APÉNDICE D

PALABRAS RESERVADAS ISO/ANSI C++

asm

Se utiliza para incluir directamente código ensamblador en su programa C++. El uso correcto de `asm` depende de la implementación.

Sintaxis

```
asm <instrucción en ensamblador>;
asm <instrucción -1>; asm <instruccion-2>;...
asm {
    secuencia de instrucciones en ensamblador
}
```

Ejemplos

```
asm push ds;
asm {
   pop ax
   inc ax
   push ax
}
```

auto

Es un especificador de almacenamiento de clases para variables temporales. Estas variables se crean al introducirse en un bloque de sentencias y se destruyen al salir de él. Las variables locales de una función tienen clase de almacenamiento `auto` por defecto (*omisión*).

bool

Tipo lógico (*boolean*) que toma valores verdadero (*true*) o falso (*false*) que puede contener los literales **true** y **false**. Sustituye al sistema tradicional de C que considera el valor cero como falso y distinto de cero como verdadero.

break

`break` permite salir del bucle `do`, `while` o `for` más interno. También se puede utilizar para salir de una sentencia `switch`.

Un ejemplo de `break` en un bucle es:

```
while (Z < 10) {
   cin >> Z;
   if (Z < 0) break;            // salir si Z es negativo
   cout << "Hola mundo, yo sigo";
}
```

case

Sirve para etiquetar los diferentes casos de la sentencia `switch`.

Sintaxis

```
case <valor> : <sentencia>;
                    ...
                   break;
```

Ejemplo

```
switch(numero)
{
   case 2 + 5: cout << "Es 7";
                  break;
   case 9    : cout << "Es 9";
                  break;
   default   : cout << "N: 7 ni 9";
}
```

catch

Se utiliza como mecanismo de tratamiento de excepciones. Su propósito es interceptar una excepción generada por `throw`. Dado su carácter experimental, no está implementada en algunos compiladores.

Sintaxis

```
catch (<excepción> {
      // código que manipula excepciones
   }
```

cdecl

La palabra reservada `cdecl` no forma parte del estándar C++. Fuerza al compilador a compilar una función de modo que su paso de parámetros siga la notación estándar de C. Se suele encontrar en compiladores que permiten la especificación de notaciones de Pascal. Se utiliza cuando se está compilando un archivo completo utilizando la opción *Pascal* y se desea que una función específica sea compatible con C/C++.

```
extern cdect printf();
void ponernums (int i, int j, int k);
cdecl main()
{
   ponernums (1, 5, 9);
}

void ponernums (int i, int j, int k)
{
   printf ("y las respuestas son: %d, %d, %d ´n", i, j, k);
}
```

char

Tipo de dato incorporado en C++. Se puede utilizar `char` para declarar variables carácter (tienen un rango de 0 a 255 en código ASCII).

class

Palabra que representa la definición de una clase. Una clase contiene variables miembros datos y funciones miembro que operan sobre esas variables; asimismo, una clase tiene un conjunto de especificadores que controlan el acceso a los miembros de la clase (`private, public` y `protected`)

Sintaxis

```
class nombre_clase: (especificadores de acceso) clase_base
{
   // miembros privados por defecto
protected:
   // miembros privados se pueden heredar
public:
   // miembros públicos
} [lista de objetos];
nombre_clase clase_1, clase_2,...;
```

Si no se especifica ningún especificador de acceso, el acceso por omisión (por defecto) es `private`. La *`lista_de_objetos`* es opcional, de modo que si no se especifica, la declaración de una clase no crea objetos de esa clase y se necesita definir el objeto como si creara una determinada variable.

const

Es un calificador de tipo, utilizado para indicar que la variable que le sigue no puede ser modificada por el programa. Esto significa que no se puede asignar un valor a esa variable, incrementarla o decrementarla; sin embargo, se puede inicializar a un valor dado, cuando se declara.

Sintaxis

```
const <nombre> = <valor>;
```

Ejemplo

```
const int edad = 46;
const in *p = 3025;
```

const_cast

El operador `const_cast<T>(e)` permite acceder a un objeto con el atributo `const` o `volatile` adjuntado. El tipo `T` debe ser del mismo tipo que el operando `e` excepto para los modificadores `const` y `volatile` y el resultado devuelto por `const_cast()` es el mismo que `e` pero de tipo `T`.

```
void func (char* cp)
{}

func(const_cast <char*>(vc));
```

continue

Se utiliza `continue` en la ejecución de un bucle. Es equivalente a ejecutar una sentencia `goto` al final del bucle. Esta sentencia afecta al bucle más interno en el que aparece.

