

CATEDRA DE COMPUTACION II

TEMAS DEL LENGUAJE C++



Tomo 2

**UNIVERSIDAD NACIONAL DE ENTRE RIOS
FACULTAD DE INGENIERIA – BIOINGENIERÍA
2004**



TABLA DE CONTENIDOS

PROGRAMACIÓN ORIENTADA A OBJETOS	5
Clases	5
Constructores y destructores	8
Sobrecarga de constructores	10
Constructor vacío	10
Constructor de copia	10
Vector de objetos	11
Punteros a clases	12
Sobrecarga de operadores	13
Ejemplo de operador binario	14
El uso de <i>this</i>	16
Ejemplo de operador unario	16
Miembros estáticos	18
Relación entre clases	19
Funciones amigas (<i>friend</i>)	19
Clases amigas (<i>friend</i>)	21
Herencia entre clases	22
Polimorfismo	24
Punteros a clases base	24
Miembros virtuales	25
Clases bases abstractas	27
Clases abstractas y polimorfismo	29
Introducción	29
Funciones virtuales puras	29
Polimorfismo y uso de contenedores	31
PLANTILLAS	33
Planteo de una inquietud	33
Funciones plantilla	33

Clases plantilla	35
Especialización de plantillas	36
Valores de parámetros para plantillas.....	37
Plantillas y proyectos de varios archivos	38
CONTENEDORAS ESTÁNDARES	40
Introducción	40
¿Cómo elegir el contenedor adecuado?.....	41
Operaciones comunes	43
Ejemplos	43
La clase <i>vector</i>	43
Creación	43
Operaciones básicas	43
Uso de <i>pop_back()</i> y <i>empty()</i>	44
Acceso a través de iteradores	45
La clase <i>deque</i>	45
Operaciones básicas	45
La clase <i>list</i>	46
Operaciones básicas	46
<i>Push_back()</i> y <i>push_front()</i>	47
Lista ordenada.....	48
<i>Unique()</i>	48
La clase <i>queue</i>	49
Operaciones básicas	49
Ciclos a través de <i>queue</i>	50
La clase <i>map</i>	50
Operaciones básicas	50
Usando iteradores	51
Ordenando objetos.....	51
La clase <i>set</i>	52
Operaciones básicas	52
MANEJO DE ERRORES.....	54
Errores estándares	56
RECURSIVIDAD	58
Ejemplo de llamadas recursivas.....	58
Definición formal de la recursividad	59
Cuando no usar la recursividad.....	60

Algoritmos de rastreo inverso.....	61
CREACIÓN DE BIBLIOTECAS.....	64
Introducción	64
Creación y uso de bibliotecas estáticas.....	65

Programación orientada a objetos

Clases

Las clases son una manera lógica de organizar datos y funciones en una estructura única. Se declaran utilizando la palabra clave **class**.

Su forma es:

```
class Nombre_de_la_clase {  
    etiqueta_de_permisos_1:  
        atributos_1;  
        métodos_1;  
    etiqueta_de_permisos_2:  
        atributos_2;  
        métodos_2;  
} nombre_del_objeto;
```

donde el `nombre_de_la_clase` es el nombre de la clase (un tipo definido por el usuario) y el campo opcional `nombre_del_objeto`, es uno o varios identificadores de objetos válidos. El cuerpo de la declaración puede contener miembros, que pueden ser tanto declaraciones de datos (atributos) o de funciones (métodos), y opcionalmente etiquetas de permisos, que pueden ser cualquiera de las 3 palabras reservadas: **private:**, **public:** o **protected:**. Estas hacen referencias a los permisos que los miembros podrán tener:

- **private:** las funciones miembro de la clase son accesibles sólo por otros miembros de la misma clase o de sus clases amigas friendo
- **protected:** los miembros son accesibles, además de por los miembros de la misma clase y de sus clases friend, por los miembros de las clases derivadas.
- **public:** son accesibles por cualquiera para el cual la clase es visible.

Si se declaran los miembros de una clase antes de incluir cualquier etiqueta, se consideran privados, por lo que se indica que los permisos por defecto para los miembros es `private`.

Por ejemplo:

```
class Rectangulo {  
    int x, y;  
    public:  
        void fijar_valores (int,int);  
        int area (void);  
} rect;
```

Declara la clase `Rectangulo` y un objeto llamado `rect` de esta clase (tipo). Esta clase contiene cuatro miembros: dos variables (atributos) del tipo `int` (`x` e `y`) en la sección `private` (debido que el permiso por defecto es el privado) y dos funciones (métodos) en la sección `public`: `fijar_valores()` y `area()`, de las que sólo se ha incluido el prototipo.

Note la diferencia entre el nombre de la clase y el nombre del objeto: En el ejemplo anterior, Rectangulo es el nombre de la clase (tal como el nombre de un tipo definido por el usuario), mientras que rect es un objeto del tipo Rectangulo. La misma diferencia existe entre int y a en la siguiente declaración:

```
int a;
```

int es el nombre de la clase (type) y a es el nombre del objeto (variable).

Por estilo de sintaxis adoptamos una C mayúscula al principio del indicador del nombre de la clase.

En las sucesivas instrucciones del cuerpo de los programas nos referiremos a los cualquiera de los miembros públicos del objeto rect como si fueran funciones normales o variables, colocando el nombre del objeto seguido por un punto y luego el miembro de la clase. Por ejemplo

```
rect.fijar_valores (3,4);  
mi_area = rect.area();
```

pero no podremos referirnos a x ni a y porque son miembros privados (private) de la clase y pueden ser únicamente referidos por otros miembros de la misma clase.

A continuación hay un ejemplo completo de Rectangulo

```
// ejemplo de clase  
#include <iostream>  
using namespace std;  
  
class Rectangulo {  
    int x, y;  
public:  
    void fijar_valores (int,int);  
    int area (void) {return (x*y);}  
};  
  
void Rectangulo::fijar_valores (int a, int b)  
{  
    x = a;  
    y = b;  
}  
  
int main () {  
    Rectangulo rect;  
    rect.fijar_valores (3,4);  
    cout << "área: " << rect.area();  
}
```

área: 12

Lo nuevo en este código es el operador :: de ámbito incluido en la definición de fijar_valores(). Se utiliza para declarar un miembro de la clase fuera de ella. Observar que se ha definido el comportamiento de la función miembro area() dentro de la definición de la clase Rectangulo debido a su extremada simplicidad. Mientras que fijar_valores() tiene solamente su prototipo declarado dentro de la clase pero su

definición está fuera. En esta declaración fuera de la clase es donde se utilizó el operador de ámbito `::`.

El operador de ámbito (`::`) permite especificar la clase a la que la función miembro que está siendo declarada pertenece, tomando exactamente las mismas propiedades de ámbito que si fuera directamente declarada dentro de la clase. Por ejemplo, en la función `fijar_valores()` del código anterior, hemos recurrido a las variables **x** e **y**, que son miembros de la clase `Rectangulo` y que son visibles solamente dentro de ella y sus miembros (debido a que son privadas).

La única razón para que se haya hecho a los miembros `x` e `y` `private` (recordar que por defecto todos los miembros de una clase definidos con la palabra reservada `class` tiene acceso privado) es porque ya se ha definido una función para introducir estos valores al objeto (`fijar_valores()`) y por lo tanto el resto del programa no tiene porqué acceder directamente a ellos. Si bien en ejemplos sencillos como los que se han desarrollado hasta el momento no se ve la utilidad de proteger estas dos variables, en proyectos más grandes es muy importante que los valores no puedan ser modificados de una manera inesperada (desde el punto de vista del objeto).

Una de las mayores ventajas de la clases es que se puede definir varios objetos diferentes de ellas. Por ejemplo, siguiendo con el ejemplo de la clase `Rectangulo`, se puede declarar el objeto `rectb` además del `rect`:

```
// ejemplo de clase
#include <iostream>
using namespace std;

class Rectangulo {
    int x, y;
public:
    void fijar_valores (int,int);
    int area (void) {return (x*y);}
};

void Rectangulo::fijar_valores (int a, int b)
{
    x = a;
    y = b;
}

int main () {
    Rectangulo rect, rectb;
    rect.fijar_valores (3,4);
    rectb.fijar_valores (5,6);
    cout << "área rect: " << rect.area() << endl;
    cout << "área rectb: " << rectb.area() << endl;
}
```

```
área rect: 12
área rectb: 30
```

Observar que no se obtiene el mismo resultado de la llamada a `rect.area()` que de la llamada `rectb.area()`. Para explicarlo de alguna manera, cada objeto de la clase `Rectangulo` tiene sus propias variables `x` e `y`, y sus propias funciones `fijar_valores()` y `area()`.

En esto esta basado el concepto de *objeto y programación orientada a objetos*. En que los datos y las funciones pertenecen al objeto, en vez del punto de vista habitual de objetos como parámetros de funciones en la programación estructurada. En esta sección y en la siguiente se discutirá sobre la ventaja de esta metodología.

En este caso concreto, la clase (tipo del objeto) del que estamos hablando es Rectangulo, del que hay dos instancias, u objetos rect y rectb, cada uno con sus propias variables y funciones miembro.

Constructores y destructores

Los objetos generalmente necesitan inicializar variables o asignar memoria dinámica durante su proceso de creación para convertirse en totalmente operativos y para evitar que retorne valores no esperados en la ejecución. Por ejemplo, ¿qué podría pasar si en el ejemplo anterior se llamara a la función `area()` antes de haber invocado a `fijar_valores()`? Probablemente se obtendría un valor indeterminado ya que a los miembros `x` e `y` nunca le fue asignado un valor.

Para evitar esto, una clase debe incluir una función especial llamada **constructor**, que puede ser declarada como una función miembro con el mismo nombre de la clase. Esta función constructor será llamada automáticamente cuando una nueva instancia de la clase sea creada (cuando declare un nuevo objeto o asigne un objeto de esta clase) y sólo en ese momento. A continuación se implementará Rectangulo de manera de incluir un constructor.

```
// ejemplo de clase
#include <iostream>
using namespace std;

class Rectangulo {
    int ancho, largo;
public:
    Rectangulo (int,int);
    int area (void) {return (ancho*largo);}
};

Rectangulo::Rectangulo (int a, int b) {
    ancho = a;
    largo = b;
}

int main () {
    Rectangulo rect (3,4);
    Rectangulo rectb (5,6);
    cout << "área rect: " << rect.area() << endl;
    cout << "área rectb: " << rectb.area() << endl;
}
```

```
área rect: 12
área rectb: 30
```

Como se puede ver, el resultado de los ejemplos es idéntico a los previos. En este caso solamente se ha reemplazado la función `fijar_valores()`, que ya no existe, por el constructor. Notar la manera en que los parámetros se pasan al constructor al momento en que se crea una instancia de la clase.

```
Rectangulo rect (3,4);  
Rectangulo rectb (5,6);
```

Se puede ver también como ni en el prototipo ni más adelante en la declaración del constructor se incluye un valor de retorno, aunque no es del tipo void, esto debe ser siempre así. Un constructor jamás devuelve un valor aunque no se especifique void. Como se ha hecho en el ejemplo previo.

El **destructor** cumple la función opuesta. Esta función se llama automáticamente cuando el objeto es desalojado de la memoria, ya sea porque el ámbito de su existencia haya finalizado (por ejemplo, si fue definido como un objeto local dentro de una función y la función finaliza) o debido a que es un objeto asignado dinámicamente y el liberado usando el operador **delete**.

El destructor debe tener el mismo nombre que el de la clase con el tilde (~) como prefijo y no debe retornar ningún valor.

El uso de los destructores es especialmente adecuado cuando un objeto asigna memoria dinámica durante su vida y en el momento de ser destruido se quiere liberar la memoria que ha utilizado.

```
// ejemplo de constructores y destructores  
#include <iostream>  
using namespace std;  
  
class Rectangulo {  
    int *ancho, *largo;  
public:  
    Rectangulo (int,int);  
    ~Rectangulo ();  
    int area (void) {return (*ancho * *largo);}  
};  
  
Rectangulo::Rectangulo (int a, int b) {  
    ancho = new int;  
    largo = new int;  
    *ancho = a;  
    *largo = b;  
}  
  
Rectangulo::~~Rectangulo () {  
    delete ancho;  
    delete largo;  
}  
  
int main () {  
    Rectangulo rect (3,4), rectb (5,6);  
    cout << "área rect: " << rect.area()  
        << endl;  
    cout << "área rectb: " << rectb.area()  
        << endl;  
    return (0);  
}
```

```
área rect: 12  
área rectb: 30
```

Sobrecarga de constructores

Al igual que cualquier otra función, un constructor puede también ser sobrecargado con varias funciones que tienen el mismo nombre pero diferente tipo o cantidad de parámetros. Recordar que el compilador ejecutará la que coincida con el nombre al momento de ser invocada. En el caso de los constructores, al momento en que el objeto es declarado.

