Definición de nuevos tipos de datos.

José Otero

[1]Departamento de informática
Universidad de Oviedo

3 de enero de 2009

- Las combinaciones de las distintas palabras reservadas utilizadas para nombrar los tipos de datos predefinidos pueden complicarse, de modo que puede ser aconsejable definir nombres alternativos más sencillos.
- En C++ esto puede conseguirse mediante la palabra reservada `typedef`.
- Si el nombre de un tipo es precedido por `typedef`, el nombre que va después del tipo será un sinónimo de este en lo sucesivo.

```
typedef unsigned int uint;   //uint = unsigned int
typedef vector<uint> tabla;  //tabla= vector<unsigned int>
tabla a(10,1);               //=vector<unsigned int> a(10,1);
uint b;                      //=que unsigned int b;
typedef vector<tabla> matriz;//matriz=vector<vector<unsigned int> >
matriz d(10,tabla(8));       //=vector<vector<unsigned int> >
                             //d(10, vector<unsigned int>(8));
```


- Enumeración: tipo que puede almacenar un conjunto de valores especificados por el usuario.
- Ejemplo: conjunto de los colores primarios rojo, amarillo, azul.

```
//definicion del tipo
enum color {rojo, amarillo, azul};
//declaracion de una variable
color a;
```

- Durante la compilación, los nombres de los valores son sustituidos por valores de tipo entero.
- Por defecto esta sustitución comienza por cero y por el valor escrito en primer lugar.
- En el ejemplo anterior: ”rojo” se sustituye por 0, ”amarillo” por 1 y ”azul” por 2.

Puede hacerse que la sustitución comience en otro valor o asignar de forma explícita todos o alguno de los valores.

```
//rojo=7 amarillo=8 azul=1
enum color {rojo=7, amarillo, azul=1};
```

Es posible convertir un valor de tipo entero a enumeración:

```
color a;
a=color(8);//asigna ''amarillo'' a a
```

- Se les puede aplicar los mismos operadores que son aplicados a los enteros.
- Puede mostrarse por la pantalla su valor numérico con `cout`.
- Pero no se pueden pedir con `cin` usando su nombre.

El siguiente ejemplo mostraría 1 por la pantalla.

```
a=azul;
cout<<a;
```


- Nuevos tipos de datos que se construyen mediante la agregación de tipos de datos (posiblemente distintos) existentes.
- Es posible acceder al valor de cada componente de la estructura, que se denomina **miembro**.
- Pueden existir el número de miembros que se desee, del mismo o distinto tipo.
- Una estructura puede formar parte (como miembro) de otra estructura.
- En C++ las estructuras se definen mediante la palabra reservada `struct`.


```
struct nombre_tipo{
   tipo1 miembro1;
   tipo2 miembro2;
   ...
};
```

Desde este momento se pueden declarar variables de tipo `nombre_tipo`:

```
nombre_tipo nombre_variable;
```

- El tipo y número de miembros es arbitrario.
- El orden de definición de los miembros no tiene impacto en las propiedades del tipo.

Ejemplo:

```
struct direccion {
   int DNI;
   vector<char> nombre;
   int edad;
};
direccion a;
```

- Un miembro de una estructura se especifica mediante el nombre de ese miembro precedido por un punto y el nombre de la variable de tipo estructura.
- Un miembro de una estructura se comporta como una variable de su tipo.

```
a.DNI=666999666;          //asignación de un entero
a.edad=33;                //asignación de otro entero
a.nombre=vector<char>(20);//asignación del tamaño de un vector
a.nombre[0]='J';          //asignación del elemento 0 de un vector
a.nombre[3]='u';          //asignación del elemento 3 de un vector
```

- Las variables de tipo `struct` pueden asignarse (si son del mismo tipo).
- Ser pasadas como argumentos de una función.
- Devueltas como resultado de una función con `return`.
- Otras operaciones, como las realizadas por los operadores relacionales, no están definidas, deben de ser programadas por el usuario.
- Dos estructuras que posean exactamente los mismos miembros y en el mismo orden en su definición no son del mismo tipo y por lo tanto no pueden ser asignadas una a otra como tales.
- Si pueden asignarse los miembros uno a uno.

