UNIDAD II - PROBLEMAS CON SIMPLES RELACIONANDO OBJETOS

OBJETIVOS DE LA SEMANA 6ta (cuatr.)
Clase teórica

- Implementado relaciones "tiene un" (Class Alternan "tiene un" Caracter.)
- Tratamiento de palabras (Class Palabras "tiene un" Caracter)
- Tratamiento de palabras, enunciados complejos.
- HERENCIA - INTRODUCCIÓN, BENEFICIOS DE LA HERENCIA, REUSABILIDAD DEL SOFTWARE, COMPARICIÓN DE CÓDIGO, CONSISTENCIA DE LA INTERFAZ, COMPONENTES DE SOFTWARE, MODELADO RAPIDO DE PROTOTIPOS, OCULTACIÓN DE INFORMACIÓN, POLIMORFISMO
- HEURÍSTICAS PARA CREAR SUBCLASES. HERENCIA Y JERARQUÍA DE CLASES. TIPOS DE HERENCIA. CLASES DERIVADAS.

Clase práctica

- class Caracter
- Class Alternan

OBJETIVOS DE LA SEMANA 7ma (cuatr.)
Clase teórica

- DECLARACION DE UNA CLASE . DERIVADA. ATRIBUTOS Y METODOS.
- DERIVACION PUBLICA, PRIVADA, PROTEGIDA. REDEFINICION Y SOBRECARGA DE METODOS.
- HERENCIA SIMPLE. CONSTRUCTORES. DESTRUCTORES. UN EJEMPLO DE JERARQUÍA DE CLASES: HERENCIA SIMPLE. FUNCIONES VIRTUALES .
- CLASES ABSTRACTAS. POLIMORFISMO. APLICACIONES.

Clase práctica

- Class Palabra
- Class Palabra1

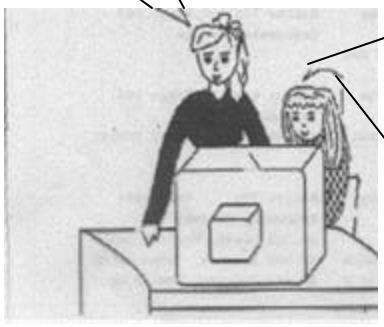
Material Disponible en el sitio labsys.

Autor: Tymoschuk, Jorge
Compaginación: Guzmán, Analía

IMPLEMENTANDO RELACIONES CON OBJETOS

RELACION "TIENE UN"

En esta Unidad continuamos con objetos ?



*Y si ... Comenzaremos por ver como podemos implementar relaciones. En general, como cada clase resuelve lo que le permite su comportamiento, y los problemas reales son mas complejos que esto, necesitamos "combinar esfuerzos", **trabajar en equipo**. Esto se logra mediante la relación "**Tiene un**", un automovil tiene un motor, habrá temas que la propia clase Automovil puede resolver, en general los detalles serán resueltos por Motor, Inyección, Carburación, Dirección ...*

Tratamiento de caracteres

Comencemos con temas conocidos. Por ejemplo, nuestra clase Caracter. Así como fué presentada en la Unidad I, es una clase orientada a un problema específico. Cuantas letras, vocales, dígitos, ... En general, nos conviene que las clases que usamos en un problema tengan una capacidad general; Por ejemplo en los problemas que siguen no queremos saber cuantos caracteres de cada tipo tenemos, así que no necesitamos de los contadores, en cambio no interesa el comportamiento de detección de tipos de caracteres. Entonces nuestra clase Caracter quedaría:

```
//  Tratamiento de caracteres, comportamiento mínimo ...
# include <iostream.h>
class Caracter{
private:                        // Parte interna de la clase
        char car;               // Nuestro caracter
public:                         // Parte publica, interface con el usuario: main()
        Caracter();             // Constructor, inicializa car en ' '
        Caracter(char car);     // Construcrtor, inicializa car mediante parámetro
        int esLetra();          // Retorna 1 minúscula, 2 mayúscula, 0 no es letra
        int esVocal();          // Retorna 1 si vocal, 0 no es vocal
        int esConso();          // Retorna 1 si consonante, 0 si no es
        int esLetMay();         // Retorna 1 si es letra mayuscula
        int esLetMin();         // Retorna 1 si es letra minúscula
        int esDigDec();         // Retorna 1 si es dígito decimal
        int esDigHex();         // Retorna 1 si es dígito hexadecimal
        int esSigPun();         // Retorna 1 si es signo puntuación
        void setCar(char)       // Inicializa car mediante parámetro
        char getCar();          // Retorna car
        int  finDatos();        // Retorna 1 si se digita #
};


Caracter::Caracter(){car=' ';}   // Constructor

Caracter::Caracter(char cara){ car = cara;}  //  Segundo constr.


void Caracter::setCar(char cara){car = cara;}


char Caracter::getCar(){
        return car;          //  Retornamos el atributo car.
}

int Caracter::esLetra(){
```

```
        int letra=0;
        if(esLetMay()) letra=1;
        if(esLetMin()) letra=1;
        return letra;              //  Retornemos lo que haya sido
} //  Fin de la función

int Caracter::esVocal(){
        int vocal=0;            // Inicialmente suponemos que no es vocal
        char voca[10]="aeiouAEIOU";
        for(int i=0;i <= sizeof(voca);i++)
                if(voca[i]==car) vocal=1;  //  Es una vocal
        return vocal;              //  Retornamos lo que sea ...
}


int Caracter::esConso(){
        int conso=0;
        if(esLetra() && !esVocal()) conso=1;
        return conso;
}

int Caracter::esLetMay(){
        int letra=0;                    //  Inicialmente suponemos que no es letra minuscula
        char mayu[26]="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        for(int i=0;i <= sizeof(mayu);i++)
                if(mayu[i]==car) letra=1;  //  Es una letra mayúscula
        return letra;
}

int Caracter::esLetMin(){
int letra=0;
        char minu[26]="abcdefghijklmnopqrstuvwxyz";
        for(int i=0;i < sizeof(minu);i++)
                if(minu[i]==car) letra=1;  //  Es una letra minúscula
        return letra;
}

int Caracter::esDigDec(){
        int digDec=0;
        char dig10[10]="1234567890";
        for(int i=0;i <= sizeof(dig10);i++)
                if(dig10[i]==car) digDec=1;
        return digDec;
}

int Caracter::esDigHex(){
        int digHex=0;
        char dig16[16]="1234567890ABCDEF";
        for(int i=0;i <= sizeof(dig16);i++)
                if(dig16[i]==car) digHex=1;
        return digHex;
}


int Caracter::esSigPun(){
int sigPun=0;
        char punct[04]=".,;:";
        for(int i=0;i <= sizeof(punct);i++)
                if(punct[i]==car) sigPun=1;
        return sigPun;
}

int Caracter::finDatos(){
        int finDat=0;
        if(car=='#') finDat=1;
        return finDat;
}
```

Ejemplo: Alternancias consonante/vocal.

Vamos a un primer ejemplo específico. Tenemos una secuencia de caracteres, terminada en el '#'. Queremos saber cuantas veces una vocal sigue inmediatamente después a una consonante y viceversa. La secuencia se inicia en vocal o consonante, indistintamente. A este par de vocal/Consonante ó consonante/vocal lo llamamos Alternancia. (Alternancia consonante/vocal o viceversa.

O sea que el concepto que debemos modelar es el de alternancia. Si el carácter anterior fue vocal y el actual es consonante, tenemos alternancia. Al revés también. Cualquier otra combinación no es alternancia.

La estrategia obligada para modelar este concepto es relacionar el carácter actual al anterior. El primer carácter, por no tener anterior, está naturalmente exceptuado de esta relación.

