Principios de Orientación a Objetos

Roni Guillermo Apaza Aceituno

Universidad Nacional de San Agustin

rapazaac@unsa.edu.pe

September 12, 2018

Programación Procedimental

- Procesos.
- Pseudocódigo.
- Algoritmo.

Paradigma

- Es una teoria o conjunto de Teorias.
- Aceptada sin cuestionar.
- Sirve de modelo para resolver problemas.

Paradigma de programación

Paradigma de programación

- Modelo Imperativo.
- Modelo no estructurado.
- Modelo procedimental.

Modelo Imperativo

- El primero en aparecer.
- Naciõ con el lenguaje máquina.
- Consiste en una secuencias de instrucciones.

Modelo Imperativo

- Instrucciones del tipo MOV.
- Tiene un modo contrario como es el modelo declarativo.
- Incluye varios modelos de programación.

Modelo no estructurado

- Esta forma de programar continuo con lenguajes como BASIC.
- Surgio el problema de usar datos definidos al principio.
- BASIC lo resolvio con GOTO.

Modelo no estructurado

- Los programas estaban en un solo bloque.
- La necesidad de saltos era la única opción.

Paradigma de programación

- Modelo procedimental.
- Modelo estructurado.
- Modelo modular.

Modelo procedimental

- Aparecen subrutinas.
- Estas pueden ser funciones o procedimientos.
- Esto se da con el lenguaje PASCAL.

Modelo estructurado

- Los programas crecen lo que crea la necesidad de legibilidad y claridad.
- Es necesario tener una serie de principios o reglas.
- Estas reglaso son:

Modelo estructurado

- Estructura jerarquica de arriba hacia abajo. (TOP - DOWN)
- Diseño en modulos.
- Las entradas y salidas estan bien definidos.

Modelo estructurado

- Solo se puede entrar atravez de un punto.
- Solo se puede salir atravez de un punto.
- Constructores de programación estructurada.

Modelo estructurado

- Constructores del tipo secuencia.
- Constructores del tipo selección.
- Constructores del tipo iteración.

Modelo estructurado

- Constructores del tipo llamada.
- Constructores del tipo procedimientos.

Modelo estructurado

- Permite reutilizar código.
- El programa es mas comprensible.

Modelo estructurado

- En esta forma de programación se afronta el problema desde arriba.
- El analisis posterior limita sucesivamente las partes.
- El analisis posterior limita claramente las partes.

Modelo estructurado

- Esta forma de pensamiento genera elementos mas sencillos.
- Mas sencillos de entender.
- Mas sencillos de programar.

Paradigma de programación

- Modelo modular.
- Modelo tipo abstracto de datos.

Modelo modular

- La división de los programas fue necesaria.
- Esta división se hizo en módulos.
- Estos módulos posibilito el trabajo en grupo.

Modelo modular

- Sin la noción de instanciación.
- Ni la exportación de tipos.

Modelo tipo abstracto de datos

- Exportan una definición de tipo.
- Disponibilizan operaciones para manipular instancias de tipo.(interfaz)
- Esta interfaz es el único medio de acceso a a la estructura de un tipo abstracto de datos.

Modelo tipo abstracto de datos

- Es posible crear múltiples instancias de un tipo.
- Esta caracteristica la diferencia de los otros modelos.
- Crear nuevos tipos y usarlos como nosotros querramos.

Paradigma de programación

- Modelo declarativo.
- Modelo funcional.
- Modelo lógico.

Modelo declarativo

- Usa bloque de construcción.
- Como funciones.
- Como la recursión.

Modelo declarativo

- Especifica mas la solución.
- No solo es calculo.
- Puede ser funcional o lógico.

Modelo funcional

- Basada en funciones como la base de este modelo.
- También hace el uso de la composición de funciones.
- No da importancia al cambio de estados.