En el siguiente ejemplo se supone definida la estructura del ejemplo anterior.

```
struct direccion {
   int DNI;
   vector<char> nombre;
   int edad;
};
direccion a,b;            //declaracion de otra variable de tipo direccion
a.DNI=666999666;          //asignacion de un entero
a.edad=33;                //asignacion de otro entero
a.nombre=vector<char>(20);//asignacion del tamaño de un vector
a.nombre[0]='J';          //asignacion del elemento 0 de un vector
a.nombre[3]='u';          //asignacion del elemento 3 de un vector
b=a;                      //a b se le puede asignar a, son del mismo tipo
struct datos {
   int DNI;
   vector<char> nombre;
   int edad;
};
datos x;                  //declaracion de una variable de tipo datos
x.DNI=a.DNI;              //se puede asignar miembro a miembro pero
x.nombre=a.nombre;        //NO la estructura completa x=a; es un ERROR
x.edad=a.edad;
```


Recapitulación:

- Hasta el momento hemos visto los recursos existentes en el Lenguaje C (precursor de C++) para definir nuevos tipos de datos.
- *A la forma tradicional* de C, mediante una estructura, se define como codificar los valores del nuevo tipo de datos, pero no las operaciones que los manipulan (funciones y operadores).
- C++ frente a C, incorpora la posibilidad de definir clases, mediante las que se definen conjuntamente la forma de codificar los valores del nuevo tipo de datos y las operaciones que los manipulan.

- Hemos comentado muy por encima como se codifican los los valores de los distintos tipos de datos.
- Las operaciones posibles sobre los distintos tipos de datos y sus propiedades no dependen de la codificación escogida.
- Determinada codificación se escoge porque las operaciones implementadas usándola mantienen las propiedades del tipo de dato en cuestión:
  - Se usa CA2 para representar enteros porque la suma sigue siendo conmutativa y el 0 sigue siendo el elemento neutro, por ejemplo.

- Hemos usado `float`, `int`, `char`...`vector<...>`.
- En algunos casos conocíamos detalles de su codificación.
- En otros casos no.
- Para usar un tipo de datos sólo es necesario conocer:
  - Las operaciones que se le pueden aplicar.
    Ej: el `%` se puede aplicar a `int` pero no a `float`
  - Sus propiedades.
    Ej: el `*` o la `/` tienen mayor precedencia que la `+` o la `-`, la asociatividad de todos ellos es de izquierda a derecha.

- La **especificación** de un tipo de datos consiste en definir las operaciones aplicables y sus propiedades.
- La **implementación** de un tipo consiste en elegir una representación del tipo en función de los tipos predefinidos y en implementar sus operaciones de forma que cumplan su especificación.
- La **encapsulación de datos** (también llamada **ocultación de información**) consiste en ocultar los detalles de la implementación de un tipo de datos a sus usuarios.
- La **abstracción de datos** es la separación entre la especificación de un tipo de datos y su **implementación**.
- Un **Tipo Abstracto de Datos** es un tipo de datos organizado de forma que la especificación de los valores y la especificación de las operaciones sobre los valores están separadas de la representación de los valores y de la implementación de las operaciones.

- Se realiza mediante la palabra reservada `class`, que generaliza la orden `struct`.
- De la misma forma que `struct`, la la palabra reservada `class` sirve para definir una ”tupla”, es decir, un agregado de tipos de datos existentes y además permite definir las operaciones que es posible aplicar a ese nuevo tipo.

Una clase consta de cuatro elementos:

- El nombre de la clase.
- Los campos que definen la tupla, también llamados **datos miembro** o **atributos**.
- Las operaciones que se definen sobre el tipo, también llamadas **funciones miembro** o **métodos**.
- Los niveles de acceso desde el programa, que controlan al grado de acceso desde las funciones definidas fuera de la clase a los miembros (datos y funciones) de ésta.

