Referencias y Punteros

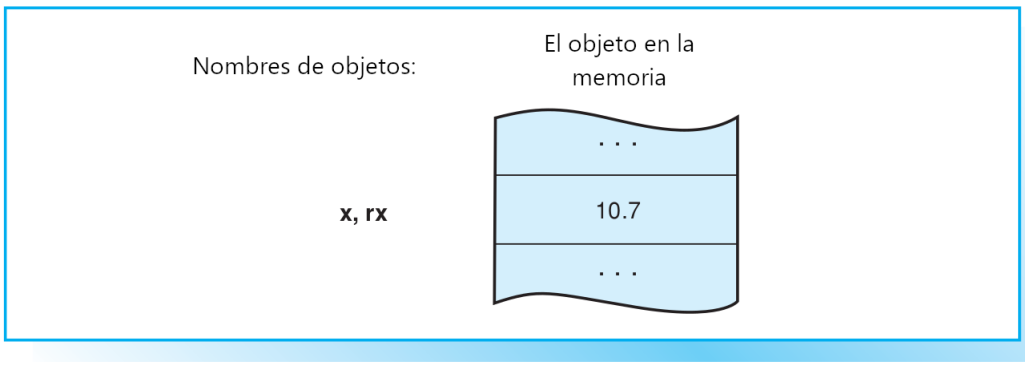
En este capítulo se describe cómo definir referencias y punteros y cómo utilizarlos como parámetros y/o valores de retorno de funciones. En este contexto, se introduce el paso por referencia y el acceso de solo lectura a los argumentos.

Referencias y Punteros

Definición de referencias

Ejemplo

```
float x = 10.7,     &rx = x;
```



Programa de ejemplo

```
// Ref1.cpp
// Demuestra la definición y el uso de referencias.
// ------------------------------------------------

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

float x = 10.7F;          // Global

int main()

{
    float &rx = x;        // Referencia local a x
//  double &ref = x;      // Error: ¡tipo diferente!

    rx *= 2;

    cout << "    x = " << x << endl    // x = 21.4
         << " rx = " << rx << endl;    // rx = 21.4
    const float& cref = x;             // Referencia de solo lectura
    cout << "cref = " << cref << endl; // ok!
//  ++cref;                            // Error: ¡sólo lectura!
    const string str = "¡Soy una cadena constante!";
//  str = "¡Eso no funciona!";         // Error: str constante!
//  string& text = str;                // Error: str constante!
    const string& text = str;          // ok!
    cout << text << endl;              // ok! Sólo leyendo.
    return 0;
}
```

Una referencia es otro nombre, o alias, para un objeto que ya existe. Definir una referencia no ocupa memoria adicional. Cualquier operación definida para la referencia se realiza con el objeto al que hace referencia. Las referencias son particularmente útiles como parámetros y valores de retorno de funciones.

Definición

El carácter & se utiliza para definir una referencia. Dado que `T` es un tipo, `T&` denota una referencia a `T`.

Ejemplo:

```
float x = 10.7;
float& rx = x;      // o:     float &rx = x;
```

`rx` es, por tanto, una forma diferente de expresar la variable `x` y pertenece al tipo "referencia a `float`". Operaciones con `rx`, como

Ejemplo:

```
--rx;     // equivalente a --x;
```

afectará automáticamente a la variable `x`. El carácter `&`, que indica una referencia, solo aparece en declaraciones y no está relacionado con el operador de dirección `&`! El operador de dirección devuelve la dirección de un objeto. Si aplica este operador a una referencia, devuelve la dirección del objeto referenciado.

Ejemplo:

```
&rx     // Dirección de x, por lo tanto es igual a &x
```

Una referencia debe *inicializarse* cuando se declara y no puede modificarse posteriormente. En otras palabras, no puede utilizar la referencia para direccionar una variable diferente en una etapa posterior.

Referencias de solo lectura

Una referencia que direcciona un objeto constante debe ser una constante en sí misma, es decir, debe definirse utilizando la palabra clave const para evitar modificar el objeto por referencia. Sin embargo, es posible, a la inversa, utilizar una *referencia a una constante* para direccionar un objeto no constante.

Ejemplo:

```
int a;     const int& cref = a;     // ok!
```

La referencia `cref` se puede utilizar para acceder en modo de solo lectura a la variable a, y se dice que es un *identificador de solo lectura*.

Un identificador de solo lectura se puede inicializar mediante una constante, a diferencia de una referencia normal:

Ejemplo:

```
const double& pi = 3.1415926;
```

Como la constante no ocupa ningún espacio de memoria, el compilador crea un objeto temporal al que luego se hace referencia.

Referencias como parámetros

Programa de ejemplo

```
// Ref2.cpp
// Demostración de funciones con parámetros
// de tipo referencia.
// ------------------------------------------------

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

// Prototypes:
bool getClient( string& name, long& nr);
void putClient( const string& name, const long& nr);
```

```
int main()
{
    string clientName;
    long   clientNr;

    cout << "\nTo input and output client data \n"
         << endl;
    if( getClient( clientName, clientNr))    // Calls
        putClient( clientName, clientNr);
    else
        cout << "Invalid input!" << endl;

    return 0;
}

bool getClient( string& name, long& nr)      // Definition
{
    cout << "\nTo input client data!\n"
         << " Name:    ";
    if( !getline( cin, name)) return false;

    cout << " Number: ";
    if( !( cin >> nr)) return false;

    return true;
}

// Definición
void putClient( const string& name, const long& nr)
{                                          // ¡El nombre y el número solo se pueden leer!
    cout << "\n-------- Client Data ---------\n"
         << "\n Name:   "; cout << name
         << "\n Number: "; cout << nr << endl;
}
```

Paso por referencia

Se puede programar un *paso por referencia* utilizando referencias o punteros como parámetros de función. Sintácticamente es más sencillo utilizar referencias, aunque no siempre está permitido.

Un parámetro de un tipo de referencia es un alias para un argumento. Cuando se llama a una función, se inicializa un parámetro de referencia con el objeto suministrado como argumento. De este modo, la función puede manipular directamente el argumento que se le pasa.

Ejemplo:

```
void test( int& a) { ++a; }
```

Con base en esta definición, la afirmación

```
test( var);     // Para una variable int var
```

incrementa la variable var. Dentro de la función, cualquier acceso a la referencia a accede automáticamente a la variable suministrada, `var`.

Si se pasa un objeto como argumento al pasar por referencia, el objeto no se copia. En cambio, la dirección del objeto se pasa a la función internamente, lo que permite que la función acceda al objeto con el que se llamó.

Comparación con el paso por valor

A diferencia de un *paso por valor* normal, una expresión, como `a+b`, no se puede utilizar como argumento. El argumento debe tener una dirección en la memoria y ser del tipo correcto.

El uso de referencias como parámetros ofrece los siguientes *beneficios*:

- Los argumentos no se copian. A diferencia del paso por valor, el tiempo de ejecución de un programa debería mejorar, especialmente si los argumentos ocupan grandes cantidades de memoria

- Una función puede utilizar el parámetro de referencia para devolver *varios* valores a la función que llama. El paso por valor permite solo un resultado como valor de retorno, a menos que recurra al uso de variables globales.