Modelo funcional

- Evita declarar y cambiar datos.
- Evita los efectos secundarios de otros lenguajes de programación.
- Un ejemplo: Haskell

Modelo lógico

- Basado en reglas lógicas.
- Usa un motor de inferencias lógicas.
- Responde preguntas.
- Un ejemplo: Prolog

Sobre los paradigmas de programación

- Todas las caracteristicas deberian ser claramente y elegantemente integradas dentro del lenguaje.
- Deberia ser posible usar caracteristicas en combinación con las soluciones alcanzadas que serian en otro caso requeridas caracteristicas separadas extras.
- Deberia haber pocas caracteristicas falsas o de proposito especial.

Sobre los paradigmas de programación

- Una caracteristica deberia ser implementada de forma que no este encima de un programa si no es requerido.
- Un usuario solamente sabe de un subconjunto del lenguaje explicito que usa para escribir un programa.

Programación procedimental

- Decide que procedimiento quieres y usa el mejor algoritmo que tu encuentres.
- Se requiere un lenguaje que pase los argumentos a una función.
- Se requiere un lenguaje que reciba el resultado de dicha función.

Programación procedimental

- Se centra en como pasar los argumentos.
- Procedimientos, rutinas y macros.
- Fortran es el mejor ejemplo, Algol60, Algol68, C y Pascal lo siguieron.

Ocultando datos (Data Hiding)

- Los procedimientos organizan los datos, por mucho tiempo.
- Por lo tanto esto incrementa el tamaño del programa.
- Asi un conjunto de procedimientos relacionados con sus datos se llama **módulo**.

Ocultando datos (Data Hiding)

- Decide que modulo quieres y parte el programa y el dato oculto en modulos.
- Donde no hay agrupación de procedimientos con datos relacionados la programación procedimiental es suficiente.
- Como hacemos un nombre oculto para el resto del programa.

Ocultando datos (Data Hiding)

- La unica forma es hacer de este elemento un elemento local en el procedimiento.
- Pero hay procedimientos que trabajan con datos globales.
- Esto ocurre en Pascal.

Ocultando datos (Data Hiding)

- En C se mejoro esto controlando los nombre que podian ser vistos dentro de un programa.
- A menos que se declare STATIC.
- En C se alcanza un grado de modularidad.

Ocultando datos (Data Hiding)

- Para el paradigma procedimental no es aceptado este procedimiento.
- Esto es considerado de bajo nivel.
- La programación modular exige esta declaración de modulos.

Ocultando datos (Data Hiding)

- Un control sobre el alcance de los nombres, import y export.
- Una forma de inicializar los modulos.
- Un estilo aceptado de uso.

Ocultando datos (Data Hiding)

- C habilita la descomposición de un programa en modulos.
- Modula-2 soporta esta técnica.

Abstrayendo datos

- Programando con modulos permite la centralización de todos los datos.
- Todos los datos estarán bajo el control de un modulo tipo administrador.
- El ejemplo del Stack.

Abstrayendo datos

```
class stack_id;  // stack_id is a type
                 // no details about stacks or stack_ids are known here

stack_id create_stack(int size); // make a stack and return its identifie
destroy_stack(stack_id);         // call when stack is no longer needed

void push(stack_id, char);
char pop(stack_id);
```

Abstrayendo datos

- Esto es mas ordenado.
- Estos tipos son distintos de los definidos para un lenguaje.
- Para definir un tipo hay que crearlo como variable.

Abstrayendo datos

- No existe el identificador de objetos.
- Un tipo sole es una variable para el compilador.
- Peor es si esa variable no obedece las reglas de alcance o las reglas de pasar argumentos.

Abstrayendo datos

- Un tipo creado a traves de un mecanismo de modulo es muy diferente a un tipo predefinido.
- Puede tener el mismo soporte para los tipos predefinidos.

Abstrayendo datos

```
void f()
{
        stack_id s1;
        stack_id s2;

        s1 = create_stack(200);
```

Abstrayendo datos

```
// Oops: forgot to create s2

char c1 = pop(s1,push(s1,'a'));
if (c1 != 'c') error("impossible");
```

Abstrayendo datos

```
    char c2 = pop(s2,push(s2,'a'));
    if (c2 != 'c') error("impossible");

    destroy(s2);
    // Oops: forgot to destroy s1
}
```

Abstrayendo datos

- El concepto de modulo soporta el paradigma de ocultar datos.
- El concepto de ocultando datos esta habilitado para esta forma de programaciõn.
- Pero no lo soporta (o sea lo promete).