Sintaxis

```
continue;
```

Ejemplo

```
for (j = 0, j  < 100; j++)
{
   if (j == 10) continue;
   suma += j;
}
```

default

Se utiliza en la sentencia `switch` para marcar el código que se ejecutará cuando ninguna de las etiquetas de `case` se corresponde con la expresión `switch`.

Sintaxis

```
default:<sentencia>;...
```

delete

El operador de asignación dinámica `delete` se utiliza para liberar la memoria apuntada por su argumento (asignada utilizando `new`)

Sintaxis

```
delete <puntero>;
delete <[elementos]> <expresión de conversión de tipo>
```

Ejemplo

```
delete objeto_prueba;
delete[100] indicadores; // borra cada uno de los 100 elementos
                         // indicadores
```

do

Se utiliza un `while` para construir bucles iterativos en los cuales las instrucciones del cuerpo del bucle se ejecutan hasta que la condición se evalúa a 0 (falso).

Sintaxis

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

Como la condición se evalúa al final del bucle, las sentencias se ejecutan al menos una vez.

Ejemplo

```
do {
   suma += j;
   j++;
} while (j <= 100);
```

double

Especificador de tipo de dato `double` que declara variables y arrays de coma flotante de doble precisión.

Sintaxis

```
double <nombre_variable>;
```

dynamic_cast

Este operador se puede utilizar para moldear (convertir explícitamente) a un puntero o a un tipo referencia

Sintaxis

```
dynamic_cast <T*>(p)
void vp = dynamic_cast <void*> (bp);
```

else

Se utiliza con `if` para controlar el flujo de ejecución con sentencias `if` cuya sintaxis es:

```
if (expresión) sentencia_1; else sentencia_2;
```

donde *sentencia_1* se ejecuta si *expresión* es distinta de 0 (verdadero) y *sentencia_2* se ejecuta si *expresión* es igual a 0 (falso).

enum

Tipo de datos de valor entero que puede tomar sus valores de una lista de constantes enumerados.

explicit

Declara un constructor explícito.

Ejemplo

```
class Punto
{
private:
   double x, y, z;
public:
   Punto() : x(0.0), y(0.0), z(0.0) {}
   explicit Punto ( double d)   // constructor explícito
      : x(d), y(d), z(d) {}
   // ...
};
```

extern

Especificador de clase de almacenamiento utilizado para indicar al compilador que una variable se declara en otra parte del programa. Si una declaración de variable comienza con la palabra reservada `extern` no es una definición, es decir, especifica el tipo y nombre de la variable e implica que una definición de esta variable se encuentra en otra parte del programa. Sin esta palabra reservada, es una definición; cada definición de variable es al mismo tiempo declaración de variable.

```
extern int n;     // declaración de n (no definición)
```

Este especificador se suele utilizar (en archivos compilados) separadamente, que comparten los mismos datos globales y se enlazan juntos. Así por ejemplo, si `test` se declara en otro archivo, como entero, la declaración siguiente se utilizará en otros archivos:

```
extern int test;
```

No utilice `extern` con frecuencia. Existen otros métodos para pasar información entre funciones. En general, utilice `extern` sólo para referirse a variables globales que se definen en otra parte. Debido a que esta característica crea espacio permanente para las variables, la mayoría de las variables locales no son `extern`.

extern "C"

El prefijo `extern "C"` permite que un programa C++ pueda llamar expresamente a una función C. Si se declara una función externa, un compilador C++ supondrá que es una función C y no producirá errores de compilación.

La solución es especificar el enlace como `"C"` cuando se declara la función:

```
// función C
extern "C" void Mensaje(void)
```

Si existen diferentes funciones se pueden declarar en una sola declaración entre llaves:

```
extern "C" {
   void Mensaje(void);
   void Listo(void);
}
```

Estas definiciones se agruparán en un archivo de cabecera, pero no se puede declarar en otra parte dentro del archivo de cabecera con enlace `"C"`.

```
extern "C" {
   // cabeceras de bibliotecas C
   #include "dibujar.h"
   #include "impresora.h"
}
```

false

Literal *boolean* de valor cero.