La detección de consonante y vocal es parte del comportamiento de la clase Carácter. Y lo precisamos aquí, imprescindiblemente. Aquí podemos hacer dos cosas:

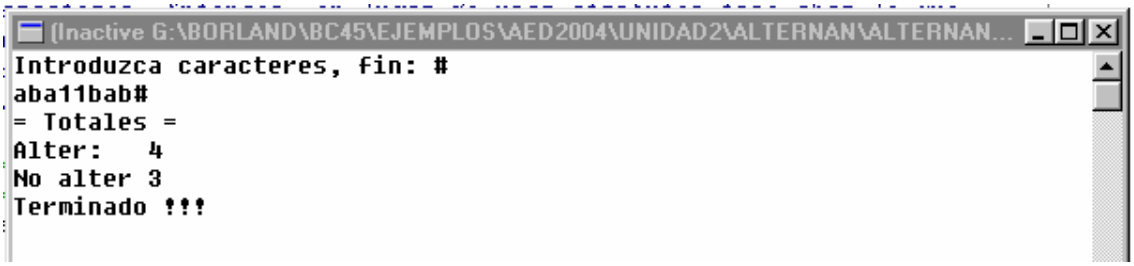
- Incorporar (usando copia y pegar) la codificación de estas funciones miembro de la clase Caracter en la clase Alternancia. Hecho esto, podremos fácilmente determinar si las variables Caracter actual y anterior (ambos tipo char) son consonantes o vocales.

- Declarar en la clase Alternancia dos objetos Caracter, lo que nos da **derecho a usar todo su comportamiento asociado**. Estamos implementando la relación "**tiene un**". La clase Alternancia **tiene un** objeto Caracter. (En realidad, necesita de dos de ellos, uno para el último carácter leído, otro para el anterior).

Tiene el alumno alguna duda de cual es la **solución correcta** ?

```
//  Alternancias consonante/vocal. El concepto de alternancia lo aplicamos
// sobre caracteres. Entonces, en lugar de usar atributos tipo char lo que
// nos conviene es emplear objetos Caracter, de los cuales aprovecharemos
// su comportamiento. En la clase Alternan tenemos cuatro funciones miembro
// propias, y dentro de ellas se invocan varias de Caracter.

# include <iostream.h>
# include <G:\Borland\BC45\Ejemplos\AED2004\Unidad2\Caracter\Caracter.cpp>
class Alternan{
private:
        int alter, noAlt;               //  Contadores,
        Caracter carAnt, carAct;
        void toString();                // la exhibición de los resultados
        int hayAlter();                 // Hay alternancia ?
public:
        Alternan();                     // el constructor,
        void implementar();             // Implementación de la alternancia
};

Alternan::Alternan(){                   //  Constructor
        alter=0; noAlt=0;               //  Contadores a cero
}

void Alternan::toString(){
        cout << "= Totales =\n";
        cout << "Alter:   "<< alter << " \n";
        cout << "No alter "<< noAlt << " \n";
        cout << "Terminado !!!\n";
}

int Alternan::hayAlter(){               // hay alternancia ?
        int hay = 0;                    // A priori no hay
        if(carAnt.esConso() && carAct.esVocal())hay=1;
        if(carAnt.esVocal() && carAct.esConso())hay=1;
        return hay;
```

```
}

void Alternan::implementar(){       // Implementación de la alternancia
       cout << "Introduzca caracteres, fin: #  \n";
       carAnt=cin.get();              //  Leemos el primero,
       carAct=cin.get();              // leemos el segundo
       while(!carAct.finDatos()){     //  Mientras no sea fin de datos ...
        if(hayAlter())alter++;
        else noAlt++;
        carAnt=carAct;
        carAct=cin.get();
       };
       toString();
}

void main(void){
Alternan demo;
demo.implementar();
};
```



Ejemplo: Tratamiento de Palabras (Cuantas palabras, cuantos caracteres)

Si consultamos el diccionario, veremos que dice de frase: Conjunto de palabras que tienen sentido. Nosotros no vamos a ser tan avanzados, solo vamos a considerar que las palabras, además de letras, pueden contener dígitos decimales. Por supuesto, no contendrán signos de puntuación. Los signos de puntuación, más los espacios en blanco separan las palabras.

O sea que tenemos frases constituidas por palabras y palabras por caracteres. El comportamiento de la clase Caracter ya lo vimos y usamos en el ejemplo anterior, clase Alternan.

Que comportamiento debemos modelar en la clase Palabras? Mínimamente debemos poder contar palabras y para ello es imprescindible que seamos capaces de detectar si el carácter que hemos acabado de leer pertenece a la palabra que estamos procesando o bien es uno de los caracteres que separan las palabras. Consideremos la frase:

El prof**e**, ... dice el alumno, ... es un burro#

Supongamos que en este momento nuestro último carácter leído es la letra '**e**' (negrita). Es una letra, pertenece a la palabra. No sabemos cual será el próximo, leamos ... Usemos la variable **car** para la lectura.

El profe**,** ... dice el alumno, ... es un burro#

Ahora tenemos car conteniendo una '**,**' (coma). No es letra ni dígito, entonces no pertenece a la palabra, podríamos decir que nos está indicando que la palabra ha terminado. Si nuestro propósito es contar palabras, es el momento de hacerlo. Sigamos leyendo.

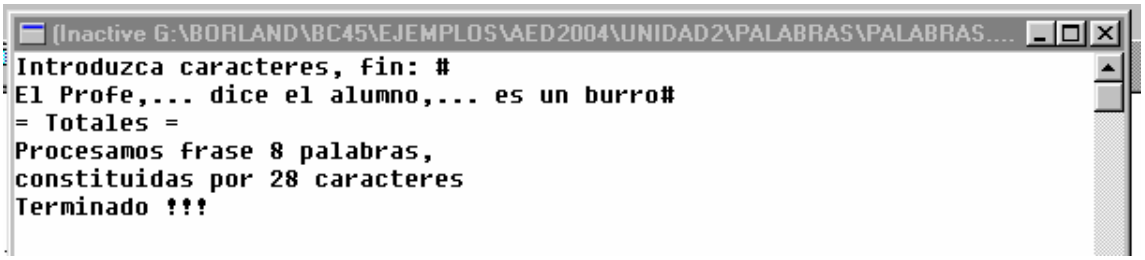
El profe, **.**.. dice el alumno, ... es un burro#

Ahora tenemos car conteniendo un '**.**' (punto). No es letra ni dígito, no pertenece a la palabra, pero tampoco podríamos decir que nos esté indicando que la palabra ha terminado. Ocurre que la palabra terminó en el carácter anterior. Descubrimos que no es suficiente considerar el contenido de car, una buena idea sería definir que **ocurre un fin de palabra cuando el carácter actual no pertenece a la palabra, pero el anterior si**(Es letra o dígito). Por ahora, usaremos esa *estrategia.*

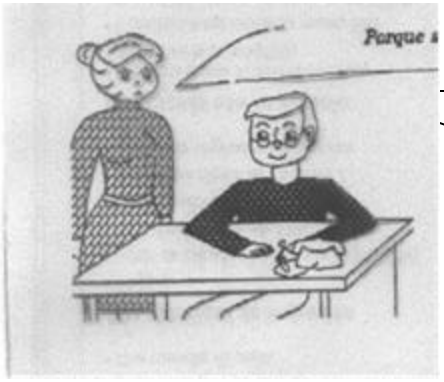
Letras, dígitos, signos punt. son tratados por la clase Caracter; esta clase provee comportamiento de lectura de caracteres, detección de fin de datos, etc ... la necesitaremos. Como en el caso anterior, estamos implementando la relación "**tiene un**". La clase Palabras necesita **tener** dos

objetos Caracter.

```
# include <G:\Borland\BC45\Ejemplos\AED2004\Unidad2\Caracter\Caracter.cpp>
class Palabras{
private:
    Caracter carAnt, carAct;
    int contCar, contPal;
    int  intPalab();                    // Estoy en el interior de una palabra ?
    int  finPalab();                    // Ha finalizado mi palabra ?
    void toString();                    // la exhibición de los resultados
public:
    Palabras();                         // Un constructor
    void implementar();                 // Implementando el enunciado
};

Palabras::Palabras(){     //  Constructor
    contCar=0; contPal=0;   //  Contadores a cero
};

int Palabras::intPalab(){               //  Estamos en el interior de la palabra ?
    int intP = 0;                       //  Suponemos lo contrario
    if(carAct.esLetra() || carAct.esDigDec()){  //  Recordemos que nuestras palabras
       intP = 1;                        //  contienen letras y/o dígitos
    }
    return intP;
};

int Palabras::finPalab(){               //  Suponemos que ya verificamos que no estamos
    int finP = 0;                   // en el interior de la palabra (Método intPalab())
    if (carAnt.esLetra() || carAnt.esDigDec()) finP = 1;
                                    // si el anterior pertenece, es fin de palabra
    return finP;
};

void Palabras::implementar(){
    cout << "Introduzca caracteres, fin: #  \n";
    carAct=cin.get();          //  L
    while(!carAct.finDatos()){
       if(intPalab())contCar++;      // dentro palabra, contamos caracteres
       else if(finPalab())contPal++;  // Finalizada, contamos palabra
       carAnt = carAct;              //  Siempre guardamos carAct
       carAct = cin.get();
    };
    if(finPalab())                   //  Al salir del ciclo puede quedar una
       contPal++;                    // palabra sin contar...
    toString();

}

void Palabras::toString(){
    cout << "= Totales =\n";
    cout << "Procesamos frase "<< contPal << " palabras, \n";
    cout << "constituidas por "<< contCar << " caracteres \n";
    cout << "Terminado !!!\n";
}

void main(void){
    Palabras pal;            // Construimos el objeto
    pal.implementar();       // Procesar los datos;
};
```



Bueno, hasta aquí está todo mas o menos claro. Pero que pasará si tenemos que resolver un enunciado con varios items, como siempre se estila en los parciales ?