Abstrayendo datos

- Ada, Clu y C++ atacan este problema.
- Permitiendo que el usuario defina sus propios tipos.
- Esto es llamado algunas veces como **tipo abstracto de dato**.

Abstrayendo datos

- Decide tu tipo.
- Llena con un conjunto de operaciones para cada tipo.

Abstrayendo datos

- No hay necesidad de mas de un objeto de un tipo.
- Los datos que ocultan el estilo de programación son suficientes.
- Definiendo mas tipos númericos se puede comprender esto mejor.

Abstrayendo datos

```
class complex {
    double re, im;
public:
    complex(double r, double i) { re=r; im=i; }
    complex(double r) { re=r; im=0; }   // float->complex conversion

    friend complex operator+(complex, complex);
    friend complex operator-(complex, complex);   // binary minus
    friend complex operator-(complex);            // unary minus
    friend complex operator*(complex, complex);
    friend complex operator/(complex, complex);
    // ...
};
```

Abstrayendo datos

- La representación es privada.
- Las variables re, in son accibles por sus funciones.
- Funciones especificadas en la declaración de la clase.

Abstrayendo datos

```
complex operator+(complex a1, complex a2)
{
    return complex(a1.re+a2.re,a1.im+a2.im);
}
```

Abstrayendo datos

- Las funciones son definidas parecidas como las anteriores.
- Lo usan así.

Abstrayendo datos

```
complex a = 2.3;
complex b = 1/a;
complex c = a+b*complex(1,2.3);
//  ...
c = -(a/b)+2;
```

Abstrayendo datos

- Los modulos son mejor expresados como tipos definidos por el usuario.
- Esta representación es deseable incluso para tipos definidos.
- El programador puede declarar un tipo.

Abstrayendo datos

- Y solamente un solo objeto de ese tipo.
- Un lenguaje puede proveer un concepto de módulo distinto de el concepto de clase.

Problemas con la abstracción de datos

- El tipo de dato abstracto define algo parecido a una caja negra.
- Para adaptarlo a nuevos usos hay que variar su definición.
- Esto lo convierte en un ambiente muy inflexible.

Problemas con la abstracción de datos

- Definamos una clase shape.
- Definamos una clase point.
- Definamos una clase color.

Problemas con la abstracción de datos

```
class point{ /* ... */ };
class color{ /* ... */ };
```

Problemas con la abstracción de datos

```
enum kind { circle, triangle, square };

class shape {
    point center;
    color col;
    kind k;
    // representation of shape
public:
    point where()          { return center; }
    void move(point to)  { center = to; draw(); }
    void draw();
    void rotate(int);
    // more operations
};
```

Problemas con la abstracción de datos

- Debe permitirse la función draw().
- Debe permitirse la función rotate().
- Para determinar el tipo de shape.

Problemas con la abstracción de datos

```
void shape::draw()
 {
        switch (k) {
        case circle:
                // draw a circle
                break;
        case triangle:
                // draw a triangle
                break;
```

Problemas con la abstracción de datos

- La función draw() debe soportar todos lo shape.
- Esta función creceria con cada tipo de shape.
- No es posible agregar un nuevo tipo de shape a menos que se acceda al código fuente.

Problemas con la abstracción de datos

- Esto para cada operación.
- La habilidad es necesaria.
- Por lo tanto es posible generar una serie de errores

Problemas con la abstracción de datos

- La elección de nuevos shape se ve entrampada.
- Por ejemplo en el tamańo.
- Esto es por que el tipo shape es general.

- Las propiedades especificas de un shape.
- No es lo mismo que las propiedades generales.
- La programación orientada a objetes toma ventaja de estas caracteristicas.

