PROGRAMACIÓN EN C/C++ ORIENTADO A ENTORNOS INFORMÁTICOS Y SISTEMAS EMBEBIDOS

Prof. Israel Pavelek

C++

- C++ fue diseñado en 1979 por Bjarme Stroustrup. Su intención fue la de extender el lenguaje de programación C incorporando la manipulación de objetos.
  Hoy en día C++ ya no se considera una extensión de C sino una lenguaje totalmente diferente

C++

- C++ fue diseñado en 1979 por Bjarme Stroustrup. Su intención fue la de extender el lenguaje de programación C incorporando la manipulación de objetos.
  Hoy en día C++ ya no se considera una extensión de C sino una lenguaje totalmente diferente

Primer programa

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

int main(){
    cout << "Hola mundo" << endl;
    return 0;
}
```

Primer programa

- Podríamos copiar el primer programa en C y funcionaria 100% pero en este formao vemos cosas nuevas.
  Vemos el uso de otra librería para entrada y salida de datos iostream En C++ todos es un flujo, pantalla, teclado, archivos, impresora etc.
- Con la línea using namespace std; nos sirve para poder utilizar cout como elemento de salida sin tener que aclarar todo el tiempo que cout pertenece a std

Si no colocaramos el namespace

```
#include <iostream>

int main (){
    std::cout << "Hola mundo" <<std::endl;
    return 0;

}
```

Cout

- Mostrar texto por pantalla en C++ es muy simple. Para imprimir una salida de texto en C++ se hace uso de la instrucción cout, junto con <<.
- Es importante tener en cuenta que la instrucción cout siempre va acompañada de << para controlar el flujo de datos que sale.

Intercalando valores

```
#include "iostream"
using namespace std;

int main(){
    int numero = 2;
    cout << "El valor de numero es " << numero << "\n";
    return 0;
}
```

Ingresando valores por teclado cin

```
#include "iostream"

using namespace std;

int main(){
    int numero;
    cout << "Ingrese un valor por favor\n";
    cin >> numero;
    //Se concatenan y muestran los valores por pantalla con c
out<<
    cout << "El valor de numero es " << numero << "\n";
    return 0;
}
```

Endl y \n

- Ambos pueden ser utilizados de la siguiente forma y hacen lo mismo

```
cout << "Ingrese un valor por favor "<< endl;
cout << "Ingrese un valor por favor\n";
```

Dec,hex,oct

- Cuando queremos cambiar como se ve una variable estos manipuladores cambian la forma en que vemos la salida o entrada del flujo

Ingresando valores por teclado cin

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

int main(){

    int var=20;

    cout << "Valor de var: 0x" << hex << var << endl;
    return 0;
}
```

También podemos agregar la biblioteca iomanip.h

- Y utilizar setbase(int n) para usar cualquier base, es como dec,oct y hex

Ingresando valores por teclado cin

```
#include <iostream>
#include <iomanip>

using namespace std;

int main(){
    int var=20;
    cout << "Valor de var: 0x" << setbase(16) <<  var << endl;
    return 0;
}
```

Otros manipuladores de iomanip.h

- Setw(int ancho) es usada para limitar el ancho de los campos de entrada
- Setfill(int caracter) establece cual será el relleno en caso de haberse establecido un ancho de campo.
- Setprecision(int num_dec) establece la cantidad máxima de decimales para los números flotantes.

flags

- La clase ios posee una serie de propiedades (en forma de flags). Los flujos de entrada y salida son objetos de esa clase, y es posible setear y resetar esos flags a través de las funciones manipuladoras:
  - Setioflags(long flag)
  - Resetioflags(long flag)

