LENGUAJE C++

Tema 3 – Clases y Objetos

Clases

- ***Clases:*** ***Estructura de Datos*** que modela de manera general un Objeto. Una ***instancia*** de una clase es un **Objeto**.
- La palabra reservada ***class*** declara una clase de objeto con el cual crear objetos de dicho tipo.
  - Una clase puede estar compuesta por:
    - ***Propiedades:*** variables y/o estructuras definidas internamente para almacenar el estado del Objeto.
    - ***Funciones Miembro:*** también llamados Métodos u Operaciones internas del Objeto. Son las funciones que le pertenecen al Objeto.
  - Estos pueden ser: *Públicos, Protegidos o Privados.*
    - **Públicos:** La *clase derivada* hereda los elementos Públicos de la *clase base*.
    - **Protegidos y Privados:** A estos elementos pueden accederlo solo las funciones miembro o las **friends.**

Sintaxis de *CLASS*

```
class [className [: base-list ]]
{
  private member-list
  public:
       public member-list
} [declarators];
```

className: se vuelve una palabra reservada.

base-list: clases de la cual la *className* se deriva (su clase base). Cada clase base puede ser precedido por especificados de acceso: **public**, **private**, **protected**.

member-list: *Declara los miembros de la clase*: puede incluir propiedades, métodos, enums, campo de bits, tipos definidos por el usuario. No acepta inicialización explicita.

Declarators: Declara uno o mas objetos de la clase tipo.

Definición de Clases

```
class pila
{ public:   int pos[Max_pila];
            int fin;
  public:   void insertar(int i);
            int quitar(void);
            int mostrar(int elemento);
};

int main()
{ int i;
  pila Xpila;
  cout <<"Prueba de Clase pila\n";
  for (i=0; i<Max_pila; i++)
      Xpila.insertar(i);
  return 0;
}
```

Tabla de privilegios de acceso a los miembros

Acceso en clase base	Clase base heredada como	Acceso desde Clase derivada
Public Protected Private	Public	Public Protected No access[1]
Public Protected Private	Protected	Protected Protected No access[1]
Public Protected Private	Private	Private Private No access[1]

[1] A menos que las declaraciones friends dentro de la clase base otorguen acceso explícitamente.

Ejemplo Clases 1

```
class A
{  int priv_A_p1;
   protected:
        int prot_A_p2;
   public:
        int pub_A_p3;
};

class C: public A
{  public:
        int C_p1;
        void f(void);
        int leerProt();
};
```

```
void C::f(void)
{       prot_A_p2=1; }

int C::leerProt(void)
{       return prot_A_p2;}

int main()
{       C xC;
        xC.pub_A_p3=1;
        xC.f();
        cout << xC.leerProt();

       return 0;
}
```

Muestra "1" por consola

Ejemplo Clases 2

```
class A
{  int priv_A_p1;
   protected:
      int prot_A_p2;
   public:
      int pub_A_p3;
};
class C: protected A
{  public:
      int C_p1;
      void f(void);
      int leerProt();
      int leerPub();
};
```

```
void C::f(void)
{     prot_A_p2=1;
      pub_A_p3=2;
}
int C::leerProt(void)
{     return prot_A_p2;}
int C::leerPub(void)
{     return pub_A_p3;}
int main()
{     C xC;
      xC.f();
      cout << xC.leerProt() << endl;
      cout << xC.leerPub()<< endl;
     return 0;
}
```

Muestra por consola "1 2"

Constructores y Destructores

- Es muy frecuente que ciertas partes de un objeto requieran:
  - Una **Iniciación** antes de empezar a trabajar con él.
    - Obtener memoria, iniciar variables, cargar datos desde un archivo, etc.
  - Una **Finalización** al terminar de trabajar con él.
    - Liberar memoria, bajar datos a un archivo, etc.
- **CONSTRUCTORES:** ***Iniciación*** automática del Objeto.
  - Función miembro de la **Class** que tiene el mismo nombre de la **Class.**
  - Es invocado cuando se instancia el Objeto.
- ***DESTRUCTORES:*** ***Finalización*** automática del Objeto.
  - Función miembro de la **Class** con el mismo nombre de la **Class**, pero va precedido por un "**~**".
  - Es invocado cuando se destruye el Objeto al salir del ámbito de creación.