Si, ya se que le preocupa. ***Una buena idea es tener comportamiento adecuado para cada uno de esos items, tanto dentro la palabra como fuera de ella****.*
Todo ese detalle de implementación debe ir en la parte privada de la clase. En la parte pública va solo lo que se usa en el main, o sea el constructor y el método que implementa. Es la ***interfase*** *de la clase con el usuario...*

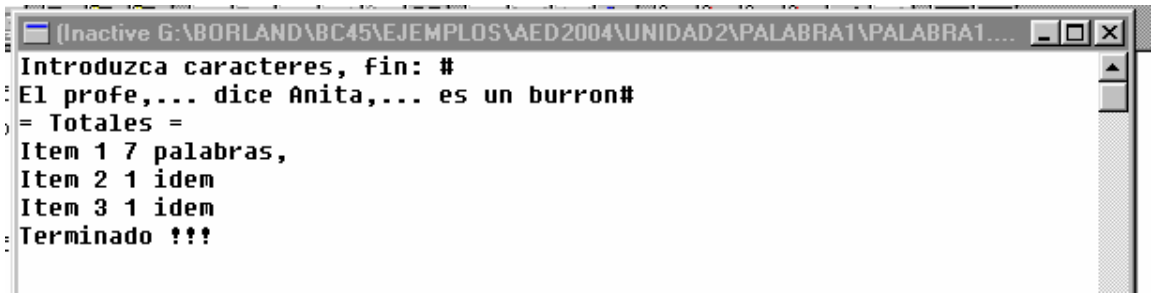
Ejemplo: Tratamiento de Palabras - Sea el siguiente enunciado:

- Cuantas palabras contiene la frase?
- Cuantas palabras tienen mayoría vocales?
- Cuantas palabras inician y terminan en consonante?

```
# include <G:\Borland\BC45\Ejemplos\AED2004\Unidad2\Caracter\Caracter.cpp>
class Palabra1{
private:
        Caracter carAnt, carAct;
        int contPal;            // Cuantas palabras contiene la frase?
        int contMayVoc;         // Cuantas palabras contienen mayoria vocales?
        int contIniTer;         // Cuantas palabra inician/terminan c/consonante
        int contVoc;            // Contador de vocales
        int contRes;            // Contador de los restantes caracteres
        int idIniCon;           // Identificador palabra inicia en consonante
        int intPalab();         // Estoy en el interior de una palabra ?
        int finPalab();         // Ha finalizado mi palabra ?
        void item1Int();         // Bloques de tratamiento de c/u de los
        void item2Int();        // items del enunciado. Los nombres terminados
        void item3Int();        // en Int tratan lo que corresponda del item
        void item1Fin();        // en el interior de la palabra, los que fina-
        void item2Fin();        // lizan en Fin idem en la finalización de la
        void item3Fin();        // palabra
        void toString();        // la exhibición de los resultados
public:
        Palabra1();             // Un constructor
        void implementar();     // Nuestro ciclo de lectura de caracteres
};

Palabra1::Palabra1(){       //  Constructor
        contPal=contMayVoc=contIniTer=contVoc=contRes=idIniCon=0;
        // Todos los contadores a cero
};
```

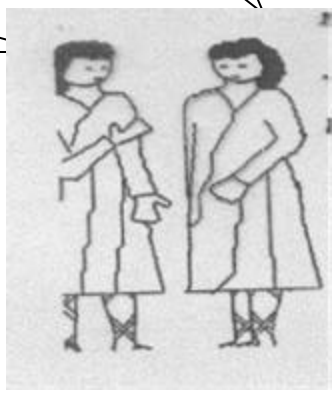
```
int Palabra1::intPalab(){     //  Estamos en el interior de la palabra ?
      int intP = 0;           //  Suponemos lo contrario
      if(carAct.esLetra() || carAct.esDigDec()){  //  Recordemos que nuestras palabras
       intP = 1;              //  contienen letras y/o dígitos
      }
      return intP;
};

int Palabra1::finPalab(){     //  Suponemos que ya verificamos que no estamos
      int finP = 0;           // en el interior de la palabra (Método intPalab())
      if (carAnt.esLetra() || carAnt.esDigDec()) finP = 1;
                              // si el anterior pertenece, es fin de palabra
      return finP;
};

void Palabra1::implementar(){
      cout << "Introduzca caracteres, fin: #  \n";
      carAct = cin.get();             //  Leemos el primero,
      while(!carAct.finDatos()){      //  Mientras no sea fin de caracteres ...
            if(intPalab()){           //  En el interior de la palabra,
                  item1Int();         //  tratamos el primer item,
                  item2Int();         //  tratamos el segundo,
                  item3Int();         //  y el último,
            }
            else if(finPalab()){      // Finalizada,
                  item1Fin();         //  tratamos el primer item,
                  item2Fin();         //  tratamos el segundo,
                  item3Fin();         //  y el último,
       }
       carAnt = carAct;               //  Siempre guardamos car en anterior
       carAct = cin.get();
      }; // while
      if(finPalab()){                 //  Al salir del ciclo puede quedar una
                                      // palabra sin procesar.
            item1Fin();               //  tratamos el primer item,
            item2Fin();               //  tratamos el segundo,
            item3Fin();               //  y el último,
       }
      toString();                     //  Exhibimos
};

void Palabra1::item1Int(){;};       //  Nada que hacer aquí por el primer item

void Palabra1::item1Fin(){contPal++;};      //  Solo contar palabras ...

void Palabra1::item2Int(){
      if (carAct.esVocal())contVoc++;
      else contRes++;
};

void Palabra1::item2Fin(){
      if (contVoc > contRes)contMayVoc++;
      contVoc=contRes=0;
};

void Palabra1::item3Int(){
      if (contVoc+contRes==1)
       if (carAct.esConso())idIniCon=1;
};

void Palabra1::item3Fin(){
      if (idIniCon && carAnt.esConso())contIniTer++;
      idIniCon=0;
};

void Palabra1::toString(){
```

```
        cout << "= Totales =\n";
        cout << "Item 1 "<< contPal << " palabras, \n";
        cout << "Item 2 "<< contMayVoc << " idem   \n";
        cout << "Item 3 "<< contIniTer << " idem    \n";
        cout << "Terminado !!!\n";
};

void main(void){
        Palabra1 demo;                  // Construimos el objeto
        demo.implementar();             // lo procesamos
};
```



*... de acuerdo, la relación **"tiene un"** es muy importante. Es la única que podemos implementar ? ...*

*De ninguna manera. En realidad, la relación mas importante es la otra, conocida como **"es un".***
*En esta relación el comportamiento y los datos asociados con la idea más abstracta forman un subconjunto del comportamiento y los datos asociados con la idea más específica. Por ejemplo, un florista (específico) **es un** comerciante (abstracto); los hechos generales de los comerciantes son aplicables a los floristas, pero para representar floristas puede requerirse de nuevos atributos y métodos.*
*El mecanismo para implementar la relación **"es un" es la herencia***



HERENCIA - INTRODUCCION

Herencia es la propiedad de que los ejemplares de una clase hija (o subclase) puedan tener acceso tanto a los datos como al comportamiento (métodos) asociados con una clase paterna (o superclase). La herencia es siempre transitiva, por lo que una clase puede heredar características de superclases alejadas muchos niveles. Esto es, si la clase Perro es una subclase de la clase Mamífero, y ésta a su vez es una subclase de la clase Animal, entonces Perro heredará atributos tanto de Mamífero como de Animal.