- Un lenguaje que permite constructores que permitan la distinción entre su expresión
- Un lenguaje que permite constructores que permiten la distinción entre su uso .
- Este lenguaje es un lenguajes orientado a objetos

- Implementamos una solución con Simula
- Primero definimos una clase con las propiedades generales de una forma.

```
class shape {
        point center;
        color col;
        // ...
public:
        point where() { return center; }
        void move(point to) { center = to; draw(); }
        virtual void draw();
        virtual void rotate(int);
        // ...
};
```

- Las funciones pueden definir una llamada a la interface.
- Pero no pueden definirse en una implementación a menos que sea en una especificica forma.
- Esto quizas hace la situación un poco **virtual**.

- En Simula y en C++ puede ser redefinido después.
- Esto puede ser hecho en una clase derivada.
- Se puede escribir una función manipulando formas.

```
void rotate_all(shape* v, int size, int angle)
// rotate all members of vector "v" of size "size" "angle" degrees
{
    for (int i = 0; i < size; i++) v[i].rotate(angle);
}
```

- Primero definimos que es una forma.
- Especificamos sus propiedades particulares.
- Se incluyen sus funciones virtuales.

```
class circle : public shape {
    int radius;
public:
    void draw() { /* ... */ };
    void rotate(int) {}    // yes, the null function
};
```

- Lo virtual va de la mano con el manejo de las funciones.
- Funciones que pueden cambiar de forma.
- Esto depende del programador.

- La clase **circulo** deriva de la clase **forma**
- Por lo tanto se dice que la clase **forma** es la clase base de la clase **circulo**
- Existe mas terminologia.

- La clase **circulo** deriva de la clase **forma**
- Por lo tanto se dice que la clase **forma** es la clase base de la clase **circulo**
- Existe mas terminologia.

- La clase **circulo** es llamada de subclase.
- La clase **forma** es llamad de superclase.

Programación Orientada a Objetos

- Este paradigma dice lo siguiente:
- Decide que clases quieres
- Provee un conjunto de operaciones para cada clase.

- Este paradigma dice lo siguiente:
- Hacer explicita la similitud usando herencia.

- Si la similitud no es suficiente.
- La abstracción de datos seria suficiente.

- La cantidad de similitud puede ser explotada atravez de los tipos.
- Explotando la herencia y las funciones virtuales.
- Esto prueba la utilidad de la programación orientada a objetos para una área definida.

Programación Orientada a Objetos

- En las areas graficas hay un enorme alcance para la programación orientada a objetos.
- En la aritmética clásica y los derivados parece que solo esta presente la abstracción de datos.
- Aun cuando la matemática se puede beneficiar de la herencia (anillos, espacio de vectores y otros).

- Encontrar similitudes entre los tipos no es una tarea trivial.
- Estas similitudes son afectadas por como fue diseñado el sistema.
- El diseño debe buscar lo comun.

- Tanto para diseñar clases.
- Tanto para diseñar bloques de construcción para otros tipos.
- Lo ideal es encontrar una clase en comun durante la examinación.

Soporte para la abstracción de datos

- Facilidades para definir un conjunto de operaciones para un tipo.
- Restringir el acceso de objetos del tipo a ese conjunto de operaciones.

Soporte para la abstracción de datos

- El programador encontrará que es necesario un lenguaje de programación refinado.
- Para la definición de nuevos tipos.
- Para el uso de esos tipos.

Soporte para la abstracción de datos

- La sobrecarga de operadores es un buen ejemplo de esto.
- Primero hay que inicializar

Inicialización y limpiar

- En el caso de que la representación de un tipo este oculta.
- Existen formas de iniciar las variables de ese tipo.
- Estos mecanismos deben ser proveidos al usuario.

Inicialización y limpiar

```
class vector {
    int  sz;
    int* v;
public:
    void init(int size);    // call init to initialize sz and v
                            // before the first use of a vector
    // ...
};

vector v;
// don't use v here
v.init(10);
// use v here
```

Inicialización y limpiar

- Esto no es elegante.
- También puede ser propenso a errores.
- Lo mejor es dejar al diseñador proveer un tipo.

Inicialización y limpiar

- El tipo sera el usado para inicializar esta clase.
- En dicha función la alocación e inicialización de esta variable puede ser hecha en una simple operación.
- Esto en vez de dos operaciones.

