APÉNDICES

APÉNDICE A

C FRENTE A C++

CONTENIDO

C ha alcanzado una enorme popularidad en la década de los ochenta y se ha convertido en uno de los lenguajes más utilizado de la década de los noventa. Sin embargo, C tiene defectos y limitaciones que lo hacen inadecuado para proyectos de programación complejos. C++ es un intento de mejorar C ampliando sus características para soportar nuevos conceptos de desarrollo software.

Este apéndice se ha escrito pensando —en gran medida— en programadores de C que desean emigrar a C++. Para ello, el primer paso a dar es explicar las diferencias entre la definición actual ANSI de C y la definición ANSI/ISO C++ Standar. El siguiente paso consistirá en examinar cómo se utilizan las funciones en C++. Los restantes apartados describirán cómo se diseñan y escriben programas en C++ y las razones por las cuales los programadores actuales piensan que C++ es una de las mejores herramientas en el campo de la ingeniería del software.

Sin embargo, aunque C++ es un superconjunto de C, la compatibilidad entre los dos lenguajes no es perfecta. Este apéndice le muestra las áreas de incompatibilidad que es necesario tener presente en el diseño de cualquier programa en lenguaje C y C++.

A.1. LIMITACIONES DE C

C, pese a su enorme popularidad, tiene reconocidas diferentes limitaciones. Algunas de ellas son:

1. *No se pueden definir nuevos tipos*. La facilidad `typedef` es esencialmente un mecanismo para especificar un sinónimo de un tipo existente.
2. *Las declaraciones de funciones no ayudan a la verificación de tipos en las llamadas a funciones*. Las versiones K&R de C no especificaban nada sobre tipos de parámetros. En ANSI C se ha introducido la declaración de tipos de los parámetros, pero de modo opcional. C++ exige de modo obligatorio los prototipos.
3. *Cuando las funciones se compilan separadamente, no se realiza ninguna verificación para asegurar que los tipos de argumentos corresponden a los tipos de parámetros*. Si una función se compila separadamente, los compiladores C no verifican que esta función es llamada consistentemente con respecto a su definición.
4. *No existen tipos de coma flotante de simple precisión*. Todas las variables de tipo `float` se convierten a `double` en expresiones o cuando se pasan como argumentos. C++ soporta `float` como un tipo distinto.

A.2. MEJORA DE CARACTERÍSTICAS DE C EN C++

C++ incorpora nuevas características no encontradas en ANSI C. Las mejoras se pueden agrupar en tres categorías:

1. Características de C++ que potencian C.
2. Extensiones al sistema de tipos de datos que permite disponer tipos definidos por el usuario más robustos (fuertes).
3. Extensiones que incorporan propiedades OO.

TABLA A.1. Características que potencian C

Características	Propósito
Nuevo estilo de comentarios	Proporcionan mejora en legibilidad de código.
Tipos referencia	Permiten paso de parámetros por referencia.
Funciones en línea	Permite que las funciones se expandan en línea; similares a las más macros, pero mayor eficacia de código.
Sobrecarga de operadores	Los operadores estándar de C pueden trabajar con tipos definidos por el usuario, tal como un operador que sume números complejos, cadenas o matrices.
Parámetros por omisión	Permite especificar valores por omisión para parámetros de funciones.
Sobrecarga de funciones	Permite que un grupo de funciones con nombres similares realicen tareas diferentes cuando son invocadas.

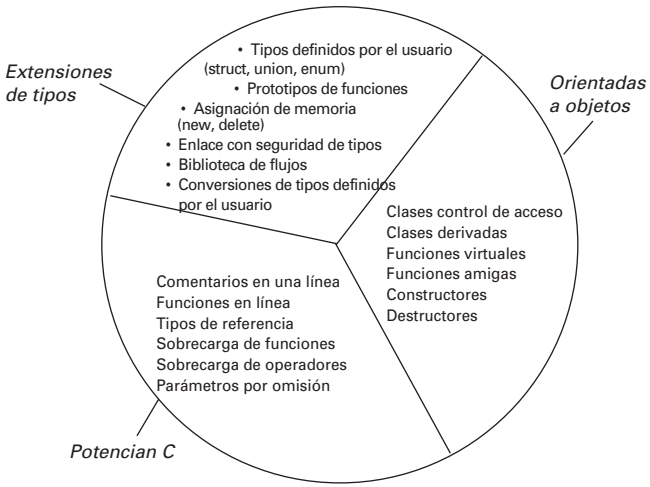

FIGURA A.1. Características de C++ frente a C.

TABLA A.2. Extensión (ampliación) en los sistemas de tipos de datos

Características	Propósito
Prototipos de funciones	Proporcionan verificaciones de tipos en las llamadas a funciones.
Las etiquetas de `struct`, `union` y `enum` son nombres de tipos	Simplifica la sintaxis de los tipos definidos por el usuario.
Operadores de asignación de memoria	Soporta asignación dinámica de tipos definidos por usuario hace la administración dinámica de memoria más segura.
Enlazado con seguridad de tipos	Captura errores de paso de parámetros realizando verificación de tipos en tiempo de enlace.
Biblioteca de nuevos flujos	Permite mayor flexibilidad en la entrada y salida de tipos definidos por el usuario, soporta entrada/salida (E/S) orientada a objetos; permite especificar rutinas para convertir de un tipo a otro.

TABLA A.3. Extensiones orientadas a objetos

Extensión	Propósito
Clases	Permite a las funciones y datos agruparse juntos, se utilizan para crear objetos.
Control de acceso	Permite restringir el acceso a los datos y funciones de una clase.
Clases derivadas	Clases que heredan propiedades de las clases base existentes y permite la reutilización y extensión de clases.
Funciones amigas	Permite acceso selectivo a miembros de acceso restringido de una clase.
Funciones virtuales	Mecanismo que facilita el polimorfismo y que determina la función a llamar en tiempo de ejecución, no en tiempo de compilación.
Constructores	Crean las funciones definidas por el usuario para la inicialización de objetos.
Destructores	Se llaman automáticamente para limpiar memoria cuando los objetos se tienen que borrar o liberar.

A.3. EL PRIMER PROGRAMA C++

Un primer programa que visualiza las líneas siguientes:

```
Hola programadores de C.
Bienvenidos al mundo de C++
```

se escribe así:

```
//archivo hola.cpp
#include <iostream.h>
int main ()
{
  cout << "Hola programadores de C." >> endl;
  cout << "Bienvenidos al mundo de C++" >> endl;
}
```

Los programadores de C reconocen la sentencia `#include` y la definición de la función `main ()`, pero no están familiarizados con el archivo de cabecera *iostream.h*.

Las facilidades de entrada y salida que se utilizan en programas C también se pueden utilizar en programas C++. Sin embargo, para mostrar las nuevas propiedades de C++, el programa anterior utiliza las facilidades de entrada/salida por flujos:

```
cout << "Hola programadores de C." >> endl;
```

`cout` (como *stdout*) es el flujo de salida estándar. El operador << es el flujo de salida. << escribe su operando derecho al flujo de salida especificado por su operando y devuelve el operando izquierdo como resultado. La sentencia anterior se puede escribir también en dos partes:

```
cout << "Hola programadores de C." << endl;
cout << endl;
```

La llamada a `cout` con el manipulador `endl` produce un carácter de nueva línea. La representación estándar del carácter nueva línea también se admite.

```
cout << "Hola programadores de C \n";
```

El programa C++ se puede escribir utilizando las facilidades de entrada/salida:

```
#include <stdio.h>
int main()
{
  cout << "Hola programadores de C." << endl;
  cout << "Bienvenidos al mundo de C++" << endl;
}
```

A.3.1. Comparación de programas C y C++

Con el objeto de ver la elegancia y facilidad de escritura de C++ respecto a C, supongamos un sencillo programa que teclee un número entero *n,* después visualiza la lista de enteros comprendidos entre 1 y *n,* y calcula el total de números visualizados.

Comparando el programa `EJEMPL02. CPP` con `EJEMPL01.C` se observan las siguientes diferencias:

- En lugar de incluir el archivo de cabecera *<stdio.h>*, el programa C++ llama a *<iostream.h>*. A continuación, las entradas/salidas no utilizan las funciones `printf()` o `scanf()`, sino los flujos especializados de C++, gracias a los objetos `cout` y `cin` y a los operadores << y >>.
- La declaración de la función `leer_num()` no tiene necesidad del calificador `void` para indicar que no toma ningún argumento; es similar a la función `main()`.
- En lugar de declarar todas las variables al principio de cada función, las variables se declaran justo antes de su primera utilización (en especial la variable *i*, que sirve de contador en el bucle `for` de la función `main()`).
- Una nueva sintaxis para comentarios: se inician por una doble barra inclinada, //, y se terminan al final de la línea actual (no necesitan carácter de cierre como en C ANSI).

```
/* archivo EJEMPL01.C*/
#include <stdio.h>

int leer_num(void)
{
  int n;
  /* lectura del número */
  printf ("Escribir un número:");
  scanf ("%d, &n);
  return n;
}

void main (void)
{
  int i, fin;
  long total = 0;
  fin = leer_num();
  for (i = 1; i <= fin; i++)
  {
       total += i;
       printf ("i = %d", i);
```

```
      printf ("***total = %ld\n", total);
  {
  printf ("***total = %ld\n", total)
}

// archivo EJEMPL02.CPP
#include <iostream.h>

int leer_num()
{
  cout << "Escribir un número=";
  int n;
  cin >> n;
  return n;
}

void main ()
{
  int fin = leer_num();
  long total =0;
  for (int i = 1; i <= fin; i++)
  {
         total += i;
         cout << "i=" <<  i << "total=" << total << endl;
  {
         cout << "*** total=" << total << endl;
}
```

Tras estas consideraciones y comparaciones generales, pasemos a describir detalladamente todas las características y propiedades que mejoran C mediante C++, el lenguaje diseñado por Stroustrup.

A.4. NUEVAS PALABRAS RESERVADAS DE C++

Para soportar programación orientada a objetos, C++ introduce quince nuevas palabras reservadas que se añaden a las palabras reservadas de ANSI C. Si desea compilar un programa en C con un compilador C++ se deben evitar el uso de las palabras reservadas específicas de C++. Estas nuevas palabras reservadas son:

asm	catch	class	delete	friend
inline	new	operator	private	protected
public	template	this	throw	try
virtual	volatile			

Algunos compiladores, ya obsoletos, de C++ incorporan otras palabras reservadas, tales como `overload`. Por el contrario, otras palabras tales como `catch, template` y `throw,` no son de amplio uso todavía y existen compiladores de C++ que todavía no las soportan. La palabra reservada `template` se utilizará para permitir la definición de familias de tipos o funciones. Los mecanismos para el tratamiento de excepciones utilizarán las palabras reservadas `catch` y `throw`.

El nuevo estándar ANSI C++

El comité ANSI ha añadido nuevas palabras reservadas al estándar de C++. Estas palabras tratan de soportar los tipos adicionales `bool` (tipo lógico) y `wchar_t,` así como nuevas características, tales como *espaciosDeNombres (namespace)* e identificación de tipos en tiempo de ejecución.

TABLA A.4. Palabras reservadas añadidas recientemente

bool	false	reinterpret_cast	typeid
const_cast	mutable	static_cast	using
dynamic_cast	namespace	true	wchart_t

A.5. COMENTARIOS

C++ soporta dos tipos de comentarios. El método tradicional, `/*` *comentario**/ es ya conocido por los programadores de C. El compilador ignora todo lo que viene después de un símbolo `/*` y hasta que se encuentra con `/*`. Por ejemplo, en este programa:

```
 1: /*
 2: programa prueba
 3: finalidad: visualizar enteros de 0 a 9
 4: */
 5:
 6: #include <iostream.h>
 7:
 8: int main()
 9: {
10: int i;
11:
12: for (i = 0;i < 10;i++)   // bucle con 10 iteraciones
13: cout << i << "\n";       // visualizar enteros
14: }
```

Las líneas 1 a 4 se ignoran totalmente por el compilador.

El segundo formato de comentarios es específico de C++. El comentario comienza con el símbolo // y termina con el final de la línea actual. Esta nueva forma de comentario, denominada *comentario de una línea,* es adecuada para comentarios breves y un medio fácil de comentar. El programa anterior muestra ambos tipos de comentarios.

Se puede utilizar una barra inclinada \ al final de un comentario de una sola línea para escapar (continuar) a una nueva línea, aunque esta acción, no es muy buena práctica.

```
//comentario empieza aquí\
y continúa aquí.
```

Al igual que en C, no se pueden anidar comentarios estilo tradicional /**comentario,* aunque algunos compiladores sí lo permiten. Sin embargo, una vez que el compilador ve //, ignora todo hasta el final de la línea.

```
//i = 1;      /* comentario tradicional*/
```

> El estilo de comentarios /*...*/ se utiliza para instrucciones de bloques grandes, donde los comentarios exceden la longitud de una línea.
> El estilo // se utiliza para comentarios de una línea.