La herencia significa que el comportamiento y los datos asociados con las clases hijas son siempre una extensión (es decir, un conjunto estrictamente más grande) de las propiedades asociadas con las clases paternas. Una subclase debe reunir todas las propiedades de la clase paterna y otras más.

BENEFICIOS DE LA HERENCIA

REUSABILIDAD DEL SOFTWARE: Cuando el comportamiento se hereda de otra clase, no necesita ser reescrito el código que proporciona ese comportamiento. Lo anterior puede parecer obvio, pero las implicaciones son importantes. Muchos programadores pasan un alto porcentaje de su tiempo reescribiendo código que ya han escrito muchas veces antes; por ejemplo, para buscar un patrón en una cadena o para insertar un nuevo elemento en una tabla. Tales funciones pueden escribirse una sola vez y reutilizarse mediante técnicas orientadas a objetos. Otros beneficios del código reusable incluyen una mayor confiabilidad (cuanto más se reutilice el código, mayores serán las oportunidades de descubrir errores) y un menor costo de mantenimiento gracias a la compartición del código por todos los usuarios.

COMPARTICION DE CODIGO: La compartición de código puede tener lugar en diversos niveles cuando se usan técnicas orientadas a objetos. En un nivel, muchos usuarios o proyectos independientes pueden emplear las mismas clases. Otra forma de compartición ocurre cuando dos o más clases diferentes, desarrolladas por un solo programador como parte de un proyecto, heredan de una clase paterna única. Por ejemplo, un conjunto y un arreglo pueden ser considerados como una forma de colección; cuando esto sucede, significa que dos o más tipos diferentes de objetos compartirán el código que heredan. Dicho código se necesita escribir sólo una vez y sólo una vez contribuirá al tamaño del programa resultante.

CONSISTENCIA DE LA INTERFAZ: Cuando múltiples clases heredan de la misma superclase nos aseguran que el comportamiento que heredan será el mismo en todos los casos. De esta manera es más fácil garantizar que las interfaces para objetos sean similares y que al usuario no se le presente una colección confusa de objetos que son casi lo mismo pero que se comportan e interactúan en forma diferente.

COMPONENTES DE SOFTWARE: La herencia nos permite construir componentes de software reutilizable. Existen comercialmente muchas bibliotecas así.

MODELADO RAPIDO DE PROTOTIPOS: Cuando un sistema de software puede construirse en su mayor parte con componentes reutilizables, el tiempo de desarrollo puede concentrarse en entender la parte del sistema que es nueva e inusitada. De esta manera, pueden generarse más rápida y fácilmente los sistemas de software, llevando a un estilo de programación conocido como *modelado rápido de prototipos o programación exploratoria.* Se desarrolla un sistema prototipo, los usuarios experimentan con él, se produce un segundo sistema basado en la experiencia con el primero, se efectúa una exploración adicional y así se repite varias veces. Este estilo de programación es particularmente útil en casos en los que los objetivos y requisitos del sistema se entienden vagamente cuando comienza el proyecto.

OCULTACION DE INFORMACION: Cuando un programador reutiliza un componente de software sólo necesita entender la naturaleza del componente y su interfaz. No es necesario que el programador tenga información detallada sobre aspectos como las técnicas usadas para implantar el componente. Así se reduce la interconexión entre los sistemas de software.

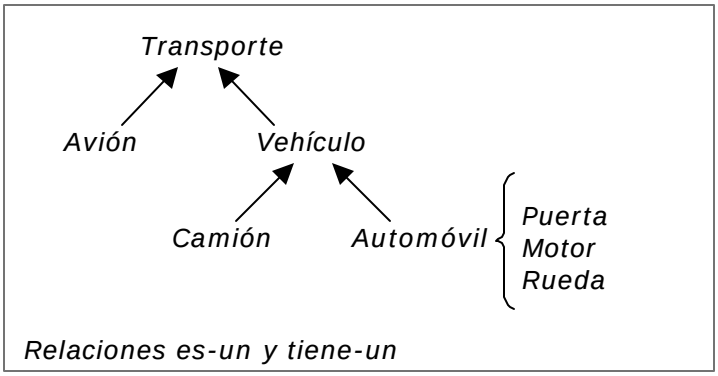
POLIMORFISMO: El polimorfismo de los lenguajes de programación permite al programador generar componentes reutilizables de alto nivel que puedan adaptarse para que se ajusten a diferentes aplicaciones mediante el cambio de sus partes de bajo nivel.

HEURISTICAS PARA CREAR SUBCLASES

La regla más importante para saber cuando una clase debe convertirse en subclase de otra o cuándo es más apropiado otro mecanismo es que, para que una clase se relacione con otra por medio de la herencia, debe haber una relación de funcionalidad entre ambas. Entonces describimos la regla ***es-un*** para captar la relación.

RELACIONES ES-UN y TIENE-UN

Hay dos relaciones de importancia conocidas coloquialmente como la relación ***es-un*** y la relación ***tiene-un*** (o es *parte-de*). La relacion "tiene un" ya la conocemos. Un gráfico puede aclarar ideas:



Relaciones es-un y tiene-un

Al decir que una subclase es una superclase, estamos diciendo que la funcionalidad y los datos que asociamos con la clase hija (o subclase) forman un superconjunto de la funcionalidad y los datos asociados con la clase paterna (o superclase).

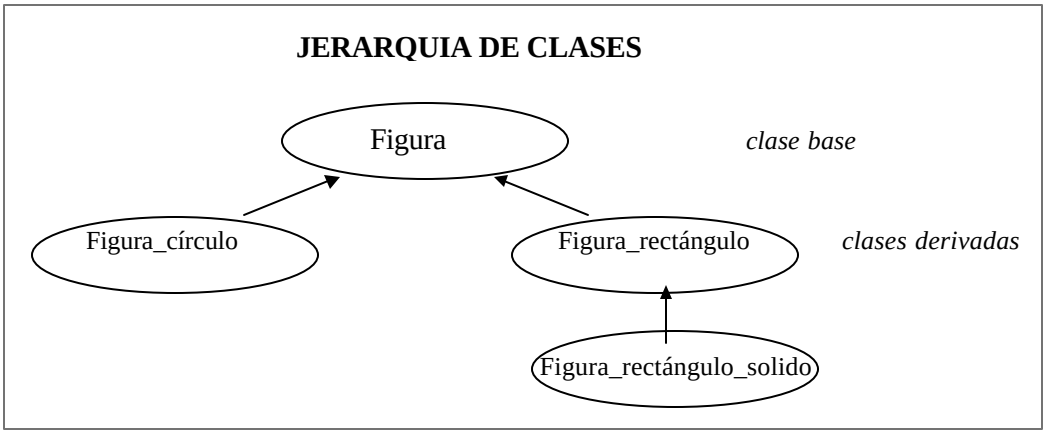
HERENCIA Y JERARQUIA DE CLASES

La herencia se manifiesta con la creación de un tipo definido por el usuario (**clase**), que puede heredar las características de otra clase ya existente o derivar las suyas a otra nueva clase. Cuando se hereda, las clases derivadas reciben las características (estructuras de datos y funciones) de la clase original, a las que se pueden añadir nuevas características o modificar las características heredadas.

La derivación de las clases consigue la reutilización efectiva del código de la clase base para sus necesidades.

C++ y Java utilizan un sistema de herencia jerárquica. Es decir se hereda una clase de otra, creando nuevas clases a partir de clases ya existentes. Solo se pueden heredar clases, no funciones ordinarias ni variables.

Una clase utilizada para derivar nuevas clases se denomina *clase base* (padre, superclase o ascendiente). Una clase creada de otra clase se denomina *clase derivada o subclase.* La terminología supone una clase base o clase padre y una clase derivada o clase hija. Esta relación supone un orden de jerarquía simple. A su vez, una clase derivada puede ser utilizada como una clase base para derivar más clases. Por consiguiente se puede construir jerarquías de clases, en las que cada clase sirve como padre o raíz de una nueva clase. La siguiente figura representa un diagrama de jerarquía de clases.