De hecho, en los casos donde se declara una clase y no se especifica ningún constructor, el compilador asume automáticamente dos constructores sobrecargados (el "constructor por defecto" y "el constructor de copia"). Por ejemplo, para la clase:

```
class CEjemplo {
public:
    int a,b,c;
    void multiplicar (int n, int m) { a=n; b=m; c=a*b; };
};
```

que no posee constructores, el compilador automáticamente asume que tiene las siguientes funciones miembro:

Constructor vacío

Es un constructor que no tiene parámetros definido y un bloque de instrucciones vacío. Aunque no hace nada, es necesario en algunos casos como en los ejemplos que se ven más adelante.

```
CEjemplo::CEjemplo () { };
```

Constructor de copia

Este es un constructor con solo un parámetro de su mismo tipo que asigna a todas las variables miembros no estáticos de la clase del objeto una copia de los valores del objeto pasado como parámetro.

```
CEjemplo::CEjemplo (const CEjemplo& rv) {
    a=rv.a; b=rv.b; c=rv.c;
}
```

Es importante destacar que ambos constructores por defecto: el constructor vacío y el constructor de copia existen sólo si ningún otro constructor es explícitamente declarado. En el caso de que algún constructor con cualquier cantidad de parámetros sea declarado ninguno de estos dos constructores por defecto existirán, por lo que si queremos que ocurra esto, deberemos definir los nuestros.

Por supuesto, se puede también sobrecargar los constructores de la clase teniendo diferentes constructores para cuando pasa parámetros entre paréntesis o cuando no especifica ninguno (vacío).

```
// sobrecargando constructores
#include <iostream>
```

```
área rect: 12
área rectb: 30
```

```
using namespace std;

class Rectangulo {
    int ancho, largo;
public:
    Rectangulo ();
    Rectangulo (int,int);
    int area (void) {
        return (ancho*largo);}
};

Rectangulo::Rectangulo () {
    ancho = 5;
    largo = 5;
}

Rectangulo::Rectangulo (int a, int b) {
    ancho = a;
    largo = b;
}

int main () {
    Rectangulo rect (3,4);
    Rectangulo rectb;
    cout << "área rect: " << rect.area()
        << endl;
    cout << "área rectb: " << rectb.area()
        << endl;
}
```

En este caso `rectb` se declaró sin parámetros, por lo que será inicializado por el constructor sin parámetros, que declara tanto a `ancho` como a `largo` con el valor de 5. Notar que si se declara un nuevo objeto y no se le quiere pasar parámetros no se debería incluir los paréntesis ().

```
Rectangulo rectb; // correcto
Rectangulo rectb(); // incorrecto!
```

Vector de objetos

La clase 'Vector' puede contener objetos con la misma simplicidad que se almacenan valores numéricos. También hay que tener en cuenta que según como se creen los elementos del vector, se llamará al constructor para cada uno de los objetos. El siguiente ejemplo ilustra esta característica:

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

class LaClase
{ private:
    int x;
public:
    LaClase() {x=rand()%100;};
```

```

    int devuelve_x() {return x;};
};

int main()
{
    vector <LaClase> ListaObj_1;
    for (unsigned i=0;i<5;i++)
    { LaClase obj_aux;
      ListaObj_1.push_back(obj_aux);
    }

    for (unsigned i=0;i<ListaObj_1.size();i++)
        cout << ListaObj_1[i].devuelve_x() << " ";
    cout << endl;

    //=====
    vector <LaClase> ListaObj_2(5);
    for (unsigned i=0;i<ListaObj_2.size();i++)
        cout << ListaObj_2[i].devuelve_x() << " ";
    cout << endl;

    cin.get();
    return (0);
}

```

El vector ListaObj_1 el llenado llamando al constructor de cada uno de los objetos, por lo tanto, el atributo x es inicializado por medio de la función rand(), dando como resultado valores distintos de x para cada uno de los objetos. En el segundo caso, ListaObj_2 es creada de la siguiente forma: se llama al constructor del primero y luego la clase vector fue codificada internamente para que copie el mismo objeto en los siguientes elementos, de esta manera se tiene el mismo valor del atributo de x para todos los objetos. Un resultado similar a este último se hubiese logrado si se utilizaba el siguiente código:

```

vector <LaClase> ListaObj_2;
ListaObj_2.resize(5);

```

Punteros a clases

Es perfectamente válido crear punteros que apunten a objetos, para realizar esto se puede considerar que una vez declarada, la clase llega a ser un tipo válido, de manera que use el nombre de la clase como el tipo para el puntero. Por ejemplo:

```

Rectangulo * punt_a_rect;

```

Es un puntero a un objeto de la clase Rectangulo.

Como ocurre con las estructuras de datos, para referirse directamente al miembro de un objeto apuntado por un puntero debemos utilizar el operador ->. A continuación se muestra un ejemplo de algunas combinaciones posibles:

```

// punteros a clases

```

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

class Rectangulo {
    int ancho, largo;
public:
    void fijar_valores (int, int);
    int area (void) {return (ancho * largo);}
};

void Rectangulo::fijar_valores (int a, int b) {
    ancho = a;
    largo = b;
}

int main () {
    Rectangulo a, *b, *c;
    Rectangulo * d = new Rectangulo[2];
    b= new Rectangulo;
    c= &a;
    a.fijar_valores (1,2);
    b->fijar_valores (3,4);
    d->fijar_valores (5,6);
    d[1].fijar_valores (7,8);
    cout << "área de: " << a.area() << endl;
    cout << "área de *b: " << b->area() << endl;
    cout << "área de *c: " << c->area() << endl;
    cout << "área de d[0]: " << d[0].area() << endl;
    cout << "área de d[1]: " << d[1].area() << endl;
    return (0);
}
```

```
área de *b: 12
área de *c: 2
área de d[0]: 30
área de d[1]: 56
```

A continuación se muestra un resumen de cómo puede leer algunos operadores de clase y punteros (*, &, ., ->, []) que aparecen en el ejemplo precedente:

```
*x puede ser leído como: apuntado por x
&x puede ser leído como: dirección de x
x.y puede ser leído como: miembro y del objeto x
(*x).y puede ser leído como: miembro y del objeto apuntado por x
x-y puede ser leído como: miembro y del objeto apuntado por x
(equivalente al anterior)
x[0] puede ser leído como: primer objeto apuntado por x
x[1] puede ser leído como: segundo objeto apuntado por x
x[n] puede ser leído como: (n+1)ésimo objeto apuntado por x
```

Sobrecarga de operadores

C++ incorpora la opción de usar operadores estandar del lenguaje entre clases además de poder usarlos entre tipos fundamentales. Por ejemplo:

```
int a, b, c;
a = b + c;
```

es perfectamente válido, dado que las distintas variables en la suma son todas de tipos fundamentales. No es así de directo cuando se quiere sumar objetos, porque estos pueden tener una variedad de atributos que debería indicarse cómo se realiza su suma.

Pero gracias a la habilidad de C++ para sobrecargar operadores, se puede hacer que objetos puedan aceptar operadores que no serían aceptados de otra forma o aún modificar el efecto de operadores que ya admiten. Aquí hay una lista de todos los operadores que pueden ser sobrecargados:

```
+      -      *      /      =      <      >      +=     -=     *=     /=     <<    >>
<<=   >>=   ==     !=     <=    >=    ++     --     %      &      ^      !      |
~     &=    ^=    |=    &&    ||    %=    []    ()    new   delete
```

Para sobrecargar un operador sólo se requiere escribir una función miembro de una clase cuyo nombre es **operador** seguido por el signo del operador que se desea sobrecargar. El prototipo es el siguiente:

```
tipo operador signo (parametros);
```

Ejemplo de operador binario

Aquí hay un ejemplo que incluye al operador **+**, que nos permitirá sumar dos complejos. La suma es tan simple como sumar las dos partes reales e imaginarias respectivamente.

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

class CComplejo {
private:
    float re, im;
public:
    CComplejo () {};
    CComplejo (float a, float b) {re=a; im=b;}
    void mostrar();
    CComplejo CComplejo::operator+ (CComplejo &valor)
    { CComplejo aux;
      aux.re = re + valor.re;
      aux.im = im + valor.im;
      return(aux); //devuelve el resultado de la suma
    }
};

void CComplejo::mostrar()
{ cout << re;
  if (im>0) cout << "+";
  cout << im << "i" << endl << endl;
}

int main () {
    CComplejo a (3,1);
    CComplejo b (1,2);
    CComplejo c;
```

```
3+1i
1+2i
**** c = a+b ***** 4+3i
*** c = a+b+c *** 8+6i
```

```

a.mostrar();
b.mostrar();

cout << "**** c = a+b ***** ";
c = a+b; // ver "a+b" como si fuera "a.+(b)"
c.mostrar();

cout << "**** c = a+b+c *** ";
c = a+b+c; // esta sintaxis también es válida
c.mostrar();

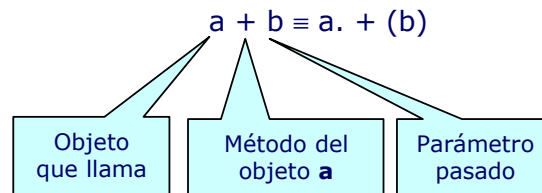
cin.get();
return (0);
}

```

La función **operator+** de la clase CComplejo es la que está a cargo de sobrecargar el operador aritmético + que nos permite hacer:

```
c = a + b;
```

Esta operación puede suponerse que tiene la siguiente equivalencia:



Observa que también se incluyó el constructor vacío (sin parámetros) y se definió con un bloque sin instrucciones (vacío):

```
CComplejo () { };
```

Esto es necesario, dado que ya existe otro constructor,

```
CComplejo (int, int);
```

y ninguno de los constructores por defecto existirá en CComplejo si no se lo declara explícitamente. De otra forma la declaración

```
CVector c;
```

incluida en main() no sería válida.

De todas maneras, se debe advertir que un bloque vacío no es una implementación recomendada para un constructor, dado que no satisface la funcionalidad mínima que un constructor debería tener, que es la inicialización de todas las variables de la clase. En este caso este constructor deja las variables **re** e **im** indefinidas. Por lo tanto, una declaración más recomendable sería:

```
CComplejo () { re=0; im=0; };
```

Que por simplicidad no fue incluida en el código.

Así como las clases incluyen por defecto un constructor vacío y un constructor de copia, también incluyen una definición por defecto del operador de asignación (=) entre dos clases del mismo tipo. Este copia el contenido de todos los datos no estáticos del parámetro objeto (el que está a la derecha del signo =) que se encuentra del lado izquierdo. Por supuesto, se puede redefinir con cualquier otra funcionalidad que se desee para este operador, como por ejemplo, copiar sólo ciertos miembros de la clase.

La sobrecarga de operadores no necesariamente obliga a mantener una relación matemática o el significado usual con el operador, aunque esto último es recomendado. Por ejemplo, no es muy lógico usar el operador + para realizar una resta entre dos clases o usar el operador == para inicializar a cero todas las variables, aunque es perfectamente posible hacerlo.

El uso de *this*

this representa dentro de una clase la dirección en memoria del objeto de esa clase que está siendo ejecutado. Es un puntero cuyo valor es siempre la dirección de memoria del objeto.

Es frecuentemente usado en las funciones **operator=** que retornan objetos por referencia (evitando el uso de objetos temporarios). Siguiendo con el ejemplo de vectores podríamos haber escrito una función operator= como esta:

```
CVector& CVector::operator= (const CVector& param)
{
    x=param.x;
    y=param.y;
    return *this;
}
```

de hecho es el probable código por defecto generado por la clase si no incluimos ninguna función operator=.

Ejemplo de operador unario

Este ejemplo muestra el operador unario de cambio de signo (-objeto) y el de incremento (++objeto).

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

class CComplejo {
private:
    float re, im;
public:
    CComplejo () {};
    CComplejo (float a, float b) {re=a; im=b;}
```

```
*** b = -a ***** -3-1i
*** -a ***** 3+1i
*** ++a ***** 4+1i
```

```
void mostrar();

CComplejo CComplejo:: operator-()
{ re = -re;
  im = -im;
  return (*this); // devuelve él mismo
}

CComplejo CComplejo:: operator++()
{ // define ++objeto
  re++;
  im++;
  return (*this);
}
};

void CComplejo::mostrar()
{ cout << re;
  if (im>0) cout << "+";
  cout << im << "i" << endl << endl;
}

int main () {
  CComplejo a (3,1);
  CComplejo b;

  cout << "*** b = -a ***** ";
  b=-a; // cambia signo y asigna
  b.mostrar();

  cout << "*** -a ***** ";
  -a; // sólo cambia de signo
  a.mostrar();

  cout << "*** ++a ***** ";
  ++a;
  a.mostrar();

  cin.get();
  return (0);
}
```

Observar que no es lo mismo el operador **++objeto** que **objeto++**. Para este último caso la codificación para la sobrecarga de este operador es con la siguiente sintaxis:

```
CComplejo CComplejo:: operator++(int notused)
{ // define objeto++
  re++;
  im++;
  return (*this);
}
```

Miembros estáticos

Los datos estáticos de una clase son conocidos también como “variables de clase”, porque su contenido no depende de ningún objeto. Sólo hay un único valor para todos los objetos de la misma clase.

Por ejemplo, puede ser usado para que una variable dentro de una clase pueda contener el número de objetos declarados de esa clase, como en el siguiente ejemplo:

```
// miembros estáticos en clases
#include <iostream>
using namespace std;

class CBoba {
public:
    static int n;
    CBoba () { n++; };
    ~CBoba () { n--; };
};

int CBoba::n=0;

int main () {
    CBoba a;
    CBoba b[5];
    CBoba * c = new CBoba;
    cout << a.n << endl;
    delete c;
    cout << CBoba::n << endl;
    return (0);
}
```

```
7
6
```

De hecho, los miembros estáticos tienen las mismas propiedades que las variables globales pero disfrutando del alcance de una clase. Por esta razón, y para evitar que sean declaradas varias veces, sólo podemos incluir el prototipo (declaración) en la clase pero no la definición (inicialización). Para inicializar un dato estático debemos incluir una declaración formal fuera de la clase, en el alcance global, como en el ejemplo previo.