Existen tres niveles de acceso:

- `public`, `protected` y `private`.
- El nivel de acceso `public` significa que el miembro es accesible desde cualquier función.
- El nivel `private` significa que el miembro solamente puede ser accedido desde otra función miembro de la misma clase.
- El nivel `protected` se verá más adelante.

NOTA: Por defecto los miembros son `private`. Desde que se especifica un nivel de acceso hasta que se especifica otro distinto, se aplica el primero a los miembros comprendidos entre ambos.

- Normalmente, el nivel de acceso a los atributos es `private`.
  - De esta forma existe *ocultación de información*.
- También puede haber métodos con nivel de acceso `private`.
  - Se usarán exclusivamente desde miembros de la propia clase.
- Habrá métodos públicos que se pueden usar desde cualquier parte del programa.
  - Son el *interfaz público* de la clase, mediante el que se acceden y modifican los valores de los atributos.
- Es posible declarar funciones externas a la clase que tengan acceso a la parte privada de una clase, mediante la palabra reservada `friend`.

Esquema general de una clase, alternativas:

```
class nombre_clase{
      //esta parte es privada
      //atributos
      tipo1 atributo1;
      tipo2 atributo2;
      //metodos privados
      //definicion+declaracion
      tipox metodox(tipoa pa, tipob pb, ...){
            ...
      }
      //metodos publicos
      //definicion+declaracion
public:
      tipoy metodoy(tipot pt, ...){
            ...
      }
};
```

```
class nombre_clase{
      //esta parte es privada
      //atributos
      tipo1 atributo1;
      tipo2 atributo2;
      //metodos privados
      //declaracion
      tipox metodox(tipoa, tipob, ...);
      //metodos publicos
      //declaracion
public:
      tipoy metodoy(tipot, ...);
};
//definicion, es necesario especificar
//a que clase pertenecen
tipox nombre_clase::metodox(tipoa pa,
                            tipob pb, ...){
      ...
      }
tipoy nombre_clase::metodoy(tipot pt, ...){
      ...
      }
```


- De una variable del tipo de una clase se suele decir que es un *objeto*.
- A los miembros de un objeto se accede mediante el nombre del objeto y el nombre del miembro, separados por un punto: `nombre_objeto.nombre_miembro`
- Si el miembro es un método, además se escriben los () y los parámetros (si existen) después del nombre del método: `nombre_objeto.nombre_metodo(...)`
  - Dentro de la definición de un método, si se escribe el nombre de un miembro, se entiende que se refiere al correspondiente del objeto que ha llamado a este método.

NOTA: dada la declaración `nombre_clase *pc;` la notación `(*pc).miembro` se puede abreviar como `pc->miembro`.

- Dependiendo de si:
  - Un miembro es público o no.
  - O bien de la parte del programa (miembro de la clase, función `friend` u otro lugar).

  se podrá acceder a el o no.

- Si el miembro es público se puede utilizar desde cualquier parte del programa.
- Si el miembro es privado sólo se puede utilizar desde un método de la propia clase o función `friend`.
- Desde un método de la clase se pueden utilizar todos los miembros.
- Desde una función que no pertenezca a la clase sólo se pueden utilizar los miembros públicos.
- Desde una función `friend` se puede acceder a todos los miembros.


```
class nombre_clase{
      //esta parte es privada
      //atributos
      tipo1 atributo1;
      tipo2 atributo2;
      //metodos privados
      tipox metodox(tipoa pa, tipob pb, ...){
         nombre_clase a;
         //correcto metodox es miembro de
         //nombre_clase
         ...a.atributo1...
         ...
      }
```

```
//metodos publicos
public:
      tipoy metodoy(tipot pt, ...){
         nombre_clase a;
         //correcto metodoy es miembro de
         //nombre_clase
         ...a.atributo1...
         //este es el del objeto que
         //llame a metodoy
         ...atributo1...
      }
};

int main()
{
//declaracion del objeto
nombre_clase nombre_objeto;
//correcto metodoy es publico
...nombre_objeto.metodoy(...)...
//incorrecto atributo1 es privado
...nombre_objeto.atributo1...
//incorrecto metodox es privado
...nombre_objeto.metodox(...)...
}
```