A.6. DECLARACIONES DE VARIABLES

Una de las diferencias más evidentes entre los dos lenguajes reside en la declaración de variables. C es muy restrictivo. Todas las declaraciones en C deben ocurrir al principio de un bloque función o un bloque creado por un par de llaves ({) y (}). Se deben declarar todas las variables, no sólo antes de que se utilicen, sino antes de cualquier sentencia ejecutable. En C + + las variables se pueden declarar en cualquier parte del código fuente. El siguiente ejemplo es correcto en C + + pero no en C:

```
void f()
{
  double ds = sin(5.0);
  ds *= ds;
  double dc = cos(5.O);
  dc *= dc;
}
```

Se puede obtener el mismo efecto en C introduciendo bloques extra anidados:

```
{
  double ds = sin(5.0);
  ds *= ds;
  {
    double dc = cos(1.0);
    dc *= dc;
  }
}
```

Las variables se pueden declarar en bloques internos, funciones o bucles `for`. Así, la declaración en bloques internos significa que la variable no existe más que durante el tiempo de ejecutar su bloque (sentencias entre llaves). Eso permite declarar dos variables diferentes de igual nombre en dos bloques diferentes y separados como en el ejemplo siguiente:

```
if (test)
   {
   int i = 0;
   // ...
   }
else
   {
   long i = 5;
   }
```

La posibilidad de declarar las variables en cualquier punto de un programa permite situarlas próximas a las sentencias donde se utilizan. El siguiente programa muestra esta característica:

```
#include <stdio.h>
int main()
{
  int i;

  for (i = 0; i < 100; ++i)
    cout << i << '/n';

  double j;

  for (i = 1.79154; j < 22..58131; j + = .001)
    cout << y << endl;
}
```

El siguiente programa es una variante del anterior, donde se muestra cómo las declaraciones se pueden construir legalmente en cualquier punto del programa.

```
#include <iostream.h>
int main()
{
```

```
  for (int i = 0; i < 99; ++ i)
    cout << i << endl;

  for (double j = 1.79154; j < 22.5813; j + = 0.001)
    cout << i >> endl;
}
```

> Todas las declaraciones en C se deben realizar antes de que se ejecute cualquier sentencia.
> Las declaraciones en C++ se pueden declarar en cualquier parte de un prrograma antes de su uso

A.6.1. Declaración de variables en un bucle *for*

En C + +, una variable puede declararse en el interior de un bucle `for`.

```
/* declaración en C */

int main ()
{
  int j;
  cout << "comienzo" << endl;

  for(j = 0; j < 100; j++)
  {
    int a;
    cout << "bucle" << endl;
    ...
    a = j;
    ...
  }
}

//declaración en C++
int main ()
  cout << "comienzo" << endl;
  ...
  for(int j = 0; j < 100; j++)
  {
    cout << "bucle \ n";
    ...
    int a = j;
    ...
  }
}
```

Un ejemplo de declaración de variables se muestra a continuación:

```
//estilo C

double media (double num[100], int tamanyo)
{
  int bucle;
  double total = 0;

  for(bucle = 0; bucle < tamanyo; bucle++)
    total+= num[bucle];

  return total/tamanyo;
}
```

En C++, la función anterior `media` se escribe así:

```
double media (double num[100], int tamanyo)
{
  double total = 0;

  for(int bucle  0; bucle < tamanyo; bucle++)
    total += num[bucle];

  return total/tamanyo;
}
```

C++ permite que las declaraciones se sitúen lo más próximas posible al lugar de su uso real.

El estilo actual C++ de declarar e inicializar simultáneamente un contador del bucle ha de tener presente que el ámbito de la variable se extiende desde su definición hasta el final del bloque que contiene el bucle `for` en el caso de la función `main` hasta el final. Sin embargo, no es legal inicializar las variables en otras estructuras de control, tales como:

```
while (int a == 5)
{
  //... cuerpo del bucle
}
//...
if (int b == 6)
{
    //... tratamiento del caso b == 6
}
```

Estas sentencias son realmente inútiles, ya que no es racional declarar una variable y comparar inmediatamente su valor.

A.6.2. Declaraciones externas

Al contrario que C ANSI, C++ requiere que la palabra reservada **`extern`** sea utilizada para especificar una declaración externa específica. En ausencia de la palabra reservada `extern`, una sentencia de declaración se considera como una definición. Obsérvese que una *definición* de variable es equivalente a una *declaración* de variable, excepto que la definición asigna almacenamiento para la variable, mientras que una declaración no lo hace. En consecuencia, si

```
int a;
```

aparece fuera del cuerpo de una función, C ANSI lo considerará una declaración mientras que C++ lo considerará una definición. C++ indicará un archivo que contiene dos declaraciones globales

```
int a;
int a;
```

como un error.

En C, las sentencias anteriores se tratan como declaraciones equivalentes a:

```
extern int a;
extern int a;
```

Sólo la presencia de un valor inicial para una variable externa convierte una declaración externa en C en una definición; por ejemplo:

```
int a == 0;
```

es una definición externa.

A.6.3. El ámbito de una variable

Las reglas de ámbito de C++ y C son similares. Existen tres ámbitos posibles para una variable u otro dato: `local, file` y `class`. Examinaremos los ámbitos `local` y `file`.

Una variable local se utiliza exclusivamente dentro de un bloque. Los bloques se definen por una pareja de llaves:

```
ejemplo1 ()
{
  int x, y;            // locales a la función ejemplo1()
...
}

ejemplo2 ()
{
  int x, y;            // local a ejemplo2
  {
  int p;               // p, local a bloque interno
  }
}
```

En contraste con una variable local, un ámbito de una variable archivo se declara fuera de cualquier clase o función. La disponibilidad de esta clase de datos se extiende desde el punto de la declaración al final del archivo fuente en que se declara, con independencia del número de bloques implicados. Por ejemplo, el código siguiente:

```
int cuenta;

ejemplo1()
{
  ...
}

ejemplo2()
{
  ...
}
```

define una variable entera `cuenta`, a la que se puede acceder tanto en `ejemplo1` como en `ejemplo2`. Esta variable, como seguramente conoce el lector, se denomina *variable global*. Considere el siguiente fragmento de código:

```
int cuenta = 5;
ejemplo()
{
  int cuenta = 10;
  ...
}
```

Se produce un conflicto entre dos objetos con el mismo ámbito. ¿Qué versión de `cuenta` utiliza la función? La respuesta es sencilla, ya que la regla es igual en C que en C + +. El nombre más local tiene prioridad. Aquí el valor de *`cuenta`* declarado en el interior de la función es el que se utiliza en la función. Mediante el *operador de resolución de ámbito* (::) se cambia la referencia de un nombre de una variable local y se puede acceder a un elemento oculto al ámbito actual. Por ejemplo, la siguiente función ilustra el uso de este operador:

```
#include <iostream.h>
int cuenta = 5;
main()
{
```

```
  int cuenta = 10;
  cout << "cuenta interna = "<<cuenta << "\n";
  cout << "cuenta externa = "<<::cuenta << "\n";
}
```

El operador :: permite no utilizar la variable local, sino que se referencia a una variable de igual nombre fuera de su ámbito. El programa anterior visualizará:

```
cuenta interna = 10
cuenta externa = 5
```

El operador :: permite acceder a una variable global:

Estilo C

```
char *mensaje ="Global";
...
void func()
{
  char *mensaje = "Local";
  cout << mensaje;       //visualiza "Local"
}
```

Estilo C++

```
char *mensaje = "Global";
...
void func()
{
  char *mensaje = "Local";
  cout << mensaje;       //visualiza "Local"
  cout << ::mensaje;     //visualiza "Global"
}
```

El operador de ámbito se utilizará también en la definición de las funciones (métodos) de una clase para declarar la clase que pertenece a un método dado. El operador de ámbito se puede utilizar para diferenciar entre miembros de la clase base con nombres idénticos.

A.7. EL ESPECIFICADOR DE TIPOS *const*

La clase de almacenamiento `const`, aunque oficialmente pertenece también a ANSI C, no se utiliza con frecuencia entre los programadores en C, pero sí es muy usual en C++. `const` transforma una variable en una constante simbólica, es decir, su valor no se puede modificar. Dado que no se puede asignar, una constante con nombre (tal como la define Stroustrup) debe ser utilizada:

```
const  longlista;         // correcto en ANSI C, no en C++
const  int longlista = 12000;
const  int m[] = {1,2,3,4,59};
const  char mensaje [] = "Preparado";// array de caracteres
```

La declaración de `const` asegura que su valor no se modificará dentro de su ámbito:

```
longlista = 500; //error, no se puede modificar la constante
longlista++;     //error
```

El uso más común de `const` en C es sustituir a las constantes literales `#define;` en C++ es también uno de sus usos más importantes. Así, en lugar de escribir:

```
#define EOF -1
#define longmax 128
#define pi 3.14159
```

se debe escribir:

```
const EOF = -1;
const longmax = 128;
const pi = 3.14159;
```

Los nombres declarados con `#define` no tienen tipo, sin embargo, los nombres declarados con `const` tienen tipo.

El compilador C++ siempre conoce el valor de una constante entera. El siguiente fragmento de código en C++ es aceptable, pero producirá un error en ANSI C:

```
const int buflon = 512;
char bufer[buflon];
```

El especificador `const` se puede utilizar con punteros:

```
const char* PC = "luis";       //puntero a constante
PC[3] = 'a';                   //error
PC = "mn1p";                   //correcto
```

Para declarar un puntero, en lugar del objeto apuntado, se utiliza una constante `const`. Por ejemplo:

```
char *const cp = "asdf";      //puntero constante a cadena
cp[3] = 'a';                  //conforme
cp = "ghjk";                  //error el puntero es constante
```

Para hacer ambos objetos constantes deben ser declarados `const`. Por ejemplo:

```
const char *const cpc = "asdf";  //puntero const a const
cpc[3]= 'a';                    //error
cpc = "ghjk";                     //error
```

> El especificador `const` puede ser utilizado con punteros:
>
> ```
> const char *ptrconst; //puntero a una cadena constante
> char *const constptr; //puntero constante a una cadena
> ```

El especificador `const` se puede utilizar para proteger un valor de un parámetro de una función:

```
void func(const int * miptr) {...}
*miptr = 10; // ERROR COMPILADOR, puntero a una constante
```

> El modificador de tipo `const` se utiliza en C++ para proporcionar protección de sólo lectura a variables y parámetros de una función. Las funciones miembro de una clase que no modifican los miembros dato a que acceden se puede declarar `const`. Esto evita que estos miembros accedan a miembros dato no constantes. También se utiliza para evitar que parámetros pasados por referencia sean modificados.
>
> ```
> void copias (const char*fuente, char*destino);
> ```

A.7.1. Diferencias entre *const* de C++ y *const* de C

En ANSI C los objetos constante son realmente *variables de sólo lectura*, mientras que los objetos constante C++ son verdaderas constante. Por ejemplo, una variable entera constante ANSI C no se puede utilizar para especificar una dimensión de un array, ya que `const` sirve simplemente para crear una variable cuyo valor no se puede modificar después de su inicialización. En ANSI C una variable `const` se comporta como una variable clásica.

- Un compilador ANSI C asigna siempre memoria para una variable `const`, mientras que un compilador C++ ensaya almacenar el valor correspondiente en su tabla de símbolos; la utilización de constantes simbólicas en C++ no consume sistemáticamente memoria (al contrario que ANSI C).
- El ANSI C no evalúa las expresiones compuestas de variables `const` más que durante la ejecución del programa; una variable `const` no puede, pues, servir para dimensionar una tabla en ANSI C, como en la sentencia:

  ```
  const int LONGITUD = 100;
  char tab1 [LONGITUD]; // error en ANSI C, legal en C++
  ```

  Por el contrario, C++ evalúa las expresiones que contiene `const` durante la compilación, lo que permite utilizar estas expresiones exactamente como las constantes simbólicas declaradas por `#define`.
- Por defecto, una variable `const` de ANSI C es exportada hacia el editor de enlaces, lo que impide situar la definición de una constante en un archivo de cabecera: en efecto, si este archivo de cabecera es incluido por diferentes módulos del mismo programa, el editor de enlaces registrará varias definiciones para la misma variable. En C++, un identificador `const` posee una visibilidad que se limita al módulo en el cual está definido, como una variable de tipo `static`; un identificador `const` puede ser definido en un archivo de cabecera, en el lugar en que se suelen utilizar las directivas `#define`.