Inicialización y limpiar

- Una vez creada la función de inicialización esta recibe algunas veces el nombre de constructor.
- Existen casos donde la construcción de objetos no es trivial, es necesario una operación para finalizar su uso.
- Esta función limpiadora es llamada de destructor.

Inicialización y limpiar

```
class vector {
     int  sz;                                  // number of elements
     int* v;                                   // pointer to integers
public:
     vector(int);                              // constructor
     ~vector();                                // destructor
     int& operator[](int index);               // subscript operator
};
```

Inicialización y limpiar

```
vector::vector(int s)
{
    if (s<=0) error("bad vector size");
    sz = s;
    v = new int[s];    // allocate an array of "s" integers
}
```

Inicialización y limpiar

```
vector::~vector()
{
    delete v;          // deallocate the memory pointed to by v
}
```

Inicialización y limpiar

- C++ no tiene un recolector de basura.
- En compensación tiene su propio administrador de alojamiento.
- El cual no requiere la intervención del usuario.

Inicialización y limpiar

- Esto es un comun uso para los mecanismos del constructor y destructor.
- Estos usos no estan relacionados con el administrador de alojamiento.

Asignación e inicialización

- Construcción y destrucción es lo necesario para muchos tipos, pero no para todos.
- Necesario para controlar la copia de todas las operaciones.

Asignación e inicialización

```
vector v1(100);
vector v2 = v1;   // make a new vector v2 initialized to v1
v1 = v2;          // assign v2 to v1
```

Asignación e inicialización

- Debe ser posible definir el significado de la asignación de v1 y la inicialización de v2.
- Tambien deberia ser posible la prohibición de operaciones de copia.
- Ambas operaciones son deseables.

Asignación e inicialización

```
class vector {
    int* v;
    int  sz;
public:
    // ...
    void operator=(vector&);   // assignment
    vector(vector&);           // initialization
};
```

Asignación e inicialización

- Las operaciones definidas por el usuario deben usarse para la inicialización y la asignación.
- Para nuestro ejemplo deberia verse asi.

Asignación e inicialización

```
vector::operator=(vector& a) // check size and copy elements
{
    if (sz != a.sz) error("bad vector size for =");
    for (int i = 0; i<sz; i++) v[i] = a.v[i];
}
```

Asignación e inicialización

- La operación de asignación esta basada en un valor antiguo
- La operación de inicialización debe ser diferente

Asignación e inicialización

```
vector::vector(vector& a)     // initialize a vector from another vecto
{
    sz = a.sz;                // same size
    v = new int[sz];          // allocate element array
    for (int i = 0; i<sz; i++) v[i] = a.v[i];    // copy elements
}
```

Asignación e inicialización

- En C++, un constructor X(X&) define también la inicialización
- También usados para manejar argumentos pasados **por valor**
- También para los valores retornados por una función.

Asignación e inicialización

```
class X {
    void operator=(X&);     // only members of X can
    X(X&);                  // copy an X
    // ...
public:
    // ...
};
```

Asignación e inicialización

- En el caso anterior podemos prohibir la asignación de una clase X.
- Haciendo la declaración de la asignación privada.
- Ada no soporta constructores, destructores, sobrecarga de asignaciones o control definido por el usuario de los argumentos pasados o retornados por una función.

Asignación e inicialización

- Esto retorna a las **tecnicas de ocultado de datos**.
- Usar un administrador de tipos a la vez que el propio tipo.

Tipos parametrizados

- En el caso de querer un arreglo que admita un tipo desconocido de elementos.
- El tipo vector debe ser expresado para tomar cualquier tipo de argumento.

Tipos parametrizados

```
class vector<class T> {      // vector of elements of type T
    T* v;
    int sz;
public:
    vector(int s)
    {
        if (s <= 0) error("bad vector size");
        v = new T[sz = s];     // allocate an array of "s" "T"
    }
    T& operator[](int i);
    int size() { return sz; }
    // ...
};
```

Tipos parametrizados

```
vector<int> v1(100);        // v1 is a vector of 100 integers
vector<complex> v2(200);    // v2 is a vector of 200 complex numbers

v2[i] = complex(v1[x],v1[y]);
```

Tipos parametrizados

- Ada, Clu y ML soportan parametrización de tipos.
- C++ no soporta parametrización de tipos.