Ejemplo

```
bool b1;
bool b2 = false;
bool b3 (true);
int i1 = true
bool function()
{
   // ...
}
```

far

El modificador de tipos `far` no es parte del estándar C++. Se emplea por compiladores diseñados para utilizar en la familia 8086 de procesadores, que fuerzan a una variable puntero a utilizar un direccionamiento de 32 bits en lugar de 16 bits.

float

`float` es un especificador de tipos de datos utilizados para declarar variables de coma flotante.

for

El bucle `for` permite inicializar e incrementar variables contadores.

Sintaxis

```
for (inicialización; condición; incremento) {
   <sentencias>
}
```

Si el bloque de sentencias sólo contiene una, no son necesarias las llaves (`{,}`). Si la condición es falsa, al comenzar el bucle éste no se ejecutará ni una sola vez.

Se puede omitir cualquiera de las tres expresiones de `for`, pero deben dejarse los puntos y comas (;). Si se omite *condición*, ésta se considerará como verdadera.

El bucle infinito es `for (;;)` y equivale a `while(1)`.

Se puede utilizar el operador coma (,) para poner varias expresiones en el interior de las diferentes partes de `for`.

Ejemplo

```
for(i = 0, j = n-1; i < n; i++, j--)
   a[i] = a[j];
```

friend

La palabra reservada `friend` se utiliza para garantizar el acceso a una parte privada (`private`) de una clase, por una función que no es miembro de ella y a las partes protegidas (`protected`) de una clase de la cual no se derivan.

Las funciones amigas pueden ser *amigas* de más de una clase.

Ejemplo

```
class Persona {
   // ...
public:
   // ...
   friend void funcamiga(int x, float y);
};
```

goto

Produce un salto en la ejecución del programa a una etiqueta de la función actual. *Su uso no está recomendado* más que para situaciones excepcionales, tales como la salida directa del bucle más interior perteneciente a una serie de bucles anidados.

Sintaxis

```
goto <etiqueta>;
...
etiqueta:
```

huge

El modificador de tipos `huge` no forma parte del C++ estándar. Se utiliza en compiladores construidos basados en la familia de microprocesadores 8086. Borland/Turbo C++ normalmente limitan el tamaño de todos los datos estáticos a 64 k; el modelo de memoria *huge* desborda ese límite, permitiendo que los datos ocupen más de 64 k.

if

La palabra reservada `if` se utiliza para ejecutar código sólo bajo ciertas condiciones. Se puede utilizar `if` sola o con `else`. Si las sentencias sólo son una, no se necesitan las llaves.

Sintaxis

```
if (condicion)
   <sentencia;...>
else
   <sentencia_esp;...>
```

```
if (condición) {
        <sentencias1>
   }
else {
   sentencias2>
}
```

Si la condición (expresión) se evalúa a cualquier valor distinto de 0, entonces se ejecutarán *sentencias_1* y si no, se ejecutarán *sentencias_2*.

Ejemplo

```
if(a==x)
 aux = 3;
 aux = 5;
```

```
if(x > 1)
   if(y == 2)
        z == 5;
```

```
if(x > 1) {
   if(y == 2)
        z == 5;
 } else
     z = 10;
```

inline

El especificador `inline` instruye al compilador para sustituir las llamadas a funciones con el código del cuerpo de la función. Esta sustitución se denomina *expansión en línea*. El uso de las funciones en línea incrementa el tamaño del programa objeto, pero puede aumentar la velocidad de ejecución, eliminando las operaciones auxiliares implicadas en llamadas a funciones. Las funciones `inline` son similares a las macros, aunque una función `inline` es mucho más segura.

Sintaxis

1. Función en línea autónoma:

```
inline <tipo> <nombre_función> (<arg1>,...) {definición función;}
```

2. Función en línea definida de una clase:

```
<tipo> <nombre_función> {definición función;}
```

Existen dos métodos de utilizar funciones en línea:

1. Declarar una función independiente precediendo a la declaración con la palabra reservada `inline`. Se deben declarar y definir antes de que se haga cualquier llamada a función.
2. Declarar una función miembro de una clase como implícitamente *en línea*.

Ejemplo

```
inline double Cuenta::Balance()
{
   return positivo;
}
inline int triple(int Z) {return Z * 3;}
```

int

Especificador de tipo para variables y arrays de enteros. Los cualificadores `short` y `long` se pueden utilizar para declarar un entero del tamaño deseado.