La argucia que se utiliza para hacer esto es el puntero oculto `this`, lo que sucede realmente es:

```
class nombre_clase{
      //esta parte es privada
      //atributos
      tipo1 atributo1;
      tipo2 atributo2;
      //metodos privados
      //argumento adicional, puntero
      //a la clase this
      tipox metodox(nombre_clase *this,
                    tipoa pa, tipob pb, ...){
         nombre_clase a;
         //correcto metodox es miembro de
         //nombre_clase
         ...a.atributo1...
         ...
      }
```

```
//metodos publicos
public:
      tipoy metodoy(nombre_clase *this,
                    tipot pt, ...){
         nombre_clase a;
         //correcto metodoy es miembro de
         //nombre_clase
         ...a.atributo1...
         //este es el del objeto que
         //llame a metodoy
         //equivale a (*this).atributo1
         ...this->atributo1...
      }
};

int main()
{
//declaracion del objeto
nombre_clase nombre_objeto;
//correcto metodoy es publico
//se le pasa un puntero al objeto
//que lo ha llamado
...metodoy(&nombre_objeto,...)...
//incorrecto atributo1 es privado
...nombre_objeto.atributo1...
}
```


Para dar valor a los atributos de una clase cuando son privados (lo usual) se utilizan métodos públicos.

- Lo habitual es utilizar *constructores*:
  - Son métodos públicos.
  - Su nombre es el de la clase.
  - Pueden sobrecargarse.
  - Suele haber al menos:
    - Un *constructor por defecto*, sin parámetros.
    - Un constructor con tantos argumentos como atributos tenga la clase y del mismo tipo.
  - En ocasiones también es necesario un *constructor de copia*.
  - Los dos primeros se invocan cuando se declara un objeto de la clase, el último cuando se pasa un objeto por valor.
  - Pueden escribirse constructores específicos para convertir tipos de datos existentes en objetos de la clase.

NOTA: No es lo mismo un constructor por defecto que un constructor con parámetros por defecto.

```
class nombre_clase{
      //esta parte es privada
      //atributos
      tipo1 atributo1;
      tipo2 atributo2;
      //metodos privados
      tipox metodox(tipoa pa, tipob pb, ...){
         ...
      }
//metodos publicos
public:
      //constructor por defecto
      nombre_clase(){
         //normalmente:
         //valores por defecto
         atributo1=...;
         atributo2=...;
         ...
      }
      //otro constructor
      nombre_clase(tipo1 p1, tipo2 p2, ...){
         //normalmente:
         atributo1=p1;
         atributo2=p2;
         ...
      }
      //constructor de copia
      nombre_clase(const nombre_clase &p){
         //normalmente:
         atributo1=p.atributo1;
         atributo2=p.atributo1;...
      }
      //convierte de int a nombre_clase
      nombre_clase(int x){
         //si es posible convertir
         //int a tipo1
         atributo1=x;...
      }
      //resto metodos publicos
      tipoy metodoy(tipot pt, ...){...}
};
tipo funcion_externa(nombre_clase x){...}
int main()
{
//llama a nombre_clase()
nombre_clase nombre1;
//llama a nombre_clase(tipo1,tipo2,...)
nombre_clase nombre2(val_tipo1,val_tipo2,...);
//llama a nombre_clase(nombre_clase const &)
...funcion_externa(nombre2)...
//llama a nombre_clase(int)
nombre1=7;
}
```