Si necesita leer argumentos, pero no copiarlos, puede definir una *referencia de solo lectura* como parámetro.

Ejemplo:

```
void display( const string& str);
```

La función `display()` contiene una cadena como argumento. Sin embargo, no genera una nueva cadena en la que se copia la cadena del argumento. En cambio, `str` es simplemente una referencia al argumento. El invocador puede estar seguro de que el argumento no se modifica dentro de la función, ya que `str` se declara como una `const`.

Referencias como valor de retorno

Programa de ejemplo

```
// Ref3.cpp
// Demuestra el uso de valores de retorno con
// tipo de referencia.
// ------------------------------------------------

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

                                                  // Returns a
double& refMin( double&, double&);                // reference to
                                                  // the minimum.

int main()
{
    double x1 = 1.1, x2 = x1 + 0.5, y;

    y = refMin( x1, x2);                  // Assigns the minimum to y.
    cout << "x1 = " << x1 << "     "
         << "x2 = " << x2 << endl;
    cout << "Minimum: " << y << endl;

    ++refMin( x1, x2);                    // ++x1, as x1 is minimal
    cout << "x1 = " << x1 << "     "      // x1 = 2.1
         << "x2 = " << x2 << endl;        // x2 = 1.6
    ++refMin( x1, x2);                    // ++x2, because x2 is
                                          // the minimum.
    cout << "x1 = " << x1 << "     "      // x1 = 2.1
         << "x2 = " << x2 << endl;        // x2 = 2.6
    refMin( x1, x2) = 10.1;               // x1 = 10.1, because
                                          // x1 is the minimum.
    cout << "x1 = " << x1 << "     "      // x1 = 10.1
         << "x2 = " << x2 << endl;        // x2 = 2.6
    refMin( x1, x2) += 5.0;               // x2 += 5.0, because
                                          // x2 is the minimum.
    cout << "x1 = " << x1 << "     "      // x1 = 10.1
         << "x2 = " << x2 << endl;        // x2 = 7.6

    return 0;
}

                                          // Devuelve una
double& refMin( double& a, double& b)     // referencia a
{                                         // el mínimo.
    return a <= b ? a : b;
}
```

Nota:

La expresión `refMin(x1,x2)` representa el objeto `x1` o el objeto `x2`, es decir, el objeto que contiene el valor más pequeño.

Devolución de referencias

El tipo de retorno de una función también puede ser un tipo de referencia. La llamada a la función representa entonces un objeto y puede utilizarse como un objeto.

Ejemplo:

```
string& message()     // Reference!
{
    static string str = "Today only cold cuts!";
    return str;
}
```

Esta función devuelve una referencia a una cadena *estática*, `str`. Preste atención al siguiente punto al devolver referencias y punteros:

El objeto al que hace referencia el valor de retorno debe existir después de salir de la función.

Sería un error crítico declarar la cadena `str` como una variable `auto` normal en la función `message()`. Esto destruiría la cadena al salir de la función y la referencia apuntaría a un objeto que ya no existía.

Llamada a una función de tipo de referencia

La función message() (mencionada anteriormente en esta sección) es del tipo "referencia a cadena". Por lo tanto, llamar a

```
message()
```

representa un objeto de tipo `string` y las siguientes declaraciones son válidas:

```
message() = "Let's go to the beer garden!";
message() += " Cheers!";
cout << "Length: " << message().length();
```

En estos ejemplos, primero se asigna un nuevo valor al objeto al que hace referencia la llamada de función. Luego, se agrega una nueva cadena antes de que se muestre la longitud de la cadena a la que se hace referencia en la tercera instrucción.

Si desea evitar modificar el objeto al que se hace referencia, puede definir el tipo de función como una referencia de solo lectura.

Ejemplo:

```
const string& message();     // Read-only!
```

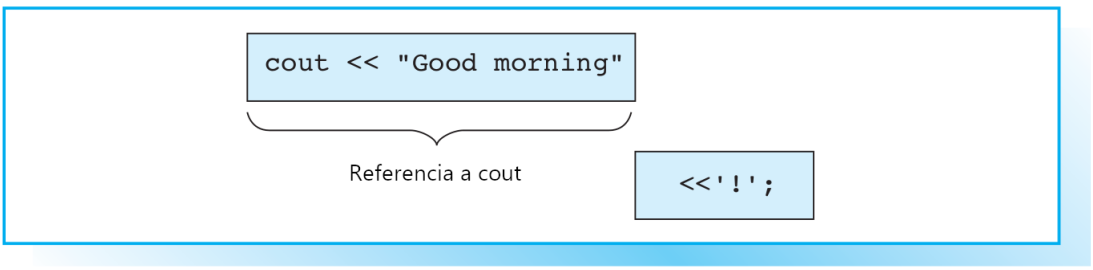
Las referencias se utilizan comúnmente como tipos de retorno cuando se sobrecargan operadores. Las operaciones que un operador debe realizar para un tipo definido por el usuario siempre se implementan mediante una función adecuada. Consulte los capítulos sobre sobrecarga de operadores más adelante en este libro para obtener más detalles. Sin embargo, en este punto se pueden proporcionar ejemplos con operadores de clases estándar.

Expresiones con tipo de referencia

Example:

Operator << of class ostream

```
cout << "Good morning" << '!';
```



Ejemplos de asignaciones de la clase string

```
// Ref4.cpp
// Expresiones con tipo de referencia ejemplificadas por
// asignaciones de cadenas.
// ------------------------------------------------

#include <iostream>
#include <string>
#include <cctype>               // For toupper()
using namespace std;

void strToUpper( string& );     // Prototype

int main()
{
    string text("Test with assignments \n");

    strToUpper(text);
    cout << text << endl;

    strToUpper( text = "Flowers");
    cout << text << endl;

    strToUpper( text += " cheer you up!\n");
    cout << text << endl;

    return 0;
}

void strToUpper( string& str)      // Converts the content
{                                  // de str a mayúsculas.
    int len = str.length();

    for( int i=0; i < len; ++i)
        str[i] = toupper( str[i]);
}
```

Cada expresión de C++ pertenece a un tipo determinado y también tiene un valor, si el tipo no es `void`. Los tipos de referencia también son válidos para las expresiones.

Los operadores de desplazamiento de la clase Stream

Los operadores << y >> utilizados para la entrada y salida de flujos son ejemplos de expresiones que devuelven una referencia a un objeto.

Ejemplo:

```
cout << " Good morning "
```

Esta expresión no es de tipo `void` sino una referencia al objeto `cout`, es decir, representa el objeto `cout`. Esto permite utilizar repetidamente el signo << en la expresión:

```
cout << "Good morning" << '!'
```

La expresión es entonces equivalente a

```
(cout << " Good morning ") << '!'
```

Las expresiones que utilizan el operador << se componen de izquierda a derecha, como se puede ver en la tabla de precedencia incluida en el apéndice.

De manera similar, la expresión variable `cin` >> representa el flujo `cin`. Esto permite el uso repetido del operador >>.