Flags

Flag	Descripción
Skipws	Saltea espacios en blanco en operaciones de entrada
Left,right	Justifica la salida a la izquierda o derecha en un campo
Internal	Rellena el campo después del signo o el indicador de base
Dec,oct,hex	Activa la conversión a decimal, octal o hexadecimal
Showbase	Visualiza el indicador de base numérica
Showpoint	Visualiza el punto decimal en flotantes
Uppercase	Visualiza valores hexadecimales en mayúscula
Showpos	Precede a los enteros positivos del signo '+'
Scientific	Establece notación científica en los flotantes
Fixed	Establece notación d e punto fijo para los flotantes
Unitbuf	Vacía los buffers después de cada escritura
stdio	Ídem sobre stdout y stderr

Ejemplo

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

int main (){
    float mat[3][4];
    for (int i=0;i<3;i++){
        for(int j=0;j<4;j++){
            cout << "\n Ingrese el número "<< i << " " << j;
            cin >> mat[i][j];
        }
    }
    cout << setfill ('~');
    cout << setprecision(2);
    cout << showpoint;
    for(int i=0;i<3;i++){
        cout << endl << endl << "\t";
        for(int j=0;j<4;j++)
            cout << " " << setw (12)<< mat[i][j];
    }
    cout << "\n\n\n\n";
    getchar();
    return 0;
}
```

Ejemplo

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

int main () {
  int n = 20;
  cout << hex << showbase << n << '\n';
  cout << hex << noshowbase << n << '\n';
  return 0;
}
```

Tipos de datos en C++

- C++ posee los tipos de datos fundamentales que C agregando valores bool que pueden tomar el valor 0 o 1, también están ya definidas las macros de false y true que valen 0 y 1 respectivamente

- Una particularidad de C++ es que podemos instanciar una variable donde queramos, a diferencia de C que solo podíamos hacerlo luego de la apertura de llaves.

clases

- En primera instancia una clase es la extensión de la estructura de C. En las estructuras definíamos formas de datos nuevas, bajo un nombre definíamos un montón de parámetros (campos).
- Las clases poseen esto mismo pero incorporan la posibilidad de agregar a esa estructura funciones propias.

Clases vs objetos

- Vamos a ver que vamos a definir clases, y en el momento que esas clases sean instanciadas, es decir que existan los espacios de memoria asignados para el trabajo de esa clase, tendremos un objeto.

miembros, métodos

- Tantos las variables propias de una clase, como las funciones son conocidas como miembros de una clase.
- Las clases entonces pueden tener variables y métodos. Los métodos son las funciones propias de la clase.
- Cada miembro de la clase puede ser público o privado, de forma que sea accesible por cualquier ente fuera de la clase o solo por la clase misma.

Ejemplo clase

```
class pareja {
   private:
      // Datos miembro de la clase "pareja"
      int a, b;
   public:
      // Metodos / Funciones miembro de la clase "pa
reja"
      void Lee(int &a2, int &b2);
      void Guarda(int a2, int b2) {
         a = a2;
         b = b2;
      }
};
void pareja::Lee(int &a2, int &b2) {
   a2 = a;
   b2 = b;
}
```

```
int main(){
   pareja par1;
   int x, y;
   par1.Guarda(12, 32);
   par1.Lee(x, y);
   cout << "Valor de par1.a: " << x << endl;
   cout << "Valor de par1.b: " << y << endl;
   return 0;
}
```

A analizar…

- Las palabras public y private definen que miembros de la clases son públicos y claes privados, si no se coloca nada por defecto todos los miembros son privados.
- En la clase podemos colocar el prototipo del método y su contenido como en guarda. O solamente colocar el prototipo y definir el método fuera. Notar la forma de mencionar que es un método y de que clase `clase::metodo`

Encapsulamiento

- Es importante notar que el acceso a las variables en este caso se hace mediante los métodos y no mediante el acceso directo, esto y la protección de datos como privados define el encapsulamiento a modo de encontrar un mejor formato para el acceso a la información.