Sintaxis

```
int <nombre_variable>;
```

Ejemplo

```
int j, x[100];
```

interrupt

Palabra reservada que no se encuentra en C++ estándar. Esta palabra significa que una rutina (función) de interrupción del sistema se puede utilizar como administrador de interrupciones.

long

Especificador de tipo de datos para declarar variables enteros, que ocupa dos veces más bytes que los enteros de tipo `short`.

mutable

Permite que un miembro de un objeto anule una constante (**`const`**); es decir, permite que los miembros dato de variables de clase que han sido declarados constantes permanezcan modificables.

```
class persona {
public
persona(const char* pnombre, int pag, unsigned long noss);
   void dinam() {++edad;}
   ...
private:
   const char * nombre;
   mutable int edad;   // modificable siempre
   ...
};
...
const persona prueba ("Carchelejo", 45, 1110111);
...
prueba.diam();            // correcto prueba.edad es mutable
```

namespace

Define un ámbito (rango).

```
namespace []Acarc {
   class mus {...};
   class poker {...};
   ...
}
```

near

El modificador `near` no está definido en C++ estándar. Se utiliza por compiladores basados en la familia de microprocesadores 8086, que permiten utilizar direccionamientos de 16 bits en lugar de 32 bits.

new

El operador `new` asigna memoria dinámica y devuelve un puntero del tipo apropiado al mismo.

Sintaxis

```
var_p = new tipo;
```

`var_p` es una variable puntero que recibirá la descripción de la memoria asignada y *`tipo`* es el tipo de dato que la memoria contiene. El operador `new` asigna automáticamente memoria para contener un elemento de datos del tipo especificado. Si la asignación solicitada falla, `new` devuelve un puntero nulo

Ejemplo

```
double * q;       // asigna memoria suficiente para contener
q = new double;   // un double
```

```
var_p = new tipo(inicializador);
```

Se inicializa la memoria asignada, especificando un inicializador (valor asignado).

```
var_p = new tipo[tamaño];
```

Asigna un array de una dimensión.

```
var_p = new tipo[tamaño1][tamaño2]...
```

Asigna arrays multidimensionales.

operator

En C++ se pueden sobrecargar las funciones y los operadores. La palabra reservada `operator` se utiliza para crear funciones de operadores sobrecargados. Se pueden sobrecargar los operadores con relación a una clase.

Sintaxis

```
nombre_clase::operator opr(lista_parámetros)
```

La lista de parámetros contiene un parámetro cuando se sobrecarga un operador unitario, y dos parámetros cuando se sobrecarga un operador binario (en este caso, el operando de la izquierda se para en el primer parámetro, y el operando de la derecha se para en el segundo parámetro). Por ejemplo, una clase `string` puede definir el operador `==` como sigue:

```
class string
{
public:
   // ...
      int operator == (const string & s) const;
   // ...
};
```

pascal

Esta palabra reservada no está definida en C++ estándar. El modificador `pascal` es específico a Turbo/Borland C++; está concebido para funciones (y punteros a funciones) que utilizan la secuencia de paso de parámetros Pascal. Las funciones declaradas de tipo `pascal` pueden ser llamadas desde rutinas C, siempre que la rutina C vea que esa función es de tipo `pascal`.

private

El especificador de acceso `private` se utiliza para declarar elementos privados de una clase; estos miembros no son accesibles a ninguna función distinta de las funciones miembro de esa clase.

Sintaxis

```
class nombre {
   // ...
private:
   // ... miembros privados
};
```

Los miembros de una clase son privados por defecto.

```
class consumidor {
   char nombre[30],                // privado por omisión
        calle[40],
        ciudad[20],
        provincia[30];
```

```
public:
   void ver_datos(void);
   void leer_datos(void);
private:
   int edad, salario;          // también son privados
};
```

En declaración de clases derivadas también se utiliza `private class` *`tipo_clase_derivada:`*`public|private|protected...` *`tipo_clase_base{...}`*

En este caso `private` provoca que todos los miembros públicos y protegidos de la clase base se vuelvan miembros privados de la clase derivada, y todos los miembros de la clase base privados permanecen en ella.

protected

El especificador de acceso `protected` marca el comienzo de los miembros de la clase a los que sólo se puede acceder por los propios miembros de la clase y por las funciones miembro de todas las clases derivadas.