Debido a que es una única variable para todos los objetos desde la misma clase, esto puede ser referido como un miembro de cualquier objeto de la clase o inclusive directamente por el nombre de la clase (por supuesto esto es válido únicamente para miembros estáticos).

```
cout << a.n;
cout << CBoba::n;
```

estas dos llamadas incluidas en el ejemplo previo se refieren a la misma variable: la variable estática n dentro de la clase CBoba.

Una vez más, debe recordarse que de hecho es una variable global. La única diferencia es el nombre fuera de la clase.

Como podemos incluir datos estáticos dentro de una clase también podemos incluir funciones estáticas. Representan lo mismo: son funciones globales que son llamadas como si fueran miembros de una clase determinada. Sólo pueden referirse a datos estáticos, en ningún caso a miembros no-estáticos de la clase, y tampoco permiten el uso de `this`, dado que hacen referencia a un objeto apuntado y estas funciones de hecho no son miembros de ningún objeto sino directamente miembros de la clase.

Relación entre clases

Funciones amigas (*friend*)

En la sección previa se vio que había tres niveles de protección interna para los diferentes miembros de una clase: `public`, `protected` y `private`. En el caso de los miembros `protected` y `private`, no se puede acceder a ningún miembro fuera de la misma clase en la que fueron declarados. Sin embargo, esta regla puede ser transgredida usando la palabra `friend` en una clase, por medio de la cual podemos permitir que una función externa obtenga acceso a los miembros `protected` y `private` de una clase.

Para permitir que una función externa tenga acceso a los miembros `private` y `protected` de una clase debemos declarar el prototipo de la función externa que obtendrá el acceso precedida de la palabra `friend` dentro de la declaración de la clase que compartirá sus miembros. En el siguiente ejemplo se declara la función amiga `duplicar`:

```
// funciones amigas
#include <iostream>
using namespace std;

class Rectangulo {
    int ancho, largo;
public:
    void fijar_valores (int, int);
    int area (void) {return (ancho * largo);}
    friend Rectangulo duplicar (Rectangulo);
};
void Rectangulo::fijar_valores (int a, int b) {
    ancho = a;
    largo = b;
}
Rectangulo duplicar (Rectangulo rectparam)
{
    Rectangulo rectres;
    rectres.ancho = rectparam.ancho*2;
    rectres.largo = rectparam.largo*2;
    return (rectres);
}
int main () {
    Rectangulo rect, rectb;
    rect.fijar_valores (2,3);
    rectb = duplicar (rect);
    cout << rectb.area();
}
```

24

```
}
}
```

Desde dentro de la función `duplicar()` que es amiga de la clase `Rectangulo`, se accede a los miembros `ancho` y `largo` de diferentes objetos de tipo `Rectangulo`. Note que ni en la declaración de `duplicar()` ni en su posterior uso en `main()` se consideró a `duplicar()` como miembro de la clase `Rectangulo`, dado que no es así.

Las funciones amigas pueden servir, por ejemplo, para realizar operaciones entre dos clases diferentes. Generalmente el uso de funciones amigas está fuera de la metodología de la programación orientada a objetos, así que cuando sea posible es mejor intentar usar miembros de la misma clase para llevar a cabo un proceso. Como en el ejemplo previo, en el que hubiera sido mejor integrar a la función `duplicar()` dentro de la clase.

Unos de los operadores que resulta muy útil sobrecargar son los flujos de entrada y salida, de esta manera podremos ingresar y enviar a la salida los valores de los tipos de datos que hayamos definido. Consideremos el caso del operador inserción de flujo `<<` el que debe tener un operando izquierdo del tipo `ostream&` (como `cout` de la expresión `cout<<objeto de clase`) por lo que según lo que sería necesario que fuera una función no miembro. De la misma manera el operador `>>` debe tener a la izquierda un operador de la clase `istream&`. Además cada una de estas funciones deberían poder acceder a los datos miembros `private` de la clase, por lo que deberíamos trabajar con funciones `friend`. A continuación se presenta un ejemplo del operador de inserción:

```
// función amiga para sobrecargar "<<"
# include <iostream>
using namespace std;

class CComplejo {
private:
    float re, im;
public:
    CComplejo () {};
    CComplejo (float a, float b) {re=a; im=b;}

    CComplejo CComplejo::operator+ (CComplejo &valor)
    { CComplejo aux;
      aux.re = re + valor.re;
      aux.im = im + valor.im;
      return (aux);
    }

    friend ostream & operator<< (ostream& sal, CComplejo c);
};

ostream& operator<< (ostream& sal, CComplejo aux)
{ cout << aux.re;
  if (aux.im>0) cout << "+";
  cout << aux.im << "i";

  return sal;    // permite concatenar con otros "<<"
};
```

```
4+3i
3+1i  1+2i
```

```

int main () {
    CComplejo a (3,1);
    CComplejo b (1,2);
    CComplejo c;

    c = a+b;
    cout << c << endl << endl;
    cout << a << "    " << b << endl << endl;

    cin.get ();
    return (0);
}

```

El nuevo operador <<() usa un ostream con el nombre sal. Normalmente sal se referirá al objeto cout, pero tengamos en cuenta que podría referirse a cualquier objeto stream de salida por ello estamos usando sal como alias para cualquiera sea. Respecto al objeto del tipo CComplejo, podríamos haberlo pasado por valor o por referencia; aquí escogimos pasarlo por referencia porque usa menos memoria y tiempo, que el pasaje por valor.

Note que el tipo que retorna la función es ostream &. Esto significa que la función retorna una referencia a un ostream, debido a que el programa pasa una referencia a una objeto al comienzo, el efecto neto es que la función devuelve exactamente un valor igual al objeto que se le pasó.

Clases amigas (*friend*)

Como existe la posibilidad de declarar funciones amigas, también se puede definir una clase como amiga de otra, permitiendo que la segunda acceda a los miembros protected y private de la primera.

```

// clases amigas
#include <iostream>
using namespace std;

class Cuadrado;
class Rectangulo {
    int ancho, largo;
public:
    int area (void)
        {return (ancho * largo);}
    void convertir (Cuadrado a);
};
class Cuadrado {
private:
    int lado;
public:
    void fijar_lado (int a)
        {lado=a;}
    friend class Rectangulo;
};
void Rectangulo::convertir (Cuadrado a) {
    ancho = a.lado;
}

```

16

```
    largo = a.lado;
}

int main () {
    Cuadrado sqr;
    Rectangulo rect;
    sqr.set_lado(4);
    rect.convertir(sqr);
    cout << rect.area();
    return (0);
}
```

En este ejemplo se declara a Rectangulo como amiga de Cuadrado para que Rectangulo pueda acceder a los miembros protected y private de Cuadrado, más concretamente a Cuadrado::lado, que define la longitud del lado del cuadrado.

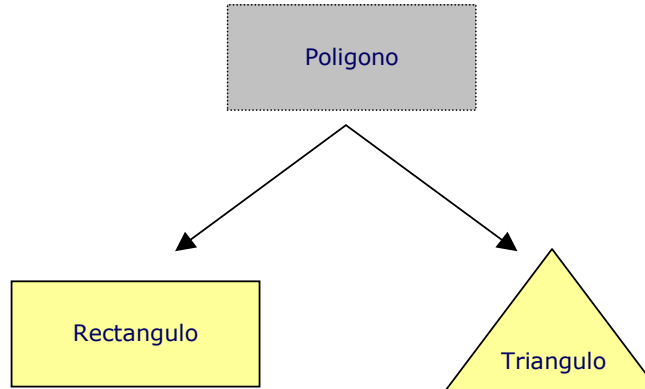
Se nota también como algo nuevo la primera instrucción del programa, que es un prototipo de la clase Cuadrado, esto es necesario porque dentro de la declaración de Rectangulo se hace referencia a Cuadrado (como parámetro en convertir()). La definición de Cuadrado es incluida posteriormente, así que si no se incluye una definición previa para Cuadrado esta clase no sería visible desde adentro de la clase Rectangulo.

Considere que las amistades no son correspondidas si no lo especificamos explícitamente. En el ejemplo de Cuadrado, Rectangulo es considerada una clase amiga, pero Rectangulo no hace lo propio con Cuadrado, así que Rectangulo puede acceder a los miembros protected y private de Cuadrado pero no a la inversa. Aunque nada impide declarar también a Cuadrado como amiga de Rectangulo.

Herencia entre clases

Una importante propiedad de las clases es la *herencia*. Esto permite crear un objeto derivado de otro, así que éste podría incluir algunas propiedades de otros objetos más las propias. Por ejemplo, si se quiere declarar una serie de clases que describan polígonos como Rectangulo o Triangulo. Ambas poseen algunas características en común, por ejemplo, que ambas pueden ser descritas por medio de sólo dos lados: altura y base.

Esto puede ser representado con una clase Poligono de la cual derivan las dos anteriores, Rectangulo y Triangulo.



La clase CPoligono contendrá miembros que son comunes a todos los polígonos. En nuestro caso: ancho y largo. Rectangulo y Triangulo sería sus clases derivadas.

Las clases derivadas heredan todos los miembros visibles de la clase base. Esto significa que si una clase base incluye un miembro A y de ella se deriva otra clase con otro miembro llamado B, la clase derivada contendrá ambos, A y B.

Para derivar una clase de otra, se usa el operador : (dos puntos) en la declaración de la clase derivada de la siguiente manera:

```
class nombre_clase_derivada: public nombre_clase_base { ... };
```

donde nombre_clase_derivada es el nombre de la clase derivada y nombre_clase_base es el nombre de la clase en la cual está basada. public puede ser reemplazado por cualquiera de los otros especificadores de nivel de acceso como protected o private, y describe el acceso para los miembros heredados, como se ve después de este ejemplo:

```
// clases derivadas
#include <iostream>
using namespace std;

class Poligono {
protected:
    int ancho, largo;
public:
    void fijar_valores (int a, int b)
        { ancho=a; largo=b;}
};
class Rectangulo: public Poligono {
public:
    int area (void)
        { return (ancho * largo); }
};
class Triangulo: public Poligono {
public:
    int area (void)
        { return (ancho * largo / 2); }
};
```

```
20
10
```

```

int main () {
    Rectangulo rect;
    Triangulo trgl;
    rect.fijar_valores (4,5);
    trgl.fijar_valores (4,5);
    cout << rect.area() << endl;
    cout << trgl.area() << endl;
    return (0);
}

```

Los objetos de las clases Rectangulo y Triangulo contienen todos los miembros de Poligono: ancho, largo y fijar_valores().

Cuando se deriva una clase, los miembros protected de la clase base pueden ser usados por otros miembros de la clase derivada, lo que no ocurre con los miembros private. Como se quiso que ancho y largo pudieran ser manipulados por miembros de las clases derivadas Rectangulo y Triangulo, y no sólo por miembros de Poligono se usó el nivel de acceso protected en vez de private.

Se puede resumir los diferentes tipos de acceso de acuerdo a quién puede acceder a ellos en la siguiente manera:

Acceso	public	Protected	private
miembros de la misma clase	Sí	Sí	sí
miembros de clases derivadas	Sí	sí	no
no-miembros	Sí	no	no

Polimorfismo

Punteros a clases base

Una de las mayores ventajas de derivar clases es que un puntero a una clase descendiente es de tipo compatible con un puntero a su clase base. Esta sección se dedica totalmente a explotar las ventajas de esta potente característica de C++. Por ejemplo, vamos a reescribir nuestro programa acerca de rectángulos y triángulos de la sección anterior, teniendo en cuenta esta propiedad.

```

// punteros a clase base
#include <iostream>
using namespace std;

class Poligono {
protected:
    int ancho, largo;
public:
    void fijar_valores (int a, int b)
    { ancho=a; largo=b; }
};

```

20
10

```
class Rectangulo: public Poligono {
public:
    int area (void)
        { return (ancho * largo); }
};

class Triangulo: public Poligono {
public:
    int area (void)
        { return (ancho * largo / 2); }
};

int main () {
    Rectangulo rect;
    Triangulo trgl;
    Poligono * ppoly1 = &rect;
    Poligono * ppoly2 = &trgl;
    ppoly1->fijar_valores (4,5);
    ppoly2->fijar_valores (4,5);
    cout << rect.area() << endl;
    cout << trgl.area() << endl;
    return (0);
}
```

La función main crea dos punteros que apuntan a objetos de la clase Poligono, que son *ppoly1 y *ppoly2. A estos se le asigna las direcciones de rect y trgl, que debido a que son objetos de clase derivadas de Poligono son asignaciones válidas.

La única limitación de usar ppoly1 y ppoly2, en vez de rect y trgl, es que tanto ppoly1 como ppoly2 son del tipo Poligono* y por lo tanto sólo podremos referirnos a los miembros que Rectangulo y Triangulo heredan de Poligono. Por esta razón cuando se llama a los miembros area() no se puede utilizarlos (*ppoly1 y *ppoly2)

Para que los punteros de la clase Poligono admitan area() como un miembro válido, también debería haber sido declarado en la clase base y no sólo en sus descendientes. (vea la siguiente sección)

Miembros virtuales

Para declarar un elemento de una clase que se va a redefinir en las clases descendientes se debe precederlo con la palabra reservada **virtual** para suplir los problemas anteriores.