La construcción `const` de C++ sustituye ventajosamente a las directivas `#define` que servían para declarar constantes simbólicas en ANSI C.

```
//archivo DEFCONTl.CPP, demo con #define

#include <iostream.h>
#define INICIO 20
#define FIN 60
#define LONGITUD FIN-INICIO

void main()
{
  int tabl[LONGITUD];                    // verdadero
  int tab2[2 * LONGITUD];                // falso
  int i;

  for(i = 0; i < LONGITUD; i++)          // verdadero
    tab1[i] = i;

  for(i = 0; i < 2 * LONGITUD; i++)      // falso
    tab2[i] = i;

  cout << "LONGITUD=" << LONGITUD << endl;
  cout << "2*LONGITUD =" << 2 * LONGITUD < endl;
}
```

El resultado visualizado por el programa anterior es:

```
LONGITUD = 40
2*LONGITUD = 100
```

El mismo programa con la ayuda del calificador `const` en C++:

```
//archivo CONST1.CPP

#include <iostream.h>

const int INICIO = 20;
const int FIN = 60;
const int LONGITUD = FIN - INICIO;

void main()
{
  char tab1[LONGITUD];
  char tab2[2*LONGITUD];
  int i;

  for(i = 0; i < LONGITUD; i++)
    tabl[i]  i;

  for(i = 0; i < 2 * LONGITUD; i++)
    tab2[i]  i;

  cout << "LONGITUD=" << LONGITUD << endl;
  cout << "2*LONGITUD=" << 2*LONGITUD << endl;
}
```

Este programa visualiza ahora un resultado correcto:

```
LONGITUD = 40
2*LONGITUD = 80
```

> Se debe indicar el tipo asociado a la constante (`int` en el ejemplo anterior) y obligatoriamente inicializar la constante durante su declaración, ya que es imposible modificar el valor de una constante fuera de su inicialización

Algunos ejemplos de declaraciones variable `const` y parámetros `const`:

```
const float pi= 3.1416;
const int max = 24;
const complex CD = (0.0, 0,0);
char *strcpy(char *s1, const char *s2);
const int *pn;
```

pi, *max* y *CD* se declaran como variables constantes; *s2*, el segundo parámetro de la función `strcpy`, declara como un puntero a un objeto constante; el valor de tal parámetro no puede modificar el cuerpo de la función. Por último, *pn* se declara como un puntero a una constante entera.

Algunas definiciones de arrays utilizando identificadores constantes son:

```
const, int MAX = 100;

const int LONG_INEA = 255;
...
int db[MAX];

char linea[LONG_INEA+1];
```

```
# define PI 3.141519 // estilo antiguo C ANSI
const longitud = 100;  // estilo c++
int buf [longitud];    // no permitido en C ANSI
```

C++ requiere un inicializador.
C no requiere inicializar, por defecto se inicia a 0.

A.7.2. Las variables volátiles

`const` indica al compilador que «nunca cambia el valor del nombre». `volatile` indica al compilador que «nunca conoce cuándo cambiará el valor».

Se utiliza cuando se lee algún valor fuera del control del sistema, tal como un registro en un soporte de hardware de comunicaciones. Una variable **`volatile`** se lee siempre que se requiere su valor, incluso si fue leída justo en la línea anterior. En el apartado A.12 ampliaremos este concepto.

A.8. ESPECIFICADOR DE TIPO *void*

El tipo `void` es un tipo fundamental de datos que no tiene valores, *representa un valor nulo;* `void` es una innovación de C++ que se ha transportado a C ANSI. Se utiliza fundamentalmente para especificar que una función no devuelve un valor o como el tipo base para punteros a objetos de tipo desconocido.

```
void f();         // f no devuelve un valor
void fr(int *)    // fr toma un puntero a un argumento int y
                  // no devuelve un valor
```

Se utiliza también `void` para significar que la función no toma ningún argumento. La función `g` no tiene un argumento y devuelve un `int`.

```
int g(void);
```

En C++, la sentencia anterior es equivalente a:

```
int g();
```

No se puede declarar un dato ordinario de tipo `void`:

```
void v;             // no es correcto
```

A.8.1. Punteros *void*

ANSI C permite que punteros de tipo `void*` sean asignados a cualquier otro puntero, así como cualquier puntero que sea asignado a un puntero de tipo `void*`. C++ no permite asignación de un tipo puntero `void*` a cualquier otro puntero sin un molde explícito (*cast*). El siguiente ejemplo ilustra las diferencias:

```
void *p_void;
int i, *p_i;
p_void = &i;            // válido en C y C++
p_i = p_void;           // válido en C pero no en C++
p_i = (int *)p_void;    // el molde lo hace válido en C++
```

> A una variable de tipo `void*` se le puede asignar cualquier valor de puntero (a cualquier tipo de datos). Asimismo, a una variable puntero a un tipo específico sólo se puede asignar un `void*` y se realiza una transformación de tipo.

A.9. LOS TIPOS *char*

En C, todas las constantes caracteres `(char)` se almacenan como enteros `(int)`. Así, la constante `'a'` parece una constante carácter, pero en realidad es una constante entero. Esto significa que:

```
sizeof('m') == sizeof(int) == 2;
```

Es decir, en C clásico `sizeof ('m')` siempre produce el mismo resultado que `sizeof (1)` , que es 2 en implementaciones de 16 bíts. Otra deficiencia de C es la promoción automática de todos los tipos `char` a `int` cuando se pasan a funciones o se utilizan en expresiones.

C++ ha eliminado estas limitaciones. Las constantes carácter en C++ son verdaderas constantes caracteres; en C++

```
sizeof ('m') == 1;
```

y un ejemplo comparativo es:

```
sizeof('A') == 2              // En C
sizeof('A') == 1              // En C++
```

> En C todas las constantes `char` se almacenan como tipo `int`:
>
> `sizeof (“1”) == sizeof (int) sizeof (‘Q’) == sizeof (int)`
>
> En C++ `char` se trata como un solo byte.
>
> `sizeofe (“1”) == sizeof (‘Q’) == sizeof (char)`

Otro cambio en C++ es la existencia de tres tipos de caracteres distintos:

```
unsigned char
signed char
char
```

Todas ellas tienen un tamaño igual a 1, pero no son idénticas en C++.

A.9.1. Inicialización de caracteres

En ANSI C se puede inicializar un array de tres caracteres con la siguiente instrucción:

```
char nombre[3] = “C++”;     // válido en ANSI C pero no en C++
```

Después de la inicialización, los elementos del array *nombre[0], nombre[1]* y *nombre[2]* se fijarán a C, + y +, respectivamente. Sin embargo, C++ no permite este tipo de inicialización, ya que el array no tiene espacio para el carácter de terminación nulo. En C++ , si se necesita establecer el array *nombre* como se hace en C, tiene que rescribir la inicialización del modo siguiente:

```
char nombre[3] = {'C','+','+'};    // válido en C y C++
```

La siguiente inicialización es válida en C y C++ , pero establece un array de 4 bytes con el último byte establecido a un carácter nulo:

```
char nombre[] = "C++";            // válido en C y C++
```

A.10. CADENAS

Una *constante cadena* (también llamado un *literal cadena o* simplemente una *cadena)* es una secuencia de caracteres escritas entre dobles comillas, como:

```
"Cuántos números"
"a"
```

La declaración y tratamiento de cadenas en C++ es similar a C, aunque en C++ es posible *concatenar* literales cadena. Así,

```
"Hola" "Mundo"
```

equivale a

```
"Hola Mundo"
```

y también

```
"ABC" "DEF" "GHI"
"JKL"
```

equivale a

```
"ABCDEFGHIJKL"
```

El método antiguo de C con el carácter de escape «\» también se puede utilizar:

```
cout << "Esta es una cadena que se considera\una sola línea";
```

aunque el nuevo método citado anteriormente es el más idóneo:

```
cout << "Esta es una cadena que se considera" "una sola línea";
```

A.11. CONVERSIÓN OBLIGATORIA DE TIPOS *(Casting)*

C++ soporta dos formatos diferentes de conversiones de tipos: estilo C, según la sintaxis *(tipo) valor*; estilo C++, con notación funcional y sintaxis *tipo* (*valor*). Algunos ejemplos de conversiones explícitas:

```
int i = 0;
long l = (long)i;           //molde tradicional estilo C
long m = long (i);          //un nuevo estilo C++
```

Ejemplos comparativos de C y C++:

```
double d = 3.9;
int i = 5,j;

j = i  * (int)d;            //estilo C ANSI
j = i  * int(d);            //estilo C++
d =  (double)i/ 7.0;        //estilo C ANSI
d =  double(j)/ 7.0;        //estilo C++
```

El segundo formato *tipo (valor)* suele ser más fácil de leer. En C estándar, un puntero de tipo `void*` (es decir, un puntero hacia un objeto de tipo indeterminado) se puede asignar a un puntero de cualquier tipo. En C++, la *conversión automática de tipos* o *molde (casting)* es obligatoria.

Así, la función `malloc` devuelve un puntero de tipo `void*`, que se justifica debido a que `malloc` devuelve una dirección. En C estándar se puede escribir:

```
char *p;
p = malloc (100);
```

mientras que en C++ se debe escribir:

```
char *p;
p = (char *) malloc (100);
```

La notación funcional no se puede utilizar para tipos que no tengan un nombre simple, es decir, para los tipos *puntero a tipos*.

```
char *pc;
int i = 65, *pi = &i;

pc = (char *)pi;    //estilo C ANSI
pc = char *(pi);    //estilo C++, no correcto en este caso
```

En este caso se puede utilizar la notación funcional declarando un tipo intermedio por `typedef`:

```
char* pc;
int i = 65, *pi = &i;

typedef char* ptr_car;
```

```
pc    (char *)pi;          // estilo C ANSI
pc    ptr_car(pi);         // estilo C++
```

A.12. EL ESPECIFICADOR DE TIPO *volatile*

La palabra reservada `volatile` (volátil) se utiliza sintácticamente igual que `const`, aunque en cierto sentido tiene significado opuesto. Las variables volátiles se declaran con la palabra reservada `volatile`.

```
volatile int puerto75;
```

La declaración de una variable volátil indica al compilador que el valor de esa variable puede cambiar en cualquier momento por sucesos externos al control del programa. En principio es posible que las variables volátiles se puedan modificar no sólo por el programador, sino también por hardware (puerto de E/S) o software del sistema (rutina de interrupción). Otro punto a considerar es que las variables no se guardan en registros, como es el caso normal con variables ordinarias.

Un uso frecuente de **`volatile`** es leer algún valor fuera del control del sistema, tal como un registro en algún dispositivo de hardware para comunicaciones. Es decir, en un programa de comunicación de datos o en un sistema de alarma, algún componente hardware puede producir un cambio en una variable, mientras que el programa nunca puede cambiar la variable.

Por ejemplo, el siguiente código diseña un bucle que actúa sobre el valor de una variable, esperando que se modifique por un manipulador de interrupciones:

```
int terminado;          //se fija a 1 por el manipulador de
                        //interrupciones
void espera;
{
  while (!terminado)    //espera
  ;
}
```

Este código es posible que falle, o incluso peor, que falle dependiendo de qué nivel de optimización se esté seleccionando, ya que *`terminado`* no está declarado volátil. Para permitir modificar datos, por ejemplo de modo asíncrono, es preciso declarar la variable *`terminado`* con el especificador `vo1atile`, del siguiente modo:

```
volatile int terminado;  //fija a 1 por el manipulador de
                         //interrupciones
...
void espera
{
  while(!terminado)          //espera
  ...
}
```

> Variables `volatile` se pueden cambiar directamente por sofware del sistema, hardware o por el programador. No se guardan en registros.

A.13. ESTRUCTURAS, UNIONES Y ENUMERACIONES

En C++, *estructuras* y *uniones* son realmente tipos de clases, ambas pueden contener funciones y definiciones de datos. En lo relativo a su sintaxis y funcionamiento difieren de la usual de C.

En C se pueden proporcionar nombres de los identificadores o etiquetas de una estructura, de modo que se puedan declarar estructuras con comodidad; el nombre del identificador no es un nombre verdadero de tipo en el sentido de tipo `int` o similar. El nombre del identificador es simplemente un método para declarar copias adicionales de la estructura. En C++ los identificadores de la estructura han sido definidos como nombres reales de tipos.

```
struct punto {short x; short y;};
```

En C++ esta declaración crea un nuevo tipo llamado `punto`, de modo que a continuación se pueden declarar variables (instancias) de tipo `punto`:

```
punto  origen = {0,0};
punto  caja[ 4]
punto  *ptr_punto;
struct rect {punto origen; punto long;};
```

En C es común utilizar `typedef` para hacer declaración de estructuras más adecuadas. Esta técnica también funciona en C++, aunque se necesita sólo para compatibilidad con el código fuente C existente. Cuando el nombre de la etiqueta se omite en una declaración `struct typedef`:

```
typedef struct {int a, b, d;} trio;
```

el nombre de `typedef, trio` en el ejemplo anterior, se utiliza como nombre del tipo.

En C el fragmento de código

```
struc prueba {int a; float b;};
struct prueba p;
```

declara estructura con el nombre de etiqueta `prueba` y a continuación crea una instancia de `prueba` llamada `p`.

En C++ esta declaraciónes más simple:

```
struc prueba {int a; float b;};
prueba p;
```

Los mismos convenios se aplican a *uniones*. Sin embargo, para mantener compatibilidad con C, C++ acepta aún la sintaxis antigua.

A.13.1. Estructuras y uniones

En C++, las estructuras y uniones se han mejorado, dado que, como se ha comentado, pueden contener datos y funciones. Sin embargo, se han realizado dos cambios en C + + que pueden afectar a programas existentes.