Tipos parametrizados

- Un tipo parametrizado debe depender por lo menos de un aspecto del tipo parametrizado
- Por ejemplo en el caso del vector debe estar definida la operación de asignación con el "="
- No es buena idea tomar una identificación parcial, puede provocar errores de compilación.

Tipos parametrizados

- Esta tecnica permite la definición de tipos donde la dependencia de los atributos es manejada a nivel de una operación individual del tipo.
- Por ejemplo si definimos a la clase vector como un clase de ordenación.

Tipos parametrizados

- Utilizaremos simbolos como $<$,$==$ y $=$.
- Para definir operaciones de ordenamiento.

Tipos parametrizados

- Por cada tipo parametrizado instanciado se crea un tipo independiente.
- Por ejemplo se crea un tipo $vector < char >$ o un $vector < complex >$.

Tipos parametrizados

- Lo ideal es tener una función independiente del tipo parametrizado.
- Por ejemplo una función tamaño.
- Un lenguaje deberia soportar varios tipos parametrizados.

Tipos parametrizados

- Un lenguaje deberia soportar varios tipos parametrizados.
- La herencia tiene una ventaja sobre los tipos parametrizados.

Manejo de Excepciones

- Cuando los programas crecen, incluso con el uso de librerias.
- Es necesario manejar las excepciones.
- Excepciones es una forma de decir errores.

Manejo de Excepciones

- Ada, Algol68 y Clu soporta el manejo de excepciones.
- C++ no soporta esto.
- C++ usa punteros a funciones.

Manejo de Excepciones

- Objetos de excepción.
- Estados de error.
- Y algunas librerias.

Manejo de Excepciones

- En el ejemplo del vector que se debe hacer en caso de que el valor ingresado pase el rango establecido.
- El diseñador de esta clase deberia estar preparado para un ambiente por defecto para esto.

Manejo de Excepciones

```
class vector {
    ...
    except vector_range {
        // define an exception called vector_range
        // and specify default code for handling it
            error("global: vector range error");
            exit(99);
    }
}
```

Manejo de Excepciones

- Dentro de *vector* :: *operator*[]() podemos invocar el codigo de manejo de excepciones.
- En vez de llamar a una función de error.

Manejo de Excepciones

```
int& vector::operator[](int i)
{
    if (0<i || sz<=i) raise vector_range;
    return v[i];
}
```

Manejo de Excepciones

- En teoria esto ejecutará una pila de llamadas.
- Esta acción hará que se encuentre al final un manejador de excepciones.
- por ejemplo $vector_range$ una vez encontrado sera ejecutado.

Manejo de Excepciones

```
void f() {
    vector v(10);
    try {                   // errors here are handled by the local
            // exception handler defined below
        // ...
        int i = g();        // g might cause a range error using some vector
        v[i] = 7;           // potential range error
    }
    except {
    vector::vector_range:
        error("f(): vector range error");
        return;
    }                       // errors here are handled by the global
            // exception handler defined in vector

    int i = g();            // g might cause a range error using some vector
    v[i] = 7;               // potential range error
}
```

Manejo de Excepciones

- La anterior es una definición del bloque para un manejador de excepciones.
- No es el único camino para definir excepciones.
- Lo mostrado se asemeja a ejemplos encontrados en Clu y Modula 2+.

Manejo de Excepciones

- Este código no se ejecutará a menos que aparezca la excepción.
- *setjmp*() y *longjmp*() son funciones para ser implementadas en C que portan estas excepciones.
- Estas funciones de manejo de excepciones no se pueden falsificar en C.

Manejo de Excepciones

- Al producirse una excepción, se hace en tiempo de ejecución.
- Por lo tanto el controlador se ejecutará después de un camino de desenredo.
- En C++ implica invocar destructores definidos en los lugares adecuados.
- *longjmp*() no lo hace y el usuario tampoco.

