución del programa de forma segura
- ♦ En general, la función que lanza la excepción puede estar en un módulo o biblioteca, y el código que la trata en otro

Excepciones - lanzamiento con throw

- ♦ Sintaxis: **throw** expresión;
- ♦ En C++ cualquier expresión puede ser una excepción
- ♦ Ejemplos:

```
throw exception(); // clase definida en <exception>  
throw 37;  
throw EDiv; // constante definida en un enumerado
```

- ♦ Al ejecutarse **throw** el control de la ejecución del programa se transfiere a la primera cláusula **catch** (siguiendo la pila de llamadas a funciones) que capture la excepción lanzada.
- ♦ Si no se captura la excepción, la ejecución del programa termina de forma incorrecta, devolviendo un error y sin destruir los objetos ni cerrar los recursos.

Excepciones - captura con try/catch

Sintaxis:

```
try {  
    // Código que puede generar excepciones  
    // o que se debe proteger de éstas, evitando su ejecución  
}  
  
// Después del bloque try, la secuencia de cláusulas catch  
catch (Tipo1[& var]) { /* código de tratamiento para Tipo1 */ }  
catch (Tipo2[& var]) { /* código de tratamiento para Tipo2 */ }  
...  
catch (TipoN[& var]) { /* código de tratamiento para TipoN */ }  
catch (...) { /* código de tratamiento else (opcional) */ }  
// "fin-try-catch" siguiente instrucción al try-catch
```

- ✦ Las cláusulas **catch** se prueban secuencialmente hasta que una captura la excepción lanzada, y el control de la ejecución del programa pasa al bloque de código de dicha cláusula.

Excepciones - captura con try/catch

- ♦ La siguiente instrucción al código de tratamiento de una cláusula **catch** es la siguiente instrucción al bloque **try-catch**
- ♦ Si ninguna cláusula **catch** captura la excepción, esta se transmite, abandonando la función (análogo a **return**), a la siguiente función en la pila de llamadas, para seguir con el mismo proceso.
- ♦ En una cláusula **catch** se puede declarar una variable local cuyo ámbito es el bloque del tratamiento **catch** (Tipo& var)
 - ▶ En caso de interceptarse, la excepción quedará capturada en var, que se destruirá automáticamente al finalizar el bloque.
- ♦ En una cláusula **catch** se puede volver a lanzar la misma excepción mediante la instrucción **throw;** // Sin argumento

Excepciones - ejemplo

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

using namespace std;

int main() {
    try {
        cout << "Throwing an integer exception...\n";
        throw 42;
    } catch (int i) {
        cout << " the integer exception was caught, with value: " << i << '\n';
    }

    try {
        cout << "Creating a vector of size 5... \n";
        vector<int> v(5);
        cout << "Accessing the 11th element of the vector...\n";
        cout << v.at(10); // vector::at() throws out_of_range
    } catch (const exception& e) { // caught by reference to base
        cout << " a standard exception was caught, with message '" << e.what() << "'\n";
    }
}
```

Excepciones - ejemplo

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

using namespace std;

int main() {
    try {
        cout << "Throwing an integer exception...\n";
        throw 42;
    } catch (int i) {
        cout << " the integer exception was caught, with value: " << i << '\n';
    }

    try {
        cout << "Creating a vector of size 5... \n";
        vector<int> v(5);
        cout << "Accessing the 11th element of the vector...\n";
        cout << v.at(10); // vector::at() throws out_of_range
    } catch (const exception& e) { // caught by reference to base
        cout << " a standard exception was caught, with message '" << e.what() << "'\n";
    }
}
```

Throwing an integer exception...

the integer exception was caught, with value: 42

Creating a vector of size 5...

Accessing the 11th element of the vector...

a standard exception was caught, with message 'out_of_range'

Excepciones - ejemplo quitarEspacios

Ejemplo: copia de un fichero en otro quitando espacios

```
#include <iostream>
#include <fstream>

void quitarEspacios(const string& fichEntrada, const string& fichSalida) {
    ifstream input(fichEntrada);
    ofstream output(fichSalida);

    if (!input.is_open()) throw Error("No se encuentra el fichero");
    else {
        char c;
        input.get(c);
        while (!input.fail()) {
            if (c != ' ') output << c;
            input.get(c);
        }
    }
}

int main() {
    try {
        quitarEspacios("in.txt", "out.txt");
        std::cout << "Hecho!" << endl;
    } catch (const Error& e) {
        std::cout << e.what() << endl;
    }
}
```

Resumen de diferencias vistas con C#

- ◆ Gestión de la memoria:
 - ▶ Pila vs. heap
 - ▶ No recolección de basura
- ◆ Inicialización de variables
- ◆ Tipos
- ◆ Arrays:
 - ▶ Arrays estáticos y arrays dinámicos
 - ▶ No guardan su longitud (hay que llevarla explícitamente)
 - ▶ Arrays multidimensionales
- ◆ Cadenas de caracteres
- ◆ Paso de parámetros
- ◆ Operadores
- ◆ Excepciones

Otras diferencias con C#

- ✦ En C++ el **switch** solo funciona con tipos enteros o **char**
- ✦ Chequeo/inferencia de contextos constantes
- ✦ Aritmética de punteros
- ✦ Separación entre interfaz e implementación
- ✦ Plantillas
- ✦ Funciones inline
- ✦ Herencia múltiple
- ✦ Etc., etc., etc.