Ejemplo:

```
int a; double x;
cin >> a >> x;     // (cin >> a) >> x;
```

Otros operadores de tipo de referencia

Otros operadores de tipo de referencia de uso común incluyen el operador de asignación simple = y las asignaciones compuestas, como += y *=. Estos operadores devuelven una referencia al operando de la izquierda. En una expresión como

`a = b o a += b`

a debe ser, por tanto, un objeto. A su vez, la expresión en sí misma representa el objeto `a`. Esto también se aplica cuando los operadores hacen referencia a objetos que pertenecen a tipos de clase. Sin embargo, la definición de clase estipula los operadores disponibles. Por ejemplo, los operadores de asignación = y += están disponibles en la cadena de clase `string`.

Ejemplo:

```
string name("Jonny ");
name += "Depp";          //Referencia a nombre
```

Dado que una expresión de este tipo representa un objeto, la expresión se puede pasar como argumento a una función que se llama por referencia. Este punto se ilustra con el ejemplo siguiente.

Definición de punteros

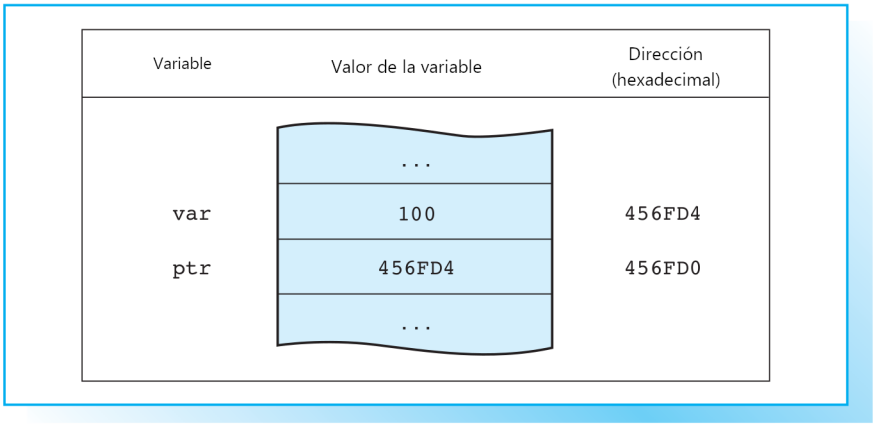
Programa de ejemplo

```
// pointer1.cpp
// Imprime los valores y direcciones de las variables.
// ---------------------------------------------------

#include <iostream>
using namespace std;

int var, *ptr;          // Definition of variables var and ptr

int main()              // Imprime los valores y direcciones
{                       // de las variables var y ptr.
    var = 100;
    ptr = &var;

    cout << " Value of var:     " << var
         << "   Address of var: " << &var
         << endl;
    cout << " Value of ptr:     " << ptr
         << "   Address of ptr: " << &ptr
         << endl;

    return 0;
}
```

Ejemplo de salida de pantalla

```
Value of var:      100     Address of var: 00456FD4
Value of ptr: 00456FD4     Address of ptr: 00456FD0
```

Las variables `var` y `ptr` en la memoria



La lógica de un programa eficiente a menudo requiere acceso a las direcciones de memoria utilizadas por los datos de un programa, en lugar de la manipulación de los datos en sí. Las listas enlazadas o los árboles cuyos elementos se generan dinámicamente en tiempo de ejecución son ejemplos típicos.

Punteros

Un *puntero* es una expresión que representa tanto la *dirección* como el *tipo* de otro objeto. El uso del operador de dirección, &, para un objeto determinado crea un puntero a ese objeto. Dado que `var` es una variable `int`,

Ejemplo:

```
&var     // Dirección del objeto var
```

es la dirección del objeto int en memoria y, por lo tanto, un puntero a `var`. Un puntero apunta a una dirección de memoria y, al mismo tiempo, indica mediante su tipo cómo se puede leer o escribir en la dirección de memoria. Por lo tanto, según el tipo, nos referimos a *punteros a char*, *punteros a int*, etc., o utilizamos una abreviatura, como *puntero char*, *puntero int*, etc.

Variables de puntero

Una expresión como `&var` es un puntero constante; sin embargo, C++ permite definir *variables de puntero*, es decir, variables que pueden almacenar la dirección de otro objeto.

Ejemplo:

```
int *ptr;          // o:          int* ptr;
```

Esta declaración define la variable `ptr`, que es de tipo `int*` (en otras palabras, *un puntero a int*). Por lo tanto, `ptr` puede almacenar la dirección de una variable `int`. En una declaración, el carácter asterisco * siempre significa "puntero a".

Los *tipos puntero* son ti pos derivados. La forma general es `T*`, donde `T` puede ser cualquier tipo dado. En el ejemplo anterior, `T` es un tipo `int`.

Los objetos del mismo tipo base `T` se pueden declarar juntos.

Ejemplo:

```
int a, *p, &r = a;     // Definición de a, p, r
```

Después de declarar una variable `ptr`, debe apuntar el puntero a una dirección de memoria. El programa siguiente hace esto mediante la instrucción

```
ptr = &var;.
```

Referencias y punteros

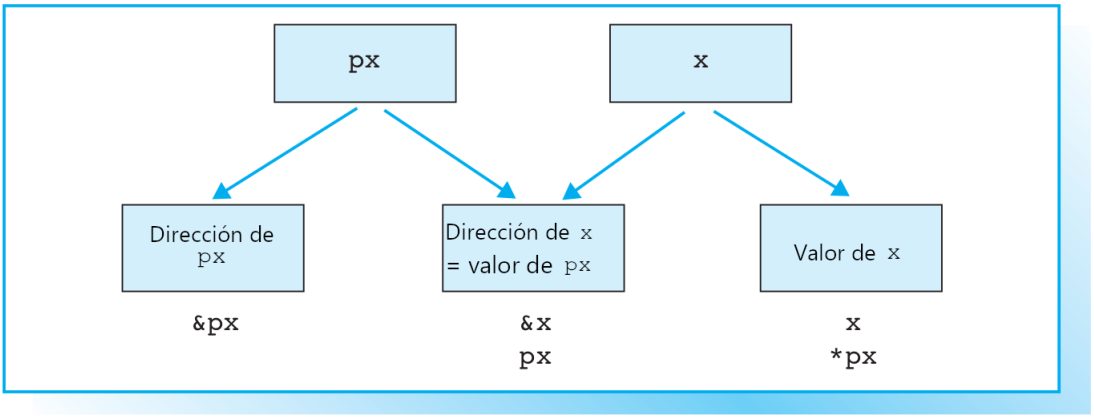
Las referencias son similares a los punteros: ambos hacen referencia a un objeto en la memoria. Sin embargo, un puntero no es simplemente un alias, sino un objeto individual que tiene una identidad separada del objeto al que hace referencia. Un puntero tiene su propia dirección de memoria y se puede manipular apuntándolo a una nueva dirección de memoria y, por lo tanto, haciendo referencia a un objeto diferente.