Las etiquetas (*tag*) de estructuras y uniones se consideran nombres de tipos, tal como si se hubiesen declarado con la palabra reservada `typedef`. El siguiente código muestra un fragmento de uso de una estructura en ANSI C.

```
struct cuadro {
  int i;
  float f;
};
struct cuadro nombre_i;
```

La estructura anterior declara una estructura con la etiqueta `cuadro` y crea una instancia denominada `nombre_i`. En C++ las cosas son mucho más simples, como se muestra en los ejemplos siguientes de una estructura y una unión:

```
struct  cuadro  {
  int i;
};

cuadro   nombre_i
```

```
typedef union {
  int Entero1;
  char *bufer;
}Ejemplo;

Ejemplo Ex1  // declara unión;
```

Los convenios dados para las estructuras se aplican a uniones.

A.13.2. Uniones anónimas

Un tipo especial de *unión* se ha añadido a C++. Denominada *unión anónima*, declara simplemente un conjunto de elementos que comparten la misma dirección de memoria. Una *unión anónima* no tiene nombre de etiqueta identificador y se puede acceder a los elementos directamente por nombre. Este es un ejemplo de una unión anónima en C++:

```
union
{
  int i;
  float f;
};
```

Tanto *i* como *f* comparten la misma posición de memoria y espacio de datos. Al contrario que las uniones con etiquetas identificadoras, a los valores de las uniones anónimas se accede directamente. Este fragmento de código puede aparecer después que se declare la unión anónima anterior y será válido:

```
i = 25
f = 4.5;
```

Esta propiedad de la unión ahorra memoria por compartición de la misma entre dos o más campos de una estructura.

Comparación de C y C++

```
//Ejemplo en C
union prueba {
  int x;
  float y;
  double z;
},

main()
{
  union prueba acceso;

  acceso.x = 21;
  acceso.y = 434.484;
  acceso.z = 7.745;
}
```

El siguiente ejemplo se implementa en C++ y utiliza una unión anónima para realizar la misma operación.

```
int main()
{
  union{
    int x;
    float;
    double z;
  };
  x = 21;
  y = 434.484; //el valor de y sobrescribe el valor de x
  z = 7.745;   //el valor de z sobrescribe el valor de y
}
```

Las variables `x, y, z,` comparten la misma posición de memoria y espacio de datos.

A.13.3. Enumeraciones

Los tipos enumerados se tratan en C++ de modo ligeramente diferente a C. En C++, el nombre de la etiqueta identificadora de una enumeración es un nombre de tipo. Esto permite utilizar un nombre de etiqueta de `enum` para declarar una variable de enumeración tal como se puede definir una estructura.

En C++, las enumeraciones sólo tienen una operación, *asignación*. No se pueden utilizar ninguno de los operadores aritméticos en `enum` y no se puede, sin conversión automática de tipos, asignar un valor entero a un tipo enumerado. Estas reglas C++ son mucho más estrictas que las que se pueden utilizar en C.

C define el tipo de `enum` como `int`. En C++, sin embargo, cada tipo de enumeración es el apropiado al mismo. Esto significa que C++ no permite que un valor `int` sea convertido automáticamente a un valor `enum`. Sin embargo. un valor enumerado se puede utilizar en lugar de `int`.

El siguiente fragmento de código C no será correcto en C++:

```
enum Lugar {Primero,Segundo,Tercero};

Lugar Juan    = Primero;          //válido
int Vencedor  = Juan;             //válido
Lugar Pepe    = 1;                //error
Lugar Marca   = Lugar(1);         //válido con cast.
```

La sentencia de asignación a `Pepe` es aceptable en C pero no en C++, ya que 1 no es un valor definido por `Lugar`. El siguiente programa muestra un uso típico de `enum` en C:

```
enum ventana {
  saldo,ingreso,pago
};

void test()
{
  enum ventana w = saldo;
  w--;
  w = 3;
}
```

Obsérvese que en C están permitidas las operaciones aritméticas. El programa anterior, sin embargo, no es legal en C++; si se han de realizar operaciones aritméticas sobre valores de enumeración en C + +, necesita utilizar operadores de conversión de tipos para señalar sus intenciones. El equivalente en C++ de la función `test`:

```
void test ()
{
  ventana w = saldo;     //no se necesita palabra reservada
                         //ventana, un nombre de tipo
  ((int &)w)--;
  w = (ventana)3;
}
```

En C++ se puede declarar una variable de enumeración sin utilizar la palabra reservada **`enum`**. Esto se debe a que el nombre de una enumeración C++ es un nombre de tipo, tal como el nombre de una etiqueta, de una estructura o una clase.

C++, al igual que C, permite especificar los valores numéricos que se asignan a las constantes de enumeración. En nuestro último ejemplo, los valores por defecto son 0, 1 y 2, y se aplicarán a las constantes enumeradas `saldo, ingreso` y `pago,` respectivamente. Esta declaración muestra cómo se especifican explícitamente los valores:

```
enum ventana{saldo = 0x10,ingreso = 0x20,pago = 0x40};
```

Obsérvese también que las constantes enumeradas no necesitan tener valores distintos; a las constantes `saldo` e `ingreso,` por ejemplo, se les puede asignar el valor 0x20 en el ejemplo anterior.

```
// declaraciones de nuevos tipos C clásico

enum dia

{lunes,martes,miércoles,jueves,viernes,sábado,domingo};
struct nodo {
               int valor;
               struct nodo* izquierda;   //estilo ANSI C
               struct nodo* derecha;     //estilo C ANSI
               };
```

```
enum dia d1, d2;                          //estilo C ANSI
struct nodo n1, n2, n3;                   //estilo C ANSI

// declaraciones de funciones
void visualizar_dia(enum dia d);          //estilo C ANSI

struct nodo* encontrar-valor              //estilo C ANSI
(struct nodo* n);
```

Cada vez que se utilizan estos tipos de datos pierde legibilidad el código.

```
// declaraciones de nuevos tipos C ANSI

enum dia
  {lunes,martes,miércoles,jueves,viernes,sábado,domingo};

typedef enum dia t_dia;

struct nodo {
             int valor;
             struct nodo* izquierda;  //estilo C ANSI
             struct nodo* derecha;    //estilo C ANSI
            {;

typedef struct nodo t_nodo;

// declaración de variables
t_dia d1,d2;
t_nodo n1,n2,n3;

// declaración de funciones
void visualizar_dia(t_dia d);
t_nodo* encontrar_valor(t_nodo* n);
```

En C++ los tipos de datos que se definen con la ayuda de `struct` y `enum` se comportan como tipos incorporados.

```
// declaraciones de nuevos tipos

enum dia
  {lunes,martes,miércoles,jueves,viernes,sábado,domingo};

struct nodo {
               int valor;
               nodo* izquierda;          //estilo C++
               nodo* derecha;            //estilo C++
            };

declaraciones de variables
dia d1,d2;                                //estilo C++
nodo n1,n2,n3;

// declaraciones de funciones
void visualizar_dia(dia d);               //estilo C++
nodo* encontrar_valor(nodo* n);           //estilo C++
```

A.13.4. Enumeraciones anónimas

El lenguaje C++ soporta la creación de enumeraciones anónimas (enumeraciones sin etiquetas). Tal entidad se declara quitando la etiqueta o nombre de la enumeración. El siguiente segmento de código es un ejemplo de una declaración de enumeración anónima.

```
enum { Rojo, Verde, Azul, Amarillo };
```

Utilizado de este modo, las constantes enumerables se pueden referenciar de igual modo que las constantes regulares, como se muestra a continuación:

```
int pantalla = Rojo;
int borde = Verde;
```

A.14. FUNCIONES EN C++

Al igual que en C, los programas C++ se componen de funciones. C++ introduce nuevas características para construir funciones más eficientes, seguras y legibles que sus equivalentes en C.

A.14.1. *main()*

C no define un formato específico para la función `main`. Así, es típico escribir la definición de la función principal:

```
void main (void)
{
      // código programa principal
}
```

Sin embargo, `main` de C++ puede tomar una de las siguientes formas (prototipos):

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

En el segundo formato, *argc* representa el número de argumentos pasados al programa C++ desde la línea de órdenes y *argv[i]* apunta al i-*ésimo* argumento de la línea de órdenes.

Al igual que en C, la terminación de la función `main` termina el programa. La función `main` puede terminar con una llamada a la función `exit` (con un argumento entero), alcanzando el final de su cuerpo, o ejecutando la sentencia `return`. El argumento de la función `exit` es el valor devuelto por el entorno. `exit` se llama normalmente con 1 para indicar fallo y con 0 si se indica éxito. La terminación de un programa por ejecución de la sentencia `return` es equivalente a la terminación del programa con la llamada a `exit`.

```
// archivo FUNC1.CPP

#include <iostream.h>

int leer_número(1)
{
  cout << "Escribir un número

  int n;
  cin >> n;                //lectura de n

  return n;
}
```

Puede existir más de una sentencia `return` en una definición de una función, como en este ejemplo:

```
// RETURN.CPP
#include <iostream.h>

char func(const int i)
{
    if(i == 0)
      return 'a';
    if(i == 1)
      return 'g';
    if(i == 5)
      return 'z';
    return 'c';
}

int main ()
{
    cout << "escribir un entero:";
    int val;
    cin >> val;
    cout << func(val) << endl; //daría un mensaje advirtiendo
                               //que no devuelve nada
}
```

En C++ cualquier otro formato de cabecera para `main` generará un error. La mayoría de los compiladores C++ también proporciona un error o un mensaje de advertencia si no devuelve un valor `main()`. Esta acción se puede realizar con una sentencia `return` o una llamada a la función `exit`. Dado que C++ fuerza a `main` a devolver un valor, es buena práctica de programación que la función `main` devuelva un valor.

A.14.2. Prototipos de funciones

Los prototipos encontrados en C ANSI han sido tomados de C + +. Un prototipo de función es una declaración que define el tipo devuelto y el tipo y número de parámetros de una función. El compilador utiliza el prototipo de la función para asegurar que al menos se proporciona el número correcto y los tipos de argumentos cada vez que se utiliza la función. En C tradicional, las funciones se declaran así:

```
int test();
```

En esta declaración no se dice nada de los tipos de parámetros aceptados por `test`. En C++ debe declararse `test` utilizando una sentencia similar a:

```
int test(char *cad, unsigned int long);
```

Esta declaración significa que `test` es una función que devuelve un valor entero `(int)` con dos parámetros, un puntero a carácter y un entero sin signo `(unsigned int)`. El compilador utiliza el prototipo para asegurar que los tipos de los argumentos que se pasan en una llamada a una función son los mismos que los tipos de los parámetros correspondientes. Esta propiedad se conoce como *verificación estricta de tipos*. Sin verificación estricta de tipos es más fácil pasar valores ilegales a funciones. Una función sin prototipo permitirá enviar un argumento entero `(int)` a un parámetro puntero o utilizar un argumento real `(float)` cuando espera un entero largo `(long)`. Estas clases de errores producen valores no válidos para los parámetros de la función. Por otra parte, cuando los tipos impropios se pasan a una función, el compilador no puede restaurar la cantidad correcta del espacio de pila y su pila se puede *romper*.

> Mientras que en ANSI C simplemante se utilizan prototipos, C++ los requiere.

Los prototipos de C++ no siguen exactamente las mismas reglas que C. En C++, este prototipo indica que `aleatorio()` es una función que no tiene argumentos y que devuelve un tipo entero.

```
int aleatorio ();
```

En C, sin embargo, el mismo prototipo indica que `aleatorio()` es una función que toma un conjunto de argumentos desconocidos y devuelve un entero `(int)`. Naturalmente, un prototipo de función de `aleatorio()` en C++ mejor que el anterior es:

```
int aleatorio(void);
```

En C++ no tenemos la facilidad de tener funciones tales como `printf` y `scanf()`, que toman diferentes números y tipos de argumentos. En C++ se utilizan puntos suspensivos para indicar qué números y tipos variables de argumentos son aceptables. El prototipo de C++ de `printf` es:

```
int printf(const char *, ... );
```

```
/* formato C */                 // formato C++
int demo1(void);                int demo1();
int demo2();                    int demo2(...);
```

Una función C++ que tiene una lista de parámetros abiertos requiere utilizar puntos suspensivos.

Recuerde que todas las funciones en C++ deben estar prototipadas. Esto significa que cada función debe tener su lista de argumentos declarados, y la definición real de una función debe corresponder exactamente con su prototipo en número y tipo de argumentos.