Sintaxis

```
class nombre {
// ...
protected:
   // miembros protegidos (sólo disponibles en clases derivadas)
};
```

Ejemplo

```
class datos_act {
   char nombre[40]
   ...
protected:
   long num_ss;
   ...
};    // num_ss es protegido

class datos_nuevos:protected datos_act {
...
};    // datos_nuevos puede acceder a num_ss
```

public

El especificador de acceso `public` especifica aquellos miembros que son accesibles públicamente, cualquier función puede acceder a estos miembros.

Sintaxis

```
class <nombre_clase> {
...
public:
//miembros públicos, disponibles a todas las funciones
//interiores o exteriores a la clase
};
```

Cuando se incluye el especificador `public` en una clase derivada, todos los miembros públicos de la clase base se convierten en miembros públicos de la clase derivada, y todos los miembros protegidos de la clase base se convierten en miembros protegidos de la clase derivada. En todos los casos los miembros privados de la clase base permanecen privados, es decir, no se heredan.

register

`register` es un especificador de almacenamiento para tipos de datos enteros, utilizados para informar al compilador de que el acceso a los datos debe ser tan rápido como sea posible. El compilador almacenará los datos enteros en un registro de la CPU, en lugar de situarlos en memoria.

Sintaxis

```
register <tipo> <nombre_variable>
```

Ejemplo

```
register int j;
```

reinterpret_cast

El operador `reinterpret_cast<T>(e)` permite a un programador ejecutar conversiones explícitas de tipos que son generalmente inseguros y dependientes de la implementación.

```
Derivadax dp1 = reinterpret_cast <Derivada*> (ip);
```

return

La sentencia `return` se utiliza para detener la ejecución de la función actual y devolver el control al llamador. Si la función devuelve un valor utiliza una sentencia

```
return expresion;
```

para devolver el valor representado por la `expresión`.

Ejemplo

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

short

`short` es un calificador de tamaño para variables enteras con y sin signo; al menos ocupa dos bytes, `short` significa en realidad `signed short int`.

signed

El modificador de tipo `signed` se utiliza para indicar que los datos almacenados en un tipo entero (`int` o `char` tienen signo). Los valores `int, long` y `short`, por defecto, tienen signo.

Ejemplo

```
signed char;        // puede tomar valores de -127 a +127
unsigned char;      // puede tomar valores de 0 a 255
```

sizeof

El operador `sizeof` determina el número de bytes que se utilizan para almacenar una variable particular o tipo de datos. Así, cuando se aplica a una variable, `sizeof` devuelve el tamaño del objeto referenciado, y cuando se aplica a una clase, `sizeof` devuelve el tamaño total de un objeto de esa clase.

```
int j;
longitud = sizeof(int);         // valor 2
cout << sizeof j;               // visualiza 2
```

static

`static` es un modificador de tipo de datos que instruye al compilador para crear almacenamiento permanente para la variable local que le precede. Si se declara una variable de tipo `static`, tendrá almacenamiento permanente y retiene su valor a lo largo de toda la vida del programa (y, por consiguiente, entre llamadas a funciones). `static` se utiliza también para ocultar datos y funciones de otros módulos o archivos.

Sintaxis

```
static <tipo> <nombre_variable>
```

Ejemplo

```
static int indice = 0;
```

static_cast

La sintaxis del operador `static_cast <>()` es:

```
static_cast <T>(e)
```

donde `T` puede ser o bien un tipo definido por el usuario o integral, puntero, referencia o `enum`. Sólo realiza verificaciones de tipo estático.

struct

La palabra reservada `struct` se utiliza para representar el tipo de datos estructura que reagrupa variables y/o funciones. En C++ una estructura tiene sintaxis idéntica a una clase, con la única diferencia de que en el acceso por defecto los miembros de una estructura tienen acceso *público*; mientras que el acceso por defecto de una clase es *privado*.

Sintaxis

```
struct nombre_estructura {
   // miembros públicos por defecto
especificador_de_acceso_1:
   tipo elemento1;
   tipo elemento2;
...
especificador_de_acceso_2:
   tipo elemento3;
   tipo elemento4;
...
};
```

```
nombre_estructura estruct_1, estruct_2;

struct empleado {
   int id;
   char nombre[40];
   void leer_infor(int i, char *n);
}
empleado e;      // declara objeto empleado e
```

switch

La sentencia `switch` se utiliza para realizar una bifurcación múltiple, dependiendo del valor de una expresión.