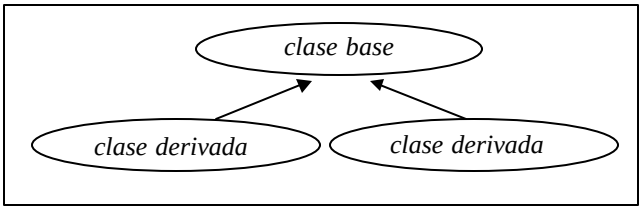
Decir que una clase Hija hereda de una clase Madre significa:

- que recupera los atributos y los métodos;
- que un objeto de la clase Hija es también un objeto de la clase Madre.

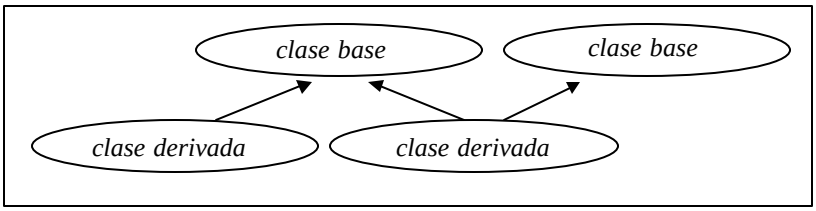
TIPOS DE HERENCIA

Existen dos tipos de herencia: simple y múltiple.

- La *herencia simple* es aquella en la que cada clase derivada hereda de una única clase. Cada clase tiene un solo ascendiente y puede tener muchos descendientes



- La *herencia múltiple* es aquella en la cual una clase derivada tiene más de una clase base.



En C++ existen los dos tipos, en Java solo la herencia simple.

Nota: En los lenguajes mas actuales la herencia múltiple no se implementa (Es como que fué una idea no muy feliz), por lo tanto no la estudiaremos.

CLASES DERIVADAS

Las clases herederas se llaman clases derivadas, clases hijas o subclases.

Las características de las clases derivadas son:

En general (para C++ y Java):

- Puede a su vez ser una clase base, dando lugar a una jerarquía de clases.
- Hereda todos los miembros de la clase base, pero solo podrá acceder a aquellos que los especificadores de acceso de la clase base lo permitan.
- Puede añadir a los miembros heredados, sus propios datos y funciones miembros.
- Los miembros heredados por una clase derivada, pueden a su vez ser heredados por más clases derivadas de ella.

En particular:

EN C++

- Las clases derivadas solo pueden acceder a los miembros públicos (public) y protegidos (protected) de la clase base o clases bases, como si fueran miembros de ella misma.
- No tiene acceso a los miembros privados de la clase base.
- Los siguientes elementos de la clase base no se heredan:
    - Constructores y destructores
    - Funciones amigas
    - Funciones y datos estáticos de la clase
    - operador de asignación sobrecargado (operator =)
    - Funciones y datos estáticos de la clase

DECLARACION DE UNA CLASE DERIVADA

EN C++

La declaración de una clase derivada tiene la siguiente sintaxis:

```
class clase_derivada : <especificadores-acceso> clase_base
{ ... };
```

Los especificadores de acceso pueden ser los ya conocidos public, protected o private:

```
class clase_derivada : (public | protected | private ) clase_base
{
        declaraciones de miembros
};
```

Por ejemplo una clase derivada de la clase comida es la clase hamburguesa:

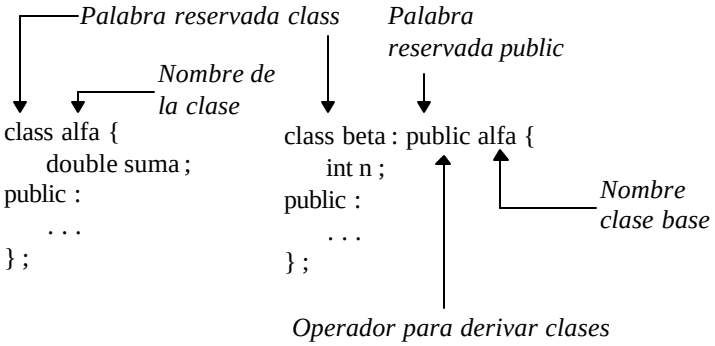
```
class hamburguesa : public comida
{ ... }
```

En este caso la etiqueta de la clase derivada es hamburguesa. La clase padre o base tiene visibilidad pública y su etiqueta (nombre) es comida.

Observe la definición de la clase beta a partir de la clase alfa:

```
class alfa {
        double suma;
public :
        // funciones públicas
};
class beta  : public alfa {
        int n;
public :
        //funciones públicas
};
```

El operador : permite definir una nueva clase utilizando una existente. La siguiente figura muestra los componentes etiquetados en cada definición de la clase.



Cada clase derivada se debe referir a una clase base declarada anteriormente, Esta clase base puede ser precompilada o puede ser definida en el mismo programa en que la clase derivada se crea. Si no se define la clase antes que la clase derivada, se debe al menos, declarar el nombre de la clase base primero como una referencia anticipada. Por ejemplo:

```
class base;
class derivada : base
{ ... }
```

ATRIBUTOS Y METODOS

Las clases derivadas retoman los atributos y los métodos de la clase base. Pero puede:

- enriquecerlos con nuevos atributos y nuevos métodos;
- redefinir los métodos.

Acceso a miembros de clases bases

Especificadores de acceso

El acceso, de la clase derivada, a los miembros de la clase base está regulado por los especificadores de acceso vistos en la unidad anterior. A continuación analizaremos el uso de estos especificadores.

EN C++

Los especificadores de acceso a las clase base, junto con los de derivación, definen como será el acceso a los miembros de la clase base desde las clases derivadas. El tipo de derivación especifica cómo recibirá la clase derivada a los miembros de la clase base. Si no se especifica un tipo de derivación, C++ supone que su tipo de herencia es privado.

Si en la derivación especificamos:

- ***public***: Los miembros públicos de la clase base son miembros públicos de la clase derivada. Los miembros protegidos de la clase base son miembros protegidos de la clase derivada.
- ***private***: Todos los miembros públicos y protegidos de la clase base son miembros privados de la clase derivada.
- ***protected***: Todos los miembros públicos y protegidos de la clase base son miembros protegidos de la clase derivada.
- Independientemente del tipo de derivación los miembros privados de la clase base son inaccesibles en la clase derivada. (Los atributos privados de la clase base incluso ocupan espacio en el objeto de la derivada, pero son inaccesibles)

Para ejemplificar utilizaremos una clase base y una derivada, derivaremos en las tres formas ya vistas y analizaremos el concepto:

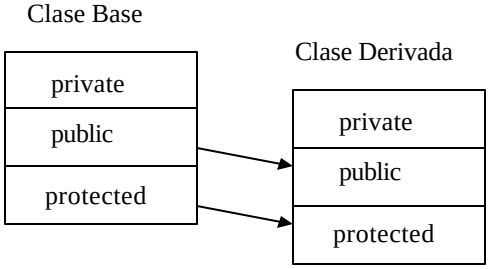
DERIVACION PUBLICA

```
class clase_base {
       int Bvalor1;
public:
       clase_base ();
       VerBValores();
protected:
       int Bvalor2;
};

       class clase_derivada : public clase_base {
              int Dvalor1;
public:
              clase_derivada ();
              VerDValores();
};
```

La clase derivada tendría los siguientes miembros :

```
private
       Dvalor1;
public:
       clase-derivada ();
       VerDValores ();
       VerBValores ();   //heredada en forma pública de clase_base
protected:
       Bvalor2;       //heredada en forma protegida de clase_base
```

Entonces:



DERIVACION PRIVADA

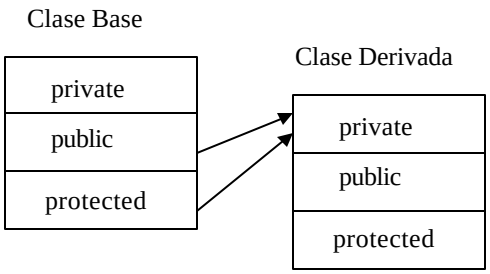
```
class clase_base {
       int Bvalor1;
```

```
public:
        clase_base ();
        VerBValores();
protected:
        int Bvalor2;
};

        class clase_derivada : private clase_base {
                int Dvalor1;
public:
                clase_derivada ();
                VerDValores();
};
```

La clase derivada tendría los siguientes miembros:

```
Private:
        Dvalor1;
        VerBValores ();  //heredada en forma privada de clase_base
        Bvalor2;         //heredada en forma privada de clase_base
public:
        clase-derivada ();
        VerDValores ();
```