Constructores

- En las clases si definimos un método con el mismo nombre de la clase, este método será invocado al instanciar la clase

Constructores

```
class pareja {
   private:
      // Datos miembro de la clase "pareja"
     int a, b;
   public:
      // Metodos / Funciones miembro de la clase "pareja"
     pareja(void);
     void Lee(int &a2, int &b2);
     void Guarda(int a2, int b2);
};

pareja::pareja (void){
    a=3;
    b=5;
}
void pareja::Lee(int &a2, int &b2) {
    a2 = a;
    b2 = b;
}
void pareja::Guarda(int a2, int &b2){
    a = a2;
    b = b2;
}
```

```
int main(){
    pareja par1;
    int x, y;
    par1.Lee(x, y);
    cout << "Valor de par1.a: " << x << endl;
    cout << "Valor de par1.b: " << y << endl;
    return 0;
}
```

Constructores

- Es importante notar que en el constructor no hay parámetros de retorno, puede si recibir un parametro o no.

Constructores

```
class pareja {
   private:
      // Datos miembro de la clase "pareja"
     int a, b;
   public:
      // Metodos / Funciones miembro de la clase "pareja"
     pareja(void);
     pareja(int a2, int b2);
     void Lee(int &a2, int &b2);
     void Guarda(int a2, int b2);
};

pareja::pareja (void){
    a=3;
    b=5;
}
pareja::pareja (int a2, int b2){
    a=a2;
    b=b2;
}
```

```
void pareja::Lee(int &a2, int &b2) {
    a2 = a;
    b2 = b;
}
void pareja::Guarda(int a2, int b2){
    a = a2;
    b = b2;
}

int main(){
    pareja par1(8,3);
    int x, y;
    par1.Lee(x, y);
    cout << "Valor de par1.a: " << x << endl;
    cout << "Valor de par1.b: " << y << endl;
    return 0;
}
```

Constructores

- En el ejemplo anterior agregamos un constructor nuevo (ahora tenemos dos), se diferencian por los parámetros que tiene cada uno. Si al instanciar la clase le pasamos los valores llama al constructor con parámetros, sino ejecuta el otro constructor.

Destructor

- Si definimos igual que un método igual que un constructor pero le agregamos adelante el símbolo ~ el método se convierte en destructor. Este método será invocado cuando se libere la memoria asignada a la clase

Constructores otra forma que funciona igual

```
class pareja {
   private:
      // Datos miembro de la clase "pareja"
    int a, b;
   public:
     // Metodos / Funciones miembro de la clase "par
eja"
    pareja() : a(3),b(5){}
    pareja(int a2, int b2) : a(a2),b(b2){}
    void Lee(int &a2, int &b2);
    void Guarda(int a2, int b2);
};
void pareja::Lee(int &a2, int &b2) {
    a2 = a;
    b2 = b;
}
void pareja::Guarda(int a2, int b2){
    a = a2;
    b = b2;
}
```

```
int main(){
    pareja par1(8,3);
    int x, y;
    par1.Lee(x, y);
    cout << "Valor de par1.a: " << x << endl;
    cout << "Valor de par1.b: " << y << endl;
    return 0;
}
```

Sobrecarga de operadores

- r=a+b;
- r=sumar(a,b);
- De acuerdo a la naturaleza de a y de b poriamos esperar que la suma fuera de diferentes formas(por ejemplo si a y b fueran vectores (física)en dichos caso podemos modificar el operador suma o definir el método suma para hacerlo. Esto se llama sobrecarga de operadores o de funciones.

Sobrecarga de funciones

```
#include <iostream>
using namespace std;
typedef int* complex;
complex cuadrado (complex);
int cuadrado (int);
float cuadrado (float);

int main (){

    int n=2;
    float f=3.2;
    int complejo[2]={1,3}; //parte real y parte  ima
ginaria
    cuadrado(n);
    cuadrado(f);
    cuadrado(complejo);
    //getchar();
    return 0;
}
```

```
complex cuadrado (complex v){
    int r[2];
    r[0]=v[0]*v[0]-v[1]*v[1];
    r[1]=2*v[0]*v[1];
    char c='+';
    int imag=r[1];
    if(imag<0){
        c='-';
        imag=-imag;
    }
    cout << "\n\n El complejo cuadrado es:";
    cout <<r[0]<<c<<imag<< "i \n\n";
    return v;
}
int cuadrado (int a){
    a=a*a;
    cout<<"\n\n El entero cuadrado es " <<a;
    return a;
}
float cuadrado (float f){
    f=f*f;
    cout<<"\n\n El flotante cuadrado es " <<f;
    return f;
}
```

Sobrecarga de funciones

- Vemos en el ejemplo anterior que C++ tiene la capacidad de acuerdo a los parámetros que le pasemos a una función de identificar cual utilizar.