El operador de indirección

Uso del operador de indirección

```
double x, y, *px;

px = &x;              // Sea px el punto x.
*px = 12.3;           // Asignar el valor 12,3 a x
*px += 4.5;           // Incrementar x en 4,5.
y = sin(*px);         // Para asignrar el seno de x a y.
```

Dirección y valor de las variables `x` y `px`



Notas sobre direcciones en un programa

- Cada variable puntero ocupa la misma cantidad de espacio, independientemente del tipo de objeto al que hace referencia. Es decir, ocupa tanto espacio como sea necesario para almacenar una dirección. En un ordenador de 32 bits, como un PC, esto son cuatro bytes.
- Las direcciones visibles en un programa son normalmente direcciones lógicas que el sistema asigna y asigna a direcciones físicas. Esto permite una gestión eficiente del almacenamiento y el intercambio de bloques de memoria actualmente no utilizados al disco duro.
- C++ garantiza que cualquier dirección válida no será igual a 0. Por tanto, el valor especial 0 se utiliza para indicar un error. Para los punteros, la constante simbólica NULL se define como 0 en los archivos de cabecera estándar. Un puntero que contiene el valor NULL también se denomina puntero NULL.

Uso de punteros para acceder a objetos

El *operador de indirección* * se utiliza para acceder a un objeto al que hace referencia un puntero:
Dado un puntero, `ptr`, `*ptr` es el objeto al que hace referencia `ptr`.
Como programador, siempre debes distinguir entre el puntero `ptr` y el objeto al que se dirige `*ptr`.
Ejemplo:

```
long   a = 10, b,          // Definición de a, b
       *ptr;               // y puntero ptr.
ptr = &a;                  // Sea ptr el que apunta hacia a.
b = *ptr;
```

Esto asigna el valor de `a` hacia `b`, ya que `ptr` apunta hacia `a`. La asignación `b = a;` devolvería el mismo resultado. La expresión `*ptr` representa el objeto `a`, y puede utilizarse en cualquier lugar en el que se pueda utilizar `a`.

El carácter asterisco `*` utilizado para definir variables de puntero no es un operador, sino que simplemente imita el uso posterior del puntero en expresiones. Por lo tanto, la definición

```
long *ptr;
```

tiene el siguiente significado: `ptr` es un tipo `long*` (puntero a `long`) y `*ptr` es un tipo `long`.

El operador de indirección `*` tiene alta precedencia, al igual que el operador de dirección &. Ambos operadores son unarios, es decir, tienen un solo operando. Esto también ayuda a distinguir el operador de redirección del operador de multiplicación binaria `*`, que siempre toma dos operandos.

Valores L

Una expresión que identifica un objeto en la memoria se conoce como *valor L* en C++. El término valor L aparece comúnmente en los mensajes de error del compilador y se deriva de la asignación. El

operando izquierdo del operador = siempre debe designar una dirección de memoria. Las expresiones que no son un valor L se denominan a menudo *valores R*.

Un nombre de variable es el ejemplo más simple de un valor L. Sin embargo, una constante o una expresión, como `x + 1`, es un valor R. El operador de indirección es un ejemplo de un operador que produce valores L. Dada una variable de puntero p, tanto p como `*p` son valores L, ya que `*p` designa el objeto al que apunta p.

Punteros como parámetros

Función de ejemplo

```
// pointer2.cpp
// Definición y llamada de la función swap().
// Demuestra el uso de punteros como parámetros.
// ------------------------------------------------------
#include <iostream>
using namespace std;

void swap( float *, float *);          // Prototype of swap()

int main()
{
    float x = 11.1F;
    float y = 22.2F;
       .
       .
       .
    swap( &x, &y );
       .
       .          // p2 = &y
       .
}           // p1 = &x

void swap( float *p1, float *p2)
{
    float temp;                        // Variable temporal

    temp = *p1;                        // En la llamada anterior
    *p1 = *p2;                         // p1 apunta a x y p2 a y
    *p2 = temp;
}
```

Objetos como argumentos

Si se pasa un objeto como argumento a una función, pueden darse dos situaciones:

- El parámetro en cuestión es del mismo tipo que el objeto que se le pasa. A la función que se llama se le pasa una copia del objeto (paso por valor)
- El parámetro en cuestión es una referencia. El parámetro es entonces un alias del argumento, es decir, la función que se llama manipula el objeto pasado por la función que llama (paso por referencia).

En el primer caso, el argumento pasado a la función no puede ser manipulado por la función. Esto no es cierto para el paso por referencia. Sin embargo, hay una tercera forma de pasar por referencia: pasar punteros a la función.

Punteros como argumentos

¿Cómo se declara un parámetro de función para permitir que se pase una dirección a la función como argumento? La respuesta es bastante sencilla: el parámetro debe declararse como una *variable puntero*.

Si, por ejemplo, la función `func()` requiere la dirección de un valor `int` como argumento, puede utilizar la siguiente declaración

Ejemplo:

```
long func( int *iPtr )
{
    // Function block
}
```

para declarar el parámetro `iPtr` como un puntero `int`. Si una función conoce la dirección de un objeto, por supuesto puede usar el operador de indirección para acceder y manipular el objeto.

En el programa siguiente, la función `swap()` intercambia los valores de las variables `x` e `y` en la función que realiza la llamada. La función `swap()` puede acceder a las variables ya que las direcciones de estas variables, es decir, `&x` y `&y`, se le pasan como argumentos.

Los parámetros `p1` y `p2` en `swap()` se declaran, por tanto, como punteros `float`. La declaración

```
swap( &x, &y);
```

inicializa los punteros `p1` y `p2` con las direcciones `x` o `y`. Cuando la función manipula las expresiones `*p1` y `*p2`, en realidad accede a las variables `x` e `y` en la función que realiza la llamada e intercambia sus valores.

Ejercicios

1. ¿Qué sucede si el parámetro en la función de ejemplo `strToUpper()` se declara como una `string&` en lugar de una `string`?

2. Escriba una función de tipo `void` llamada `circle()` para calcular la circunferencia y el área de un círculo. El radio y dos variables se pasan a la función, que por lo tanto tiene tres parámetros:

   Parámetros:

   Una referencia de solo lectura a `double` para el radio y dos referencias a `double` que la función usa para almacenar el área y la circunferencia del círculo.

   Pruebe la función `circle()` generando una tabla que contenga el radio, la circunferencia y el área para los radios `0.5, 1.0, 1.5, . . . , 10.0`.

   Nota:

   Dado un círculo con radio `r`:

   Área $= \pi * r * r$ y circunferencia $= 2 * \pi * r$ donde $\pi = 3{,}1415926535$

3. *a*. La versión opuesta de la función `swap()` se puede compilar sin producir ningún mensaje de error. Sin embargo, la función no intercambiará los valores de `x` e `y` cuando se llame a `swap(&x,&y);`. ¿Qué es lo que está mal?

   b. Pruebe la versión de puntero correcta de la función `swap()` que se encuentra en este capítulo. Luego escriba y pruebe una versión de la función `swap()` que use referencias en lugar de punteros.

4. Cree una función `quadEquation()` que calcule las soluciones de ecuaciones cuadráticas. La fórmula para calcular ecuaciones cuadráticas se muestran a continuación.

   La ecuación cuadrática: $ax^2 + bx + c = 0$ tiene soluciones reales:

   $$x_{12} = \frac{-b\sqrt{b^2-4ac}}{2a}$$

   Si el discriminante satisface: $b^2 - 4ac >= 0$

   Si el valor de $(b^2 - 4ac)$ es negativo, no existe una solución real.