Entonces:



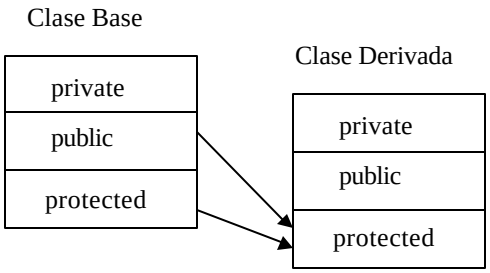
DERIVACION PROTEGIDA

```
class clase_base {
        int Bvalor1;
public:
        clase_base ();
        VerBValores();
protected:
        int Bvalor2;
};

        class clase_derivada : protected clase_base {
                int Dvalor1;
public:
                clase_derivada ();
                VerDValores();
};
```

La clase derivada tendría los siguientes miembros:

```
Private:
        Dvalor1;
public :
        clase-derivada ();
        VerDValores ();
protected:
        VerBValores (); //heredada en forma protegida de clase_base
        Bvalor2;       //heredada en forma protegida de clase_base
```

Entonces:



REDEFINICION DE METODOS

Cuando se hace heredar una clase de otra, se pueden redefinir ciertos métodos, a fin de refinarlos, o bien de modificarlos. El método lleva el mismo nombre y la misma signatura, pero sólo se aplica a los objetos de la subclase o a sus descendientes.
Así, el programa siguiente:

En C++

```
include <iostream.h>

class Madre {
public:
	void habla () {
		cout << " Soy de la clase Madre";
	}
};

class Hija : public Madre {
public:
	 void habla () {
		cout << "Soy de la clase Hija";
	}
};

class Nieta : public Hija {
public:
	void NuevoMetodoSinHistoria () {
		cout << "No es llamado";
	}
};

void main() {
	Madre m;
	m.habla();
	Hija h;
	h.habla ();
	Nieta n;
	n.habla ();		//llama al método habla de la clase hija
}
```

dará el resultado siguiente:

Los objetos de la clase Madre utilizan el método de la clase Madre, los de la clase Hija el método redefinido en la clase Hija y los de la clase Nieta utilizan el método de la clase Hija, porque este método no ha sido redefinido en la clase Nieta.

Redefinición y sobrecarga

Hemos visto que una clase que hereda de otra podía redefinir ciertos métodos. Hemos visto también que se podía dar el mismo nombre a dos métodos diferentes, a poco que tengan parámetros diferentes. ¿Se trata del mismo mecanismo?

¡En absoluto! La sobrecarga permite distinguir dos métodos de la misma clase que pueden ser llamados uno u otro sobre el mismo objeto, pero que poseen parámetros diferentes. La redefinición distingue dos métodos de dos clases de las cuales una es ancestro de la otra y que tienen ambos los mismos parámetros.

Por ejemplo:

En C++

```
clase Madre {
public:
        void metodo () { ... }
        void metodo (int param) { ... }          //el método está sobrecargado
};

class Hija : public Madre {
public:
        void metodo () {...}     //redefinición de metodo() de Madre
};
```

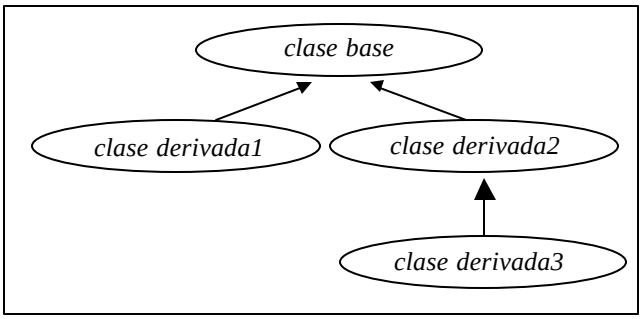
HERENCIA SIMPLE

Como ya dijimos la *herencia simple* es aquella en la que cada clase derivada hereda de una única clase y su síntaxis para declarar funciones miembro es la siguiente:

EN C++

```
class clase_base
{ ... };

class clase_derivada : (public | protected | private ) clase_base
{...};
```

Un diagrama representativo de herencia simple puede ser el siguiente:



En este ejemplo se muestran tres derivaciones simples de clases, la clase derivada1 y la clase derivada2 derivan de la clase llamada base y la clase derivada3 deriva de la clase llamada derivada2.

CONSTRUCTORES

Cuando se construye una Hija (o clase derivada), se llama al constructor de Hija. Pero sabemos que una Hija es también e implícitamente una Madre (clase base). Es necesario pues que el constructor de Madre sea invocado. Esta invocación es implícita o explícita según la presencia o no de constructores explícitos.

Puede ser implícita si el constructor de Madre no toma parámetros. En este caso, ya sea el constructor de Hija implícito o no, la llamada al constructor de Madre es automática.

Si el constructor de Madre espera parámetros, hay que pasárselos. Es necesario pues definir un constructor para Hija y, en este constructor, realizar la llamada al constructor de Madre.

Cuando una clase se deriva de otra, el orden de construcción es el siguiente:

1. Constructores de la clase base.

2. Constructores de los miembros de la clase derivada.
3. Ejecución de las instrucciones contenidas en el cuerpo del constructor de la clase derivada.

Si se llama a un constructor de una clase derivada, un constructor de su clase base debe ser siempre llamado igualmente. Si la llamada no especifica ningún constructor de la clase base, entonces se llamará al constructor por defecto de la clase base.

En resumen, cuando una clase base define un constructor, éste se debe llamar durante la creación de cada instancia de una clase derivada, para garantizar la buena inicialización de los datos miembros que la clase derivada hereda de la clase base. En este caso, la clase derivada debe definir a su vez un constructor que llama al constructor de la clase base proporcionándole los argumentos requeridos:

EN C++

La llamada al constructor de la clase base se debe realizar después de la cabecera de la función y antes de las llaves de comienzo de la misma o inmediatamente después.

```
constructor de la                          llamada al constructor
clase derivada                          de la clase base
derivada :: derivada (tipo1 x, tipo2 y)  : base (x,y)
{
  ...
}
```

Aquí, derivada es la clase derivada, base es la clase base, y x,y son variables utilizadas por el constructor de la clase base. Si por ejemplo, al constructor derivada se pasan los argumentos 10 y 20, este mecanismo pasa 10 y 20 al constructor base. Este formato aparece en la definición de la función, pero no en el prototipo.

Ejemplo 1, usando constructor por defecto:

```
class base {
public:
        base () {cout <<"\n Base creada"; }
};

class derivada : public base {
public:
        derivada () { cout <<"\n Se ha creado la clase derivada"; }
};
void main ()
{
        derivada x;
        ...
}
```

La ejecución produce un objeto x del tipo derivada y la salida es :

Base creada
Se ha creado la clase derivada

Ejemplo 2, usando constructor con parámetros, llamado explícitamente:

```
class base {
        int x, y;
public:
        base (int px, int py) { x = px;  y = py; }
        ...
};
class derivada : public base {
        int z;
public:
        derivada (int pz, int px, int py);
        ...
};

derivada :: derivada (int pz, int px, int py) : base (px, py)
{
        z = pz;
```

}

DESTRUCTORES

Por definición, un destructor de clases no toma parámetros. Al contrario que los constructores, una función destructor de una clase derivada se ejecuta antes que el destructor de la clase base. La razón es que la destrucción de la clase base implica la destrucción de la clase derivada, el destructor derivado debe ser ejecutado antes que se destruya.

EN C++

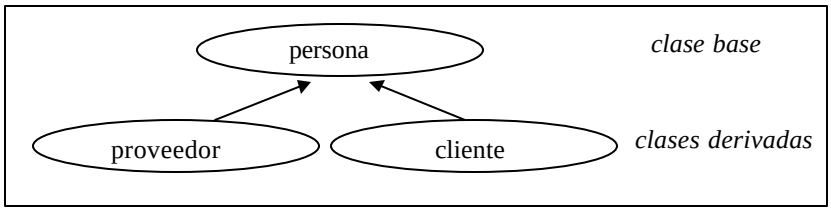
Ejemplo

```
#include <iostream.h>
class base {
public:
        base () {cout << "\n Base creada"; }
        ~base () {cout << "\n Destruida Base"; }
};
class derivada : public base {
public:
        derivada () { cout <<"\n Se ha creado la clase derivada"; }
        ~derivada () { cout <<"\n Se ha destruido derivada"; }
};
void main ()
{
derivada x;
cout << "\n --------------";
}
```

En resumen, si un constructor se define en la clase base y en la clase derivada, C++ llama (ejecuta) primero el constructor de la clase base y después el de la clase derivada. Al contrario de los constructores, el destructor de una clase derivada se ejecuta antes que el destructor de la clase base.