Sintaxis

```
swtich (expresion)
{
   case <valor> : sentencia;...
   ...
   default : sentencia;...
};
```

Se debe utilizar `break` para separar el código de una etiqueta de `case` de otra. Una etiqueta `default` marca el código que se ejecutará si ninguna de las etiquetas de `case` se corresponde con el valor de la expresión.

Ejemplo

```
switch(x) {
   case ´A´:
      cout << "caso A\n";
      break;
   case ´B´:
   case ´C´:
      cout << "caso C\n";
      break;
   default:
      cout << "fin\n";
      break;
};
```

template

La palabra reservada `template` se utiliza para crear funciones y clases genéricas. En aquellos compiladores en que está implementada `template`, una plantilla definirá una familia de clases o funciones. Por ejemplo, una plantilla de clases para una clase `Pila` permitirá crear pilas de diferentes tipos de datos, tales como `int, float` o `char*`. En esencia, para crear una plantilla de funciones o clases, se definirá una clase o función con un *parámetro*. Por esta razón, las plantillas se conocen también como *tipos parametrizado* o *tipos genéricos*.

Una funci*ón plantilla* adopta el formato:

```
template <class tipo de dato>
tipo nombre_función(lista de parámetros)
{
   // cuerpo de la función
}
```

El siguiente programa crea una función genérica que intercambia los valores de dos variables con independencia de sus tipos.

```
#include <iostream.h>

template <class Z>          // plantilla
void intercambio (Z &a, Z &b)
{
   Z aux;

   aux = a;
   a   = b;
   b   = aux;
}

main()
{
   int i = 50, j = 30;
   float x = 34.25, y = 16.45;

   intercambio(i, j);       // se intercambian enteros
   intercambio(x, y);       // se intercambian reales

   cout << "valores de i, j actuales" << i << "" << J << endl;
   cout << "valores de x, y actuales" << x << "" << y << endl;

   return 0;
}
```

Una declaración de clase genérica o clase plantilla es:

```
template <class tipo_de_dato>
class nombre_clase {
   // ...
};
```

Una instancia de una clase genérica se instanciará con:

```
nombre_clase <tipo> objeto;
```

El siguiente es un ejemplo de una clase genérica `Pila`*:*

```
template <class Tipo>
class Pila {
public:
   Pila();
   Boolean meter(const Tipo);    // poner un elemento en la pila
   Boolean sacar(Tipo&);         // quitar un elemento de la pila
private:
   tipo Elementos[MaxElementos]; // elementos de la pila
   int la_cima;                  // cima de la pila
};
```

Instancias del tipo `Pila` son:

```
Pila <int> pila_ent;                  // pila de enteros
Pila <char> pila_car;                 // pila de caracteres
```

La sección de implementación de la `pila` se especifica de un modo similar utilizando la construcción `template`. Ésta se utiliza para especificar cuál es el nombre que actúa como un parámetro tipo antes de especificar la declaración de cada función miembro.

```
template <class Tipo>
Pila <Tipo> :: Pila()
{
   la_cima = -1;
}

template <class Tipo>
Boolean Pila <Tipo> :: meter(const Tipo item)
{
   if(la_cima <MaxElementos-1) {
      Elementos[++ la_cima]-item;
      return TRUE;
   } else {
      return FALSE;
   }
}

template <class Tipo>
Boolean Pila <Tipo>:: quitar(Tipo& item)
{
   if(la_cima < 0) {
      return FALSE;
   } else {
      item = Elementos[la_cima --];
      return TRUE;
   }
}
```

Una clase parametrizada puede tener muchos tipos como parámetros. Por ejemplo, una clase `Pila` puede tener dos tipos parametrizados utilizando

```
template <class Tipo1, class Tipo2> class Pila {
   ...
   ...
   }
```

this

`this` es un puntero al objeto actual, que significa «la dirección del objeto para el cual esta función fue llamada».