Coerciones

- Hacer que un tipo se comporte como otro tipo.
- Las coeerciones de números de coma flotante y números complejos definidos por el usuario.
- Estos son usados para constructores del tipo doble, son útiles en C++

Coerciones

- El programador puede confiar que el compilador lo agregará.
- Sin ambiguedades.

Coerciones

```
complex a = complex(1);
complex b = 1;                    // implicit: 1 -> complex(1
a = b+complex(2);
a = b+2;                          // implicit: 2 -> complex(2
```

Coerciones

- Introducidas en C++ para manejar la aritmética mezclada.
- Muchos tipos definidos por el usuario son usados para cálculos.
- Desde matrices hasta direcciones de datos.

Coerciones

- Deberian tener una asignación natural a otros tipos.
- El uso de estas coerciones aun de un solo modo es útil para el punto de vista organizador del programa.

Coerciones

```
complex a = 2;
complex b = a+2;   // interpreted as operator+(a,complex(2))
b = 2+a;           // interpreted as operator+(complex(2),a)
```

Coerciones

- Se puede interpretas + una vez definida su función.
- Esta función maneja los operandos por sistema de tipos.
- La clase complex es escrita sin la necesidad de modificar el concepto de entero.

Coerciones

- En un sistema orientado a objetos puro contrastaria con esto.
- Las operaciones serian asi.

Coerciones

```
a+2;        // a.operator+(2)
2+a;        // 2.operator+(a)
```

Coerciones

- Es necesario modifcar la clase entero para hacer $2 + a$ legal.
- No es recomendable modificar el codigo de un sistema.
- Especialmente cuando se añaden nuevas caracteristicas.

Coerciones

- La programación orientada a objetos agrega caracteristicas superiores.
- Estas caracteristicas superiores pueden ser agregadas al sistema sin modificar código.
- Especificamente para este ejemplo la abstracción de datos ofrece una mejor solución.

Iterador

- Un buen lenguaje debe definir bien las estructuras de control.
- En nuestro caso un bucle.
- Este bucle debe poder soportar la sobrecarga de operadores.

Implementación de características

- La abstracción de datos puede ser proveida por el compilador.
- La parametrización de tipos puede ser soportada con un enlazador con conocimientos en semantica.
- El manejo de excepciones necesita un soporte para un entorno corriendo al instante.

Soporte para programación orientada a objetos

- El soporte básico esta enlazado con un clase de mecanismo para la herencia.
- Que las funciones niembro dependan del tipo de un objeto actual.
- Para los casos que el tipo es desconocido para el tiempo de compilación.

Llamada de mecanismos

- Llamada a una función normal.
- Llamada a una función virtual.
- Una invcación a un metodo

Checkeo de tipos

- El chequeo de tipos estaticos permite solo las operaciones especificadas en la declaración de la clase.
- Este chequeo es diferente para tipos dinamicos e invocación de metodos.
- El usuario en este ultimo caso puede probar funciones no especificadas.

Herencia

- Un lenguaje de programación no puede considerarse orientado a objetos sin la herencia.
- Pero puede existir el caos sin una manera sistematica de tratar la asociaciõn de métodos y estructuras de datos.
- Para expresar al usuario en una forma estándar como se comportará ese objeto.

Herencia

- Para hacer eso es necesario tener un mecanismo de herencia.
- También debemos hacer este mecanismo compatible con las funciones y métodos virtuales.
- Aunque dificil en forma hace poderoso al lenguaje.

Herencia

- Esta forma lo haria mucho mejor que un lenguaje de abstracción simple.
- Simula y C++ estan estructuados con jerarquias de clase sin funciones virtuales.
- La capacidad de expresar elementos comunes es muy deseable (factoring)

Herencia

- En nuestro ejemplo del shape(forma) seria dificil sin poder hacer una representación común.
- Sin funciones virtuales el usuario necesitará recurrir a "campos para tipos".
- Por lo tanto los problemas de falta de modularidad permanecerian.

