Aportaciones de C++

Antonio Espín Herranz

Aportaciones de C++

- Permite la POO con herencia múltiple.
- Añade nuevas palabras reservadas.
- Una nueva biblioteca de funciones.
- Nuevos operadores.
- Programación con plantillas.
- Chequeo de Excepciones.
- Trabajar con Hilos.
- Permite sobrecargar funciones y operadores.
- Crear referencias.
- Nuevas funciones en E / S.
- Clases contenedoras.
- Operadores **new** y **delete** para trabajar con memoria dinámica. **Peligrosos**.
- Punteros inteligentes.

Funciones

- C++ añade el chequeo de tipo y prototipo de funciones.

- La declaración del prototipo de una función es requerida siempre que se invoque a la función antes de la definición.

Estructura

```
// Declaración del prototipo. Normalmente se ubicará en un fichero .H
void funcion1();

// Función principal.
int main(){
    funcion1(); // Llamada a la función.
}

// Implementación de la función:
void funcion1(){
    ....
}
```

Comentarios

- Los comentarios podemos utilizar los de C.
  / Este comentario
  para varias líneas ****/***

- Este de C++ para una sola línea:

  // Comentario de una sola línea.

Tipos de datos primitivos

- **Tipos predefinidos:**
  Para enteros:
  int, short, long

  Para reales:
  double, float

  Para caracteres:
  char

  Para booleanos:
  bool: con valores true / false.

Tipo	Número de Bits
char	8
short	16
int	32
float	32
double	64

- Si utilizamos el modificador unsigned (unsigned char), se considera un tipo distinto aunque ocupe lo mismo.
- Mediante ***sizeof(tipo o variable)*** podemos obtener lo que ocupa.

Tipos de datos primitivos

- Los tipos de datos enumerados se consideran nuevos tipos.
- No podemos utilizar el tipo int para asignar un tipo enumerado.
- enum **colores** {red, green, blue};
- int miColor = 2; // Error en C++.
  - **colores uno = azul;**

Variables

- <tipo> identificador;
  - Identificador, letras, numeros <= 256, _ y que no empiecen por número.
    - Estas reglas se aplicarán a funciones, nombres de clases, etc.
  - En cuanto a definición de variables de una clase:

    Nombre_de_Clase nombre_objeto;

Constantes

- En C++ se puede asignar un literal de cadena a un char * (puntero a char). Pero no lo podemos modificar.
  - **const char *nombre = “Ana”;**

- Se puede definir como constante el puntero o la variable:
  - char a[] = "abcd";
  - const char *pc1 = a; // Defino como cte la variable.
  - char * const pc2 = a; // Defino como cte el puntero.
  - pc1 = "aaaa"; // puedo modificar el puntero.
  - pc2[0] = 'z'; // puedo modificar el contenido.
  - const char * const pc = a; // NO puedo modificar nada.

- A nivel global se considera una variable calificada con **const** como **static**.

Volatile

- Indica al compilador que esa variable va a ser modificada por factores externos al compilador. Por ejemplo: Una **IRQ** (rutina de interrupción), otro proceso.

- El compilador asume que esa variable puede cambiar aunque el no la cambia.

- La ubicación de dicha variable será en pila, no en registros. El compilador no optimiza dicha variable. **volatile** int a = 1;

Alcance de las variables

- Podemos definir variables dentro de una sentencia, y sólo estarán disponibles dentro de la misma.
  - for (**int** i=0 ; i < 10; i++) // **La i solo existe dentro del for**.

- Dentro de una función o a nivel global.

- Operador de ámbito **::** para diferenciar **entre local** y **global** con el mismo nombre.

Ejemplo

```
int var = 100;

int main(int argc, char* argv[]){
    int var = 200;
    // Con el operador :: accede a la variable global.
    printf("var global: %d, var local: %d\n\n", ::var, var);
}
```

Operadores

- Aritméticos binarios: + - * / %
- Aritméticos unarios: ++ -- (prefijos / postfijos).
- Notación corta: += -= *= /= %= &= |= ^=
- De relación: == > >= < <= !=
- De asignación: =
- Lógicos: && || !
- Sobre bits: & | ^ ~ >> <<
- sizeof(tipo / variable) → Devuelve el número de bytes de un tipo o una variable.
- Operador ternario: **? :** b = (a == 6) ? 3 : c;
- Tener en cuenta la prioridad de los operadores.
- Uso de paréntesis.