Observar el ejemplo siguiente:

```
// miembros virtuales
#include <iostream>
using namespace std;

class Poligono {
protected:
```

```
20
10
0
```

```
int ancho, largo;
public:
void fijar_valores (int a, int b)
{ ancho=a; largo=b; }
virtual int area (void)
{ return (0); }
};

class Rectangulo: public Poligono {
public:
int area (void)
{ return (ancho * largo); }
};

class Triangulo: public Poligono {
public:
int area (void)
{ return (ancho * largo / 2); }
};

int main () {
Rectangulo rect;
Triangulo trgl;
Poligono poly;
Poligono * ppoly1 = &rect;
Poligono * ppoly2 = &trgl;
Poligono * ppoly3 = &poly;
ppoly1->fijar_valores (4,5);
ppoly2->fijar_valores (4,5);
ppoly3->fijar_valores (4,5);
cout << ppoly1->area() << endl;
cout << ppoly2->area() << endl;
cout << ppoly3->area() << endl;
return (0);
}
```

Ahora las tres clases (Poligono, Rectangulo y Triangulo) tienen los mismos miembros: ancho, largo, fijar_valores() y area().

area() se ha definido como virtual debido a que luego se redefine en las clases descendientes.

Se puede verificar que si se elimina esta palabra reservada (virtual) del código y entonces ejecuta el programa el resultado será 0 para los tres polígonos, en vez de 20,10,0. Esto podría deberse a que en vez de llamar a la función area() correspondiente a cada objeto (Rectangulo::area(), Triangulo::area() y Poligono::area(), respectivamente), se llamará Poligono::area() para todos los casos ya que se utiliza un puntero a Poligono.

Por lo tanto es la palabra virtual la que indica que los miembros de clases descendientes que posean el mismo nombre que uno de la clase base, se llame adecuadamente cuando se utilicen punteros.

Clases bases abstractas

Las clases base abstractas muchas veces son muy similares a la clase Poligono del ejemplo previo. La única diferencia es que en el ejemplo anterior se ha definido la función válida `area()` para los objetos de la clase Poligono, mientras que en una clase base abstracta se puede simplemente dejar sin definir esta función agregándole `=0` (igual a cero) en la declaración de la función.

La clase Poligono puede entonces ser como:

```
// clase abstracta Poligono
class Poligono {
protected:
    int ancho, largo;
public:
    void fijar_valores (int a, int b)
        { ancho=a; largo=b; }
    virtual int area (void) =0;
};
```

Se ha agregado `=0` a `virtual int area (void)` en vez de especificar una implementación para la función. Este tipo de funciones reciben el nombre de funciones virtuales puras, y todas las clase que contienen funciones virtuales puras son consideradas clases base abstractas.

La gran diferencia de una clase base abstracta es que no pueden ser creadas instancias (objetos) de ella. Pero se puede crear punteros a ella. Por lo que una declaración como:

```
Poligono poly;
```

Podría ser incorrecta para una clase base abstracta como la declarada antes. Sin embargo los punteros:

```
Poligono * ppoly1;
Poligono * ppoly2;
```

Son perfectamente válidos. Esto es debido a que las funciones virtuales puras no están definidas y es imposible crear un objeto si no tiene todos sus miembros definidos. Sin embargo un puntero a un objeto de una clase descendiente donde esta función ha sido definida es perfectamente válido.

A continuación se muestra un ejemplo completo:

```
// miembros virtuales
#include <iostream>
using namespace std;

class Poligono {
protected:
    int ancho, largo;
public:
    void fijar_valores (int a, int b)
```

```
20
10
```

```

        { ancho=a; largo=b; }
        virtual int area (void) =0;
    };

class Rectangulo: public Poligono {
public:
    int area (void)
        { return (ancho * largo); }
};

class Triangulo: public Poligono {
public:
    int area (void)
        { return (ancho * largo / 2); }
};

int main () {
    Rectangulo rect;
    Triangulo trgl;
    Poligono * ppoly1 = &rect;
    Poligono * ppoly2 = &trgl;
    ppoly1->fijar_valores (4,5);
    ppoly2->fijar_valores (4,5);
    cout << ppoly1->area() << endl;
    cout << ppoly2->area() << endl;
    return (0);
}

```

Revisando el programa se puede notar que se puede referir a los objetos de clases diferentes usando un tipo de puntero único (Poligono*). Esto puede ser tremendamente útil. Imaginar ahora que se puede crear una función miembro de Poligono que tiene la capacidad de imprimir en la pantalla el resultado de la función area() independientemente de qué clase derivada sea:

```

// ejemplo miembros virtuales
#include <iostream>
using namespace std;

class Poligono {
protected:
    int ancho, largo;
public:
    void fijar_valores (int a, int b)
        { ancho=a; largo=b; }
    virtual int area (void) =0;
    void imprimir_area (void)
        { cout << this->area() << endl; }
};

class Rectangulo: public Poligono {
public:
    int area (void)
        { return (ancho * largo); }
};

```

20
10

```
class Triangulo: public Poligono {
public:
    int area (void)
    { return (ancho * largo / 2); }
};

int main () {
    Rectangulo rect;
    Triangulo trgl;
    Poligono * ppoly1 = &rect;
    Poligono * ppoly2 = &trgl;
    ppoly1->fijar_valores (4,5);
    ppoly2->fijar_valores (4,5);
    ppoly1->imprimir_area();
    ppoly2->imprimir_area();
    return (0);
}
```

Recordar que this representa un puntero al objeto cuyo código está siendo ejecutado.

Las clases abstractas y los métodos virtuales le otorgan a C++ las características polimórficas que hacen a la programación orientada a objetos un instrumento muy útil. Por supuesto, se ha visto el uso más simple de estas características, pero imaginar estas características aplicadas a arreglos de objetos u objetos asignados a través de memoria dinámica.

Clases abstractas y polimorfismo

Introducción

Los primeros contactos con las clases abstractas, funciones virtuales y el polimorfismo ocasionan un misticismo sobre lo que está sucediendo. Muy lejos de ser una realidad, las características de este concepto son utilizadas por nosotros en la construcción del conocimientos que usamos a diario.

Este documento presenta un ejemplo de uso de las funciones virtuales en C++ para mostrar algunas características de la sintaxis, sin por ello pretender ser un documento sobre los aspectos conceptuales que encierran las funciones virtuales.

Funciones virtuales puras

En el siguiente ejemplo se trabajará con "*funciones virtuales puras*", que pueden ser reconocidas por la palabra **virtual** y también porque la función es seguida por "**=0**". Si se intenta realizar un objeto a partir de esta clase, el compilador va a prevenir que se realice. Se fuerza a que alguna clase derivada provea una definición de esta función.

C++ provee un mecanismo para hacer este llamado en las funciones virtuales puras. La sintaxis es la siguiente:

```
virtual void X() = 0;
```

En el ejemplo siguiente se realizará una clase abstracta Instrumento formando parte de una representación de instrumentos musicales. El objetivo de Instrumento es crear una interfaz común para todas las clases derivadas desde él y no una particular implementación, por lo tanto, crear un objeto del tipo Instrumento, no tiene sentido y debería prevenirse, por ejemplo, dando un mensaje de error.

Si se trabaja con una genuina clase abstracta (como Instrumento), los objetos de esta clase no tendrán sentido.

El siguiente ejemplo modela un conjunto de instrumentos y la posibilidad de ejecutar una nota con ellos:

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

enum nota { DO, RE, MI, FA, SOL, LA, SIb };

class Instrumento { // funciones virtuales puras
public:
    Instrumento() {cout << "Constructor de Instrumento" << endl;}
    virtual void ejecutar(nota) =0;
    virtual string nombre() =0;
    virtual ~Instrumento() {cout << "Destructor de Instrumento" << endl;}
};

class Piano : public Instrumento {
public:
    Piano() { cout << "Constructor de Piano" << endl;}
    void ejecutar(nota) { cout << "Piano::ejecutar" << endl; }
    string nombre() { return "Piano"; }
    ~Piano() { cout << "Destructor de Piano" << endl; }
};

class Violin : public Instrumento {
public:
    Violin() { cout << "Constructor de Violin" << endl;}
    void ejecutar(nota) { cout << "Violin::ejecutar" << endl; }
    string nombre() { return "Violin"; }
    ~Violin() { cout << "Destructor de Violin" << endl; }
};

void EjecutaNotas(Instrumento& i) {
    i.ejecutar(DO);
    i.ejecutar(RE);
    i.ejecutar(SOL);
}

int main() {

    Instrumento* a;

    int opcion;
    cout << "[1] Piano - [2] Violin: ";
    cin >> opcion;

    switch (opcion) {
```

```

    case 1: a = new Piano; break;
    default: a = new Violin;
}

cin.get(); //=====

cout << "Instrumento a ejecutar : " << a->nombre() << endl;
a->ejecutar(SOL);

cin.get(); //=====

EjecutaNotas (*a);

cin.get(); //=====

cout << "COMIENZO DESTRUCTORES" << endl;
delete a;

cin.get();
return (0);
}

```

No es necesario repetir la palabra reservada "virtual" en ninguna de las redefiniciones de la función miembro en las clases derivadas porque si una función es declarada como "virtual" en la clase base, ella es virtual en todas las clases derivadas. Sin embargo, es un buen hábito de programación dejar claro (mediante la palabra "virtual") cuales son las funciones miembro virtuales y cuales no.

Polimorfismo y uso de contenedores

Aunque los contenedores (*containers*) en las STL pueden alojar objetos por valor (es decir, pueden contener completamente a un objeto) probablemente no sea conveniente en algunos diseños realizar esto.

En muchos casos de programación orientada a objetos, es necesario crear objetos con **new** y realizar *upcasting* con la clase base como fue visto anteriormente, reduciendo la complejidad del código y otorgando extensibilidad. Este uso polimórfico de los objetos debe ser desarrollado mediante un contenedor de punteros.

El siguiente código corresponde al uso de contenedores con el ejemplo anterior. Se ejemplifica el uso de la STL vector para contener punteros a varios tipos de Instrumentos.

```

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

enum nota { DO, RE, MI, FA, SOL, LA, SIb };

class Instrumento { // funciones virtuales puras
public:
    Instrumento() {cout << "Constructor de Instrumento" << endl;}
    virtual void ejecutar(nota) =0;
    virtual string nombre() =0;
}

```

```
    virtual ~Instrumento() {cout << "Destructor de Instrumento" << endl;}
};

class Piano : public Instrumento {
public:
    Piano() { cout << "Constructor de Piano" << endl;}
    void ejecutar(nota) { cout << "Piano::ejecutar" << endl; }
    string nombre() { return "Piano"; }
    ~Piano() { cout << "Destructor de Piano" << endl; }
};

class Violin : public Instrumento {
public:
    Violin() { cout << "Constructor de Violin" << endl;}
    void ejecutar(nota) { cout << "Violin::ejecutar" << endl; }
    string nombre() { return "Violin"; }
    ~Violin() { cout << "Destructor de Violin" << endl; }
};

void EjecutaNotas(Instrumento& i) {
    i.ejecutar(DO);
    i.ejecutar(RE);
    i.ejecutar(SOL);
}

typedef vector<Instrumento*> Contenedor;

int main() {
    Contenedor orquesta;

    orquesta.push_back(new Piano);
    orquesta.push_back(new Violin);
    orquesta.push_back(new Violin);

    cout << orquesta[1]->nombre() << endl;
    orquesta[1]->ejecutar(SOL);

    cin.get();

    unsigned i;
    for(i = 0; i < orquesta.size(); i++)
    { cout << "Instrumento a ejecutar : " << orquesta[i]->nombre() << endl;
      EjecutaNotas (*orquesta[i]);
    }

    cout << "COMIENZO DESTRUCTORES" << endl;

    for(i = 0; i < orquesta.size(); i++)
        delete orquesta[i];

    cin.get();
    return (0);
}
```

Plantillas

Planteo de una inquietud

Muchas veces se tiene que repetir el siguiente código:

```
int promedio(int num1, int num2, int num3)
{ return (int)((num1+num2+num3)/3); }

float promedio(float num1, float num2, float num3)
{ return (float)((num1+num2+num3)/3); }

double promedio(double num1, double num2, double num3)
{ return (double)((num1+num2+num3)/3); }
```

¿No sería mejor que pudiéramos escribir algo más genérico como:

```
?? promedio(?? num1, ?? num2, ?? num3)
{ return (??)((num1+num2+num3)/3); }
```

... y que el compilador se encargue de poner los tipos adecuados?

Funciones plantilla

Las plantillas permiten crear funciones genéricas que admiten cualquier tipo de datos como parámetro y devuelven valores sin tener que sobrecargar la función con todos los tipos de datos posibles.

Su prototipo puede ser:

```
template <class identificador> declaracion_de_funcion;
```

Por ejemplo, para crear una función plantilla que retorna el mayor de dos parámetros puede ser:

```
template <class TipoGenerico>
TipoGenerico mayor (TipoGenerico a, TipoGenerico b) {
    return (a>b?a:b);
}
```

Como especifica la primer línea, se ha creado una plantilla para un tipo genérico de dato que hemos llamado *TipoGenerico*. Por consiguiente, en la función, *TipoGenerico* se convierte en un tipo de dato válido y es usado como tipo para sus dos parámetros *a* y *b*, y como tipo para el valor que retorna la función *mayor*.

TipoGenerico todavía no representa ningún tipo de dato concreto; cuando la función *mayor* sea llamada se deberá llamarla con cualquier tipo de dato válido. Este tipo de dato servirá como *patrón* y reemplazará a *TipoGenerico* en la función. Por ejemplo, para llamar a *mayor* y comparar dos valores enteros de tipo *int* se escribe:

```
int x,y;
mayor (x,y);
```

así que mayor será llamada como si cada aparición de *TipoGenerico* fuera reemplazada por *int*.

A continuación se presenta el código completo:

```
// función template
#include <iostream>
using namespace std;

template <class T>
T mayor (T a, T b) {
    T resultado;
    resultado = (a>b)? a : b;
    return (resultado);
}

int main () {
    int i=5, j=6, k;
    long l=10, m=5, n;
    k=mayor(i, j);
    n=mayor(l,m);
    cout << k << endl;
    cout << n << endl;
    cin.get();
    return (0);
}
```

```
6
10
```

En este caso se ha llamado al tipo genérico como T, en vez de TipoGenerico, porque es más corto y además es uno de los identificadores más usuales para plantillas, aunque es válido usar cualquier identificador.

En el ejemplo anterior se usó la misma función Mayor() con argumentos de tipo int y long habiéndose escrito una única implementación de la función. Lo que es lo mismo, se ha escrito una función plantilla y se llamó con dos patrones diferentes.

Como se puede ver, dentro de la función plantilla Mayor() el tipo T puede ser usado para declarar nuevos objetos:

```
T resultado;
```

resultado es un objeto de tipo T, como a y b, o sea, del tipo que se encierra entre los signos <> a llamar a la función plantilla.

También se puede hacer funciones plantilla que admitan más de un tipo de dato genérico. Por ejemplo:

```
template <class T, class U>
T minimo (T a, U b) {
    return (a<b?a:b);
}
```

En este caso, la función plantilla *minimo()* admite dos parámetros de distinto tipo y devuelve un objeto del mismo tipo que el primer parámetro pasado (T). Por ejemplo, después de la declaración se podrá llamar a la función de la siguiente manera:

```
int i, j;  
float f;  
i = Minimo (j, f);
```

aún si j y f son de distinto tipo.

Clases plantilla

También existe la posibilidad de crear clases plantilla, por lo que una clase puede tener miembros basados en tipos genéricos que no necesitan ser definidos al momento de crear la clase o contener funciones que usen estos miembros genéricos. Por ejemplo:

```
template <class T>  
class par {  
    T valores[2];  
public:  
    par (T primero, T segundo)  
    {  
        valores[0]=primero; valores[1]=segundo;  
    }  
};
```

La clase recién definida sirve para almacenar dos elementos de cualquier tipo válido. Por ejemplo, si se quiere declarar un objeto de esta clase para almacenar dos valores de tipo int con los valores 115 y 36 se podría escribir:

```
par <int> mis_ints(115, 36);
```

esta misma clase serviría también para crear un objeto que almacene cualquier otro tipo:

```
par<float> mis_floats(3.0, 2.18);
```

La única función de esta dentro de la declaración de la clase, sin embargo si esto no es así y se define una función fuera de la declaración de la clase siempre se debe preceder esta definición con el prefijo **template <...>**:

```
// clases templates  
#include <iostream>  
using namespace std;  
  
template <class T>  
class par {  
    T valor1, valor2;  
public:  
    par(T primero, T segundo)
```

100

```

        {valor1=primero; valor2=segundo;}
        T mayor();
};

template <class T>
T par<T>::mayor()
{
    T retval;
    retval = valor1>valor2? valor1 :
valor2;
    return retval;
}

int main (){
    par <int> mi_objeto(100, 75);
    cout << mi_objeto.mayor();
    return (0);
}

```

Observar como la definición de la función Mayor comienza:

```

template <class T>
T par<T>::Mayor ()

```

Todas las "T" que aparecen son necesarias, razón por la cual cuando se declaran funciones que pertenezcan a clases plantillas se debe seguir un formato similar a éste (la segunda T hace referencia al tipo de dato que retorna la función, así que esto puede variar).

Especialización de plantillas

La especialización de plantillas permite a una plantilla realizar una implementación específica según sea el tipo de dato que se ingrese como patrón. Por ejemplo, suponga que la clase plantilla *par* del ejemplo anterior necesitara una función para devolver el resultado de la operación módulo entre los dos parámetros que se le pasan, pero sólo se quiere que funcione cuando el tipo de los parámetros es int, y para el resto de los tipos siempre devuelva 0. Esto se hace de la siguiente manera:

```

// especialización de template
#include <iostream>
using namespace std;

template <class T>
class par {
    T valor1, valor2;
public:
    par (T primero, T segundo)
        {valor1=primero; valor2=segundo;}
    T modulo () {return (0);}
};

// particularizado para int:
template <>

```

```

25
0

```

```

class par <int> {
    int valor1, valor2;
public:
    par (int primero, int segundo)
        {valor1=primero; valor2=segundo;}
    int modulo ();
};

template <>
int par<int>::modulo() {
    return valor1%valor2;
}

int main () {
    par <int> mis_enteros (100,75);
    par <float> mis_floats (100.0,75.0);
    cout << mis_enteros.modulo() << endl;
    cout << mis_floats.modulo() << endl;
    return (0);
}

```

En el código la especialización es definida de la siguiente manera:

```
template <> class nombre_clase <tipo>
```

La especialización es una parte de la plantilla, por lo que se debe comenzar la declaración con **template <>**. Y como de hecho es una especialización para un tipo concreto de dato, el tipo genérico no puede ser usado aquí, y los primeros signos <> deben aparecer vacíos. Luego del nombre de la clase se debe incluir el tipo que está siendo especializado encerrado entre los signos <>.

Cuando se quiere especializar una plantilla también se deben redefinir los miembros adecuándolos a la especialización (en el ejemplo anterior se incluyó su propio constructor, aunque es idéntico al de la plantilla genérica). La razón es que ningún miembro es heredado de la plantilla genérica a la especialización.

Valores de parámetros para plantillas

Además de los argumentos precedidos por **class** o **typename** que representan un tipo genérico, las funciones plantilla y las clases plantilla pueden incluir otros parámetros que no son tipos, si no que pueden ser valores constantes, como por ejemplo tipo de datos simples. Como un ejemplo, observar esta clase plantilla que sirve para almacenar arreglos:

```

// template para arreglos
#include <iostream>
using namespace std;

template <class T, int N> //T:tipo de dato
class arreglo {          //N:tamaño array estático
    T memblock[N];
public:

```

```

100
3.1416

```

```

void fijar_miembro (T valor, int x);
T mostrar_miembro (int x);
};

template <class T, int N>
void arreglo<T,N>::fijar_miembro (T valor, int x) {
    membloc[x]=valor;
}

template <class T, int N>
T arreglo<T,N>::mostrar_miembro (int x) {
    return membloc[x];
}

int main () {
    arreglo <int,5> mis_enteros;
    arreglo <float,5> mis_floats;
    mis_enteros.fijar_miembro (100,0);
    mis_floats.fijar_miembro (3.1416,3);
    cout << mis_enteros.mostrar_miembro(0) << endl;
    cout << mis_floats.mostrar_miembro(3) << endl;
    cin.get();
    return (0);
}

```

También es posible fijar valores por defecto para cualquier parámetro en una plantilla de la misma forma que en las funciones comunes.

Algunas plantillas posibles vistas arriba:

```

template <class T> // la más usual: parámetro de una clase
template <class T, class U> // parámetros de dos clases
template <class T, int N> // una clase y un entero
template <class T = char> // con tipo por defecto
template <int Tfunc (int)> // una función como parámetro

```

Plantillas y proyectos de varios archivos

Desde el punto de vista del compilador, las plantillas no son funciones o clases normales. Son compilados de acuerdo a la demanda. Esto significa que el código de una función plantilla no es compilado hasta que una instancia (la creación de un objeto de este tipo) es requerida. En ese momento el compilador genera desde la plantilla una función específica del tipo solicitado.

Cuando se trabaja en un proyecto grande, es normal dividir el código de un programa en diferentes archivos fuente. En estos casos generalmente la interfaz y la implementación son separadas. Tomando como ejemplo una biblioteca de funciones, la interfaz consiste generalmente en los prototipos de todas las funciones que pueden ser llamadas, que son declaradas por lo común en un archivo de encabezamiento con la extensión .h, y la implementación (la definición de estas funciones) es un archivo independiente que contiene el código C++ de las mismas.

El funcionamiento (similar a las macros) de las plantillas, nos fuerza a imponer una restricción para los proyectos de varios archivos: la implementación (definición) de una función plantilla o clase plantilla debe estar en el mismo archivo que la declaración. Esto significa que no se puede separar la interfaz en un archivo de encabezamiento por separado y debemos incluir tanto la interfaz como la implementación en cualquier archivo que use la plantilla.

Volviendo a la biblioteca de funciones, si se quisiera componer una librería de plantillas, en lugar de crear un archivo de encabezamiento (.h) se debería crear un "archivo de plantillas" que contenga la interfaz y la implementación de las funciones plantilla (no hay convención sobre la extensión que poseen este tipo de archivos además de .h o directamente sin extensión). La inclusión en más de una vez del mismo archivo de plantillas en un proyecto no genera errores de linkado (vinculación) dado que son compilados por demanda y los compiladores que permiten plantillas deberían estar preparados para no generar código duplicado en estos casos.

Contenedoras Estándares

Introducción

En el capítulo referido a arreglos y vectores se pudieron comparar las ventajas importantes que poseían los vectores dinámicos de la clase **vector**. Por ejemplo, a diferencia de los arreglos estáticos, los vectores dinámicos se podían redimensionar de acuerdo a las necesidades del programa. Además, los arreglos no podían pasarse por valor ni ser devueltos por a una función mientras que con un vector estas operaciones se realizaban de forma natural, como si se tratase de una variable simple cualquiera. Los vectores tienen muchas ventajas más y forman parte de un conjunto de clases muy importantes denominadas “contenedoras”.

A lo largo del tiempo, la necesidad y complejidad de los programas fue creciendo exponencialmente pero las capacidades de los lenguajes no lo hicieron de igual forma. Los programadores se vieron obligados a realizar desarrollos propios o utilizar otros de terceros para poder generar programas de la complejidad necesaria en tiempos razonables. Pero la suma anárquica de aportes individuales requirió cada vez más un estándar que unificara y resolviera los problemas a los que la mayoría de los programadores se enfrentaban. Uno de los más críticos siempre fue la administración de memoria.

La biblioteca estándar de plantillas de C++ (*Standard Template Library, STL*) permite tratar con listas dinámicas y provee una colección de algoritmos listos para utilizar. La STL se basa en tres elementos fundamentales: contenedoras, iteradores y algoritmos. En particular, en este capítulo se tratarán las clases contenedoras pero previamente es necesario introducirse en el uso de los iteradores, que son necesarios en la mayoría de los métodos de las clases contenedoras.

Los iteradores son objetos que actúan en forma similar a los índices. Así como los arreglos estáticos evolucionaron en los vectores dinámicos, puede pensarse que los punteros lo hicieron hacia los actuales iteradores. Mediante estos objetos es posible recorrer los elementos de cualquier contenedor casi del mismo modo en que se usaría un índice para recorrer los componentes de un arreglo. Hay muchos tipos de iteradores pero las clasificaciones más importantes son: iteradores de acceso secuencial e iteradores de acceso aleatorio. El primer tipo de iteradores permite que se acceda a los elementos de la contenedora siguiendo un orden predeterminado, por ejemplo, de uno en uno desde el principio hasta el final (acceso hacia adelante) o también de uno en uno pero en cualquier dirección (acceso bidireccional). Los iteradores de acceso aleatorio permiten saltar de un elemento a otro de la contenedora sin importar que tan lejos estén. Por ejemplo, se puede saltar del elemento 3 al 15 y luego volver al 7. Como ya se verá, estas operaciones pueden ser más o menos eficientes dependiendo de la clase contenedora que se trate.

¿Cómo elegir el contenedor adecuado?

Las clases contenedoras tienen como fin general el de guardar una secuencia de objetos en memoria. Sin embargo, a pesar de que todas se manejen similarmente, cada una posee características particulares que la hace más apta para determinado tipo de situaciones en un programa.

Hay tres conceptos fundamentales a considerar a la hora de elegir la contenedora más apropiada para un determinado problema:

- **Tipo de acceso:** en las listas de acceso **secuencial** se deben leer todos los datos, posición por posición, hasta llegar a la posición deseada. Esto es como los cassettes, para acceder al segundo tema primero hay que pasar por arriba del primero. En las listas de acceso **aleatorio** se puede saltar de un dato a otro sin pasar por todos los intermedios. Siguiendo con la analogía, en un compact disk es posible ir directamente al tema 2 sin tener que escuchar el 1, ni pasar por arriba de él con el botón de avance rápido.
- **Velocidad de acceso:** las listas de acceso aleatorio son rápidas para escribir o leer elementos que están distantes dentro de la estructura. Por ejemplo, escribir un dato en el elemento 1 es tan rápido como escribirlo en el 4359. Por el contrario, en las de acceso secuencial, escribir un dato en el elemento 1 puede ser 4359 veces más lento que escribirlo en el elemento 4359. En general, las listas de acceso aleatorio proporcionan velocidad de acceso **constante** a cualquier elemento y las de acceso secuencial proveen velocidad de acceso **proporcional** a la distancia al elemento.
- **Velocidad de inserción/eliminación:** en este caso interesa que tan eficiente es la lista para realizar cambios en su estructura. En muchas ocasiones es necesario insertar un dato en medio de otros dos o al final de una lista. En general, dada la estructura interna de cada tipo de lista, las que poseen acceso secuencial proveen velocidad de inserción o eliminación **constante**, es decir, no importa la posición del elemento en cuestión ni del tamaño de la lista. Por el contrario, en las listas de acceso aleatorio la inserción de elementos es **proporcional** al tamaño de la lista y además, sustancialmente mayor a la de las anteriores.

A continuación se describen las clases contenedoras más importantes:

Contenedora	Descripción
list	<p>Es eficiente cuando se deber realizar muchas operaciones de inserción o eliminación de elementos en el medio de la lista. Provee únicamente acceso secuencial en tiempo constante en ambas direcciones. Provee velocidad de inserción/eliminación constante, independiente de la posición del elemento y del tamaño de la lista. Es implementada como una lista doblemente ligada.</p>
deque	<p>Es eficiente cuando se insertan o eliminan muchos elementos al principio o al final de la lista. Provee acceso aleatorio en tiempo constante a los elementos (aunque más lento que el vector). Provee velocidad de inserción/eliminación constante al comienzo o al final y velocidad proporcional si se realizan en el medio de la lista. Es implementada como una cola con doble final.</p>
vector	<p>Es eficiente cuando se necesitan las características típicas de un arreglo, sobre todo, alta velocidad de acceso aleatorio. Sin embargo, no es conveniente cuando se deben realizar muchas operaciones de inserción o eliminación. Provee acceso aleatorio en tiempo constante a los elementos (la más rápida de todas las contenedoras). El tiempo de inseción o elimiación es constante al final o al principio de la lista (aunque no tan eficiente como deque o list) El tiempo de inseción o elimiación es proporcional a su tamaño cuando se realiza en el medio. Es implementado como un arreglo dinámico asegurando la contigüidad de sus elementos en la memoria.</p>
map	<p>Es un tipo muy particular de contenedor asociativo que posee un algoritmo de búsqueda que permite acceder a los elementos individuales a través de claves. Por ejemplo, el map se puede indexar por medio de cadenas <code>mapa["Ramon"]="Lopez";</code> Provee acceso secuencial en tiempo constate. Si el acceso se base en una clave, se realiza un proceso de búsqueda que tarda un tiempo proporcional al logaritmo de la cantidad de elementos de la lista.</p>
set	<p>Modela un conjunto de elementos en lugar de una secuencia. Es especialmente útil cuando simplemente se desea saber ssi un elemento está o no en la lista, sin importar su verdadera posición. Provee acceso secuencial en tiempo constante. Se accede al elemento por su valor (por ejemplo preguntando si está o no está) y la velocidad es proporcional al logaritmo de la cantidad de elementos de la lista.</p>

Existe además un conjunto de contenedoras que son simplemente una simplificación o versión limitada de las anteriores. Estos contenedores son conocidos como *adaptadores*. Por ejemplo, la contenedora **queue** (cola) es una simplificación de **deque** que solamente permite insertar elementos al final y eliminarlos del principio. La contenedora **stack** (pila) también es una simplificación de **deque** que solamente permite insertar y eliminar elementos en el final.

Operaciones comunes

Dado que todas contenedoras persiguen el mismo fin general, existe un conjunto de métodos y operadores similares que permiten realizar las tareas básicas de administración. A continuación se describen algunos de los métodos generales que están presentes en la mayoría de las contenedoras:

Método	Descripción
size()	Devuelve la cantidad de elementos.
empty()	Devuelve true si la lista está vacía.
clear()	Elimina todos los elementos de la lista.
resize(num, dato)	Agrega varios elementos al final.
at(pos) [pos]	Devuelve el elemento de la posición <i>pos</i> .
back() front()	Primer (front) y último (back) elemento.
push_front(dato) push_back(dato)	Agregan elementos al principio o al final.
pop_front() pop_back()	Elimina el primer o el último elemento.
insert(iter, dato)	Inserta uno o más elementos en cualquier posición.
erase(iter) erase(iter1, iter2)	Elimina uno o más elementos en cualquier posición.
begin()	Devuelve un iterador al primer elemento de la lista.
end()	Devuelve un iterador al último elemento de la lista.

No se tiene que aprender “de memoria” los detalles de cada una de las llamadas a los métodos anteriores, más aun considerando todas las variantes de acuerdo a la clase contenedora de que se trate. Es recomendable que mientras se programa se tenga siempre a mano un manual de referencia que le permita revisar rápidamente este tipo de detalles. Estos manuales de referencia pueden conseguirse gratuitamente en Internet e imprimir las tablas más importantes o tenerlas en formato html o hlp.

Ejemplos

La clase *vector*

Creación