Los prototipos de funciones pueden significar un poco más de trabajo cuando se escribe inicialmente un programa; sin embargo, los prototipos pueden tener un valor inapreciable en prevenir errores difíciles de encontrar. C++ fue diseñado para prevenir muchos de los problemas del programador imprudente que pasa tipos incorrectos a métodos. Si realmente se necesita pasar un `long` como un `int`, se puede hacer utilizando un *molde* (conversión forzosa de tipos). Algunos prototipos completos de funciones:

```
int f1(void);   //devuelve un tipo int, no toma argumentos
void f2();      //como f4 () en c++ pero no en ANSI C
float f3(float, int,  char, double);      // devuelve float
void f4(void); // no toma argumentos, devuelve nada
```

Cabeceras de funciones tipo prototipo

```
/*estilo C              //estilo c++

void widget (a, b);     void widget (int a, double b)
  int a,
  double b;
{
  //...
}
```

El estilo de C++ es más fácil comprender que el viejo estiloC.

La Tabla A.4 resume las diferencias entre prototipos de funciones C y C++.

TABLA A.4. Prototipos C y C++

Prototipo	Formato C	Formato C++
`f (void);`	`int f (void)` *función devuelve* `int`	`void f (void);` *función no devuelve un valor*
`int f ();`	`int f () ;` *no verificar argumentos de función*	*no toma argumentos*
`int f(...);`	*no reconocida por la mayoría de los compiladores C*	`int f(...);` *no verifica argumentos de la función*

A.14.3. Una declaración típica de funciones y prototipos

El lenguaje C++ es más estricto que el ANSI C en los tipos de datos manipulados. En consecuencia, C++ impone declarar el número y tipo de argumentos que recibe una función, así como el tipo de valor devuelto. Estas declaraciones son similares a los prototipos de funciones de la norma ANSI, merced a los cuales el compilador puede verificar la coherencia de su código fuente entre las declaraciones de las funciones y sus llamadas efectivas. Sin embargo, mientras que la utilización de prototipos es opcional en ANSI C, son obligatorios en C++ para que un programa sea aceptado por el compilador.

```
// Archivo PROTO1.CPP

#include <iostream.h>

void discriminante(a, b, c);   //ERROR: antiguo estilo C
   double a, b, c;
{
   double d;
   d = b*b-4*a*c;
   visualizar(d);              //ERROR: ningún prototipo
                               //antes de utilizar
}

void visualizar(d)
   double d;
{
   cout << "Discriminante =" << d << endl;
}

void main ()
   cout << "Teclear a b c";
   double a, b, c;
   cin >> a >>b >> c;
   discriminante(a, b, c);
{

//Archivo PROTO2.CPP

#include <iostream.h>

void visualizar(double d);     //declaración de prototipo

//Estilos C++ y C ANSI

void discriminante(double a, double b, double c);

{
   double d;
   d = b*b-4*a*c;
   visualizar(d);              //válida
}

// Utilización de estilo C++ y C ANSI

void visualizar(double d)
{
   cout << "Discriminante ="   << d << endl;
}

void main ()
{
   cout << "Teclear a b c
   double a, b, c;
   cin >> a >> b >> c;
   discriminante(a, b, c);
}
```

La declaración de un prototipo de función no debe necesariamente llevar los nombres de los argumentos formales de la función, sino únicamente sus tipos. Así, las dos declaraciones siguientes son equivalentes:

```
// prototipo con nombres de argumentos formales
void discriminante(double a, double b, double c);

// prototipo sin los nombres de los argumentos formales
void discriminante(double, double, double);
```

A.14.4. Funciones en línea

En C se utilizan definiciones de macros para crear segmentos cortos de código. En C++ se incorporan funciones en línea que son similares a macros. Las funciones en línea instruyen al compilador para sustituir el cuerpo completo de la función en cada llamada a la función en línea.

Cuando una línea de la cabecera de la definición de una función contiene la palabra **`inline`**, esa función no se compila como una parte independiente del código llamado. En su lugar, la función se inserta siempre que una llamada a esa función aparece en un programa. Por ejemplo, C++ compila esta función como un código en línea.

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

Esta función no existe realmente como una función que se pueda llamar. En su lugar, el compilador inserta el código que realiza su tarea siempre que aparece una llamada a `suma` en su programa. Los traductores de C crean macros para las funciones en línea e insertan esas macros en el programa C resultante y apunta dónde es llamada la función. Un compilador C++ precompila la rutina e inserta las instrucciones precompiladas en las posiciones apropiadas.

Existen algunas reglas para utilizar funciones en línea. Las funciones en línea deben ser definidas antes de que puedan ser utilizadas. Esto se debe a que el código para la función en línea debe ser precompilado antes que se

pueda insertar en el programa. Por consiguiente, el siguiente código no se compilará como se esperaba.

```
#include <stdio.h>
int suma(int a, int b);

int main()
{
  int x = suma(1,2);
  printf("%i\n",x);
}

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

Cfront, el traductor estándar C ++ de AT&T, proporcionará un error que indica se ha declarado y utilizado *suma* como una función real, pero se ha definido como una función en línea. Se deben definir siempre sus funciones en línea antes de ser referenciadas.

En esencia, la función en línea aumenta el tamaño del programa, ya que se ha insertado el código de la función en lugar de generar una llamada a la función. Sin embargo, las funciones en línea aumentan su eficiencia, ya que incrementan su velocidad de ejecución.

Dado que el cuerpo de la función en línea se duplica siempre que se llama a la función, se deben utilizar funciones en línea *sólo* cuando las funciones sean de tamaño muy limitado (algunos bytes) y sean críticas en términos de velocidad de ejecución.

Un buen ejemplo de funciones en línea aparece en la definición de la clase `complex`, que se incluye con la mayoría de los compiladores C++ y aparece en el archivo de cabecera *<complex.h>*.

Ejemplos de funciones en línea

```
//Archivo : Mathlocal.h
//Prototipos (declaraciones de funciones)

inline int alos (int i);
inline int min (int v1, int v2);
int mcd (int v1, int v2);
```

Las funciones declaradas en línea deben ser sencillas, con sólo unas pocas sentencias de programas. Deben ser llamadas sólo un número limitado de veces y no deben ser recursivas.

Las funciones miembro declaradas con una definición en la especificación de la clase se consideran automáticamente en línea por el compilador:

```
class Rect
public:
   Rect ();
   ~Rect ();
   int izquierda()            { return x; }
   int derecha()              { return x + anchura; }
   int cima()                 { return y; }
   int fondo()                { return y + altura;}
private:
   int x, y;
   int anchura, altura;
};
```

A.14.5. Ventajas sobre las macros

Las funciones en línea son como macros del preprocesador, ya que el compilador sustituye el cuerpo de la función completo para cada llamada a una función en línea. Las funciones y las macros difieren esencialmente en aspectos fundamentales:

> «Al contrario que a las macros, el compilador trata a las funciones en línea como funciones verdaderas.»

Para ver cómo puede ser esto un factor importante, considérese el siguiente ejemplo. Supongamos que se ha definido una macro llamada `multiplicar` de la forma siguiente:

```
#define multiplicar (x,y) (x*y)
```

si tuviera que utilizar esta macro:

```
x = multiplicar(4+1,6);   //se pretende buscar 4+1 por 6
```

el preprocesador transformará el lado derecho de esta sentencia en el código siguiente:

```
x= (4+1*6);
```

esta expresión se evalúa a 10 en lugar de proporcionar el resultado de (4 + 1) por 6, que será 30; naturalmente, una solución a este problema es utilizar paréntesis. Sin embargo, considere lo que sucede cuando se define una función en línea igual que una macro.

```
#include <stdio.h>

// definir funciones en línea para multiplicar dos enteros
```

```
inline int multiplicar(int x, int y)
{
   return(x*y)
}

//versión sobrecargada que multiplica dos doubles

inline double multiplicar(double x, double y)
{
   return(x*y);
}

main()
{
   cout << "Producto de 5 y 6=" << multiplicar (4 + 1, 6) << endl;
   cout << "Producto de 3.1 y 10.0 =" <<
           multiplicar(3.0 + .1, 10.0) << endl;
}
```

La ejecución produce la salida:

```
producto de 5 y 6 = 30
producto de 3.1 y 10.0 = 31.000.000
```

Las funciones en línea evitan los errores de las macros, aumentan la eficiencia y además pueden ser sobrecargadas.

A.14.6. Argumentos por omisión

En C++ existe otra mejora a las funciones en C, es que se pueden especificar los valores por omisión para los argumentos cuando se proporciona un prototipo de una función. Es decir, determinados argumentos se pueden omitir e inicializar a un valor por defecto u omisión, y a continuación omitir el argumento real en la llamada. La sintaxis es la siguiente:

```
int defecto(int a = 5, int b = 6);
```

Por consiguiente:

```
una llamada a  defecto()       es igual que  defecto(5,6)
una llamada a  defecto(10)     es igual que  defecto(10,6)
una llamada a  defecto(10,20)  es igual que  defecto(10,20)
```

Los argumentos por omisión son los últimos de la lista solamente. Supongamos otro prototipo:

```
void g(int i=1, int j=2, int k=3);
```

y analicemos diversas llamadas como ejemplos suplementarios:

```
g(5,6,7);  es correcto (no se tienen en cuenta los valores
           por omisión).
g(5,6);    es igualmente correcto, k toma automáticamente
           el valor 3.
g(5);      es correcto; j y k toman automáticamente los
           valores 2 y 3.
g();       es correcto; i, j, k toman respectivamente los
           valores 1, 2 y 3.
g(,7,8);   error de compilación.
g(,2);     error de compilación.
```

Examinemos el ejemplo siguiente. Se define una función `crear_ventana` que establece una ventana (una región rectangular) en una pantalla de gráficos y se rellena con un color de fondo, que se puede fijar a valores específicos por omisión para la posición, tamaño y color del fondo de la ventana, como sigue:

```
//Una función con valores con argumentos por omisión
//Supongamos que ventana es un tipo definido por el usuario

ventana crear_ventana(int x=0, int y=0, int anchura=100,
                      int altura=50, int pixel=0);
```

Con esta declaración de la función `crear_ventana` se puede utilizar cualquiera de las llamadas siguientes para crear nuevas ventanas:

```
ventana v;

//La siguiente llamada equivale a
//crear_ventana(0,0,100,50,0)
v=crear_ventana();

//La siguiente llamada equivale a
//crear_ventana(100,0,100,50,0);
v=crear_ventana(100);

//La siguiente llamada equivale a
//crear_ventana(30,20,100,50,0);
v=crear_ventana(30,20);
```

Obsérvese que es imposible un valor especificado para el argumento altura sin especificar los valores de `x`, `y` y `anchura`, ya que altura viene después de ellos y el compilador sólo puede corresponder argumentos por posición. En otras palabras, el primer argumento que se especifica en una llamada a `crear_ventana`, siempre se corresponde con `x`, el segundo se

corresponde con y, etc. Por consiguiente, sólo se pueden dejar los argumentos finales de la lista sin especificar, y además se tienen que inicializar todos de derecha a izquierda y consecutivamente:

```
//prototipo con error
void mackoy(int a=1, int b, int c=3, int d=4);

//prototipo válido
void makena(int a, int b=2, int c=3, int d=4);
```

Veamos algunas llamadas a las funciones:

```
// conforme, args. para cada parámetro
makena(10,15,20,25);

// correcto, los dos últimos parámetros toman el valor por omisión
makena(12,15);

// error, se saltan parámetros deben ser consecutivos
makena(3,10,,12);
```

Un último ejemplo es el siguiente:

```
#include <stdio.h>
void f(int i, int j=2);  //se especifica valor por defecto

main()
{
  f(4,5);
  f(6);
}

void f(int i, int j)
{
  cout << "i:" << i  << "y j:" << j << endl;
}
```

La salida es:

```
i : 4 y j : 5
i : 6 y j : 2
```

El programa ARGOMI.CPP utiliza argumentos por omisión en las llamadas sucesivas a la función visualizar.