Sintaxis

```
this
```

Ejemplo

```
return this;
return *this;
```

Si se desea devolver el propio objeto, en lugar de un puntero al mismo, se debe utilizar `*this`.

throw

La función `throw` se utiliza para llamar al mecanismo de tratamiento de excepciones.

```
try
{
  thow MiExcepción ();
}
```

try

Indica el comienzo de un bloque de manejadores de excepciones. Un bloque `try` tiene el formato:

```
try
   sentencia compuesta
   lista de manejadores

void test()
{
   try {
      ...
      throw i;
   }
   catch(int n)
   {
   if (i > 0)       // maneja valores positivos
      ...
      return;
   }
   else {              // maneja parcialmente i <= 0
      ...
      throw;           // rethrow
   }
}
```

true

Valor verdadero (cierto). Uno de los dos valores enumerados del tipo `bool`. El literal `true(1)` se representa por un valor distinto de cero (normalmente 1).

```
enum Boolean { FALSE, TRUE };

bool boolean;
int i1 = true;
bool b3 (true);
```

typedef

`typedef` se utiliza para dar un nuevo nombre a un tipo de dato existente. Esta característica mejora la legibilidad de un programa.

Sintaxis

```
typedef tipo_existente nuevo_nombre;
typedef float real;
typedef struct {
   float x, y;
{ punto;
```

typename

Una declaración **class** se puede anidar dentro de otra declaración **template class**. Si, sin embargo, se utiliza un nombre dentro de una declaración **template class** que no se declare explícitamente como un tipo o clase, se supone que el nombre no nombra un tipo a menos que el nombre esté modificado por la palabra reservada **typename**.

```
template <class T>
class A
{
   private:
      T datos;
   public:
      A (): datos () {}
      class X     // clase local X
      {};
      Xx;               // correcto, T::X es un nombre de tipo
      Yy;               // error: Y no es un nombre de tipo
      T::Zz             // error: T::Z no es un nombre de tipo
      typename T::I i;  // correcto, T::J es un nombre de tipo
      typename T::J * ip; // correcto, puntero a T::I
};
```

union

Tipo de dato que se utiliza para asignar almacenamiento de diferentes elementos de datos en la misma posición. La declaración de `union` es la misma que la de `struct`, con la diferencia de que en una unión todos los elementos de datos de la declaración comparten la misma posición de almacenamiento.

Sintaxis

```
union <nombre_union>
{
   <tipo> <nombre_miembro>
   ...
}
```

C++ dispone también de una unión anónima, cuyo formato es:

```
union {lista_de_miembros};
```

unsigned

El calificador de tipos `unsigned` se utiliza para tipos de datos enteros (`char, int, short, int` y `long int`) que informa al compilador que la

variable se utilizará para almacenar sólo valores no negativos. Este calificador doble el máximo valor que se pueda almacenar en esa variable.

Sintaxis

```
unsigned <tipo entero> <nombre_variable>
```

Ejemplo

```
unsigned char lista[1000];
unsigned t;                  // equivale a unsigned int t
```

using

Declaración `using` y directiva `using`. Una declaración `using` permite a un cliente tener acceso a todos los nombres del espacio de nombres.

```
using namespace mcd;
using namespace LJAabc;
juegos primero;
```

virtual

Especificador de una función que declara una función miembro de una clase que se redefinirá por una clase derivada

```
class B {
public:
   int i;
   virtual void imprimir_t() const
      { cout << i << "interior a B" << endl; }
};
```

void

`void` es un tipo de dato que se utiliza para indicar la no existencia de un valor de retorno o argumentos en una declaración y definición de una función. Se puede utilizar también `void*` para declarar un puntero a cualquier tipo de objeto dado.

Sintaxis

```
void func(void);
```

Ejemplo

```
void una_funcion(void *ptr);
```

volatile

El calificador de tipo `volatile` se utiliza para indicar que la variable que sigue a `volatile` se puede modificar por factores externos al control del programa. Por ejemplo, las variables que se cambian por hardware, tales como reloj de tiempo real, interrupciones u otras entradas, se declararán `volatile`.

wchar_t

Tipo carácter ancho (*wide*). Se utiliza para representar conjuntos de caracteres que requerirá más de los 255 caracteres estándar. Está concebido para juegos de caracteres que requieren caracteres no representables por `char`, tales como el alfabeto japonés kana.

while

La sentencia `while` permite construir un bucle cuyas sentencias interiores se ejecutan hasta que una *condición* o *expresión* se hace falsa (cero).

Sintaxis

```
while (condicion) {
   <sentencias;...>
}
```

Un bucle `while` con una sola sentencia en su cuerpo se escribe así:

```
while (condicion) sentencia;
```

Ejemplo

```
// suma de 100 numeros
suma = 0;
while(j <= 10)
{
   suma += j;
   j++;
}
j = 0;
```