- De la misma forma que se puede prohibir que un método pueda modificar un parámetro declarándolo `const`, se puede prohibir que un método pueda modificar los atributos del objeto que lo llama añadiendo `const` después de los '()'
- Es posible permitir a una función externa a la clase el acceso a la parte privada de una clase, declarándola `friend`


```
class nombre_clase{
      tipo1 atributo1;
      tipo2 atributo2;
      tipox metodox(tipoa pa,
                    tipob pb,
                    ...){
         ...
      }
public:
      nombre_clase(){
         atributo1=...;
         atributo2=...;
         ...
      }
      nombre_clase(tipo1 p1,
                   tipo2 p2,
                   ...){
         atributo1=p1;
         atributo2=p2;
         ...
      }
      tipoy metodoy(tipot pt, ...){
      //correcto metodoy pertenece a la clase
      ...atributo1...
      ...
      }
      tipoy metodoz(tipot pt, ...)const{
      //incorrecto metodoz es const
      //...atributo1...
      ...
      }
      //declaracion como friend de funcion_externa2
      friend tipo funcion_externa2(nombre_clase);
};
tipo funcion_externa1(nombre_clase x){
  //incorrecto, no pertenece a la clase
  //...x.atributo1...
  ...
}
tipo funcion_externa2(nombre_clase x){
  //correcto, no pertenece a la clase
  //pero se declaro dentro de la clase
  //como friend
  ...x.atributo1...
  ...
}
int main()
{
nombre_clase nombre1, nombre2(val_tipo1,val_tipo2,...);
...funcion_externa1(nombre2)...
...funcion_externa2(nombre2)...
}
```


- C++ permite **reusar código**
  - de un modo seguro (minimizando el riesgo de cometer errores) y
  - rápido (sin necesidad de escribir muchas líneas de código).
- Es posible emplear clases existentes para definir otras nuevas
  - añadiendo funcionalidades que no existían
  - sin modificar la o las clases originales y
  - sin repetir código (copiar y pegar).
- Los mecanismos para realizar esta tarea son la **composición** y la **herencia**. En esta sección veremos el primero de los mecanismos.

- Se usa composición cuando alguno de los atributos de una clase es un objeto de otra clase.
- También se puede decir que las clases están anidadas.
- Para dar valor a los atributos del objeto que es a su vez atributo de la otra clase, se usan *inicializadores*:
  - Llamadas a los constructores de la clase a la que pertenezca/ el/los atributo/s.
  - Se escriben antes de las '{}' de los constructores de la clase compuesta.
  - Se separan de los '()' de los constructores de la clase compuesta por ':'.
  - Si hay varios, se separan por comas.
  - Se ejecutan antes del código que va entre las '{}'.


```
class componente1{//una clase
      tipo1 a1;
      tipo2 a2;
      public:
      componente1(){
      ...
      }
      componente1(tipo1 p1, tipo2 p2){
      ...
      }
      ...
};
class componente2{//otra clase
      tipox ax;
      tipoy ay;
      public:
      componente2(){
      ...
      }
      componente2(tipox px, tipoy py){
      ...
      }
      ...
};
```

```
class compuesta{//una clase compuesta
      //dos atributos, objetos de las otras clases
      componente1 a;
      componente2 b;
      public:
      //constructor por defecto
      compuesta ():a(),b(){ //inicializadores
      ...
      }
      //constructor desde objetos de las otras clases
      compuesta (componente1 c1, componente2 c2){
         a=c1;
         b=c2;
      ...
      }
      //constructor a partir de los valores de los
      //atributos de los objetos de las otras clases
      compuesta (tipo1 v1, tipo2 v2, tipox vx,
                tipoy vy):a(v1,v2),b(vx,vy){//inic.
      ...
      }
      //alternativa
      //compuesta (tipo1 v1, tipo2 v2, tipox vx, tipoy vy){
      //
      //   a=componente1(v1,v2);
      //   b=componente2(vx,vy);...
      //}....
};
```