```
//Prueba de argumentos por omisión: ARGOMI.CPP
//Programa C++ que impone un array con diferentes elementos
//por línea
//C++ no permite inicializar arrays o estructuras local
//cuando se declaran

#include <iostream.h>
int a[10] = {81,78,45,78,91,45,76,87,54,99};

main()
{
  void visualizar (const int a[], int n, int cuenta = l);

  cout << "Array modelo, uno por línea. \n";
  visualizar(a, 10);
  cout << "Array modelo, tres por línea. \n";
  visualizar(a, 10, 3);
  return 0;
}

void visualizar(const int a[], int n, int cuenta)
{
  int i;
  for (i = 1; i <= n; i++)
  {
      cout << a[i-11];
      if (i%cuenta == 0 ||i == n)
        cout << "\n";
      else if (i! = 0)
        cout <<" ";
  }
}
```

Comentario

Primer argumento (array a visualizar). Segundo argumento (número de elementos del array). Tercer argumento (número de elementos a visualizar en cada línea, por omisión = l).

A.14.7. Funciones con un número variable de parámetros (el parámetro...)

A veces es imposible listar el tipo y número de todos los argumentos que se pueden pasar a una función. En estos casos, los puntos suspensivos («...») se pueden especificar dentro del formato de la función.

En C++ se pueden introducir los puntos suspensivos en la declaración de parámetros formales de una función para indicar que la función se lla-

mará con diferentes conjuntos de argumentos en ocasiones diferentes. Así, la declaración siguiente:

```
void func(int n, char car, ...);
```

indica que `func` se definirá de tal forma que las llamadas deben tener al menos dos argumentos, uno `int` y otro `char`, pero pueden tener también cualquier número de argumentos adicionales. En C++ se puede omitir la coma que precede a los puntos suspensivos.

Los puntos suspensivos pueden tomar cualquiera de los dos formatos en las correspondientes llamadas:

```
func(lista_arg,...);          //la coma es opcional
func(...);
```

Esta característica se encuentra en C, aunque normalmente no se piensa en ella. Sin embargo, un ejemplo claro de aplicación de esta propiedad se encuentra en la función `printf()`, que permite utilizar funciones con un número variable de parámetros en C. C++ soporta también esta función, cuyo prototipo recordemos es

```
printf(const char*...);
```

Las siguientes dos declaraciones *no* son equivalentes:

```
void f()
void f(...);
```

En el primer ejemplo, `f()` se declara como una función que no toma argumentos; en la segunda, `f(...)` se declara como una función que puede tomar cero o más argumentos. Las llamadas

```
f(valor);
f (xyz, a, b, c);
```

son invocaciones legales *sólo* de la segunda declaración. La llamada

```
f();
```

es una invocación legal de ambas funciones.

Un uso típico de la sintaxis puntos suspensivos implica una declaración de funciones en la que un conjunto de parámetros está seguida por el especificador de parámetros variables. Por ejemplo,

```
int ejemplo(int, int, double...)
```

especifica una función que requiere dos argumentos enteros y un argumento `double` y un número adicional de argumentos que es desconocido en tiempo de compilación.

Un buen ejemplo de funciones con números variables de argumentos es la familia de funciones `printf` definidas en el archivo de cabecera *<stdio.h>*.

```
int fprintf(FILE *flujo, const char *  formato, ...);
int printf(const char *formato, ...);
int sprintf(char *buffer, const char *formato, ...);
```

A.15. LLAMADA A FUNCIONES C. PROGRAMAS MIXTOS C/C++

La directiva `"extern c"` permite utilizar funciones compiladas con ANSI C. Para indicar al compilador C++ que las funciones han sido compiladas, según el convenio ANSI C, se deben seguir las siguientes reglas:

1. Prototipo de función precedido por `extern "C"`:

   ```
   extern "C" int sort(int a[], int n);
   ```

2. Las sentencias `include` se encierran en un bloque `extern C`:

   ```
   extern "C {                        extern "C" {
     #include "cmplx.h."               typedef struct {
     #include "fourier.h"              float r, i;
                                                } complex;
                                                 complex
   nuevocomplejo(float x, float y);
                                      float

   real(complex a);                   float
                                      float
   imag (comp1ex a);
   }
   ```

3. Especificar varias funciones con el modificador `"C"`.

   ```
   extern "C" {
     char *memcpy(char *t, char *s);
     int strlen(char *t);
   }
   ```

La mayoría de los compiladores C++ del mercado disponen de archivos `#include` concebidos para poder ser utilizados indistintamente a partir de programas en lenguaje C y C++.

Comprobación: véase archivo *studio.h* en \BORLAND\INCLUDE; están construidos según el modelo:

```
#ifdef _cplusplus
extern "C" {
#endif
void _Cdecl...
...
#ifdef _cplusplus
}
#endif
```

A.16. EL TIPO REFERENCIA

En C, cuando se llama a una función con argumentos, los valores de los argumentos se copian en una área especial de memoria conocida como la *pila*. La función utiliza estas copias para su operación. Este efecto se denomina *llamada por valor*. Para ver este efecto, consideremos el siguiente código:

```
void dosveces(int a)
{
  a *= 2;
}

...
int x = 5;

// llamada a la función dosveces

dosveces (x) ;
printf("x=%d\n",x);
```

Este programa imprime 5 como valor de x, no 10, pese a que la función dosveces multiplica su argumento por 2. Esto se debe a que la función dosveces recibe una copia de *x* y cualquier cambio que haga a esa copia se pierde al retornar desde la función.

En C, el único método para cambiar el valor de la variable a través de una función es pasar explícitamente la dirección de la variable a la función. Así por ejemplo, para doblar el valor de una variable, se puede escribir la función dosveces de la forma siguiente:

```
void dosveces (int *a)

{
  *a *= 2;
}
...
int x= 5;

// llamada a dosveces con la dirección de x como argumentos
dosveces(&x); printf("x=%d\n",x);
```

Esta vez el programa visualiza 10 como resultado. Por consiguiente, se pueden pasar punteros para modificar variables a través de una llamada a función, pero la sintaxis es poco clara. En la función se tiene que desreferenciar el argumento utilizando el operador *.

C++ ha añadido un tipo de dato referencia que proporciona un medio de pasar argumentos por referencia introduciendo el concepto de una *referencia,* que en esencia consiste en definir un alias o nombre alternativo para cualquier instancia de datos. La sintaxis es añadir un ampersand (&) al nombre del tipo de dato. En una declaración se utiliza el ampersand para especificar el tipo de dato referencia, de igual forma que el asterisco se utiliza para declarar un puntero:

```
double d = 0;
double &dr = d;   //dr es una referencia a d
double *dp;       //dp es un puntero a double
dr += 5.0;        //sumar 5.0 a dr
dp = &dr;         //apunta dp a dr

int i = 5;
int *P-i = &i;    //un puntero a int inicializando para
                  //apuntar a i
int &r_i = i;     //una referencia a la variable int i
```

De hecho, se puede utilizar *r_i* en cualquier parte que utilice *i* o **p_i*. Así, se escribe:

```
r_i + = 10;                    //suma 10 a i
```

i cambiará a 15, ya que *r_i* es simplemente otro nombre para *i*.

Utilizando tipos referencia, se puede reescribir la función dosveces para multiplicar un entero por 2 de un modo más simple:

```
void dosveces(int  & a)
{
  a *= 2;
}
int x = 5;

// llamada pasada por referencia

dosveces (x);
cout << "x=" << x << endl;
```

El siguiente programa contiene un procedimiento trivial denominado *max* `()` , que toma parámetros referencia y devuelve una referencia a uno de ellos.

```
#include <iostream.h>
int& max(int &a, int &b)
{
  if (a>b)
    return a; return b;
}

void main
{
  int x = 20, y = 30;
  max(x,y)--;
  cout << x << y << endl;
}
```

La salida de este programa es:

```
20      29
```

El mismo efecto se hubiera podido realizar también con punteros, pero no había sido tan legible.

```
int i;
incremento (i);
...
void incremento (int & variable_referencia)
{
variable_referencia++;
}
```

Una referencia ordinaria se debe inicializar con alguna variable del tipo apropiado. Por ejemplo, una referencia a un tipo `double` se debe inicializar con una variable `double`:

```
double di;
double &rd1 = di;    //conforme, rd1 es un alias de di
```

no con una `(const double)`:

```
const double d2 = 3.0;
double &rd2 = d2;   //error
```

ni con una constante `double`.

```
double &rd3 = 4.0;   // error
```

La función típica `intercambio` empleada en ordenación de datos diseñada en C es:

```
// estilo C

void intercambio(int  *a, int *b)
{
  int aux;

  aux = *a;
  *a = *b;
  *b = aux;
}
...

//llamada a la función intercambio

intercambio(&i1, &i2);
```

Una versión C++ de `intercambio` será:

```
// estilo C++

void intercambio(int &a, int &b)
{
  int aux = a;
  a = b;
  b = aux;
}
...

//llamada a la función intercambio

intercambio (i1,i2);
```

Un programa completo que hace uso de los parámetros referencias a través de la función es:

```
//Programa C++ que utiliza una función intercambio
//para intercambiar dos valores int

#include <iostream.h>

int main()
{
  void intercambio(int &xptr, int &yptr);
  int s = 10, t = 20;

  cout << "Antes de intercambio, s="  << s << t=", t=" << t << endl;
```

```
  intercambio(s,t);
  cout << "Despues de intercambio, s=" << s << ", t="  << t << endl
  ;
  return 0;
}

void intercambio(int &x, int &y)
{
  int aux;

  aux = x;
    x = y;
    y = aux;
}
```

C++ pasa automáticamente la información de la dirección de *i1* e *i2* como argumentos a la función `intercambio`.

> La diferencia entre un puntero a algo y una referencia a algo es que el puntero necesita ser desreferenciado y la referencia no.

Cuando se usan como parámetros función, las referencias C++ son similares a parámetros **`var`** en Pascal y Modula-2. Las referencias son particularmente útiles cuando se pasan estructuras y objetos grandes a una función. Utilizando una referencia para un parámetro sólo se pasa la dirección, no la estructura completa o el objeto. Esta acción no sólo ahorra espacio de pila y tiempo, sino que facilita el uso de los parámetros estructura/objeto dentro de la propia función.

Los parámetros referencia son eficientes y se deben por eficiencia utilizar siempre que estructuras grandes se pasen a funciones. A veces se desea pasar algo por referencia, incluso aunque no se desee modificar el valor. Esto se puede hacer con el calificador `const`:

```
void ref(const int &datosuno);
```

La función `ref` aceptará un valor entero *(datosuno)* pasado por referencia, pero, dado que tiene definido `const`, significará que no se puede cambiar. El calificador `const` permite utilizar referencias eficientes sin sacrificar integridad de los datos.

Como ya se ha comentado, aunque las referencias son muy similares a los punteros, no son punteros. No se pueden utilizar para asignar memoria dinámica, ni se pueden manipular matemáticamente. El propósito final de las referencias es permitir que las funciones cambien sus argumentos y que acepten estructuras y objetos como parámetros de un modo más claro.

El siguiente programa pasa un tipo de dato estructura por referencia:

```
//programa demo referencia: visualiza una estructura
//utilizando referencias para pasar la estructura
//programa C++ que visualiza información de empleado

#include <stdio.h>

struct empleado
{
  long numero;    //identificación de empleado
  char *nombre, *telefono;     //nombre y teléfono
  int edad;                    //edad del empleado
};

int main()
{
  void ver_emp(const empleado  &emp);
  empleado emp;
  emp.numero = 1001;           //llenar información
  emp.edad = 46;
  emp.nombre = "Mark Mackoy";
  emp.telefono = "91-4374220";
  ver_emp(emp);                //escribir empleado
  return 0;
}

void ver_emp (const empleado &e) //referencia a empleado
{
  cout << "Empleado:" << e.numero << endl;
  cout << "Nombre:" << e.nombre« endl;
  cout "Edad:" e.edad << endl;
  cout << "Telefono:" << e.telefon << endl;
}
```

A.17. SOBRECARGA

C++ añade dos propiedades muy importantes a la capacidad de una función. Se pueden tener múltiples funciones con igual nombre mediante un procedimiento denominado *sobrecarga*. Además, C++ permite definir sus propias funciones para operadores simbólicos tales como +, –, o bien *.

La sobrecarga es un concepto con doble sentido. Su tarea más evidente es permitir que el mismo nombre de una función sea utilizado por diferentes implementaciones de la función. Menos evidente, pero no por ello menos importantes es la característica de sobrecargar operadores creando funciones que se invocan cuando se utiliza un operador simbólico. Ambas

características son muy potentes y exigen un conocimiento juicioso y prudente por parte del programador.

A.17.1. Sobrecarga de funciones

En C, como en la mayoría de los lenguajes de computadoras, cada función debe tener un único nombre. A veces esto puede ser enojoso. Por ejemplo, en C existen diferentes funciones que devuelven el valor absoluto de un argumento numérico. Dado que dichas funciones realizan tareas similares pero trabajan con diferentes tipos de datos, se requieren, pues, tres funciones diferentes con nombres distintos:

```
int abs(int i);
long labs(long l);
double fabs(double d);
```

C ++ resuelve esta paradoja, permitiendo crear tres funciones diferentes con el mismo nombre. Esta es la propiedad denominada *sobrecarga*. Por ejemplo, en C++ se pueden definir estas tres funciones para declarar la función `abs`:

```
int abs(int x);
long abs(long 1);
double abs(double d);
```

El compilador C++ selecciona la función correcta comparando los tipos de argumentos en la llamada con los que se especifican en la declaración de función. Así pues, se pueden utilizar las funciones anteriores como sigue:

```
int i, difer = -5;
long desplazar;
double x;

i = abs(difer);                 //se llama a abs(int)
desplazar = abs (-21956L);      //se llama a abs(long)
x = abs (-455);                 //se llama a abs(double)
```

Otro ejemplo de función sobrecargada es `sort`, que ordena arrays o listas de diferentes tipos:

```
void sort(int a[], int n);
void sort(float a[], int n);
void sort(char *a[], int n);
void sort (int a[]);
```

Cuando se sobrecargan funciones en C++ se tiene que asegurar que el número y tipo de argumentos de todas las funciones sobrecargadas son diferentes. C++ no permite sobrecarga de funciones que difieran sólo en el tipo de valor devuelto. Por consiguiente, no puede sobrecargar funciones tales como