Sobrecarga de operadores

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

class vector{
    public:
        int x;
        int y;
        int z;
    vector(int x1,int y1,int z1) : x(x1),y(y1),z(z1)
{}
    vector() : x(0),y(0),z(0){}
};
vector operator + (const vector v, const vector w){
    vector r;
    r.x=v.x+w.x;
    r.y=v.y+w.y;
    r.z=v.z+w.z;
    return r;
};
```

```
int main (){
    vector v1(2,4,6),v2(3,2,5),vr;
    vr=v1+v2;
    cout<< "X" << "\t"<< "Y" << "\t"<< "Z\n";
    cout<< vr.x << "\t"<< vr.y << "\t"<< vr.z;
    return 0;

}
```

La palabra reservada operator

- La palabra reservada operator seguida del operador que queremos sobrecargar es la clave para poder asignar como se van a comportar los objetos cuando opere con ellos.

Operadores New y Delete

- Los operadores new y delete están relacionados con la asignación de memoria dinámica (a medida que la necesitamos).
- Si poseemos una clase alumno, no es necesario instanciar 100 clases “por las dudas”, sino que podemos instanciar una clase a medida que es necesario y eliminar una objeto cuando no es necesario.

Operador new

- El operador new gestiona un bloque de memoria dinámica del tamaño del tipo de dato que se le indica y retorna la dirección de inicio de ese bloque con formato de puntero al tipo especificado.
- En caso de fracasar la asignación, el valor retornado es NULL

Operador new

- Sintaxis:
  - tipo * puntero =new tipo;
  - tipo * puntero =new tipo [tamaño];

Ejemplo operador new

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

class alfa{
    public:
        int a;
        float b;
};


int main (){
    alfa* p=new alfa;
    return 0;

}
```

En este caso instanciamos una clase (creamos un objeto) p del tipo alfa

Operador delete

- El operador delete libera la memoria gestionada a través de new.

- Sintaxis:
  - Delete <dirección de bloque>;

Ejemplo operador new

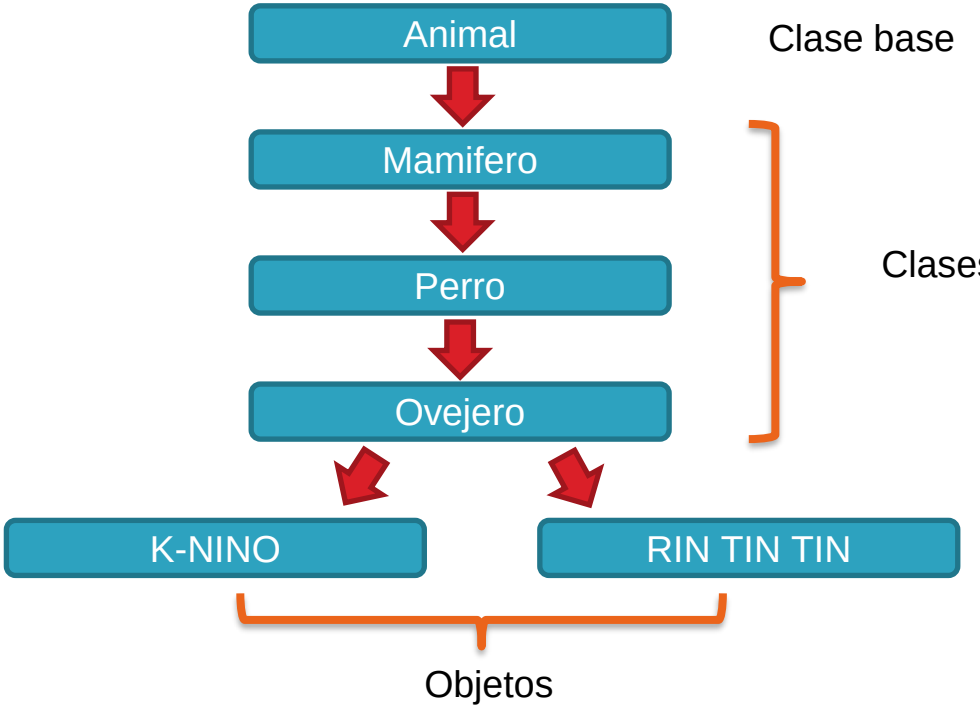
```
#include <iostream>
using namespace std;

class alfa{
    public:
        int a;
        float b;
};


int main (){
    alfa* p=new alfa;
    delete p;
    return 0;

}
```