Prefijos y sufijos: **Considerar**

```
int a, b;

a = 0; b = 9;
a = ++b;

a = 0; b = 9;
a = b++;
```

Operadores de C++

- **::**
  - Ámbito, acceso a una var. Global o a un miembro de una clase.
- **this**
  - Puntero que hace referencia al propio objeto que ha recibido el mensaje.
- **&**
  - Obtener la referencia de una variable. Para pasar parámetros por referencia.
- **new**
  - Crear objetos de forma dinámica.
- **delete**
  - Destruye un objeto creado dinámicamente.

Operadores de C++

- **.***
  - Operador para acceder al miembro de una clase cuando el miembro es referenciado por un puntero, objeto.*pmiembro.
- **→***
  - Operador para acceder al miembro de una clase cuando ambos son referenciados por punteros, objeto→*pmiembro.
- **typeid**
  - Identificador de tipo. cout << typeid(a).name(); // Nombre del tipo a.
- **???_cast**
  - Conversiones forzadas.

Precedencia de operadores

Precedencia de Operadores y Asociatividad	
Operador	Asociatividad
() [] -> .	izq. a dcha.
! ~ ++ -- - (tipo) * & sizeof	[[[dcha. a izq.]]]
* / %	izq. a dcha.
+ -	izq. a dcha.
<< >>	izq. a dcha.
< <= > >=	izq. a dcha.
== !=	izq. a dcha.
&	izq. a dcha.
^	izq. a dcha.
|	izq. a dcha.
&&	izq. a dcha.
||	izq. a dcha.
?:	[[[dcha. a izq.]]]
= += -= *= /= %= &= ^= |= <<= >>=	[[[dcha. a izq.]]]
,	izq. a dcha.

Sobrecarga de operadores

- Podemos asociar funciones con operaciones para habilitar el uso convencional del operador que define esa operación.
- Utilizamos la palabra reservada **operator**.
- Por ejemplo, si definimos una estructura Punto2D puede ser útil definir un operador + para esta clase y utilizarlo con la notación habitual.

Ejemplo

```
struct punto2d {
	int x, y;
};


punto2d operator +(punto2d a, punto2d b){
	punto2d c;

	c.x = a.x + b.x;
	c.y = a.y + b.y;

	return c;
}


void visualizar(const char *s, const punto2d &a){
	printf("%s = (%d,%d)\n", s, a.x, a.y);
}
```

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

	punto2d a = {1,2}, b = {3,4}, c;

	// llamada al operador +
	c = a + b;

	// Otra forma:
	c = operator +(a,b);

	visualizar("a", a);
	visualizar("b", b);
	visualizar("c", c);

	return 0;
}
```

Parámetros por Omisión

- Podemos definir parámetros por omisión dentro de una función.
- Si especificamos el prototipo de la función, los parámetros por defecto deben especificarse en este, si no en la propia declaración.
- double potencia(double n, int = 2)

Ejemplo

```
void visualizar( int a = 1, float b = 2.5F, double c = 3.456 ){
        printf("parámetro 1 = %d, ", a);
        printf("parámetro 2 = %g, ", b);
        printf("parámetro 3 = %g, \n", c);
}

void main(){
    visualizar();
    visualizar(1);
    etc…
}

// Esta función permite la llamada con 0, 1, 2 o 3 parámetros.
```

Ejemplo II

```
double raiz(double n, int = 2);

void main(){
   // Llamada a la función raíz …
}

// Implementación de la función raiz:
double raiz(double n, int r){
   // código
}
```

Funciones inline

- En C++ se pueden definir funciones en línea.
- El compilador puede reemplazar cualquier llamada a la función.
- Hay que indicarlo mediante la palabra ***inline***.
- Es útil para funciones pequeñas.
- Cualquier función miembro definida dentro de una clase se considera inline.
- inline menor(int a, int b){ return ((x < y)?x:y);}

Sobrecarga de funciones

- Consiste en llamar a dos o mas funciones con el mismo nombre dentro del mismo ámbito, pero con parámetros de distinto tipo o número.

- No se da sobrecarga con el tipo devuelto por una función.

- void visualizar(char *s);
- void visualizar(long n, char s);

  // Se resuelve mediante el tipo y número de los parámetros.

Macros

- Se aconseja utilizar funciones inline en vez de macros. Ya que las funciones chequean el tipo.

- Se utiliza la directiva **#define.**
- OJO con los efectos colaterales.