```
vector<int> iv;           // crea un vector de int de longitud cero
vector<char> cv(20);     // crea un vector con 20 elementos char
vector<char> cv(5, 'x'); // inicializa con 5 elementos char
vector<int> iv2(iv);    // crea un vector a igual a otro
```

Operaciones básicas

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

```
int main()
{
```

```

vector<int> v(10);
unsigned int i;

// muestra el tamaño original de v
cout << "Tamaño = " << v.size() << endl;

// asigna los elementos al vector
for(i=0; i<10; i++) v[i] = i;

// muestra el contenido del vector
cout << "Contenido actual:" << endl;
for(i=0; i<v.size(); i++)
    cout << v[i] << " ";
cout << endl << endl;

cout << "Expandiendo el vector" << endl;
// poner más valores al final del vector y agrandar lo necesario
for(i=0; i<5; i++) v.push_back(i + 10);

// mostrar los valores actuales de v
cout << "Tamaño actual = " << v.size() << endl;

// cambiar el contenido del vector
for(i=0; i<v.size(); i++) v[i] = -v[i];

// mostrar el contenido del vector
cout << "Contenido actual:" << endl;
for(i=0; i<v.size(); i++)
    cout << v[i] << " ";
cout << endl << endl;

cin.get();
return (0);
}

```

Uso de *pop_back()* y *empty()*

```

// Uso de pop_back() y empty().
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<char> v;
    unsigned int i;

    for(i=0; i<10; i++)
        v.push_back(i + 'A');

    cout << "Contenido del vector original:" << endl;
    for(i=0; i<v.size(); i++)
        cout << v[i] << " ";

    cout << endl << endl;

    do {
        v.pop_back(); // remover un elemento del final

        cout << "El vector ahora contiene:" << endl;
    }
}

```

```

    for(i=0; i<v.size(); i++)
        cout << v[i] << " ";
    cout << endl;

} while( !v.empty() );

    cin.get();
    return (0);
}

```

Acceso a través de iteradores

// Acceso a los elementos de un vector a través de un iterador.

```

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

int main()
{
    vector<int> v(10);          // crea un vector de longitud 10
    vector<int>::iterator p; // crea un iterador
    unsigned int i;

    // asigna elementos
    p = v.begin();
    i = 0;
    while(p != v.end()) {
        *p = i; // asigna i a v a través del iterador p
        p++;   // avanza el iterador
        i++;
    }

    // cambia el contenido del vector
    p = v.begin();
    while(p != v.end()) {
        *p = *p * 2;
        p++;
    }

    // muestra el contenido del vector
    cout << "Contenido original:" << endl;
    p = v.begin();
    while(p != v.end()) {
        cout << *p << " ";
        p++;
    }
    cout << endl << endl;

    cin.get();
    return (0);
}

```

La clase *deque*

Operaciones básicas

```

#include <iostream>
#include <deque>
#include <cstring>
using namespace std;

```

```

int main()
{
    deque<char> q1;
    char str[] = "-Usando una deque.";
    unsigned int i;

    // cargamos el texto en forma simétrica
    for(i=0; i<strlen(str); i++) {
        q1.push_front(str[i]);
        q1.push_back(str[i]);
    }

    cout << "Mostramos q1 (texto simétrico):" << endl;
    for(i=0; i<q1.size(); i++)
        cout << q1[i];
    cout << endl << endl;

    // remover valores de la deque desde el principio
    for(i=0; i<strlen(str); i++) q1.pop_front();

    cout << "q1 después de quitar del principio:" << endl;
    for(i=0; i<q1.size(); i++)
        cout << q1[i];
    cout << endl << endl;

    deque<char> q2(q1); // construir una copia de q1
    cout << "Contenido original de q2" << endl;
    for(i=0; i<q2.size(); i++)
        cout << q2[i];

    cout << endl << endl;

    // apuntar con un iterador a la primera ocurrencia de 'a'
    deque<char>::iterator p = q1.begin();
    while(p != q1.end()) {
        if(*p == 'a') break;
        p++;
    }

    // insertar q2 en q1
    q1.insert(p, q2.begin(), q2.end());

    cout << "q1 después de la inserción del original en la primer 'a':"
        << endl << endl;
    for(i=0; i<q1.size(); i++)
        cout << q1[i];

    cout << endl << endl;

    cin.get();
    return (0);
}

```

La clase *list*

Operaciones básicas

```
// El programa crea una lista de enteros.
```

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

int main()
{
    list<int> lst; // crea una lista vacía
    unsigned int i;

    for(i=0; i<10; i++) lst.push_back(i);

    cout << "Tamaño = " << lst.size() << endl;

    cout << "Contenido: ";
    list<int>::iterator p = lst.begin();
    while(p != lst.end()) {
        cout << *p << " ";
        p++;
    }
    cout << endl << endl;

    // cambia el contenido de la lista
    p = lst.begin();
    while(p != lst.end()) {
        *p = *p + 100;
        p++;
    }

    cout << "Contenido modificado: ";
    p = lst.begin();
    while(p != lst.end()) {
        cout << *p << " ";
        p++;
    }

    cin.get();
    return (0);
}
```

Push_back() y push_front()

```
// Muestra la diferencia entre push_back() y push_front()
#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<int> lst1, lst2;
    unsigned int i;

    for(i=0; i<10; i++) lst1.push_back(i);
    for(i=0; i<10; i++) lst2.push_front(i);

    list<int>::iterator p;

    cout << "Contenido de la lista lst1: " << endl;
    p = lst1.begin();
    while(p != lst1.end()) {
        cout << *p << " ";
    }
}
```

```

    p++;
}
cout << endl << endl ;

cout << "Contenido de la lista lst2:" << endl;
p = lst2.begin();
while(p != lst2.end()) {
    cout << *p << " ";
    p++;
}

cin.get();
return (0);
}

```

Lista ordenada

```

// Lista ordenada.
// El siguiente programa crea una lista de números aleatorios enteros
// y los pone en una lista ordenada.
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;

int main()
{
    list<int> lst;
    unsigned int i;

    // crea una lista de enteros aleatorios
    for(i=0; i<10; i++)
        lst.push_back(rand()%50);

    cout << "Contenido original:" << endl;
    list<int>::iterator p = lst.begin();
    while(p != lst.end()) {
        cout << *p << " ";
        p++;
    }
    cout << endl << endl ;

    // ordena la lista
    lst.sort();

    cout << "Contenido ordenado:" << endl;
    p = lst.begin();
    while(p != lst.end()) {
        cout << *p << " ";
        p++;
    }

    cin.get();
    return (0);
}

```

Unique()

```

// Ejemplo de unique().
// La función unique() remueve elementos consecutivos duplicados.
#include <iostream>

```

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

int main()
{
    list<int> lst;
    list<int>::iterator p;

    for(int i=0; i<5; i++)
        for(int j=0; j<3; j++) lst.push_back(i);

    lst.push_back(1); // repetido no consecutivo

    cout << "Lista original: ";
    for(p=lst.begin(); p!=lst.end(); p++)
        cout << *p << " ";
    cout << endl << endl;

    lst.unique(); // remueve elementos consecutivos duplicados

    cout << "Lista modificada: ";
    for(p=lst.begin(); p!=lst.end(); p++)
        cout << *p << " ";
    cout << endl << endl;

    cin.get();
    return (0);
}
```

La clase *queue*

Operaciones básicas

```
// Ejemplo simple de queue.
#include <iostream>
#include <queue>
#include <string>
using namespace std;

int main()
{
    queue<string> q;

    cout << "Almacenando los textos: uno, dos, tres, cuatro" << endl << endl;
    q.push("uno");
    q.push("dos");
    q.push("tres");
    q.push("cuatro");

    while(!q.empty()) {
        cout << "Mostrando y quitando elemento: ";
        cout << q.front() << endl;
        q.pop();
    }

    cin.get();
    return (0);
}
```

Ciclos a través de *queue*

```
/* Ciclos a través de queue.
Debido a que queue es una secuencia controlada, ud. no puede obtener
un iterador a la queue. Tampoco se puede acceder a la queue via
el operador []. */
#include <iostream>
#include <queue>
#include <string>
using namespace std;

int main()
{
    queue<string> q;

    q.push("uno");
    q.push("dos");
    q.push("tres");
    q.push("cuatro");

    cout << "Contenido de la queue: ";
    for(unsigned int i=0; i<q.size(); i++) {
        cout << q.front() << " ";

        // remover del frente y agregar atrás
        q.push(q.front());
        q.pop();
    }
    cout << endl << endl;

    cout << "Ahora, remueve elementos:" << endl;
    while(!q.empty()) {
        cout << "Quitando elementos: ";
        cout << q.front() << endl;
        q.pop();
    }

    cin.get();
    return (0);
}
```

La clase *map*

Operaciones básicas

```
// El programa almacena el par de valores código_ASCII/valor.
#include <iostream>
#include <map>
using namespace std;

int main()
{
    map<char, int> m;
    int i;

    // put pairs into map using [ ]
    for(i=0; i<26; i++) m['A'+i]=65+i;

    // buscamos el código ASCII de una letra
    char ch='A';
```

```
cout << "El código de " << ch << " es " << m[ch];

cin.get();
return (0);
}
```

Usando iteradores

```
// Ciclo a través de map usando iteradores.
#include <iostream>
#include <map>
using namespace std;

int main()
{
    map<char, int> m;
    int i;