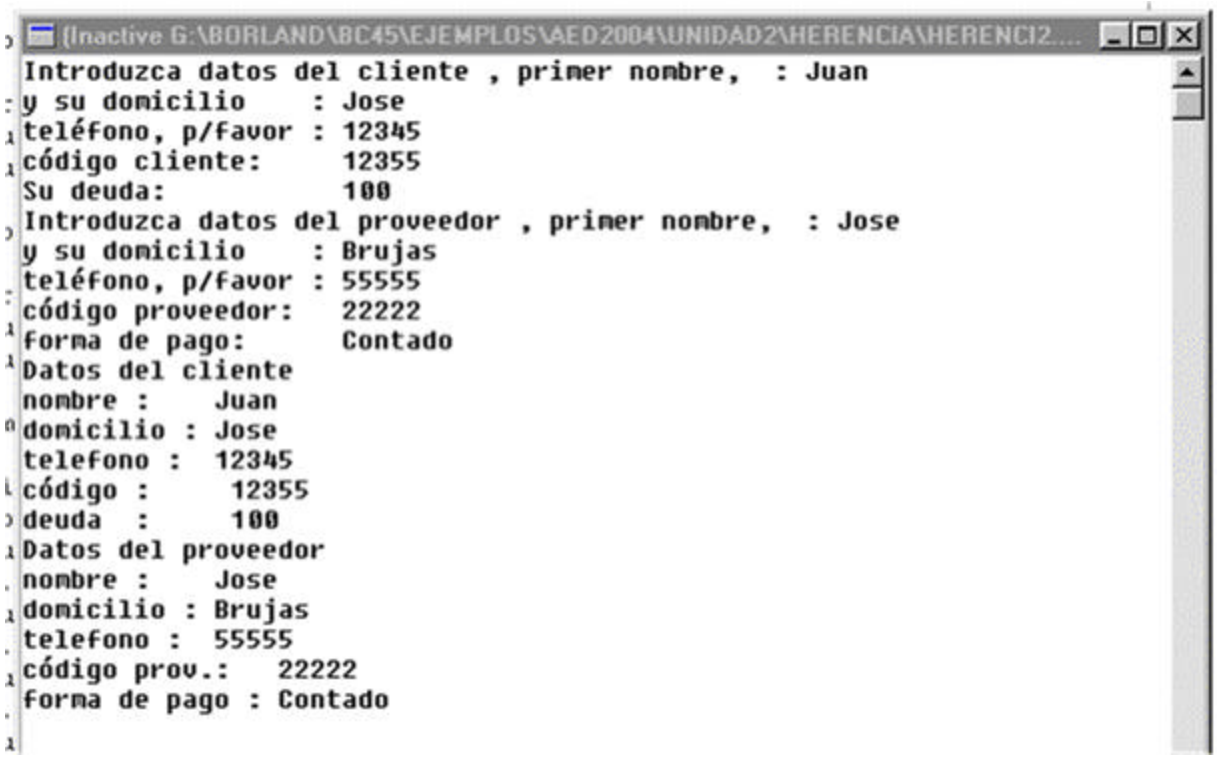
UN EJEMPLO DE JERARQUÍA DE CLASES: HERENCIA SIMPLE

Se posee una clase persona, que manipula el nombre, domicilio y teléfono de la persona. A partir de esta clase se derivan otras dos clases: clientes y proveedor, estas contienen información adicional tal como código y estado de cuenta o código y forma de pago, etc.




```
#include <iostream.h>
      class persona {           //clase base persona
            char nombre[30];
            char domicilio[30];
            char telefono[30];
      public:
            void leerdatos();
            void verdatos();
      };
      void persona :: leerdatos()
      {
            cout << ", primer nombre,  : ";
            cin >> nombre;
            cout << "y su domicilio    : ";
            cin >> domicilio;
            cout << "teléfono, p/favor : ";
            cin >> telefono;
      }
      void persona :: verdatos()
      {
            cout << "nombre :    " << nombre << endl;
            cout << "domicilio : " << domicilio << endl;
            cout << "telefono :  " << telefono << endl;
      }
      class cliente : public persona //clase cliente derivada de persona
      {
            int codigo;         // Código del cliente.
            int deuda;          // estado de la deuda del cliente.
      public:
            void leerdatos();
            void verdatos();
      };
      void cliente :: leerdatos()
      {
            persona::leerdatos();     //llama a leerdatos() de la clase persona
            cout << "código cliente:     ";  cin >> codigo;
            cout << "Su deuda:           ";  cin >> deuda;
      }
      void cliente :: verdatos()
      {
            persona::verdatos();
            cout << "código :     " << codigo << endl;
            cout << "deuda  :     " << deuda << endl;
      }
      class proveedor: public persona  //clase proveedor derivada de persona
      {
            int codigo;               //código del proveedor.
            char forma_pago[30];      //contado, a 30 días, 60 días, etc.
      public:
            void leerdatos();
            void verdatos();
      };
      void proveedor :: leerdatos()
      {
            persona::leerdatos();
            cout << "código proveedor:   ";        cin >> codigo;
```

```
        cout << "forma de pago:      ";        cin >> forma_pago;
}
void proveedor :: verdatos()
{
        persona::verdatos();
        cout << "código prov.:   " << codigo << endl;
        cout << "forma de pago : " << forma_pago <<endl;
}
void main()
{
        cliente c1;
        proveedor p1;
        cout <<  "Introduzca datos del cliente ";
        c1.leerdatos();
        cout <<  "Introduzca datos del proveedor ";
        p1.leerdatos();
        cout <<  "Datos del cliente"<<endl;
        c1.verdatos();
        cout <<  "Datos del proveedor "<<endl;
        p1.verdatos();
}
```



POLIMORFISMO

Una de las características de más importancia de la programación orientada a objetos es la capacidad de que diferentes objetos responden a órdenes similares de modo diferente. Esta característica se denomina polimorfismo y los objetos que lo soportan se llaman objetos polimórficos.

El polimorfismo consiste en llamar a un método según el tipo estático de una variable, basándose en el núcleo para llamar a la versión correcta del método, realizándose esto en tiempo de ejecución.

El polimorfismo se realiza por uno de estos métodos:

- Con sobrecarga de funciones (C++ y Java)
- Con funciones virtuales (C++)

Para mostrar como se implementa el polimorfismo usaremos un programa que contenga dos objetos que responda a las mismas ordenes de modo diferente:

En C++

Se tienen dos objetos, un monitor y una impresora que se programan para imprimir un mensaje.

```
class monitor {
public:
        void imprimir (char *mensaje)
        { cout << mensaje; }
};

class impresora {
public:
        void imprimir (char *mensaje)
        { ofstream imp ("PRN");   //define un flujo
           if (! imp)
           {  cerr <, "no se puede acceder a la impresora";
               exit(1);
           }
           imp << mensaje;  // imprime el mensaje
        }
};

void main ()
{
        monitor M;
        impresora I;
        M.imprimir("Hola");
        I.imprimir("Hola");
}
```

En este caso, el polimorfismo se crea utilizando funciones redefinidas.

En el polimorfismo utilizado en este ejemplo, el compilador determina que función imprimir() ha de llamar en tiempo de compilación ya que conoce el objeto implicado. La función se vincula al objeto en tiempo de compilación. Este tipo de ligadura se conoce como *ligadura estática*, *temprana* o *previa*.

Otro tipo de ligadura de función denominada *ligadura dinámica*, *tardía* o *postergada* es la opuesta de la ligadura estática. En este caso el compilador no indica en tiempo de compilación que objeto se está utilizando cuando se llama a una función. Esta forma de enlace tardío es muy potente. Esta forma de ligadura tardía en C++, se realiza mediante herencia y funciones virtuales.