```
#define MENOR(x, y) ((x) < (y) ? (x) : (y))
// Con la llamada m = MENOR(a--, b--);
// El valor del menor se decrementa dos veces, por la sustitución de la macro.
```

- **Utilizar funciones inline**, que no tienen estos efectos.

Referencias

- Una referencia es un nombre alternativo para un objeto.
- Se puede utilizar para pasar parámetros por referencia y en el valor retornado.
  - Declaración: tipo**&** referencia = objeto;
  - Las operaciones realizadas en la referencia se reflejan en el objeto original.
  - Las referencias siempre hay que inicializarlas en el momento.
  - No son copias de la variable, hace referencia a la misma variable.

Referencias II

- Las referencias no es lo mismo que los punteros.
  - Las referencias es obligatorio inicializarlas.
  - No se puede alterar el objeto que referencia, siempre hace referencia al mismo.
  - Tampoco se puede aplicar la aritmética de los punteros.
  - Al declararlas cada referencia debe llevar su &.
- Cuando se hace una modificación en la referencia se plasma en el objeto que referencia.

Paso de parámetros por Puntero / Referencia

- Podemos utilizar la referencias para pasar parámetros por referencia a una función.
  - 1ª Forma: (con punteros)
    void cambiar(int *a, int *b){ …} // Trabajamos con *a y *b.
    // Llamada: cambiar(&x, &y);
    Con punteros podemos distinguir entre la dirección y el contenido de la variable.

  - 2ª Forma: (con referencias)
    void cambiar(int &a, int &b){ .. } //Trabajamos con a y b.
    // Llamada: cambiar(x, y);
    Con la referencia solo accedemos al dato.

  - Una referencia a una constante proporciona el beneficio de los punteros pero no se puede modificar.
    Void funcion(const int &dato) // No se puede modificar.

Devolver una referencia

- Útil cuando:
  - La función se pueda utilizar a la izquierda y a la derecha (permite modificar el objeto devuelto).
  - Devolver un objeto grande.
  - Encadenar operaciones.

- Estas funciones suelen ser miembros de estructuras.

Ejemplo

```
struct punto {
  // Atributos
  int x; // coordenada x
  int y; // coordenada y

  // Métodos
  int &cx() // devuelve una referencia a x
  {
    return x;
  }

  int &cy() // devuelve una referencia a y
  {
    return y;
  }
};
// Fin de la estructura de datos punto
```

```
int main()
{
    punto origen;

    // Utilizar cx() y cy() como l-values
    origen.cx() = 60;
    origen.cy() = 80;

    // Utilizar cx() y cy() como r-values
    printf("x = %d\ny = %d\n", origen.cx(), origen.cy());
}
```

Espacios de nombres

- Un espacio de nombres es un ámbito.
- Podemos definirnos nuestros propios espacios de nombres.

```
namespace nombre {
    // Declaraciones
}
```

- Se suelen definir a un ámbito global.
- En C++ la biblioteca estándar está representada por el espacio de nombres **std**.
- La podemos utilizar de la forma:

```
using namespace std;
```

Utilizar using namespace std; o no

- El uso de using namespaceno está relacionado de ninguna forma con el rendimiento. Sin embargo considera el siguiente escenario: Estas utilizando dos bibliotecas llamadas Foo y Bar y en un momento dado decides importar los espacios de nombres:
  - using namespace foo;
  - using namespace bar;

- Todo funciona bien, puedes llamar Bla() de Foo y a quux() de Bar y sin problemas.

- Pero un día actualizas a una nueva versión de Foo 2.0 que ofrece una función llamada quux(). El resultado es tienes un conflicto: Tanto Foo 2.0 como Bar importan quux() en el espacio de nombres global. Corregir el error puede requerir bastante esfuerzo sobre todo si los parámetros de ambas funciones son iguales.

- Si en vez de importar los espacios de nombres has utilizado foo::Bla() y bar::quux(), introduccir foo::quux() no requiere esfuerzos adicionales.

Los flujos en C++

- Al igual que en C tenemos tres flujos en C++ también:
  - **cin**: Entrada estándar (teclado).
  - **cout**: Salida estándar (consola).
  - **cerr**: Salida estándar (consola).

- Todos están incluidos en el fichero <iostream> de la biblioteca estándar de C++.

Leer de teclado