Ejemplo Constructor/Destructor

```
class pila
{  int pos[Max_pila];
   int fin;
   public:
       pila(void); //CONSTRUCTOR
       ~pila(void); //DESTRUCTOR
       void insertar(int i);
       int quitar(void);
       int mostrar(int elemento);
       int leerFin(void);
};
pila::pila(void)
{  fin=0;
   for (int i=0; i<Max_pila; i++)  pos[i]=0;
}

pila::~pila(void)
{  fin=0;
   cout << "\n\n!!!OBJETO destruido!!!";
}
```

Herencia

- La Herencia es el proceso mediante el cual un objeto puede adquirir las propiedades y/o métodos de otro.
  - Permite definir clases particulares, a partir de clases generales, agregándole las Propiedades y métodos necesarios.
- Tipos de Herencia:
  - Simple: el nuevo Objeto es función de una sola clase.
  - Múltiple: el nuevo Objeto es función de varias clases.

Ejemplo Herencia Simple

```
class SerVivo
{  public:
        int nacimiento;
        int energia;
   public:
        int comer();
        int dormir();
};

class Hombre:public SerVivo
{  public:
        int sexo;
        int color;
   public:
        int cazar();
        int caminar();
};
```

```
class Planta:public SerVivo
{  public:
        int especie;
        int TipoTierra;
   public:
        int florecer();
};


int main()
{     Hombre xMen;
      Planta xFlor;
      xMen.sexo=1;
      xMen.nacimiento=22;
      xFlor.nacimiento=1;
      return 0;
}
```

Ejemplo Herencia Múltiple

```
class A
{ public:
      int A_p1;
      int A_p2;
};
class B
{ public:
      int B_p1;
      int B_p2;
};
class C: public A, public B
{ public:
      int C_p1;
};
```

```
int main()
{     C xC;
      A xA;
      B xB;

      xC.A_p1=1;
      xC.B_p1=1;
      xC.C_p1=1;
      return 0;
}
```

Relaciones BÁSICAS entre Clases

Asociación

- La asociación es una relación bidireccional.
  - Dada una instancia de cliente podríamos encontrar el objeto que denota sus compras.
- Posee cardinalidad y esta puede ser:
  - UNO a UNO: producto – nro de producto
  - UNO a MUCHOS: proveedor – productos
  - MUCHOS a MUCHOS: productos – facturas

Polimorfismo

- Un mismo método puede tener comportamientos diferentes.
- El Objeto reacciona de modo diferente ante los mismos mensajes.
  - ***El compilador seleccionará la rutina correcta según el dato que se le pase.***
  - ***Ejemplo: Sobrecarga de Funciones.***
    - `void Imprimir (int iNum);`
    - `void Imprimir (char cCar);`
    - `void Imprimir (float fNum);`

Uso de funciones en línea

- Cuando se llama a una Función, el compilador resuelve el direccionamiento (donde está la función, tamaño, parámetros) y realiza operaciones antes de ejecutarla (buscarla, alocarla, inicializarla).
- Una *Función inline* ya está ensamblada en línea cuando se la invoca, no hay que ir a buscarla.
  - En funciones de mucho uso, puede ahorrar tiempo.
- Se la puede implementar de dos maneras:
  - Definiendo los Métodos dentro de la misma **class** declarante.
  - Utilizando la palabra reservada ***inline***

Ejemplo Función inline

- Se puede hacer

```
class PP
    {  double f(int i);
       int x();
    };
```

- Y luego

```
PP::f(int i)
    {
       return i**i**i;
    }
```

- O se puede hacer

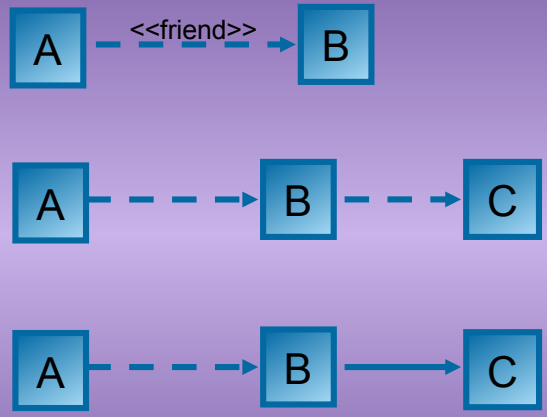
```
class PP
    { double f(int i);
          {return i**i**i;}
       int x();
    };
```