   Argumentos: Los coeficientes a, b, c y dos punteros a ambas soluciones.

   Devuelve: `false`, si no hay una solución real disponible, `true` en caso contrario.

   Pruebe la función usando estas ecuaciones cuadráticas y compare sus soluciones.

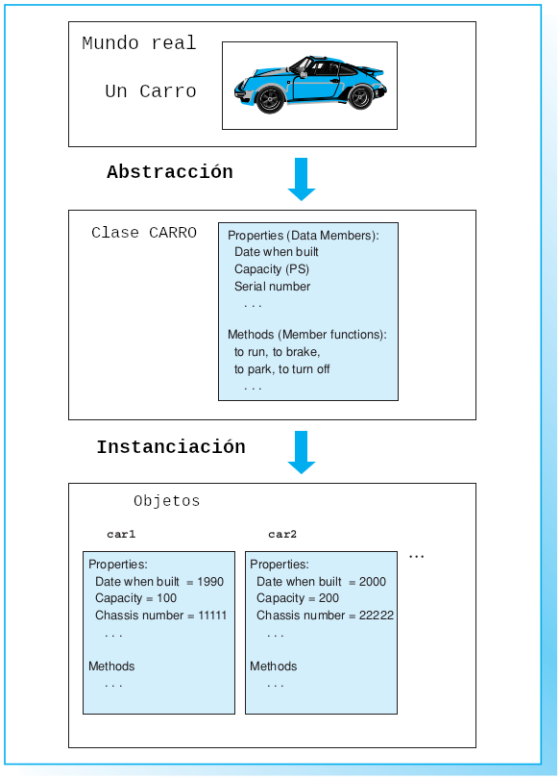
Ecuación cuadrática	Soluciones
$2x^2 - 2x - 1{,}5 = 0$	$x_1 = 1{,}5, x_2 = -0{,}5$
$x_2 - 6x + 9 = 0$	$x_1 = 3{,}0, x_2 = 3{,}0$
$2x^2 + 2 = 0$	none

Definición de Clases

Definición de clases

En este capítulo se describe cómo se definen las clases y cómo se utilizan las instancias de clases, es decir, los objetos. Además, se presentan las estructuras y las uniones como ejemplos de clases especiales.

El concepto de clase



Las clases son el elemento del lenguaje de C++ más importante para la programación orientada a objetos (OOP). Una clase define las propiedades y capacidades de un objeto.

Abstracción de datos

Los seres humanos utilizan la abstracción para gestionar situaciones complejas. Los objetos y procesos se reducen a lo básico y se hace referencia a ellos en términos genéricos. Las clases permiten un uso más directo de los resultados de este tipo de abstracción en el desarrollo de software.

El primer paso para resolver un problema es el análisis. En la programación orientada a objetos, el análisis comprende la identificación y descripción de objetos y el reconocimiento de sus relaciones mutuas. Las descripciones de objetos son los componentes básicos de las clases.

En C++, una clase es un tipo definido por el usuario. Contiene *miembros de datos*, que describen las propiedades de la clase, y *funciones miembro*, o *métodos*, que describen las capacidades de los objetos. Las clases son simplemente patrones que se utilizan para instanciar o crear objetos del tipo de clase. En otras palabras, un objeto es una variable de una clase determinada.

Encapsulación de datos

Cuando se define una clase, también se especifican los miembros privados, es decir, los miembros que no están disponibles para el acceso externo, y los miembros públicos de esa clase. Un programa de aplicación accede a los objetos utilizando los métodos públicos de la clase y activando así sus capacidades.

El acceso a los datos de los objetos rara vez es directo, es decir, los datos de los objetos normalmente se declaran como privados y luego se leen o modifican mediante métodos con declaraciones públicas para garantizar el acceso correcto a los datos.

Un aspecto importante de esta técnica es el hecho de que los programas de aplicación no necesitan conocer la estructura interna de los datos. Si es necesario, la estructura interna de los datos del programa puede incluso modificarse. Siempre que las interfaces de los métodos públicos permanezcan inalteradas, cambios como estos no afectarán al programa de aplicación. Esto permite mejorar una aplicación programando una versión mejorada de la clase sin cambiar un solo byte de la aplicación.

Por tanto, se considera que un objeto encapsula su estructura privada, se protege a sí mismo de las influencias externas y se gestiona a sí mismo mediante sus propios métodos. Esto describe el concepto de encapsulación de datos de forma concisa.

Definición de clases

Esquema de definición

```
class Demo
{
    private:
                // Miembros y métodos de datos privados aquí

    public:
                // Miembros y métodos de datos públicos aquí

};
```

Ejemplo de una clase

```
// account.h
// Definiendo la clase Account.
// ------------------------------------------------
#ifndef _ACCOUNT_          // Avoid multiple inclusions.
#define _ACCOUNT_

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

class Account
{
    private:               // Miembros protegidos:
        string name;       // Titular de la cuenta
        unsigned long nr;  // Número de la cuenta
        double balance;    // Saldo de la cuenta

    public:                // Interfaz pública:
        bool init( const string&, unsigned long, double);
        void display();
};

#endif    // _ACCOUNT_
```

Una definición de clase especifica el nombre de la clase y los nombres y tipos de los *miembros* de la clase.

La definición comienza con la palabra clave `class` seguida del nombre de la clase. Los miembros de datos y los métodos se declaran en el bloque de código siguiente. Los miembros de datos y las funciones miembro pueden pertenecer a cualquier tipo válido, incluso a otra clase definida previamente. Al mismo tiempo, los miembros de clase se dividen en:

- miembros `private`, a los que no se puede acceder externamente
- miembros `public`, a los que se puede acceder externamente.

Los miembros `public` forman la denominada *interfaz pública* de la clase.

Se muestra una definición esquemática de una clase. La sección `private` generalmente contiene miembros de datos y la sección `public` contiene los métodos de acceso a los datos. Esto permite la *encapsulación* de datos.

El siguiente ejemplo incluye una clase denominada `Account` que se utiliza para representar una cuenta bancaria. Los miembros de datos, como el nombre del titular de la cuenta, el número de cuenta y el saldo de la cuenta, se declaran como `private`. Además, hay dos métodos públicos, `init()` para fines de inicialización y `display()`, que se utiliza para mostrar los datos en la pantalla.

Las etiquetas `private:` y `public:` se pueden usar a discreción del programador dentro de una clase:

- Puede usar las etiquetas con la frecuencia que necesite, o no usarlas en absoluto, y en cualquier orden. Una sección marcada como `private:` o `public:` es válida hasta que aparezca la siguiente etiqueta `public:` o `private:`
- El valor predeterminado para el acceso de miembros es `private`. Si omite las etiquetas `private` y `public`, se supone que todos los miembros de la clase son `private`.

Nombres

Cada pieza de software utiliza un conjunto de reglas de nombres. Estas reglas suelen reflejar la plataforma de destino y las bibliotecas de clases utilizadas. Para los fines de este libro, decidimos mantener las convenciones de nombres estándar para distinguir las clases y los miembros de las clases. Los nombres de las clases comienzan con una letra mayúscula y los nombres de los miembros con una letra minúscula.