- En esta sección se explicará como crear y eliminar objetos a voluntad, mediante los operadores `new` y `delete`.
- El operador `new` toma como argumento un nombre de un tipo y devuelve un puntero a ese tipo, el cual contiene una dirección de memoria no usada, de modo que marca el comienzo de un bloque, capaz de albergar un valor del tipo al cual apunta el puntero.
- Un objeto creado por `new` se dice que está en el **almacenamiento libre** o **memoria dinámica**.
- El operador `delete` toma como argumento un puntero a una variable dinámica y libera el espacio que esta variable ocupa para que pueda ser empleado por otras aplicaciones a `new`.


```
struct pareja{
   float re,im;
};
int main()
{
pareja *p;    //dec. del puntero a pareja
p=new pareja; //aplicación de new a pareja,
              //asigna el resultado a p
p->re=3.0;    //se puede usar (*p) como si
p->im=1.0;    //fuese de tipo pareja
(*p).re=3.0;  //lo mismo
(*p).im=1.0;
delete p;     //libera el espacio de memoria
}
```

- Cuando se usa `new` y `delete` para gestionar la memoria de algún atributo de una clase es necesario escribir el *destructor* de la clase.
- El **destructor** de una clase es un método que es llamado automáticamente cuando el objeto se destruye, con el fin de liberar la memoria asignada cuando el objeto deje de existir:
  - Salir del ámbito si es automático.
  - Llamada a `delete` si es dinámico.
  - Final del programa si es global o estático.
- El destructor tiene el mismo nombre que el constructor pero precedido de una vírgula (~).
  - No tiene argumentos.
  - No se puede llamar de forma explícita.


```
#include<iostream>
using namespace std;
struct datos{
  int a,b,c;
};
class estructura_dinamica{
  datos *A;
public:
  estructura_dinamica(int a, int b, int c){
    A=new datos;
    A->a=a;
    A->b=b;
    A->c=c;
    cout<<"Objeto creado"<<endl;
  }
  ~estructura_dinamica(){
    delete A;
    cout<<"Objeto destruido"<<endl;
  }
  int suma(){
    return A->a+A->b+A->c;
  }
};
```

```
void f(int n){
  int i;
  //llamada al constructor
  estructura_dinamica a(1,2,3);
  cout<<n+a.suma()<<endl;
  //se destruye, llamada al destructor
}
int main()
{
  f(2);
  f(3);
}
```

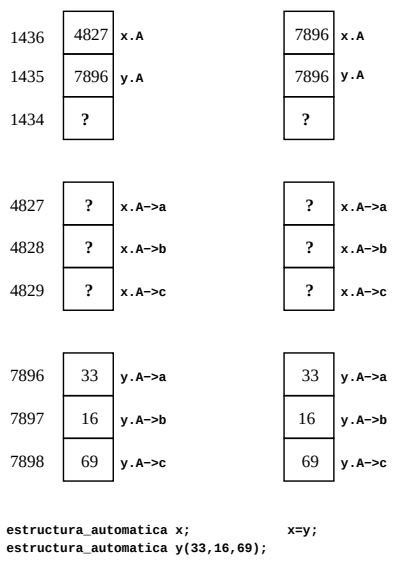
El programa anterior muestra en pantalla el siguiente texto:

```
Objeto creado
8
Objeto destruido
Objeto creado
9
Objeto destruido
```


- Cuando no se usan `new` y `delete`, la asignación de objetos y el paso por valor no ofrecen ningún problema.
  - Los atributos de un objeto se copian en los del otro.
- Si se usan `new` y `delete` es necesario escribir el operador de asignación y el constructor de copia, de modo que se realicen esas operaciones correctamente.
- Dada la definición de `estructura_dinamica` anterior, este programa produce un error en tiempo de ejecución.

```
int main()
{
estructura_dinamica x;
estructura_dinamica y(33,16,69);//declaracion de variables
x=y;                            //provocará un error
//fin del programa, se llamará al destructor de x y de y
//la segunda llamada provoca un error en tiempo de ejecución
}
```


Representación gráfica del programa incorrecto anterior. Obsérvese como se asigna el atributo de y a x, de modo que x e y son el mismo objeto porque su atributo apunta a la misma zona de memoria. La memoria reservada para x permanece sin usar.