FUNCIONES VIRTUALES

Una característica importante de las clases derivadas es la función miembro virtual. Con frecuencia, cuando se derivan un grupo de clases de una dase base, se necesita que ciertas funciones miembro que son análogas en su propósito final, operen de modo diferente en cada clase. Por ejemplo, la función visualizar() que muestra los valores de la clase base necesitar visualizar valores diferentes en el caso de ser llamada desde una clase derivada. Posiblemente, alguna de las clases derivadas puede utilizar la función visualizar() como definida en la clase base, mientras que otras pueden requerir elementos adicionales o incluso formatos totalmente nuevos. El mecanismo de función virtual ajusta estas clases de cambios.

Se pueden dar nombres diferentes a cada una de las funciones miembros visualizar de cada clase diferente. Aunque esta solución es fácilmente realizable no es la ideal. La necesidad de crear muchos nombres únicos para funciones miembro rompe un posible diseño unificado y hace el código resultante más difícil de leer y utilizar.

Una solución mejor a este problema es sobrecargar el nombre de la función miembro. Se puede, por ejemplo, crear la función miembro visualizar() tanto en la clase base como en la clase derivada. Esta solución es adecuada para conjuntos simples de clases, pero puede hacerse compleja en el caso de numerosas clases y jerarquías. La mejor solución, en todos los casos, es identificar aquellas funciones miembro de la clase base que se puedan necesitar utilizar en clases derivadas y a continuación declararlas virtuales.

En C++ , por defecto las funciones tienen ligaduras estática; si se desea ligadura dinámica se debe definir explícitamente:

virtual char * visualizar() ;

Las funciones miembro *virtuales* son similares, a las funciones miembro ordinarias con una diferencia importante: las llamadas a funciones virtuales se enlazan dinámicamente en tiempo de ejecución (en lugar de en tiempo de compilación) a la función que llama.

La función debe ser declarada como *virtual* en la primera clase que está presente (siguiendo el orden de derivación).

La palabra clave *virtual* permite a una función ser definida en una clase base y en su clase derivada bajo el mismo nombre.

La funciones miembros y los destructores pueden ser virtuales, los constructores no pueden ser virtuales porque no se pueden heredar, esto significa que no se puede utilizar un constructor de la clase base para inicializar directamente un objeto de la clase derivada.

Al utilizar funciones virtuales se pueden utilizar punteros a una clase base para referenciarse a objetos de una clase derivada.

Función virtual pura: es una función virtual que no está definida y no tiene cuerpo, se declaran en la clase base y deben estar definidas en cualquier clase derivada de tal clase base. Para declarar una función virtual como pura se necesita la siguiente declaración:

virtual tipo nombre_función (lista de parámetros) =0;

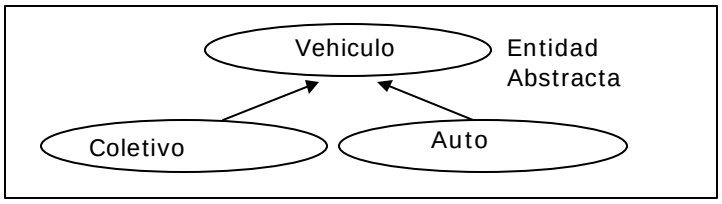
CLASES ABSTRACTAS

Son clases que sólo se pueden utilizar como clases bases; no se pueden utilizar para declarar objetos, se diseñan solamente para ser heredadas. Tales clases se utilizan para definir conceptos abstractos tales como figura y mueble, sin proporcionar detalles específicos.

EN C++

Una clase es una *clase abstracta si tiene al menos una función virtual pura.*

Por ejemplo, el siguiente gráfico nos muestra una generalización de clases



Así, la clase Vehiculo puede definirse como sigue:

```
class Vehiculo {
        int numpasajeros;
        int Peso;
public:
        void Arrancar () = 0;                          //virtual pura
        void Correr ()= 0;                                     //virtual pura
        void TransportarPasajero (int num) {
                numpasajeros = num;
                Arrancar ();
                Correr ();
        }
};
```

POLIMORFISMO

Para conseguir polimorfismo en C++, se tienen que seguir los siguientes pasos:

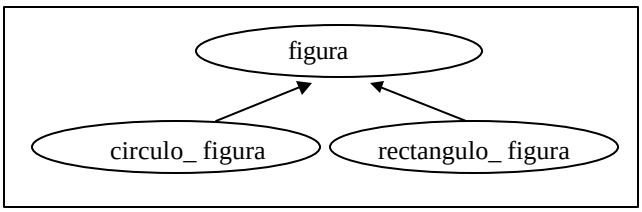
1. Crear una jerarquía de clases con las operaciones importantes definidas por funciones miembro que son declaradas virtuales en la clase base. Si las clases base son tales que no se puede proporcionar implementaciones de estas funciones, se pueden declarar virtuales puras. La función dibujar de la clase figura sería:
   virtual void dibujar() = 0;

2. Proporcionar implementaciones concretas de las clases virtuales en las clases derivadas. Cada clase derivada tiene su propia versión de las funciones. Por ejemplo, la implementación de la función dibujar variará de una figura a otra.
3. Manipular instancias de estas clases a través de una referencia o un puntero, esto se debe a la ligadura dinámica y las llamadas a funciones utilizando el mecanismo de función virtual.

APLICACIONES DE POLIMORFISMO

1. Clase figura

Esta clase sirve para definir figuras geométricas. Se declara abstracta ya que nunca se crearán instancias de ella. Se utiliza para encapsular datos y funciones comunes a todas las clases derivadas, y utilizar herencia y polimorfismo.




```
#include <iostream.h>
#include <math.h>

class figura {                          //clase abstracta no se crean
public:                                 //instancias de esta clase.
        virtual void area()     {;}
        virtual void dibujar()          {;}
};

class circulo : public figura {                 //clase circulo
        double x, y;
        double radio;
public:
        circulo(double x, double y, double radio);
        void area();
        void dibujar();
};

class rectangulo : public figura {      //clase rectángulo
        double x1, y1;
        double x2, y2;
public:
        rectangulo(double x1, double y1, double x2, double y2);
        void area();
        void dibujar();
};

circulo :: circulo(double xc, double yc, double r)
{
        x=xc;
        y=yc;
        radio=r;
}
void circulo :: area()
{
        double area = 3.141592 * radio * radio;
        cout << "Soy círculo de radio   "<<radio<<" y área "<<area <<endl;
}
void circulo :: dibujar()
{
```

```
        cout << "circulo de radio " << radio;
        cout << ", con centro en " << x << ", " << y << endl;
}

rectangulo :: rectangulo(double xvi, double yvi,
 double xvd, double yvd)
{
        x1=xvi;                 y1=yvi;
        x2=xvd;                 y2=yvd;
}
void rectangulo :: area()
{
        double area =  fabs ((x1-x2) * (y1-y2) );
        cout << "Rectángulo, absisa "<<fabs(x1-x2)<<" , ordenada "<<fabs(y1-y2)<<" y área
"<<area <<endl;
}
void rectangulo :: dibujar()
{
        cout << "rectángulo con vértices " << x1 << ", " << y1<< ", ";
        cout << x2 << ", " <<y2 << endl;
}
void main()
{       int i;
        figura *figuras[5];             //  Array de punteros a la clase base.
        figuras[0]= new circulo(120.,120.,60.);         // Un puntero a una clase base siempre
        figuras[1]= new rectangulo(80.,40.,150.,60.);// puede recibir punteros a objetos de
        figuras[2]= new circulo(200.,120.,60.);         // clases derivadas. En tiempo de ejecu-
        figuras[3]= new circulo(120.,200.,30.);         // ción C detecta la clase a la que perte-
        figuras[4]= new rectangulo(40.,40.,80.,60.);  // nece el puntero y ejecuta la función
        cout << "Calculando áreas polimórficamente"<<endl; // miembro de esa clase.
        for (i = 0; i < 5;  i++)
                figuras[i]->area(); // Misma expresión p/ calcular área de círculo, rectángulo.
        cout <<endl<<"Dibujando figuras, idem"<<endl;
        for (i = 0; i < 5;  i++)
                figuras[i] -> dibujar();          // Idem, dibujando polimorficamente ...
        cout <<"\n Procedo a destruir las figuras"<<endl;
        for (i = 0; i < 5;  i++)
                delete figuras[i];
        cout <<"\n...y hemos terminado !!!"<<endl;

}
```