Los miembros de diferentes clases pueden compartir el mismo nombre. Por lo tanto, un miembro de otra clase también podría llamarse `display()`.

Definición de métodos

Métodos de la clase `Account`

```
// account.cpp
// Defines methods init() and display().
// -------------------------------------------------
#include "account.h"          // Class definition
#include <iostream>
#include <iomanip>
using namespace std;

// The method init() copies the given arguments
// into the private members of the class.
bool Account::init(const string& i_name,
                   unsigned long i_nr,
                   double        i_balance)
{
    if( i_name.size() < 1)   // No empty name
          return false;
    name    = i_name;
    nr      = i_nr;
    balance = i_balance;
    return true;
}

// The method display() outputs private data.
void Account::display()
{
    cout << fixed << setprecision(2)
         << "---------------------------------------\n"
         << "Account holder:    " << name    << '\n'
         << "Account number:    " << nr      << '\n'
         << "Account balance:   " << balance << '\n'
         << "---------------------------------------\n"
         << endl;
}
```

Una definición de clase no está completa sin la definición de métodos. Solo así se pueden utilizar los objetos de la clase.

Sintaxis

Cuando se define un método, también se debe proporcionar el nombre de la clase, separándolo del nombre de la función mediante el operador de resolución de ámbito ::.

Sintaxis:

```
type class_name::function_name(parameter_list)
{       . . .       }
```

Si no se proporciona el nombre de la clase, se produce una definición de función global.

Dentro de un método, *todos* los miembros de una clase se pueden designar directamente utilizando sus nombres. La pertenencia a la clase se asume automáticamente. En particular, los métodos que pertenecen a la misma clase pueden llamarse entre sí directamente.

El acceso a los miembros privados solo es posible dentro de los métodos que pertenecen a la misma clase. Por lo tanto, los miembros `private` están completamente controlados por la clase.

La definición de una clase no asigna automáticamente memoria para los miembros de datos de esa clase. Para asignar memoria, debe definir un objeto. Cuando se llama a un método para un objeto determinado, el método puede manipular los datos de este objeto.

Programación modular

Una clase normalmente se define en varios archivos fuente. En este caso, deberá colocar la definición de clase en un archivo de encabezado/*header file*. Si coloca la definición de la clase `Account` en el archivo `Account.h`, cualquier archivo fuente, incluido el archivo de encabezado, puede utilizar la clase `Account`.

Los métodos siempre deben definirse dentro de un archivo fuente. Esto significaría definir los métodos para la clase `Account` en un archivo fuente llamado `Account.cpp`, por ejemplo.

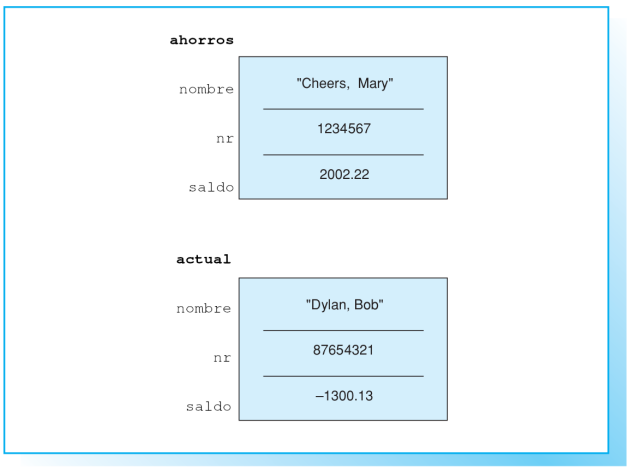
El código fuente del programa de aplicación, por ejemplo, el código que contiene la función `main`, es independiente de la clase y se puede almacenar en archivos fuente separados. Separar las clases de los programas de aplicación facilita la reutilización de las clases.

En un entorno de desarrollo integrado, un programador definirá un *proyecto* para ayudar a administrar los diversos módulos del programa insertando todos los archivos fuente en el proyecto.

Cuando se compila y vincula el proyecto, los archivos fuente modificados se vuelven a compilar automáticamente y se vinculan al programa de aplicación.

Definición de objetos

Los objetos `actuales` y `guardados` en la memoria



La definición de una clase también define un nuevo tipo para el cual se pueden definir variables, es decir, objetos. Un objeto también se conoce como una instancia de una clase.

Definición de objetos

Un objeto se define de la forma habitual proporcionando el tipo y el nombre del objeto.
Sintaxis:

```
class_name object_name1 [, object_name2,...]
```

La siguiente declaración define un objeto current de tipo `Account`:
Ejemplo:

```
Account current;      // o: class Account ...
```

Ahora se asigna memoria para los miembros de datos del objeto `current`. El objeto `current` en sí contiene los miembros `name`, `nr` y `balance`.

Objetos en memoria

Si se declaran varios objetos del mismo tipo de clase, como en
Ejemplo:

```
Account current, savings;
```

cada objeto tiene sus propios miembros de datos. Incluso el objeto `savings` contiene los miembros `name`, `nr` y `balance`. Sin embargo, estos miembros de datos ocupan una posición diferente en la memoria que los miembros de datos que pertenecen a `current`.

Se llaman los mismos métodos para ambos objetos. Solo existe una instancia del código de máquina para un método en la memoria; esto se aplica incluso si no se han definido objetos para la clase.

Siempre se llama a un método para una instancia particular y luego se manipulan los miembros de datos de `this` objeto. Esto da como resultado el contenido de la memoria como se muestra en la página opuesta, cuando se llama al método `init()` para cada objeto con los valores que se muestran.

Inicialización de objetos

Los objetos que pertenecen a la clase `Account` se definieron originalmente pero no se inicializaron. Por lo tanto, cada objeto miembro se define pero no se inicializa explícitamente. La cadena name, está vacía, ya que está definida en la clase `string`. Sin embargo, los valores iniciales de los miembros nr y balance son desconocidos. Como es el caso de otras variables, estos miembros de datos tendrán el valor predeterminado 0 si el objeto se declara global o `static`.

Puede definir exactamente cómo se crea y se destruye un objeto. Estas tareas las realizan los *constructores* y *destructores*. Los constructores son específicamente responsables de inicializar objetos; se brindan más detalles más adelante.

Uso de objetos

Programa de ejemplo

```
// account_t.cpp
// Uses objects of class Account.
// ------------------------------------------------

#include "Account.h"

int main()
{
    Account current1, current2;

    current1.init("Cheers, Mary", 1234567, -1200.99);
    current1.display();
```

```
//  current1.balance += 100;   // Error: private member

    current2 = current1;       // ok: Assignment of
                               // objects is possible.
    current2.display();        // ok

                               // New values for current2
    current2.init("Jones, Tom", 3512347, 199.40);

    current2.display();
                               // Para usar una referencia:
    Account& mtr = current1;   // mtr es un nombre de alias
                               // para el objeto current1.
    mtr.display();             // mtr se puede usar simplemente
                               // como objeto current1.
    return 0;
}
```

Operador de acceso a miembros de clase

Un programa de aplicación que manipula los objetos de una clase puede acceder únicamente a los miembros `public` de esos objetos. Para ello, utiliza el *operador de acceso a miembros de clase* (abreviado como *operador de punto*).