- O se puede hacer

```
inline PP::f(int i)
    { return i**i**i;}
```

Funciones amigas

- El modificador ***"friend"*** puede aplicarse a clases o funciones para inhibir el sistema de protección.
- La Relación de ***Amistad*** tiene características:
  - **La amistad no es simétrica.** Si A es amigo de B, B no tiene por qué ser amigo de A. (normalmente A no sabe que B no se considera su amigo).
  - **La amistad no puede transferirse**: si A es amigo de B y B es amigo de C, no implica que A sea amigo de C. (Es falso lo de "los amigos de mis amigos son mis amigos").
  - **La amistad no puede heredarse**. Si A es amigo de B, y C es una clase derivada de B, A no es amigo de C. (Los hijos de mis amigos, no tienen por qué ser amigos míos).
- Aumenta la eficiencia cuando 2 clases deben compartir una misma función, la cual puede pertenecer a una de ellas.
- La Amistad puede ser:
  - Externa: usa una función que no pertenece al objeto.
  - Interna: se usa una función que pertenece a algún Objeto



Ejemplo friend Externa

```
class A {  public:
                A(int i=0) {a=i;} //constructor
                void pubVer() {cout << a << endl;}
           private:
                int a;
                friend void extVer(A); // "Ver" es amiga de la clase A
};
void extVer(A Xa)
{// La función extVer puede acceder a miembros privados de A, pues es "amiga" de A
     cout << Xa.a << endl;
}
int main(int argc, char *argv[])
{
     A Na(10);
     extVer(Na); // Ver el valor de Na.a
     Na.pubVer(); // Equivalente a la anterior
     return 0;
}
```

Muestra por pantalla 10 10.

Ejemplo friend Interna

```
class B {  public:
                B(int i=0) {b=i;}; // inicializo
                void Ver() {cout << b << endl;} // muestro la var privada
                bool EsMayor(A Xa) {return b > Xa.a; } // Compara b con a
           private:
                int b;
};
class A {public:       A(int i=0) {a=i;};    // inicializo
   void Ver() {cout << a << endl;}
   private:      int a;
   friend bool B::EsMayor(A Xa); // accede a miembros privados de clase A
   };

void main(void)
{  A Na(10);
   B Nb(12);
   Na.Ver();
   Nb.Ver();
   if(Nb.EsMayor(Na)) cout << "Nb es mayor que Na" << endl;
   else cout << "Nb no es mayor que Na" << endl;
}
```

MUESTRA POR CONSOLA: 10 12 Nb es mayor que Na

Matrices y Punteros

- Se pueden crear matrices de OBJETOS de igual manera que con cualquier otro tipo de dato
  - ***tipobase*** *matriz[extension];…* ***miObj m[100];***
- Un puntero es una variable que contiene la dirección de memoria de otro objeto u variable
  - Si la variable X contiene la dirección de Y, se dice que X ***apunta a*** Y
  - ***tipobase*** **nombrepuntero;*

```
char  p, *j;
p = 'a';
j = &p;
miObj xObj, *ptrObj;
ptrObj= &xObj;
```

Dirección de Memoria		Variable de memoria
1000	*j	1003
1001		
1002		
1003	p	'a'
1004		
….		

Ejemplo Matrices y Punteros

```
class Pantalla
{   int colores; //numero de colores
    public:
        void SetColor(int num) {colores=num;};
        int GetColor(void){return colores;};
};
int main()
{   Pantalla monitores[3];
    Pantalla *ptrObj;
    int i;
    cout << "Cargando monitores"<< endl;
    for (i=0; i< 3; i++) monitores[i].SetColor(i);
    cout << "Mostrando monitores"<< endl;
    for (i=0; i<3; i++)
        cout << "Monitor[" <<i<< "] =" << monitores[i].GetColor() << " Colores";
    cout << "Cargando al Puntero el monitor 2"<< endl;
    ptrObj= &monitores[2];
    cout << "El ptr apunta al Monitor que tiene " << ptrObj->GetColor() << " Colores";
    cout << "Cargando al Puntero el monitor 1"<< endl;
    ptrObj= &monitores[1];
    cout << "El ptr apunta al que tiene " << ptrObj->GetColor() << " Colores";
}
```

SALIDAS

- Cargando monitores
- Mostrando monitores
- Monitor[0] tiene 0 Colores
- Monitor[1] tiene 1 Colores
- Monitor[2] tiene 2 Colores
- Cargando al Puntero el monitor 2
- El ptr apunta al Monitor que tiene 2 Colores
- Cargando al Puntero el monitor 1
- El ptr apunta al Monitor que tiene 1 Colores

Polimorfismo

Un mismo método/operación tiene comportamientos diferentes, según tipo y número de parámetros.

- *Una Interface, Múltiples implementaciones:* El compilador analiza los parámetros y elige. Si ningún método se adapta, aplicará reglas de cast.