```
#include <iostream>
using namespace std;

void main(int argc, char *argv[]){

    int var;

    cout << "Introduzca el valor de la variable: ";
    cin >> var;
    cout << "El valor de variable es:" << var << endl;
}
```

Representa un ámbito.

Definido para la biblioteca estándar de C++, si no lo ponemos tendríamos que incluir: #include <iostream.h>

Si no utilizáramos **using** tendríamos que poner **std::cout**

Formatear la salida

- Formatos similares a printf:
- Incluidos en: #include <iomanip>
- Tenemos dos formas de aplicar formatos:
  - **Indicadores**:
    - Están definidos dentro de la clase ios de la biblioteca iostream, para acceder a ellos necesitamos el operador **::**
    - nombreClase::indicador
    - Por ejemplo: con el indicador ios::showpos coloca un signo + en los números positivos.
    - El método para poder establecer un indicador es: setf del objeto cout.
    - Uso:

```
cout.setf (ios::showpos);
cout << 1;
// Fuerza la salida +1.
```

Formatear la salida

- Indicadores para escribir la salida de un número en dec, hex y oct:
  - cout.seft(ios::oct, ios::basefield);
  - cout.seft(ios::dec, ios::basefield);
  - cout.seft(ios::hex, ios::basefield);

- La otra forma es mediante **manipuladores**, más cómodo permiten insertarse directamente en un flujo mediante el operador <<.

Formatear la salida

- Para octal: cout << oct << 23;

- Varios:
  - cout << oct << 23 << dec << 23 << hex << y;
  - También podemos leer un número en hex:
    - cin >> **hex** >> var2;
    - cout << "El otro numero es: " << var2;

Formatear la salida: Ancho y Relleno

- Mediante el indicador:

```
cout.width(5);
cout.fill('#');
cout << 23 << endl;
```

- Mediante el manipulador:

```
cout << setw(5)  << setfill('#') << 23 << endl;
```

Formatear la salida: Alineación

- Indicadores: (left, right, internal).
  - cout.setf(ios:right, ios:adjustfield);
  - cout.setf(ios:left, ios:adjustfield);

- Manipuladores:
  - Insertándolos directamente en el flujo:

    cout << setw(5) << setfill('#') << -23 << endl;
    cout << setw(5) << **left** << -23 << endl;
    cout << setw(5) << **internal** << -23 << endl;

  - Salida: por defecto alinea a la derecha:

    ##-23
    -23##
    -##23

Formatear la salida: Precisión

- Para números en punto flotante:
  - fijo, científico y general.

- **Indicadores**: mediante setf

```
cout << 1234.56789 << endl;
cout.setf(ios::scientific, ios::floatfield);
cout << 1234.56789 << endl;
cout.setf(ios::fixed, ios::floatfield);
cout << 1234.56789 << endl;
```

- **Manipuladores**:

Se añaden directamente al flujo:

```
cout << 1234.56789 << endl
<< scientific << 1234.56789 << endl
<< fixed << 1234.56789 << endl;
```

Salida:

1234.57

1.234568e+003

1234.567890

Además:

cout.precision(4);

O:

cout << setprecision(4);

endl: Salta de línea.

ends: inserta carácter nulo en la cadena.

Excepciones

- Mediante las Excepciones C++ notifica errores que permite que sean capturados por el programador para su tratamiento.

- Por ejemplo el contenedor vector cuando accede a una posición no válida lanza una excepción “**out_of_range**”.

- La Excepciones están definidas en el fichero de cabecera <**stdexcept**>.

Excepciones

- Se añaden las palabras reservadas: try, catch, throw.
- Las palabras **try / catch** las vamos a utilizar para capturar y tratar una excepción.
- En el caso de que queramos provocarla podemos hacer uso de **throw**.

Capturar Excepciones

```
int main() {
    try {
        // Código que puede lanzar Excepciones.