```
double calcular(int)          float calcular(int)
```

ya que su lista de argumentos es idéntica.

> Las funciones sobrecargadas deben diferir al menos en el tipo o número de parámetros que aceptan.

El siguiente programa implementa cuatro funciones `suma`, cada una de ellas sobrecargada para realizar operaciones de suma de enteros, reales, cadenas (concatenación) y suma de un entero y una constante (42), todas ellas con dos argumentos, excepto la última, que sólo tiene un argumento:

```
#include<string.h>

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

double suma(double a, double b)
{
  return a+b;
}

char *suma(char *a, char *b)
{
  strcat(a,b);
  return a;
}

int suma(int i)
{
  return i + 42;
}

main()
{
  int i = suma (42, 17);
  double d = suma(42.0, 17.0);
  char s1 [] = "abc";
  char s2 [] = "xyz";
  suma (s1, s2);
}
```

Asegúrese que los tipos diferentes de los parámetros de las funciones sobrecargadas son realmente tipos diferentes. Un tipo `typedef` es simplemente un alias de un tipo existente y no constituye un tipo original de su propiedad. Así, el siguiente segmento de código no es correcto:

```
typedef INT int;

// ambos prototipos son idénticos
void deferror(int x);
void deferror(INT x);
```

El ejemplo anterior no se compilará correctamente, debido a que el compilador no tiene medio de diferenciar entre las dos versiones de `deferror`. Un `INT` es sólo otro nombre para un `int`. `const` y punteros referencia en una función pueden diferir en el uso del calificador `const`, como sucede en este fragmento de código:

```
void func(char *ch);
void func(const char * ch);

int main()
{
  const char c1 = "a";
  char c2 = "b";

  func(&c1);     //llamada a void func(const char *ch);
  func(&c2);     //llamada a void func(char *ch);
}
```

Es muy mala práctica de programación tener funciones sobrecargadas que realizan acciones diferentes. Las funciones con igual nombre deben tener el mismo propósito general. Crear una implementación de la función `abs` que devuelva la raíz cuadrada de un número le producirá con seguridad confusión y muchos quebraderos de cabeza.

Le sugiero que utilice las funciones sobrecargadas con moderación. Su propósito final es proporcionar un nombre nemotécnico para funciones similares pero con estructuras ligeramente diferentes. El uso abusivo de funciones sobrecargadas puede hacer un programa ilegible.

A.17.2. Aplicación de sobrecarga de funciones

La función `visualizar` escribe cualquier tipo de objeto en la salida estándar: una versión visualiza un tipo `empleado`, otra versión una cadena, otra un tipo `int`, etc. Para conseguir esta característica se sobrecarga la citada función.

```
//Programa C++ que visualiza diferentes objetos con
//sobrecarga de funciones: archivo ver.cpp

#include <stdio.h>

struct empleado
{
  long numero;                  //número de empleado
  char *nombre, *telefono;      //nombre y número de teléfono
  int edad;                     //edad de empleado
};

void visualizar(const empleado &e); //visualizar prototipos
void visualizar(char *s);
void visualizar(int i);
void visualizar(long 1);

int main()
{
  empleado emp;
  emp.numero = 1001;
  emp.edad = 46;                //rellenar info de empleado
  emp.nombre = "Mark Mackoy";
  emp.telefono = "99-4374220";
  visualizar(emp);              //escribir un empleado
  return 0;
}

void visualizar(const empleado &e)
{
  visualizar("Empleado: "); visualizar(e.numero);
    visualizar("\n");
  visualizar("Nombre:   "); visualizar(e.nombre);
    visualizar("\n      ");
  visualizar("Edad:     "); visualizar(e.edad);
    visualizar("\n");
  visualizar("Telefono: "); visualizar(e.telefono)
    visualizar("\n");
}

void visualizar(char *s)
{
    cout *s;
}

void visualizar(int i)
{
  cout << i;
}

void visualizar(long 1)
{
  cout << 1;
}
```

Cuando se llama a `visualizar`, el compilador decide a qué 'versión' llamar examinando los tipos de los parámetros que se pasan. Con `visualizar(emp)`, por ejemplo, se pasa un tipo `empleado`, de modo que se ejecuta la versión `visualizar` que espera un empleado. Sin embargo, con `visualizar (e.edad)` se pasa un `int`, así que se ejecuta la versión de `visualizar` que espera un tipo `int`.

La aplicación anterior se puede mostrar con más claridad recurriendo a implementar el archivo de especificación `empleado.h`, el archivo de implementación `empleado.cpp` y una función principal que hace uso de dichos archivos.

```
//FICHERO: empleado.h

#ifndef EMPLEADO
#define EMPLEADO

struct empleado {
  long numero;
  char * nombre;
  char * telefono;
  int edad;
};

void visualizar(const empleado & e);
void visualizar(const char * s);
void visualizar(const int i);
void visualizar(const long l);

#endif

//FICHERO(ARCHIVO): empleado.cpp

#include "empleado.h"
#include <stdio.h>

void visualizar(const empleado & e){
         visualizar("Número:");
         visualizar(e.numero);
         visualizar("\n");
         visualizar("Nombre:");
         visualizar(e.nombre);
         visualizar("\n");
         visualizar("Teléfono:");
         visualizar(e.telefono);
         visualizar("\n");
         visualizar("Edad:");
         visualizar(e.edad);
         visualizar("\n");
}

void visualizar(const char * s) {
  cout << s;
}

void visualizar(const int i){
  cout << i;
}

void visualizar (const long 1) {
  cout << 1;
}

//FICHERO (ARCHIVO): principa.cpp

#include "empleado.h"

int main() {
  empleado emp;

  emp.numero = 1001;
  emp.edad = 46;
  emp.nombre = "Mark Mackoy";
  emp.telefono = "91-437422011;

  visualizar(emp);

  return 0;
}
```

A.17.3. Sobrecarga de operadores

Al igual que C++ permite definir diferentes funciones con el mismo nombre pero con variación en sus argumentos, también permite redefinir el significado de operaciones tales como +, –, *, %, + =, – =, ..., para cualquier clase. En otras palabras, se puede sobrecargar el significado de los operadores. La sobrecarga de operadores es un concepto interesante que en la mayoría de los lenguajes existe, aunque adquiere su máxima potencia en C + +

Así por ejemplo, se pueden sumar dos enteros utilizando el operador +:

```
a = 5 + 2;
```

y también se pueden sumar dos números reales `(float)` utilizando el mismo operador:

```
a = 5.5 + 2.8;
```

Se utiliza el mismo operador para realizar la misma función sobre dos tipos de datos diferentes. El operador + está sobrecargado para realizar la suma de dos tipos enteros en el primer caso y la suma de dos tipos reales en el segundo.

Veamos otro ejemplo. El operador `*` se utiliza para multiplicar dos tipos de datos:

```
a = a * b;
```

Sin embargo, el operador * se puede también utilizar para especificar un tipo puntero cuando se declara:

```
int *ptr;
```

La sentencia anterior declara `ptr` como un puntero a un tipo entero. El operador `*` se puede utilizar también para desreferenciar una variable puntero a fin de manipular el contenido de lo que se almacena en la posición en memoria que apunta `ptr`:

```
*ptr = 5;
```

La sentencia anterior establece el contenido de la posición en memoria almacenada en `ptr` a 5. Este es otro caso de sobrecarga de operadores.

A.18. ASIGNACIÓN DINÁMICA DE MEMORIA

La asignación dinámica de memoria permite que un programa en ejecución pueda hacer uso del almacenamiento denominado montículo *(heap)*. La asignación dinámica de memoria proporciona a un programa gran flexibilidad, ya que el programa puede decidir durante la ejecución cuánta memoria se utiliza y para qué propósito. En C, la memoria se asigna mediante las funciones de biblioteca `malloc` y `free`. C++ define un nuevo método de hacer asignación dinárnica mediante los operadores `new` y `delete`. Los operadores `new` y `delete` permiten a los programadores controlar asignacion y liberación del almacenamiento tipo montículo.

A.18.1. El operador *new*

El operador `new` es más sencillo de utilizar que la función `malloc()` cuando los tipos de las variables a asignar dinámicamente se vuelven complejas. Compare en el ejemplo siguiente la sintaxis de las asignaciones de memoria que utilizan `malloc()` y la sintaxis de las que utilizan `new`.

El operador `new` puede ser sobrecargado y utilizarse de dos formas diferentes:

```
new tipo                          asigna un único item o elemento
new tipo[número de elementos]     asigna un array

char *car1, *car[40];

// Asignación de un carácter
car1 = (char*) malloc(1);         //Estilo C ANSI
car1 = new char;                  //Estilo C++

// Asignación de una cadena de 40 caracteres
car40 = (char*) malloc(40);       //Estilo C ANSI
car40 = new char[40];             //Estilo C++

double *doble1, *doble50;

// Asignación de un tipo doble
doble1 = (double*) malloc(sizeof(double));//Estilo C ANSI
doble1 = new double;                       //Estilo C++

// Asignación de un array de 50 elementos tipo double
doble50 = (double*) malloc(50 *sizeof(double));
                                  //Estilo C ANSI
doble50 = new double[50];         //Estilo C++

struct nudo {
               int valor;
               nudo *izquierda;
               nudo *derecha;
            };
nudo *nudo],                      *nudoN;

//Asignación de un nudo
nudo1 = (nudo*) malloc (sizeof(nudo)); //Estilo C ANSI
nudo1 = new nudo;                      //Estilo C++

// Asignación de N nudos
int N;
cout << "cuantos nudos:?";
cin >> N;
nudoN = (nudo*) malloc(N*sizeof(nudo)); //Estilo C ANSI
nudoN = new nudo[N]                     //Estilo C++
```

Declara una matriz de 20 enteros con `malloc`:

```
{
  int *p;
```

```
  ...
  p = (int *) malloc (20 * sizeof(int));
  ...
  free(p);
}
```

A diferencia de `malloc()`, que es una función, `new` es un *operador* que se aplica a un tipo de datos `T`. El operador new le reenvía un puntero sobre un bloque de memoria capaz de contener el número de objetos de tipo `T` requerido; la utilización del operador `sizeof ()` con `new` es inútil, pues `new` «conoce» automáticamente el número de bytes necesarios para cada objeto de tipo `T`. De igual forma, el puntero devuelto por `new` se asigna directamente a un puntero de tipo `T*`, sin que sea necesaria una conversión explícita. Al contrario que `malloc()`, el operador new efectúa una verificación de tipos:

```
nudo1 = new doble;// ERROR: conversión p(double*)->(nudo*)
```

Una función tradicional clásica que utiliza memoria dinámica es:

```
void func(void)
{
   int *i;

   i = (int*)malloc(sizeof(int));
   *i = 10;
   printf("%d",*i);
   free(i);
}
```

En C++, la función anterior se puede rescribir así:

```
void func()
{
   int *i = new int;
   *i = 10;
   cout << * i;
   delete i;
}
```

La sintaxis C++ es más clara y más fácil de utilizar. Para asignar un array de 10 enteros, utilice la sentencia siguiente:

```
int *i = new int[10];
```

Otra expresión de inicialización es

```
int *ptri = new int[0];
```

que asigna almacenamiento para un tipo entero `(int)`, inicializa el almacenamiento a 0 y a continuación guarda un puntero al almacenamiento en la variable `ptri`.

Método C

```
int *P=(int*) malloc (sizeof (int));
```

Método C++

```
int *P = new int;
```

Utiliza `new` y `delete` en lugar de `malloc()` y `free()` siempre que sea posible. Los operadores `new` y `delete` utilizarán verificación de tipos y trabajan en unión con constructores y destructores.

A.18.2. El puntero nulo/cero

Si `new` no puede asignar la cantidad solicitada de memoria, porque por ejemplo no queda bastante memoria disponible, ¿qué sucede? En este caso, la función `malloc()` devuelve un puntero `NULL` (constante definida en la biblioteca estándar de ANSI C que manipulan punteros: *<stdef.h>*, *<stdlib.h>*, *<stdio.h>*, etc.).

El operador `new` reenvía el valor 0 (cero). En efecto, la constante 0 (cero) es un valor legal para un puntero C + +; además, la constante 0 es compatible con todos los tipos de punteros; el puntero 0 juega el mismo papel en C++ que la constante `NULL` en ANSI C. El sistema para comprobar si la memoria solicitada ha sido asignada realmente es:

```
int *tab_10_ent = new int[10];
if(tab_10_ent == 0)
{
  // ... falta memoria
}
```

A.18.3. El operador *delete*

El operador `delete` libera la memoria asignada con `new`. Este operador funciona de modo similar a la función `free()` para la memoria asignada con `malloc()`. No se puede utilizar `free()` sobre un puntero inicializado con `new`, ni `delete` sobre un puntero inicializado con `malloc()`. Los formatos de `delete` son:

`delete` *`direccion`*	Libera un espacio previamente asignado por `new` a la dirección indicada.
`delete` *`[]direccion`*	Libera espacio asignado si es un array; versiones antiguas de C++ exigían *n* (número de elementos a destruir).