Existen 2 formas de lograr el Polimorfismo

- Durante la COMPILACION:
  - **SobreCarga de Funciones:** Se definen varias funciones con el mismo nombre, con número y/o tipo de parámetros distintos.
  - **SobreCarga de Operadores:** La mayoría de los operadores en C++ están sobrecargados. "**+**" suma int o float. **"*"** multip o indi.
- Durante la EJECUCION:
  - **Funciones Virtuales:** Una función virtual es una función que se declara como virtual en una clase base y que se redefine en un o mas clases derivadas.
  - **Clases Abstractas:** Funciones Virtuales Puras (solo interface).

Ejemplo Sobrecarga de Funciones

```
int mayor(int a, int b);
char mayor(char a, char b);
double mayor(double a, double b);

int main()
{  cout << mayor('a', 'f') << endl;
   cout << mayor(15, 35) << endl;
   cout << mayor(10.254, 12.452) << endl;
   return 0;
}
int mayor(int a, int b)
{  if(a > b) return a; else return b;}

char mayor(char a, char b)
{  if(a > b) return a; else return b; }

double mayor(double a, double b)
{  if(a > b) return a; else return b; }
```

Sale por consola f 35 12.452

Sobrecarga de Operadores

- La palabra clave ***operator*** declara una función especificando lo que el “operador” significa cuando lo aplico.
- Sintaxis:
  - Prototipo:

    <tipo> ***operator*** <operador> (<argumentos>);
  - Definición del Código del Operador:

    <tipo> ***operator*** <operador> (<argumentos>) { <sentencias>;}
- Limitaciones para la sobrecarga de operadores:
  - No se pueden sobrecargar los operadores "**.**", "**.***", "**::**" y "**?:**".
  - Los operadores "**=**", "**[]**", "**->**", "**()**", "**new**" y "**delete**", pueden ser sobrecargados como miembros de una clase.

Ejemplo Sobrecarga de Operadores

```
struct complejo { float i,r;};

complejo operator +(complejo a, complejo b); // Prototipo del operador
                                             + para complejos
int main()
{  complejo x = {10,32};
   complejo y = {21,12};
   complejo z;
   z = x + y; // Uso del operador sobrecargado + con complejos
   cout << z.i << "," << z.r << endl;
}
complejo operator +(complejo a, complejo b)
{  complejo temp = {a.i+b.i, a.r+b.r};
   return temp;
}
```

Sale por consola 31,44

Funciones Virtuales

- **C++** determina a que función llamar en tiempo de ejecución, basándose en el tipo de objeto al cual apunta. Si se apuntan a diferentes objetos, se llama a diferentes versiones del Función Virtual
- Una función virtual se la declara como ***virtual*** en la clase base. Sin embargo, cuando se la redefine en las clases derivadas, no hace falta repetir la palabra ***virtual.***
- Se **debe acceder a estas funciones virtuales mediante un puntero que apunte a la clase Base.**

Ejemplo Funciones Virtuales

```
class Base
{   public:
        virtual void quien() {cout << "----Base----" <<endl;};
};
class prim_d:public Base
{  public:
        void quien() { cout << "prim_D. Primera Derivacion";};
};
class seg_d:public Base
{  public:
        void quien() { cout << "seg_D. Segunda Derivacion"
  <<endl;};
};
class ter_d:public Base
{      };
```

Ejemplo Funciones Virtuales

```
int main()
{ Base objBase, *ptr;
  prim_d objPrim;
   seg_d objSeg;
   ter_d objTer;
   ptr=&objBase;
   ptr->quien(); //accede al quien de BASE
   ptr=&objPrim;
   ptr->quien(); //accede al quien de Prim_d
   ptr=&objSeg;
   ptr->quien(); //accede al quien de Seg_d
   ptr=&objTer;
   ptr->quien(); //accede al quien de Base
}
```

SALIDAS

----Base----
prim_D. Primera Derivacion
seg_D. Segunda Derivacion
----Base----

Bibliografía & Licencia

- *Como programar en C++, 9na Ed*, Deitel, H.M. y Deitel, P.J., Pearson
- *Programación en C++, Un enfoque práctico*, Joyanes Aguilar, L., McGraw-Hill.
- *Thinking in C++, 2da Ed*, Bruce Eckel, Prentice Hall PTR.

- Autor:

Matías E. García

Prof. & Tec. en Informática Aplicada

www.profmatiasgarcia.com.ar

info@profmatiasgarcia.com.ar