Sintaxis:

```
object.member
```

Donde `member` es un miembro de datos o un método.

Ejemplo:

```
Account current;
current.init("Jones, Tom",1234567,-1200.99);
```

La expresión `current.init` representa el método público `init` de la clase `Account`. Este método se llama con tres argumentos para `current`.

La llamada `init()` no se puede reemplazar por asignaciones directas.

Ejemplo:

```
current.name    = "Dylan, Bob";     // Error:
current.nr      = 1234567;          // private
current.balance = -1200.99;         // members
```

No se permite el acceso a los miembros `private` de un objeto fuera de la clase. Por lo tanto, es imposible mostrar miembros individuales de la clase `Account` en la pantalla.

Ejemplo:

```
cout << current.balance;     // Error
current.display();           // ok
```

El método `display()` muestra todos los miembros de datos del objeto actual. Un método como `display()` solo se puede llamar para un objeto. La declaración

```
display();
```

generaría un mensaje de error, ya que no existe una función global llamada `display()`. ¿Qué datos debería mostrar la función?

Asignación de objetos

El operador de asignación = es el único operador que se define para todas las clases de forma predeterminada. Sin embargo, los objetos de origen y destino deben pertenecer a la misma clase. La asignación se realiza para asignar los miembros de datos individuales del objeto de origen a los miembros correspondientes del objeto de destino.

Ejemplo:

```
Account current1, current2;
current2.init("Marley, Bob",350123, 1000.0);
current1 = current2;
```

Esto copia los miembros de datos de `current2` a los miembros correspondientes de `current1`.

Punteros a objetos

Programa de ejemplo

```
// ptrObj.cpp
// Uses pointers to objects of class Account.
// ------------------------------------------------

#include "Account.h"                      // Includes <iostream>, <string>
bool getAccount( Account *pAccount);      // Prototype

int main()
{
    Account current1, current2, *ptr = &current1;

    ptr->init("Cheer, Mary",              // current1.init(...)
              3512345, 99.40);
    ptr->display();                       // current1.display()

    ptr = &current2;                      // Let ptr point to current2
    if( getAccount( ptr))                 // Input and output a new
        ptr->display();                   // account.
    else
        cout << "Invalid input!" << endl;
    return 0;
}

// ------------------------------------------------
// getAccount() reads data for a new account
// and adds it into the argument.

bool getAccount( Account *pAccount )
{
    string name, line(50,'-');            // Local variables
    unsigned long nr;
    double startcapital;

    cout << line << '\n'
         << "Enter data for a new account: \n"
         << "Account holder: ";
    if( !getline(cin,name) || name.size() == 0)
        return false;
    cout << "Account number: ";
    if( !(cin >> nr))  return false;
    cout << "Starting capital: ";
    if( !(cin >> startcapital)) return false;
    // All input ok
    pAccount->init( name, nr, startcapital);
    return true;
}
```

Un objeto de una clase tiene una dirección de memoria, como cualquier otro objeto. Puede asignar esta dirección a un puntero adecuado.

Ejemplo:

```
Account savings("Mac, Rita",654321, 123.5);
Account *ptrAccount = &savings;
```

Esto define el objeto `savings` y una variable de puntero llamada `ptrAccount`. El puntero `ptrAccount` se inicializa de modo que apunte al objeto `savings`. Esto hace que `*ptrAccount` sea el objeto savings en sí. Luego, puede utilizar la instrucción

Ejemplo:

```
(*ptrAccount).display();
```

para llamar al método `display()` para el objeto `saving`. En este caso, se deben utilizar paréntesis, ya que el operador **.** tiene mayor precedencia que el operador *****.

Operador de flecha

Puede utilizar el operador de acceso a miembros de clase **->** (en breve: operador de flecha) en lugar de una combinación de ***** y **.**

Sintaxis:

```
objectPointer->member
```

Esta expresión es equivalente a

```
(*objectPointer).member
```

El operador **->** está formado por un signo menos y un signo mayor que.

Ejemplo:

```
ptrAccount->display();
```

Esta instrucción llama al método `display()` para el objeto al que hace referencia `ptrAccount`, es decir, para el objeto `savings`. La instrucción es equivalente a la instrucción del ejemplo anterior.

La diferencia entre los operadores de acceso a miembros de clase **.** y **->** es que el operando izquierdo del operador de punto debe ser un objeto, mientras que el operando izquierdo del operador de flecha debe ser un puntero a un objeto.

El programa de ejemplo

Los punteros a objetos se utilizan a menudo como parámetros de función. Una función que obtiene la dirección de un objeto como argumento puede manipular el objeto referenciado directamente. El ejemplo de la página opuesta ilustra este punto. Utiliza la función `getAccount()` para leer los datos de una nueva cuenta. Cuando se llama, se pasa la dirección de la cuenta:

```
getAccount(ptr) // or: getAccount(&current1)
```

La función puede entonces utilizar el puntero `ptr` y el método `init()` para escribir nuevos datos en el objeto referenciado.

Estructuras

Programa de ejemplo

```
// structs.cpp
// Defines and uses a struct.
// ------------------------------------------------

#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
struct Representative       // Defining struct Representative
{
    string name;            // Name of a representative.
    double sales;           // Sales per month.
};

inline void print( const Representative& v)
{
    cout << fixed << setprecision(2)
         << left << setw(20) << v.name
         << right << setw(10) << v.sales << endl;
}

int main()
```

```
{
    Representative rita, john;
    rita.name    = "Strom, Rita";
    rita.sales = 37000.37;
    john.name = "Quick, John";
    john.sales = 23001.23;

    rita.sales += 1700.11;     // More Sales
    cout << " Representative     Sales\n"
         << "--------------------------------" << endl;
    print( rita);
    print( john);
    cout << "\nTotal of sales: "
         << rita.sales + john.sales << endl;
    Representative *ptr = &john;          // Pointer ptr.
                                          // Who gets the
    if( john.sales < rita.sales)
        // most sales?
    ptr = &rita;
    cout << "\nSalesman of the month: "
         << ptr->name << endl;            // Representative's name
                                          // pointed to by ptr.
    return 0;
}
```

Registros

En un lenguaje clásico y procedimental como C, se juntan varios datos que pertenecen juntos de manera lógica para formar un *registro*. Los datos extensos, como los datos de los artículos en existencias de un fabricante de automóviles, se pueden organizar para facilitar su visualización y almacenar en archivos.

Desde el punto de vista de un lenguaje orientado a objetos, un registro es simplemente una clase que contiene solo miembros de datos públicos y ningún método. Por lo tanto, puede usar la palabra clave class para definir la estructura de un registro en C++.

Ejemplo:

```
class Date
{ public:    short month, day, year; };
```

Sin embargo, es una práctica común utilizar la palabra clave `struct`, que también está disponible en el lenguaje de programación C, para definir registros. La definición anterior de `Date` con los miembros `día`, `mes` y `año` es, por lo tanto, equivalente a:

Ejemplo:

```
struct Date { short month, day, year; };
```

Las palabras clave `class` y `struct`

También puede utilizar la palabra clave struct para definir una clase, como la clase Account.