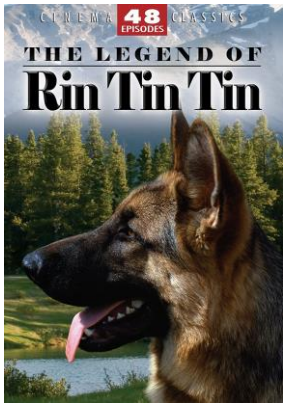
Herencia

- La herencia es una propiedad de la programación orientada a objetos mediante la cual una clase (llamada clase derivada) adopta los atributos de una o más clases (llamadas clases base)

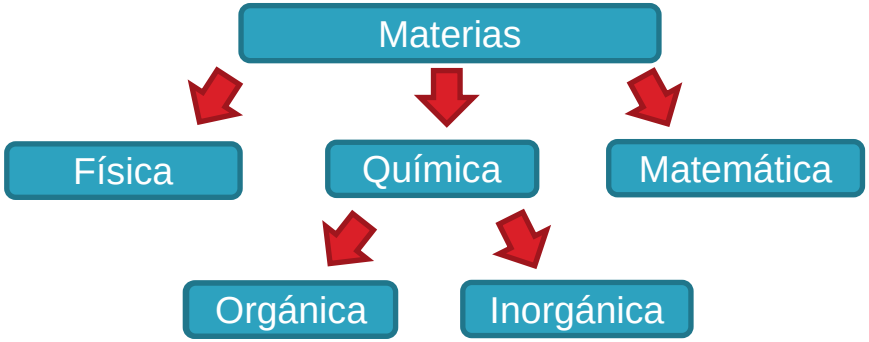
Herencia

JAMES BELUSHI
K-NINO

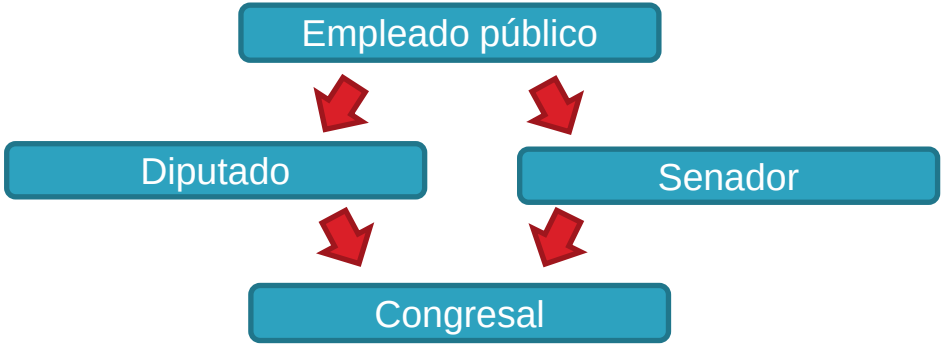

Animal
Mamifero
Perro
Ovejero
K-NINO
RIN TIN TIN
Clase base
Clases derivadas
Objetos


48
EPISODES
THE LEGEND OF
Rin Tin Tin

Herencia Simple y Herencia Multiple



Herencia Simple



Herencia Multiple

Clase abstracta

- Es una clase que solo sirve para ser heredada y nunca será instanciada.