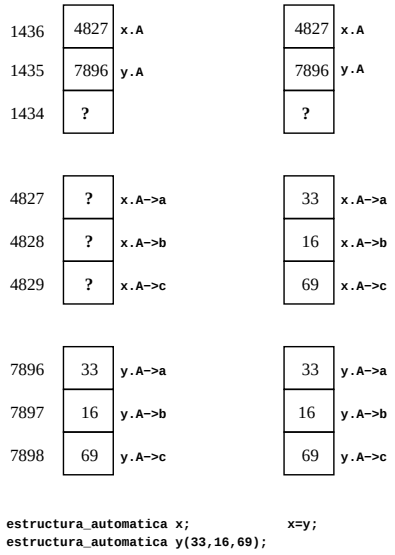


El operador de asignación se debería haber redefinido así:

```
void operator=(const  estructura_dinamica &z){
  //copia el miembro a del atributo A de z
  //en el miembro a del atributo A del objeto
  //que llama al operador =
  A->a=z.A->a;
  //analogamente para b
  A->b=z.A->b;
  //analogamente para c
  A->c=z.A->c;
}
```

NOTA: Este operador no se puede concatenar.

Representación gráfica del programa anterior redefiniendo adecuadamente el operador =. Obsérvese como NO se asigna el atributo de y a x, de modo que x e y son objetos distintos porque su atributo apunta zonas de memoria distintas. La memoria reservada para x se utiliza adecuadamente.



- Una situación similar se produce cuando se utiliza el paso por valor de variables como argumentos de funciones:
  - Cada parámetro real de la llamada se copia en el parámetro formal correspondiente en la definición de la función.
  - Previamente se ha de crear un nuevo objeto, en el que se copiará el valor del parámetro real.
- Si se utiliza memoria dinámica, se empleará el operador `new`.
- Se sobrecarga el constructor, definiendo una versión que recibe como argumento una referencia constante a una variable del mismo tipo.
- El constructor de copia de una clase `T` tendrá un prototipo `T(const T&x);`.

En el caso de `estructura_dinamica` el constructor de copia se define así:

```
estructura_dinamica(const estructura_dinamica &a){
  //asignación de memoria al atributo del objeto
  //que llama al constructor
  A=new datos;
  //asignación de los miembros de A
  A->a=a.A->a;
  A->b=a.A->b;
  A->c=a.A->c;
 }
```

Programa completo, se han añadido acciones de salida para observar la ejecución de constructores y operadores.

```
#include<iostream>
using namespace std;
struct datos{
  int a,b,c;
};
class estructura_dinamica{
  datos *A;
public:
  estructura_dinamica(int a, int b, int c){
    A=new datos;
    A->a=a;
    A->b=b;
    A->c=c;
    cout<<"Constructor a partir de enteros"
        <<endl;
  }
  ~estructura_dinamica(){
    delete A;
    cout<<"Destructor"<<endl;
  }
  estructura_dinamica(){
    //constructor por defecto
    A=new datos;
    A->a=0;
    A->b=0;
    A->c=0;
    cout<<"Constructor por defecto"<<endl;
  }
  estructura_dinamica (
          const estructura_dinamica &a){
    A=new datos;
    A->a=a.A->a;
    A->b=a.A->b;
    A->c=a.A->c;
    cout<<"Constructor de copia"<<endl;
  }
  void operator=(const estructura_dinamica &z){
    A->a=z.A->a;
    A->b=z.A->b;
    A->c=z.A->c;
    cout<<"Operador de asignacion"<<endl;
  }
  int suma(){
    return A->a+A->b+A->c;}
};
void f_valor(estructura_dinamica a){
  cout<<a.suma()<<endl;
}
void f_referencia(estructura_dinamica &a){
  cout<<a.suma()<<endl;
}
int main(){
  estructura_dinamica a(1,2,3),b;
  b=a;
  f_valor(b);
  f_referencia(b);
}
```