Ejemplo:

```
struct Account {
    private:          //  . . .  como antes
    public:           //  . . .
};
```

Las palabras clave `class` y `struct` solo varían con respecto a la encapsulación de datos; el valor predeterminado para el acceso a los miembros de una clase definida como `struct` es `public`. A diferencia de una clase definida mediante la palabra clave class, todos los miembros de la clase son `public` a menos que se utilice una etiqueta `private`. Esto permite al programador mantener la compatibilidad con C.

Ejemplo:

```
Date future;
future.year = 2100;     // ok! Public data
```

Los registros en el verdadero sentido de la palabra, es decir, los objetos de una clase que contienen sólo miembros `public`, se pueden inicializar mediante una lista durante la definición.

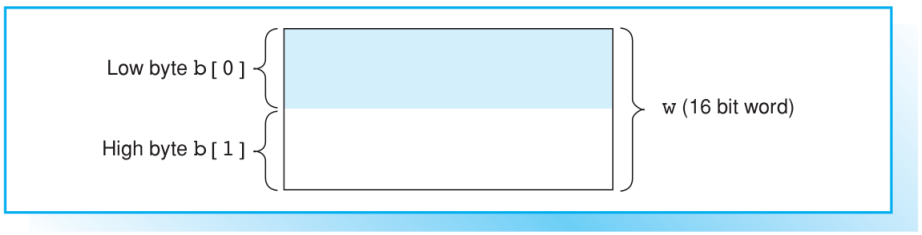
Ejemplo:

```
Date birthday = { 1, 29, 1987};
```

El primer elemento de la lista inicializa el primer miembro de datos del objeto, y así sucesivamente.

Uniones

Un objeto de unión `WordByte` en memoria



Definición y uso de unión `WordByte`

```
// unions.cpp
// Defines and uses a union.
// -------------------------------------------------
#include <iostream>
using namespace std;

union WordByte
{
    private:
      unsigned short w;        // 16 bits
      unsigned char b[2];      // Two bytes: b[0], b[1]
    public:                  // Word- and byte-access:
      unsigned short& word()   { return w; }
      unsigned char& lowByte() { return b[0]; }
      unsigned char& highByte(){ return b[1]; }
};

int main()
{
    WordByte wb;
    wb.word() = 256;
    cout << "\nWord:      " << (int)wb.word();
    cout << "\nLow-byte:  " << (int)wb.lowByte()
         << "\nHigh-byte: " << (int)wb.highByte()
         << endl;

return 0;
}
```

Salida de pantalla del programa

```
Word:     256
Low-Byte: 0
High-Byte: 1
```

Uso de la memoria

En las clases normales, cada miembro de datos que pertenece a un objeto tiene su propio espacio de memoria independiente. Sin embargo, una *union* es una clase cuyos miembros se almacenan en el mismo espacio de memoria. Cada miembro de datos tiene la misma dirección de inicio en la memoria. Por supuesto, una unión no puede almacenar varios miembros de datos en la misma dirección *simultáneamente*. Sin embargo, una unión proporciona un uso más versátil del espacio de memoria.

Definición

Sintácticamente hablando, una unión se distingue de una clase definida como una clase o estructura solo por la palabra clave union.

Ejemplo:

```
union Number
{
    long   n;
    double x;
};
Number number1, number2;
```

Este ejemplo define la unión `Number` y dos objetos del mismo tipo. La unión Número se puede utilizar para almacenar números enteros o de punto flotante.

A menos que se utilice una etiqueta `private`, se supone que todos los miembros de la unión son `public`. Esto es similar a la configuración predeterminada para las estructuras. Esto permite el acceso directo a los miembros `n` y `x` en la unión `Number`.

Ejemplo:

```
number1.n = 12345; // Storing an integer
number1.n *= 3;    // and multiply by 3.
number2.x = 2.77;  // Floating point number
```

El programador debe asegurarse de que el contenido actual de la unión se interprete correctamente. Esto normalmente se logra utilizando un campo de tipo adicional que identifica el contenido actual.

El tamaño de un objeto de tipo de unión se deriva del miembro de datos más largo, ya que todos los miembros de datos comienzan en la misma dirección de memoria. Si observamos nuestro ejemplo, la unión Number, este tamaño está definido por el miembro double, que por defecto es `8 == sizeof(double)` byte.

El ejemplo opuesto define la unión `WordByte` que le permite leer o escribir en un espacio de memoria de 16 bits byte por byte o como una unidad.

Ejercicios

Un programa necesita una clase para representar la fecha.

- Defina la clase `Date` para este propósito utilizando tres miembros de datos enteros para día, mes y año. Además, declare los siguientes métodos:

  ```
  void init( int month, int day, int year);
  void init(void);
  void print(void);
  ```

  Almacene la definición de la clase `Date` en un archivo de encabezado.

- Implemente los métodos para la clase `Date` en un archivo fuente independiente:

  1. El método `print()` genera la fecha en la salida estándar utilizando el formato `Mes-Día-Año`.
  2. El método `init()` utiliza tres parámetros y copia los valores que se le pasan a los miembros correspondientes. No se requiere una verificación de rango en esta etapa, pero se agregará más adelante.
  3. El método `init()` sin parámetros escribe la fecha actual en los miembros correspondientes.

  Nota

  Use las funciones declaradas en `ctime`

  ```
  time_t time(time_t *ptrSec)
  ```

  ```
  struct tm *localtime(const time_t *ptrSec);
  ```

  La estructura `tm` y las llamadas de muestra a esta función se incluyen en la parte opuesta. El tipo `time_t` se define como `long` en `ctime`.

La función `time()` devuelve la hora del sistema expresada como un número de segundos y escribe este valor en la variable a la que hace referencia `ptr-Sec`. Este valor se puede pasar a la función `localtime()` que convierte el número de segundos al tipo local fecha `tm` y devuelve un puntero a esta estructura.

- Pruebe la clase `Date` utilizando un programa de aplicación que, una vez más, se almacena en un archivo de origen independiente. Para ello, defina dos objetos para la clase y muestre la fecha actual. Utilice asignaciones de objetos y, como ejercicio adicional, referencias y punteros a objetos.

Estructura `tm` en el archivo de encabezado `ctime`

```
struct tm
{
    int tm_sec;          // 0 - 59(60)
    int tm_min;          // 0 - 59
    int tm_hour;         // 0 - 23
    int tm_mday;         // Day of month: 1 - 31
    int tm_mon;          // Month: 0 - 11 (January == 0)
    int tm_year;         // Years since 1900 (Year - 1900)
    int tm_wday;         // Weekday: 0 - 6 (Sunday == 0)
    int tm_yday;         // Day of year: 0 - 365
    int tm_isdst;        // Flag for summer-time
};
```

Ejemplos de llamadas a las funciones time() y localtime()

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

    struct tm *ptr;          // Pointer to struct tm.
    time_t sec;              // For seconds.
    . . .
    time(&sec);              // To get the present time.
    ptr = localtime(&sec);   // To initialize a struct of
                             // type tm and return a
                             // pointer to it.

    cout << "Today is the "      << ptr->tm_yday + 1
         << ". day of the year " << ptr->tm_year
         << endl;
    . . .
```

Referencias

Kirch-Prinz U, Prinz P *A Complete Guide to Programming in C++* Jones and Bartlett Publishers, 2002