    } catch (excepcion1){
        // Tratamiento para la excepcion 1.

    } catch (excepcion2){
        // Tratamiento para la excepcion 2.
    }
}
```

Concepto de Memoria dinámica

- La reserva de memoria dinámica se lleva a cabo mediante dos operadores: **new** y **delete**.

- En C la reserva / liberación se realizaba mediante malloc, calloc, realloc, free.

- En C++ cuando vamos a reservar o liberar memoria tenemos que tener en cuenta si estamos tratando con un array o no.

Operadores new y delete

- Los podemos utilizar para reservar / liberar un objeto o una matriz.

- void *operator new(size_t n, const std:nothrow_t&) throw();
- void *operator new[](size_t n, const std:nothrow_t&) throw();

- void *operator new(size_t n) throw (bad_alloc);
- void *operator new[](size_t n) throw (bad_alloc);

- void *operator delete(void *);
- void *operator delete[](void *);

- La memoria se asigna en el área de almacenamiento libre.

- En el caso de los objetos el tamaño viene especificado por su tipo.

- En el caso de las matrices el tamaño de un elemento viene especificado por su tipo, pero hay que indicar explícitamente el tamaño de la matriz.

- Devuelve un puntero al tipo indicado en la reserva.

Reserva de memoria en tiempo de ejecución

- Tenemos dos tipos de notaciones:
  - int *p1 = new int; // Sin paréntesis.
  - float *pf = new (float); // Con paréntesis.

  - Para estructuras:

```
struct complejo {
    float real, img;
}

complejo *p = new complejo;
```

  - Para matrices:

```
int *a = new int[n];
```

  - También lo vamos a utilizar con objetos.
    puntero_a_objeto = new clase_objeto(parámetros);

Memoria insuficiente

- Cuando invocamos al operador **new** tenemos que comprobar si hay memoria suficiente.
- Necesario: **#include <new>**

```
// Si no hay mem. Devuelve 0.
int *p = 0;
p = new (nothrow) int[n];
if (p==0){
    cout << “sin mem.”;
    return -1;
}
```

```
// Si no hay mem. Devuelve 0.
int *p = 0;
try {
    p = new int[n];
} catch (bad_alloc e){
    cout << “sin mem.”;
    return -1;
}
```

Liberación de memoria en tiempo de ejecución

- Mediante el operador **delete** podemos liberar la memoria reservada por **new**.
- No pone a 0 el puntero que lo referencia.
  - new – delete
  - new [] – delete []
- Si se aplica el operador delete a un puntero con valor 0, no se produce ningún error.
- La liberación de matrices se debe realizar de forma inversa a como se reservó.
  - **Lo mas interno primero y luego por último el puntero exterior.**

Formas de asignar memoria en C++

- **Estática**:
  - Para objetos que van a “vivir” durante la ejecución del todo el programa.
    - Variables globales y estáticas.

- **Automática**:
  - Cuando se asigna memoria a parámetros de funciones y para variables locales. Se ubican en la pila.

- **Libre**:
  - O asignación dinámica, cuando se hace uso del operador new.

  - Cuando se asigna dinámicamente memoria a un objeto y no se libera antes de que deje de existir la variable que lo referencia se puede producir una laguna de memoria.

Punteros inteligentes

- Las funciones de C: malloc, calloc, realloc y los operadores de C++: new y delete son propensas a errores.

- La liberación incorrecta de memoria en un programa de gran envergadura puede ocasionar errores en tiempo de ejecución difíciles de encontrar.

- A partir de C++ 11, se añadieron los punteros inteligentes. Simulan el comportamiento de un puntero corriente pero añaden nuevas características adicionales como un recolector de basura y comprobador de límites.

- Estos punteros intentan prevenir las pérdidas de memoria sin penalizar en la eficiencia. Algunos trabajan llevando la cuenta de referencias, otros mediante asignación de un objeto a un único puntero.

Enlaces

- Referencia a CPP
  - http://www.cplusplus.com/reference/

- Ejecución en la Web:
  - http://cpp.sh/