    // put pairs into map
    for(i=0; i<26; i++)
        m.insert(pair<char, int>('A'+i, 65+i));

    map<char, int>::iterator p;

    // Muestra el contenido de map
    for(p = m.begin(); p != m.end(); p++) {
        cout << p->first << " le corresponde el código ASCII: ";
        cout << p->second << endl;
    }

    cin.get();
    return (0);
}
```

Ordenando objetos

```
// Almacenando Objetos en un Map
// Teniendo en cuenta que map tiene los códigos ordenados, el
// programa define el operador < para objetos del mismo tipo.
// Se usará map para crear un directorio telefónico.
#include <iostream>
#include <map>
#include <string>
using namespace std;

// Esta es la clase clave.
class nombre {
    string str;
public:
    nombre() { str = ""; }
    nombre(string s) { str = s; }
    string get() { return str; }
};

// se define el operador < para objetos de la clase nombre.
bool operator<(nombre a, nombre b)
{
    return a.get() < b.get();
}

// Esto es lo que se almacenará en el map.
```

```
class NumeroTelefonico {
    string str;
public:
    NumeroTelefonico() { str = ""; }
    NumeroTelefonico(string s) { str = s; }
    string get() { return str; }
};

int main()
{
    map<nombre, NumeroTelefonico> guia;

    // se ingresa los nombres y números telefónicos
    guia.insert(pair<nombre, NumeroTelefonico>(nombre("Jonhatan"),
        NumeroTelefonico("555-4533")));
    guia.insert(pair<nombre, NumeroTelefonico>(nombre("Sole"),
        NumeroTelefonico("555-9678")));
    guia.insert(pair<nombre, NumeroTelefonico>(nombre("Pablo"),
        NumeroTelefonico("555-8195")));
    guia.insert(pair<nombre, NumeroTelefonico>(nombre("Patricia"),
        NumeroTelefonico("555-0809")));

    // dado un nombre, encuentra el número
    string str;
    cout << "Ingrese un nombre: ";
    getline(cin, str);

    map<nombre, NumeroTelefonico>::iterator p;

    p = guia.find(nombre(str));
    if(p != guia.end())
        cout << "Numero telefónico: " << p->second.get();
    else
        cout << "El nombre no está en la guía." << endl;

    cin.get();
    return (0);
}
```

La clase set

Operaciones básicas

```
// El programa almacena un conjunto de strings.
// Observar que las string son almacenadas automáticamente
// en orden ascendente. Elimina elementos repetidos.
#include <iostream>
#include <set>
#include <string>
using namespace std;

int main()
{
    set<string> s;

    // crear un conjunto de strings
    s.insert("Telescopio");
    s.insert("Casa");
    s.insert("Computadora");
}
```

```
s.insert("Ratón");
s.insert("Casa");

set<string>::iterator p = s.begin();

cout << "Contenido (ordenado automáticamente):" << endl;
do {
    cout << *p << " "; //muestra el contenido
    p++;
} while(p != s.end());
cout << endl;

// busca un elemento
p = s.find("Telescopio");
if(p != s.end())
    cout << endl << "Encontrado Telescopio" << endl;

cin.get();
return (0);
}
```

Manejo de errores

Durante el desarrollo de un programa, pueden darse ciertos casos donde no se tenga la certeza de que un fragmento de código vaya a trabajar bien, ya sea porque trata de acceder a recursos que no existen o porque se sale de un rango esperado, etc.

Este tipo de situaciones anómalas se incluyen en lo que se consideran excepciones o errores y C++ ha incorporado recientemente tres nuevos operadores que ayudan a manejar estas situaciones: **try**, **throw** y **catch**.

Su forma de uso es la siguiente:

```
try {
    // código que se intentará ejecutar
    throw posible_error;
}
catch (tipo error)
{
    // código a ser ejecutado en caso de error
}
```

El funcionamiento es el siguiente:

- El código dentro del bloque **try** es ejecutado normalmente. En caso de que ocurra un error, este código debe usar **throw** y un parámetro para devolver un error. El tipo de parámetro detalla el error y puede ser de cualquier tipo válido.
- Si un error ocurre, es decir, si se ha ejecutado una instrucción **throw** dentro del bloque **try**, el bloque **catch** es ejecutado recibiendo como parámetro el error pasado por **throw**.

Por ejemplo:

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

int main () {
    float f;
    try
    {
        cout << "Raiz cuadrada de: ";
        cin >> f;
        if (f<0) throw 1;

        cout << " es " << sqrt(f);
    }

    catch (int i)
    {
        cout<<"Error "<<i<<" en ingreso (valor negativo)"
            << endl;
    }
}
```

```
Error 1 en ingreso
(valor negativo)
```

```
}  
return (0);  
}
```

En este ejemplo, si se ingresa un valor menor a cero *throw* es ejecutado, el bloque *try* finaliza y todos los objetos creados dentro del bloque *try* son destruidos. Después de esto, el control es pasado al bloque *catch* correspondiente (que sólo se ejecuta en caso de error). Finalmente, el programa continúa luego del bloque *catch*.

La sintaxis de *throw* es similar a la de *return*: sólo un parámetro que no necesariamente va entre paréntesis.

El bloque *catch* debe ir justo después del bloque *try*, sin incluir ningún tipo de código entre ellos. Los parámetros que *catch* acepta pueden ser de cualquier tipo válido. Es más, *catch* puede ser sobrecargado de manera que pueda aceptar diferentes tipos como parámetros. **En ese caso el bloque *catch* ejecutado es el que concuerda con el tipo de error enviado (el parámetro de *throw*):**

```
// excepciones: múltiples bloques catch  
#include <iostream>  
using namespace std;  
  
void raiz(double valor)  
{  
    try  
    {  
        cout << "Raiz cuadrada de: " << valor;  
  
        if (valor<0) throw 1;  
        if (valor>10) throw "número muy grande";  
  
        cout << " es " << sqrt(valor) << endl;  
    }  
  
    catch (int i)  
    {  
        cout << " Error " << i  
            << " en ingreso (valor negativo)"<< endl;  
    }  
  
    catch (char* s)  
    {  
        cout << "Mensaje: " << s << endl;  
    }  
}  
  
int main () {  
  
    raiz(2.0);  
    raiz(-4.0);  
    raiz(100.0);  
    return (0);  
}
```

```
Raiz cuadrada de: 2 es  
1.41421  
Raiz cuadrada de: -4 Error  
1 en ingreso (valor  
negativo)  
Raiz cuadrada de: 100  
Mensaje: número muy grande
```

En este caso pueden sucederse, al menos, dos diferentes errores:

- Que el valor ingresado sea menor que cero: en este caso un error es devuelto que será registrado por **catch (int i)**, dado que el parámetro es un número entero.
- Que el valor ingresado sea mayor que 10 (supongamos que por la características del programa no es admisible número mayores a 10): en este caso el error devuelto será registrado por **catch (char* s)**.

También se puede definir un bloque catch que capture todos los errores independientemente del tipo que devuelve **throw**. Para esto se debe escribir *tres puntos* en vez del tipo de parámetro y nombre aceptado por catch:

```
try {
    // código aquí
}
catch (...) {
    cout << "Error";
}
```

Otra posibilidad, es anidar bloques try-catch dentro de bloques try más externos. En estos casos, se tiene la posibilidad de que un bloque catch interno traspase el error recibido al nivel externo, para esto se usa la expresión **throw**; sin argumentos. Por ejemplo:

```
try {
    try {
        // código aquí
    }
    catch (int n) {
        throw;
    }
}
catch (...) {
    cout << "Error";
}
```

Errores estándares

Algunas funciones de la biblioteca estándar C++ envían errores que pueden ser capturados si se incluyen dentro de un bloque **try**. Estos errores son enviados con una clase derivada de **std::error** como tipo. Esta clase (std::error) es definida en el archivo de encabezamiento estándar **<error>** de C++ y sirve como patrón para la jerarquía estándar de errores.

Debido a que esta es una jerarquía de clases, si se incluye un bloque catch para capturar cualquiera de los errores de la jerarquía usando el argumento por referencia (agregando el ampersand & después del tipo) también se capturarán todos sus derivados (reglas de herencia en C++).

El siguiente ejemplo atrapa un error de tipo **bad_typeid** (derivado de **error**) que es generado al requerir información acerca del tipo apuntado por un puntero nulo:

```
// excepciones estándares

#include <iostream>
#include <exception>
#include <typeinfo>
using namespace std;

class A {virtual f() {}};

int main () {
    try {
        A * a = NULL;
        typeid (*a);
    }
    catch (std::exception& e)
    {
        cout << "Excepción: " << e.what();
    }
    return (0);
}
```

Excepción: Attempted typeid of
NULL pointer

Se puede usar las clases de errores de jerarquía estándar para devolver errores definidos por el usuario o derivar nuevas clases a partir de ellos.

Recursividad

Una función recursiva es una llamada a ella misma, o una llamada a otra función que luego llama a la original. Cada vez que se llama a una función, se reserva espacio para almacenar el conjunto completo de las nuevas variables.

Ejemplo de llamadas recursivas

En Matemáticas se definen funciones o valores a partir de la metodología de cómo se calcula. Por ejemplo, el número **pi** es definido como "la razón de la circunferencia de un círculo a su diámetro". Esto es equivalente a establecer las siguientes instrucciones: se obtiene la circunferencia de un círculo y el diámetro, se divide el primero por el último y se llama al resultado pi. Este proceso especificado termina con un resultado definido.

Otro ejemplo, es el de la función factorial: "Dado un entero positivo n, el factorial de n se define como el producto de todos los enteros entre n y 1". Por ejemplo, 5 factorial es igual a $5*4*3*2*1 = 120$. La definición la podemos escribir de la siguiente forma:

$$\begin{array}{l} n! = 1 \rightarrow \text{si } n = 0 \\ n! = n*(n-1)*(n-2)*\dots*1 \rightarrow \text{si } n > 0 \end{array}$$

A modo de ejemplo, se calculará el factorial de 5. Aplicando la definición anterior:

- 1) $5! = 5*4!$ -> para $4!$ se aplica nuevamente la definición...
- 2) $4! = 4*3!$
- 3) $3! = 3*2!$
- 4) $2! = 2*1!$
- 5) $1! = 1*0!$
- 6) $0! = 1$ (por definición)

En la línea 6 se llega a un valor que es definido directamente que impone la finalización de las llamadas. "Se vuelve" de la línea 6 a la línea 5, retornando el valor obtenido y así sucesivamente.

El código para realizar este cálculo es el siguiente:

```
// recursividad - factorial
#include <iostream>
using namespace std;

int fact (int n) {
    if (n==0) return (1);
    else return (n*fact(n-1));
}

int main ( ) {
    int r, m;
    cin >> m;
```

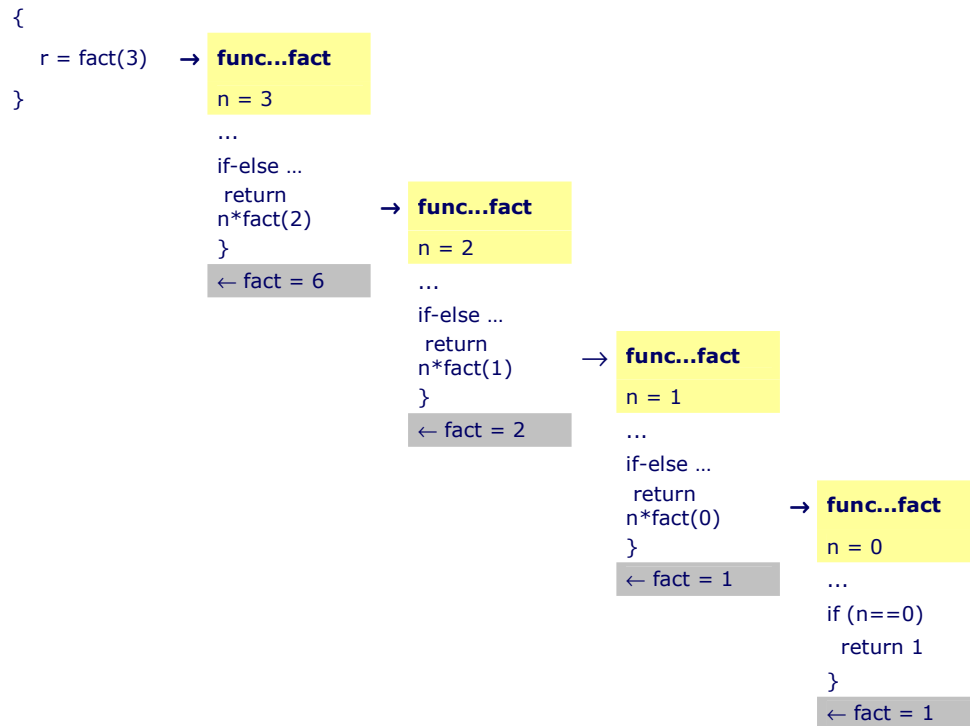
```
3
6
```

```

r = fact(m);
cout << r;
return (0);
}

```

Para el cálculo del factorial de 3, esquemáticamente las llamadas a la función "fact" y el valor de las variables son las siguientes:



Cada vez que una rutina recursiva se llama a sí misma, debe ser asignada un área de datos completamente nueva para esa llamada en particular. Esta área de datos contiene todos los parámetros, variables locales y la dirección de retorno. Un área de datos está asociada no solamente con una función, sino con la llamada en particular a esa función. Cada llamada hace que se asigne una nueva área de datos, y cada referencia a un elemento en el área de datos de la subrutina se hace en el área de datos de la llamada más reciente. Igualmente, cada retorno hace que el área de datos actual sea liberada, y que el área de datos que había sido asignada inmediatamente antes se convierta en la actual.