Ejemplo herencia simple

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

class animal {
    public:
        void come();
        void duerme();
        void respira();

};
void animal::come(void){}
void animal::duerme(void){}
void animal::respira(void){}

class mamifero: public animal{
    public:
        int tipo_sangre_caliente;
        void amamantar();

};
void mamifero::amamantar(void){}
```

```cpp
class perro : public mamifero{
    public:
        int tipo_pelo;
        void ladrido();
        void trucos();

};
void perro::ladrido(void){}
void perro::trucos(void){}



class ovejero :public perro{
    public:
        bool pelo_largo;
        int tatuje_POA;
        int displasia;
};

int main (){
    ovejero Rintin= new ovejero;
    (*Rintin).ladrido();
    return 0;
}
```

Puntero a estructura u objeto

```
int main (){
    ovejero * Rintin= new ovejero;
    (*Rintin).ladrido();
    return 0;
}
```

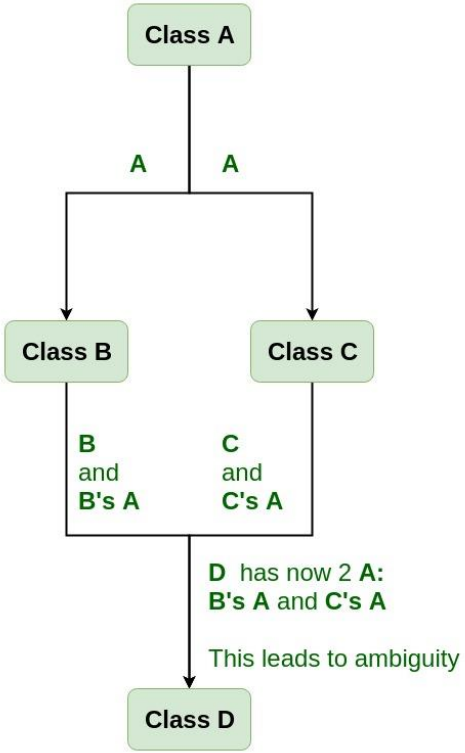
```
int main (){
    ovejero * Rintin= new ovejero;
    Rintin->ladrido();
    return 0;
}
```

Ejemplo herencia multiple

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

class animal {
    public:
        void come();
        void duerme();
        void respira();

};
void animal::come(void){}
void animal::duerme(void){}
void animal::respira(void){}

class carnivoro{
    public:
        int numero_de_dientes;
};
```

```
class perro : public animal,public carnivoro{
    public:
        int tipo_pelo;
        void ladrido();
        void trucos();

};
void perro::ladrido(void){}
void perro::trucos(void){}

int main (){
    perro * Rintin= new perro;
    Rintin->ladrido();
    return 0;
}
```

Problemas en herencia múltiple



¿Que pasa cuando desde la clase C se quiere acceder a una propiedad definida en la clase A?
¿La invoca desde la clase B o desde la clase C?

Probemos el código

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

class A {
public:
    void show(){
        cout << "Hello form A \n";
    }
};

class B : public A {
};

class C : public A {
};

class D : public B, public C {
};

int main(){
    D object;
    object.show();
    return 0;
}
```

¿Funciona?

Solución funciones y propiedades virtuales

```cpp
#include <iostream>
using namespace std;
class A {
public:
    int a;
    A():a(10){}
};
class B : public virtual A {
};
class C : public virtual A {
};
class D : public B, public C {
};
int main(){
    D objeto;
    cout << "a = " << objeto.a << endl;
    return 0;
}
```

```cpp
#include <iostream>
using namespace std;
class A {
public:
    void show(){
        cout << "Hola desde A \n";
    }
};
class B : public virtual A {
};
class C : public virtual A {
};
class D : public B, public C {
};
int main(){
    D objeto;
    objeto.show();
}
```