Herencia

- Esto afirma que la derivación de clases (subclases) es muy importante.
- Puede ampliarse mas allá de la programación orientada a objetos.
- Haciendo de las clases derivadas especializaciones de la clase base.

Herencia

- Derivar clases es una poderosa herramienta para crear nuevos tipos.
- Restando o agregando caracteristicas.
- La relación de clase resultante con la clase base no puede entenderse con la especialización, talvez con factoring.

Herencia

- Esta derivación es una herramienta poderosa.
- No se sabe como será usada a futuro.
- Pero de momento no se puede decir que sea indebido.

Herencia múltiple

- Cuando una clase hereda atributos de otra clase, muestra su utilidad además de solo ser la misma clase.
- Ahora esta afirmación sustentaria la afirmación de la utilidad de una clase que herede de dos clases.
- Esta forma de herencia es llamada de Herencia múltiple

Encapsulación

- Si necesitaramos proteger a un miembro de la clase.
- En un lenguaje de programación orientado a objetos, la respuesta sería todos las operaciones definidas para ese objeto.
- Todas las funciones miembro.

Encapsulación

- Es imposible definir todas las funciones que puedan acceder a una función miembro.
- Se puede definir por herencia una función miembro atravez de una clase derivada.
- Esto evita accidentes por que no se crea una clase por accidente.

Encapsulación

- Para un lenguaje orientado a la abstracción de datos la respuesta es diferente.
- En una clase comun se enumera las funciones que necesitan acceso en la declaración de clase.
- No es necesario ser funciones miembro.

Encapsulación

- En C++ una función no miembro que tiene acceso a a miembros privados es llamada de "friend".
- Esta acción es importante especificar como "friend" a mas de una clase.
- Por lo tanto es necesario y ventajoso tener una lista de "friend" y miembros para entender al programa.

Encapsulación

```
class B {
                        // class members are default private
    int i1;
    void f1();
protected:
    int i2;
    void f2();
public:
    int i3;
    void f3();

friend void g(B*);      // any function can be designated as a frien
};
```

Encapsulación

- Miembros privados no son generalmente accesibles.
- Miembros protegidos igual no son accesibles.

Encapsulación

```
void h(B* p)
{
    p->f1();        // error: B::f1 is private
    p->f2();        // error: B::f2 is protected
    p->f3();        // fine:  B::f1 is public
}
```

Encapsulación

- Miembros protegidos pueden ser accesibles para los miembros de una clase derivada.
- Miembros privados no son accesibles de una clase derivada.

Encapsulación

```
class D : public B {
public:
    void g()
    {
    f1();       // error: B::f1 is private
    f2();       // fine:  B::f2 is protected, but D is derived from B
    f3();       // fine:  B::f1 is public
    }
};
```

Encapsulación

- Las funciones "Friend" tienen acceso a los miembros privados y protegidos.
- Esto ocurre como funciones miembro.

Encapsulación

```
void g(B* p)
{
    p->f1();      // fine: B::f1 is private, but g() is a friend of B
    p->f2();      // fine: B::f2 is protected, but g() is a friend of
    p->f3();      // fine: B::f1 is public
}
```

Encapsulación

- Esto representa un problema aun con la encapsulación implementada.
- Se incrementa con el tamaño del programa.
- Se incrementa con el número de usuarios.
- Se incrementa con la dispersión de los usuarios.

Implementación de características

- El soporte para la programación orientada a objetos es proporcionada por el sistema de tiempo en ejecución (run-times).
- También es soportada por el entorno de programación.
- La programación orientada a objetos se basa en mejoras de los lenguajes de programación basados en la abstracción de datos.

Implementación de características

- El lenguaje de programación orientado a objetos borra la distinción entre el lenguaje de programación y su entorno.
- Los tipos definidos por el usuario pueden ser mas poderosos y para proposito general.
- Es necesario mas bibliotecas, mas depuradores y medidas de rendimiento y monitoreo.

Implementación de características

- Es necesario mejorar el sistema de tiempo de ejecución.
- Lo ideal seria integrar todo en un entorno de programación unificado.
- Por ejemplo Smalltalk.

Preguntas