Definición formal de la recursividad

Se dice que un procedimiento de cálculo es recursivo, si en parte está formado por sí mismo o se define en función de sí mismo.

Los algoritmos recursivos son idóneos cuando el problema a realizar, la función a calcular o las variables a procesar están dadas en forma recursiva. En general, un

programa recursivo puede expresarse como una composición **P** compuesta de un conjunto de sentencias **S** (que pueden contener a **P**) y **P**. La notación es la siguiente:

$$P = P[S,P]$$

Un requisito fundamental es el que las llamadas a todo proceso recursivo están sujetas a una condición **B** que haga terminar las llamadas. Esta condición se expresa en la siguiente notación:

```
funcion P = if B then P[S,P] else Fin;
```

Una manera práctica de asegurar la terminación consiste en asociar un parámetro, que al llamar recursivamente al proceso **P** se modifique de tal manera que en algún momento sea falsa la condición **B**.

En las aplicaciones prácticas la profundidad de la recursión debe ser finita, y además pequeña, ya que cada vez que se activa el proceso **P** por una llamada recursiva es necesario disponer de memoria para todas las variables locales. Hay variables con valores "**presentes**" y "**pendientes**" que tienen igual nombre, pero los valores son de momentos distintos, por lo tanto, de lugares distintos de memoria.

Cuando no usar la recursividad

Los algoritmos recursivos son adecuados cuando el problema o los datos están dados en forma recursiva, pero puede darse el caso de que el algoritmo recursivo no sea la mejor manera de resolver el problema. Siempre que un algoritmo pueda ser implementado iterativo, debe hacerse así.

Los problemas en los cuales es adecuado abstenerse de la recursión, presentan la composición siguiente:

Esquemas donde hay sólo una llamada P al final o al inicio de la composición, que se puede representar de la siguiente manera:

```
función P = if (B) S else P
```

```
función P = S; if (B) P
```

Estos esquemas están asociados con cálculos de valores que están asociados con funciones elementales de recurrencia.

El ejemplo de cálculo del factorial en forma recursiva puede ser realizado en forma iterativa de la siguiente forma:

```
int function fact ( int n )
{
    int prod=1;
    if (n == 0)
```

```

return (1);          // n! = 1 si n = 0
else
  while (n!=0)
  {
    prod = n*prod;   // n! = n*(n-1)*(n-2)*...*1 si n>0
    n = n-1;
  }
return (prod);
}

```

En general estos procesos deben transcribirse según la forma siguiente:

P = [inicializar; while (B) S]

Algoritmos de rastreo inverso

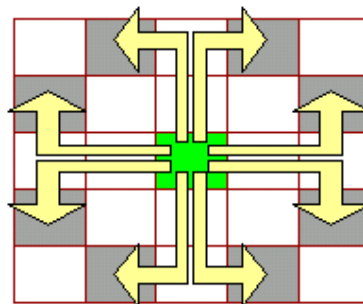
Un aspecto particularmente interesante es determinar los algoritmos para encontrar la solución de problemas sin seguir una regla específica de cálculo, sino por ensayo y error ("tanteo"). El patrón común consiste en descomponer el proceso de tanteo en tareas parciales. Con frecuencia éstas se expresan en término recursivos y consisten en explorar un número finito de subtareas. Sin embargo, cuando se dan incongruencias con las exigencias del problema, el algoritmo rastrea en forma inversa, suprimiendo la parte más reciente de la solución y ensayando después otra posibilidad hasta llegar al final.

El rastreo inverso es útil en situaciones donde muchas posibilidades pueden aparecer inicialmente, pero quedan pocas tras aplicar reglas posteriores.

Se presenta un ejemplo que consiste en la visita de todas las celdas de un tablero de ajedrez por medio de un caballo que realiza los movimientos permitidos para esta pieza. Además, debe pasar solamente una vez por cada casilla.

La simulación deberá comenzar desde un par de coordenadas iniciales cualesquiera. Deberá moverse según las reglas del ajedrez e "intentará" encontrar el itinerario que permita visitar cada una de las celdas sólo una vez.

Los ocho movimientos permitidos del caballo de ajedrez ubicado en una celda del tablero son los siguientes:



El procedimiento consiste en probar si es posible un próximo movimiento. Si es posible, se efectúa, pero si no, se verifica si se debe a que se ha llenado el tablero (fin del problema), o porque se ha llegado a un punto imposible. Si este último es el caso, se debe remover el movimiento anterior ("volviendo a la posición anterior") y probar con un movimiento distinto.

```

void SaltoDelCaballo::moverCaballo(unsigned col,unsigned fil,unsigned nroMov)
{
    unsigned deltaPosible = 0;
    Posicion proxPos;

    if (nroMov==cantidadPasos)
        exito = true;
    else
    {
        nroMov++;
        do
        {
            proxPos.col = movs[deltaPosible].col + col ;
            proxPos.fil = movs[deltaPosible].fil + fil ;
            if ( !( (proxPos.col<0) ||
                    (proxPos.col>=tablero.size()) ||
                    (proxPos.fil<0) ||
                    (proxPos.fil>=tablero[0].size()) ||
                    (tablero[proxPos.col][proxPos.fil]!=0)
                  )
                )
            {
                tablero[proxPos.col][proxPos.fil]=nroMov;
                moverCaballo(proxPos.col,proxPos.fil,nroMov);
            }
            deltaPosible++;
        } while ( (deltaPosible < maxMovs) && (!exito) );

        if (!exito)
            tablero[col][fil]=0;
    }
}

```

La representación de la lista de movimientos permitidos se realiza en un vector de 8 estructuras que contienen los movimientos posibles y afectan a las coordenadas fila y columna.

Las llamadas recursivas quedan pendientes con sus variables locales. Cuando se ejecuta el fin "}" del método, comienza a ejecutarse el pendiente anterior con sus variables locales (lo que significa que "retrocede" a fila y columna anterior) en el lugar que quedó y continúa ejecutándose la línea siguiente de la que se realizó la llamada.

	1	2	3	4	5
1		*		*	
2	*				Ⓢ
3			Ⓢ		
4	*				*
5		*		*	

A continuación se comentará algunos ciclos de la ejecución del programa.

Suponiendo que luego de varios saltos llegó a 3,3 a partir de 2,5. Prueba hacer un próximo salto e intenta los 8 movimientos posibles debido a que resultaba `tablero[col][fil]!=0`. Sale del bucle debido a que no se cumple `while deltaPosible<maxMovs`. Continúa con la línea de código `if (!exito)...` a la posición (3,3) y coloca en esta posición un cero. Llega al fin del método. Esto implica que vuelva a lo que tenía pendiente regresando al final de la línea `moverCaballo...` con las variables del procedimiento anterior, es decir, con el casillero anterior, el (2,5). De allí va a la línea `deltaPosible++`; y hace un nuevo salto para proseguir adelante ejecutando el `do-while`.

Ver la importancia de las variables **locales** y **globales**.

Creación de bibliotecas

Uno de los conceptos más potentes que existen en los lenguajes modernos de programación es la reutilización de código.

Esto significa usar el mismo código en diferentes partes de un programa. Esto puede ser realizado con macros y también con múltiples módulos que separen el código.

Las bibliotecas, que los sistemas operativos como Linux y Windows permiten usar, son formas mucho más potentes que las anteriormente dichas.

Las bibliotecas permiten compartir código (en general ya compilado) entre programas que no están escritos en ellos. Considere, por ejemplo, la función "sumaPolinomios()". Esta función puede ser usada en miles de programas de su sistema. En vez de tener una copia separada en cada uno de los programas que la usan, todos ellos pueden llamar directamente a la biblioteca que contiene el único código existente.

Introducción

Las bibliotecas pueden ser de dos tipos: estáticas y dinámicas. La diferencia está dada por el efecto producido cuando se unen las bibliotecas con el programa que las usa. Este enlace (link) generará el programa ejecutable final, tomando código de todos los módulos y uniéndolos en el programa ejecutable final.

Mediante bibliotecas estáticas, este proceso se realiza en el momento de la compilación del programa. El código de las bibliotecas es enlazado dentro del programa ejecutable y siempre lo acompaña. Esto significa que si la biblioteca es cambiada (por ejemplo por una nueva versión) deberá recompilarse el programa que la utiliza.

Trabajar con bibliotecas dinámicas significa que el enlace queda "preparado" en el momento de la compilación, pero realmente se realiza en el momento de la ejecución, por lo tanto el programa ejecutable final necesitará de la presencia de la biblioteca para que pueda funcionar. Esto significa que si la biblioteca es cambiada (por ejemplo, por mejoras) no es necesario recompilar el programa, sino que la actualización será automática en la nueva ejecución del programa.

Otra diferencia es que con las bibliotecas estáticas, si hay más de un programa que ejecuta la misma biblioteca, cada uno de ellos contendrá internamente a la biblioteca. Con las dinámicas hay ahorro de memoria porque sólo existe una biblioteca en memoria que es compartida por todos los programas que la utilizan.

Otra característica de las bibliotecas dinámicas es la capacidad de sobrescribir (o sobreutilizar) la característica de cualquier biblioteca que esté usando. Por ejemplo, se podría agregar nuevas características a funciones que ofrece el compilador. Esto se realiza precargando la nueva biblioteca antes que la original. De esta forma se podría reemplazar bibliotecas completamente.

Un problema del uso de las bibliotecas dinámicas, surge en los casos en que una nueva versión introduce incompatibilidades con la versión previa. En el caso de las estáticas, el programador crea una hermandad (biblioteca más su código) que permaneciera.

Respecto a la velocidad del proceso en el uso de bibliotecas estáticas respecto a dinámicas, para cada caso en particular deberá realizarse pruebas de desempeño debido a que las optimizaciones de los compiladores y sistemas operativos modernos pueden crear ejecuciones con resultados diferentes a los que intuitivamente parecería que podría ser respecto a los de desempeños (eficiencia en tiempo).

Creación y uso de bibliotecas estáticas

La creación de bibliotecas estáticas es muy simple. Esencialmente, se debe usar el programa "ar" (que generalmente se provee junto con el compilador) para combinar un número de archivos ".o" en una única biblioteca. Luego se necesita ejecutar el programa "ranlib" que agrega cierta información a la biblioteca.

El ejemplo que se muestra a continuación es muy simple. Se realiza una biblioteca para sumar y restar números enteros mediante el siguiente código:

```
// === Archivo "bibli.h" =====  
  
int Sumar(int a, int b);  
int Restar(int a, int b);  
  
=====
```

```
// === Archivo "bibli.cpp" =====  
  
int Sumar(int a, int b)  
{ int resultado;  
  
    resultado=a+b;  
    return (resultado);  
}  
  
int Restar(int a, int b)  
{ int resultado;  
  
    resultado=a-b;  
    return (resultado);  
}  
  
=====
```

```
// === Archivo "makefile" =====  
  
## este makefile es para generar bibliotecas estaticas. Archivo ".a"  
  
NOMBRE=libbibli  
CXX=g++
```

```
# linux
OPCIONES= -g -Wall
#DIRECT= -I../util -I/bin
#BIBLIOT=-L/lib -lm -L/usr/lib/

# cygwin
#OPCIONES= -g -Wall
#DIRECT=-I../util -I/bin -I/usr/include
#BIBLIOT= -L/lib -lm

all:
    ${CXX} -c ${OPCIONES} ${DIRECT} bibli.cpp -o bibli.o ${BIBLIOT}
    ar cr ${NOMBRE}.a bibli.o
    ranlib ${NOMBRE}.a
```

En el archivo correspondiente al makefile, observar el uso del parámetro "-c" y el uso del programa "ar" y "ranlib".

Por convención los nombres de las bibliotecas deber ir precedidas por el prefijo "lib" y el sufijo ".a" para las estáticas.

El uso de make para ejecutar los comandos del makefile, generara la biblioteca "**libbibli.a**".

El ejemplo siguiente usa la biblioteca creada mediante el agregado de la linea:

```
#include "bibli.h"
```

y la existencia del archivo "libbibli.a" que estará en el mismo directorio del programa en el momento de la compilación o encontrará su ubicación mediante la nomenclatura usada en el makefile para encontrar las otras bibliotecas (por ejemplo, colocándolas en la carpeta que contine las bibliotecas del usuario: -L/usr/lib/)

```
// Ejemplo de programa de uso de la biblioteca "bibli.h"
```

```
#include <iostream>
#include "bibli.h"

int main() {

    int a,b;
    cout << "Ingresar dos numero enteros: ";
    cin >> a >> b;

    cout << "Suma: " << Sumar(a,b) << endl;
    cout << "Resta: " << Restar(a,b) << endl;

    cin.get();
}
```

Para la compilacion del programa, un ejemplo de archivo makefile es el siguiente:

```
MAIN=main
CXX=g++
```

```
# linux
OPCIONES=-g -Wall
DIRECT=-I../util -I/bin
BIBLIOTECAS=-L/lib -lm -L/usr/lib/ -L. -lbibli

all:
    ${CXX} ${OPCIONES} ${DIRECT} ${MAIN}.cpp -o ${MAIN} ${BIBLIOTECAS}
```

Así puede usar la biblioteca. Se puede poner el ".a" en /usr/local/lib. También, puede simplemente colocarse en el directorio corriente. La opción "-L." le dice al linker que mire en el directorio actual, indicado por el punto. Normalmente, mira en el directorio de las bibliotecas solamente. -lbibli esta invocando a la biblioteca "libbibli.a".