```
// archivo deletenew.cpp
void main()
{
  double *d;
  d = new double;       //Asignación de un solo objeto
  *d  = 12.34;
  delete d;             //Liberación de un solo objeto

  d = new double[10];   //Asigna un array de objetos
  for(int i=0; i>10; i++)
    d[i] = double(i) / 7.0;
  delete[]d;          //Liberación de un array de objetos
```

```
//Estilo C                      //Estilo C++
                                struc Paciente {
struct Paciente                   char nombre [20];
  char nombre [20]                char sexo;
  char sexo;                      unsigned id;
  unsigned id;                  };
};                              struct Paciente *p;
struct Paciente *p;             p = new Paciente;
p = (struct Paciente));         //...
malloc (size(Paciente));        delete p;
/* ... */
free (p);
```

El operador `delete` «simple» se utiliza para liberar la memoria asignada a un único objeto, y el operador `delete []` cuando se asigna un array o tabla de objetos. En este último caso no es necesario precisar el número de objetos inicialmente asignados.

Los operadores `new` y `delete` sustituyen ventajosamente a las funciones `malloc()` y `free()` para todos los problemas de asignación dinámica de memoria. Sin embargo, estos operadores alcanzan su mayor rendimiento en la gestión correcta de clases en C++, autorizando la inicialización de un objeto que pertenece a una clase después de la asignación dinámica de ese objeto.

A.18.4. Ventajas de *new* y *delete*

`new` es superior a `malloc` por varias razones:

1. `new` conoce cuánta memoria se asigna a cada tipo de variable.
2. `malloc` debe indicar cuánta memoria asignar.
3. `new` provoca que un constructor se llame para el objeto asignado; `malloc` no puede.

`new` es mucho más limpia que `malloc`. Con `malloc` se tiene que declarar explícitamente cuánta memoria asignar y se debe utilizar conversión forzosa de tipos `(cast)`. Con `malloc` es fácil producir errores, pero con `new` simplemente basta especificar el tipo de objeto que está creando y `new` realiza el resto.

```
int *p = new(int);         //llamada como una función
int *p = new int[10];      //declara un array de 10 enteros
```

Aunque `new` es útil para asignar memoria a variables de tipos incorporados, la potencia real reside en asignar objetos dinámicos que tienen constructores. Cuando `new` se utiliza para tales objetos, entonces se llama automáticamente uno de los constructores.

```
reloj *r = new reloj(12,30,45); // asigna e inicializa
```

`delete` libera memoria dinámica asignada por `new`:

```
int *p = new int;
delete p;
reloj *r = new reloj(4,15,55); //asigna reloj
delete reloj;                  //libera reloj
```

`delete` produce una llamada al destructor en este orden:

1. El destructor se llama.
2. La memoria se libera.

`delete` es más segura que `free()`, ya que protege al intentar liberar memoria apuntada por un puntero nulo.

A.19. ORGANIZACIÓN DE UN PROGRAMA C++

Un programa C++ se construye sobre la base de archivos `.cpp` y `.h`, estructurados sobre la base de un archivo principal de la función `main`, con extensión `.cpp`.

Archivo principal: `nomina.cpp`

```
#include "empleado.h"
...
main ()
{
  ...
}
```

Cabecera clase: `empleado.h`

```
class empleado{
  ...
};
```

Implementación de la clase: `empleado. cpp`

```
#include "empleado.h"
...
empleado :: empleado (int 1, char *n, float s)
{
  ...
}

void empleado :: ver_salario (float horas)
{
  ...
}
```

Archivo de cabecera de la clase

```
//empleado.h

#ifndef H_EMPLEADO
#define H_EMPLEADO

class empleado {
...
};
#endif
```

Archivo fuente de la clase

```
//empleado.cpp

#include <stdio.h>
#include <string.h>
#include "empleado.h"


empleado: :empleado ("int i, ...)
```

Archivo principal

```
//nomina.cpp

#include "empleado.h"
...
main()
{
...
}
```

A.19.1. Evitar definiciones múltiples

En cada archivo de cabecera que tiene una clase se debe verificar, en primer lugar, que el archivo no ha sido ya incluido en este archivo. Esto se realiza mediante un indicador (macro) del preprocesador. Si el indicador no está activado, el archivo no se incluye; se debe activar el indicador y declarar la clase. Si el indicador está activado, la clase ha sido ya declarada, de modo que se ignora el código que declara la clase. El archivo de cabecera será similar a éste:

```
#ifndef INDICADOR_CLASE
#define INDICADOR_CLASE
//Declaración de la clase
#endif INDICADOR_CLASE
```

Como se puede ver, la primera vez que el archivo de cabecera se incluye la declaración de la clase se incluirá por el procesador, pero se ignorará la definición de la clase en todas las siguientes ocasiones.

El nombre de la macro o indicador puede ser cualquier nombre único, pero un método fiable a seguir es tomar el nombre del archivo de cabecera de sustituir el punto de la extensión por un subrayado y añadir delante o después del indicador o macro una H (de *header)* y un subrayado, o cualquier otra palabra similar que recuerde el nombre del archivo de cabecera.

```
//archivo de cabecera: PRUEBA.H

#ifndef PRUEBA_H_
#define PRUEBA_H_
class uno {
  int i, j, k;
public :
  llamar() {i = j = k = 0;}
};

#endif //PRUEBA_H_
```

En resumen, un archivo de cabecera en C++ se construye normalmente así:

```
#ifndef NOMBRE_CLASE
#define NOMBRE_CLASE
//especificación de la clase
//que sólo incluirá el archivo fuente una vez
#endif
```

Esto permite que un archivo de cabecera se pueda incluir muchas veces en un programa, pero sólo se incluye el contenido del archivo una vez. Esto puede ocurrir, por ejemplo, en la siguiente situación, donde un programa utiliza las especificaciones siguientes de las clases para un Coche y una Persona, que contienen ambos la inclusión de una especificación de una clase para contener un nombre.

Especificación de la clase Persona

```
#include "nombre.h"
//utiliza la clase Nombre
class Persona {
  //especificación de la clase
};
```

Especificación de la clase Coche

```
#include "nombre.h"
//utiliza la clase Nombre
class Coche {
  //especificación de la clase
};
```

A.19.2. Evitar incluir archivos de cabecera más de una vez

Se deben utilizar en los archivos de cabecera líneas de directivas condicionales:

```
#ifndef _DEMO_H
#define _DEMO_H

//Declaraciones que se
incluyen una vez
...
#endif
```

```
//empleado.h

#ifndef H_EMPLEADO
#define H_EMPLEADO

class empleado{
private:
int id;
```

```
...
public:
empleado(int i, char *n, float w);

...
};
```

Existe el problema de tener el mismo archivo de cabecera incluido más de una vez en una compilación de archivos fuente. El problema con la inclusión de un archivo de cabecera más de una vez es que las definiciones que contiene se multiplicarán y en muchos casos esta circunstancia producirá errores de compilación.

Se puede evitar el problema de tener múltiples definiciones fácilmente. Para ello se puede utilizar en cada archivo de cabecera las directivas del compilador `#ifndef` , `#define` y `#endif` para colocar una protección a sus definiciones. El método es el indicado anteriormente para definir la clase `empleado`.

La idea básica consiste en definir constantes macros para cada archivo de cabecera. Estas constantes se denominarán guardianes de macros y se utilizarán para guardar contra compilaciones múltiples de archivos de cabecera.

La primera vez que se compila el archivo de cabecera `empleado.h`, el valor de la guardia de la macro `H_EMPLEADO` está indefinido. Por consiguiente, el archivo de cabecera completo se compilará incluyendo la sentencia que define un valor para `H_EMPLEADO`.

Si el archivo de cabecera se incluyera más de una vez en el mismo archivo fuente, o bien directa o indirectamente, el compilador verá que `H_EMPLEADO` está definido y saltará sobre el contenido del archivo de cabecera después de la primera inclusión.

```
#include <stdio.h>

#include <string.h>
#empleado.h Declaración clase
#ifndef H_EMPLEADO
#define H_EMPLEADO

class empleado{
private:
  int id;
  char nombre[40];
  float tasa;
public:
  empleado(int i, char *n, float s);
  void ver_salario(float horas);
};
#endif

//empleado.cpp, implementaciones
#include <stdio.h>
#include <string.h>
#include "empleado.h"

empleado :: empleado(int i, char *n, float s);
{
  id = i;
  strcpy(nombre,h);
  tasa = s;
}
void empleado :: ver_salario(float horas)
{
  cout<< "Empleado: " << id << ":" << nombre << endl;
  cout << "Horas trabajadas:" << horas << endl;
  cout << "Cantidad pagada:" << horas*tasa << endl;
}

#include "empleado.h"

int main ()
{
  empleado mortimer(1, "juan mortimer",1500);
  empleado mackoy(2, "pepe mackoy",2200);
  mortimer.ver_salario(400);
  mackoy.ver_salario(5200);
  return 0;
}
```

Una ejecución del programa anterior daría una salida similar a la siguiente:

```
Empleado:1:juan mortimer
Horas trabajadas: 400.000
Cantidad pagada: 600000.000

Empleado:2:pepe mackoy
Horas trabajadas: 5200.000
Cantidad pagada: 11440000.000
```

RESUMEN

C++ es una extensión del lenguaje C, que se conoció en sus orígenes con el nombre de «C++ *con clases*». El inventor de C++, Bjarne Stroustrup, denominó a C++ «*a better* C» (un C mejor).

La mayor atribución al mundo de la programación de C++ ha sido la *clase*. Además, C++ ha añadido a ANSI C nuevas palabras reservadas y operadores, funciones en línea y sobrecargadas, operadores sobrecargados, nuevas técnicas de gestión de memoria y otras características complementarias.

En la actualidad, la versión que ha sido estandarizada por el comité ANSI es la número 3, que soporta esencialmente las características de genericidad y manejo de excepciones.

ANSI C y C++ utilizan una sintaxis casi idéntica, así como los mismos tipos de bucles, tipos de datos, punteros y otros elementos.

Los compiladores más usuales son AT&T versión 3.0, Borland C++ 4.5 y 5 Turbo C++ 4.5 y Visual C++ 5.0/6.0, todos ellos siguen prácticamente la norma AT&T C ++ 3.0.

EJERCICIOS

A.1. Explicar cuál es la razón por la que el siguiente programa se detiene con un mensaje "Divide error":

```
//errordiv.cpp
#include <stdio.h>
int valor;
main ()
{
  int n = 100/valor;
  printf ("n = % dª n",k);
  return 0;
```

A.2. Escribir un programa que solicite un nombre, a continuación, visualice el mensaje «Hola señor/señora x», donde x es reemplazada por el texto introducido.

A.3. Escribir un programa que solicite un número entero con signo, a continuación visualice el número equivalente sin signo en código decimal y hexadecimal.

A.4. Escribir un programa que determine si un número entero es par o impar.

A.5. Escribir un programa que solicite un número entre 1 y 100. Si se introduce un número de valor fuera de rango, el programa debe visualizar un mensaje de error y volver a solicitar la introducción de un número.

A.6. En matemáticas, el factorial de n (Número entero no negativo) se define mediante la ecuación:

n! = 1 * 2 * 3 * ... * n;

Nota: El factorial de 0 es 1 (!0 = 1)
El factorial de 1 es 1 (!1 = 1)
El factorial de 2 es 2 (! 2 = 2 * 1 = 2)

A.7. Escribir funciones que calculen el volumen (V) y la superficie (S) de una esfera, utilizando las fórmulas:

$V = 4/3\pi^3$ $S = 4\pi r^2$

A.8. Una serie armónica se define mediante la serie:

$1+1/2 + \frac{1}{3} + ... + 1/n$

Escribir una función que devuelve el número de términos requeridos para cumplir la expresión [series armónicas >límite].

A.9. Escribir un programa que solicite repetidamente introducir parejas de números, hasta que al menos un número de la pareja sea cero. Por cada pareja, el programa debe utilizar una función para calcular la media armónica de los números.

Media armónica = 2.0 * x * y /(x + y)

A.10. Escribir una función que calcule la longitud de una cadena que es un argumento.

A.11. Diseñar y escribir una función que haga notar una serie de valores de una lista una posición a la derecha. Es decir, dados los valores 10, 11, 12, 13, 14, la llamada a la función cambia la lista a 14, 10, 11, 12, 13, una nueva llamada cambia a 13, 14, 10, 11, 12, etc.

A.12. Escribir una función que devuelva la moda de una serie de valores reales `(double)` almacenados en un array.

A.13. Declarar una estructura que represente un número de teléfono (código del país, código del área —prefijo provincial— y número). Declarar asimismo una estructura anidada con dos números de teléfono: una para comunicación por voz y el otro para fax.
Definir un array que contenga 50 estructuras del tipo número de teléfono.

A.14. Escribir una función que tome un argumento, la dirección de una cadena e imprima la cadena una vez. Si se proporciona un segundo argumento `int` y no es cero, la función imprime la cadena un número de veces igual al número de veces que la función ha sido llamada en ese punto.