Salida por pantalla del programa anterior (izquierda) y explicación (derecha)

```
Constructor a partir de enteros
Constructor por defecto
Operador de asignacion
Constructor de copia
6
Destructor
6
Destructor
Destructor
```

```
Declaración de a
Declaración de b
b=a;
Llamada a f_valor
Salida en f_valor
Fin de llamada a f_valor
Salida en f_referencia
Fin de programa se destruye a
Fin de programa se destruye b
```


- El tipo `vector` es un **tipo paramétrico**.
- Se pueden construir vectores a partir de cualquier tipo:
  `vector<T> nombre;`
  Donde `T` es cualquier tipo existente en C++ y `nombre` es un identificador
- C++ permite definir estos tipos mediante la siguiente sintaxis:

```
template<class T> class nombre_clase{
                  //aqui se definiria la clase
                  ...
};
```

  Donde `T` es un tipo cualquiera.
- El nombre del tipo creado de esta forma es:
  `nombre_clase<T>`

- De esta forma el tipo de los atributos de una clase se fija en tiempo de compilación, en función de como se declaren los objetos de dicha clase.
  - El compilador examina las declaraciones de templates y *genera* las versiones adecuadas sustituyendo el nombre simbólico de los tipos por los nombres que aparecen en las declaraciones.
  - Se generan todas las versiones, correspondiéndose con todas las declaraciones con tipos distintos.
- Por ejemplo, si se declara `nombre_clase<int> a;` y `nombre_clase<float> b;` se genera una versión con `int` en lugar de `T` y otra con `float` en lugar de `T`
- Un template puede construirse a partir de varios tipos:

```
template<class T1, class T2> class nombre_clase{
                             //aqui se define la clase
                             ...
};
```

Declaración + definición

```
template <class T1, class T2> class par{
  T1 uno;
  T2 dos;
public:
  par(){}
  par(T1 a, T2 b){
    uno=a;
    dos=b;
  }
  T1 primero(void)const{
    return uno;
  }
  par<T1,T2> suma(const par<T1, T2> &b)const{
    return par(uno+b.uno,dos+b.dos);
  }
};
int main(){
  par<int,float> a(1,7.3),b(0,0.3),c;
  c=a.suma(b);
  par<int,int> x(1,2),y(3,4),z;
  z=x.suma(y);
}
```

Definición de nuevos tipos de datos.
Tipos parametrizados: templates

Primero declaración y después definición

```
template <class T1, class T2> class par{
  T1 uno;
  T2 dos;
public:
  par();
  par(T1, T2);
  T1 primero(void)const;
  par<T1,T2> suma(const par<T1, T2> &)const;
};
template <class T1, class T2>  par<T1,T2>::par(void){}
template <class T1, class T2>  par<T1,T2>::par(T1 a, T2 b){
  uno=a;
  dos=b;
}
template <class T1, class T2>  T1 par<T1,T2>::primero(void)const{
  return uno;
}
template <class T1, class T2>   par<T1,T2> par<T1,T2>::suma(const par<T1, T2> &b)const{
  return par(uno+b.uno,dos+b.dos);
}
int main(){
  par<int,float> a(1,7.3),b(0,0.3),c;
  c=a.suma(b);
  par<int,int> x(1,2),y(3,4),z;
  z=x.suma(y);
}
```

Definición de nuevos tipos de datos.
Tipos parametrizados: templates
Acciones genéricas

- Del mismo modo que se pueden definir tipos parametrizados, se pueden definir acciones parametrizadas o genéricas.
- Si los argumentos de una acción (función u operador) pueden ser de cualquier tipo (en cierto sentido el *tipo* de los argumentos se comporta como un argumento más) se dice que la acción es genérica, por ejemplo:

```
#include<iostream>
using namespace std;
template<typename T> void swap(T &a,T &b)
{
T c=a;
a=b;
b=c;
}
int main()
{
int a1=1;b1=2;
swap(a1,b1);//se llama a la función swap(int,int)
cout<<a1<<' '<<b1<<endl;
float a2=1.0,b2=2.0;
swap(a2,b2);//se llama a la función swap(float,float)
cout<<a2<<' '<<b2<<endl;
}
```