RESUMEN LENGUAJE C++

E/S de datos

<iostream> archivo de cabecera

Salida ------> cout << " .....";
Entrada ----> cin >> i; da a i un valor leído desde el teclado

```
cout << "su número es " << i << "\n";
```

(Suponiendo que i=100, mostrará: su número es 100)

Se pueden usar cualquiera de las funciones de E/S de C (scanf( ), printf( ), ...) pero se considera que estos siguen más la filosofía de C++.

Comentarios

En C++ los comentarios se definen de dos formas. Los comentarios del tipo C funcionan de la misma manera tanto en C++ como en C. Sin embargo se pueden definir también comentarios de una sola línea usando //. Cuando comienza un comentario usando //, se ignora lo que sigue hasta el final de la línea.

Miembros public y private

Los miembros de una clase pueden ser datos o funciones, que pueden definirse como públicos (accedidos desde cualquier parte del programa), protegidos o privados (sólo pueden ser accedidos por las funciones propias de la clase donde se definen) mediante las palabras public, protected y private.

Declaración de clases

Una clase puede ser definida de tres formas:

1ª) Mediante la palabra **struct**: por defecto todos los miembros son públicos.

www.profmatiasgarcia.com.ar

```
struct  cuadrado
{
 double CalcularArea ( );
 void Leerdatos (double Lado1, double Lado2);
private :
 double Lado1;
 double Lado2;
};
```

2ª) Mediante la palabra **union**: por defecto los miembros son públicos y los datos comparten espacio de memoria.

```
union  Nombre_Persona
{
  void MuestraNombre ( );
  void MuestraApellido ( );
private :
  char Nombre_Completo [30];
  char Nombre_y_Apellido [2] [15];
};
```

3ª) Mediante la palabra **class**: los miembros son privados por defecto. Es la forma usual de declarar clases.

```
class vehiculo
{
   int Numero_Ruedas;
   int Numero_Ocupantes;
public :
   void MostrarNumeroOcupantes ( );
}
```

Para crear un objeto en C++ primero debe definirse su forma general usando la palabra reservada **class**. Una clase es sintácticamente similar a una estructura.

Ejemplo: esta clase define un tipo llamado *cola,* que se usa para crear un objeto cola:

```
// esto crea la clase cola
class cola {
     int  c[100];
     int  ppio, fin;
```

www.profmatiasgarcia.com.ar

```
public :
    void ini ( );
    void meter (int i);
    int sacar ( );
}
```

Se puede crear un objeto de este tipo usando el nombre de la clase. Por ejemplo, creamos un objeto llamado *intcola* del tipo *cola:*

```
cola   intcola;
```

También se pueden crear variables cuando se está definiendo una clase, poniendo los nombres de las variables después de la llave de cierre, igual que en una estructura.

Forma general de una declaración de clase:

```
class  nombre_clase {
     datos y funciones privados;
public :
     datos y funciones publicos;
} lista de nombres de objetos;
```

Las funciones deben ser declaradas en el interior de la estructura class, mientras que su código se define normalmente fuera de ella. A la hora de definir el código de una función miembro de una clase se le debe indicar al compilador a qué clase pertenece la función, esto se indica precediendo al nombre de la función el nombre de la clase y un par de signos de dos puntos "::". Es necesario ya que en C++ está permitido que clases distintas declaren funciones distintas pero con el mismo nombre.

Ejemplo:

```
void cola::meter (int i)
{
 if (ppio=100) {
   cout << "la cola está llena";
   return;
  }
 ppio++;
 c[ppio]=i;
}
```

A :: se le denomina operador de resolución de ámbito.

www.profmatiasgarcia.com.ar

Para llamar a una función de una clase desde una parte del programa que no sea parte de la propia clase, se debe utilizar el nombre del objeto y el operador punto "."

Ejemplo:

```
cola  a,b;
a.ini ( );
```

Constructores y destructores

Es normal que una parte de un objeto necesite una inicialización antes de poder usarse. Un *constructor* no es más que una función miembro que aglutina un conjunto de instrucciones que permiten inicializar los objetos de una clase. El nombre de esa función debe coincidir con el nombre de la clase.

Ejemplo:

```
//esto crea la clase cola
class cola {
    int c[100];
    int ppio,fin;
public :
    cola (void)       //constructor
    void meter (int i);
    int sacar (void);
 };
```

El constructor **cola ( )** no tiene especificado tipo de dato devuelto. En C++ las funciones constructoras no pueden devolver valores.
La función cola se codifica como:

```
//Función constructora
cola::cola (void)
{
 ppio=fin=0;
 cout << "cola inicializada\n";
}
```

En muchos casos un objeto debe realizar alguna acción o acciones cuando se destruye. Hay muchas razones por las que se puede necesitar un destructor. Por ejemplo, un objeto puede tener que liberar memoria que previamente se le ha asignado o reservado.
Un destructor recibe el mismo nombre que la clase pero precedido por el carácter "~".

www.profmatiasgarcia.com.ar

El siguiente ejemplo es de la clase cola y sus funciones constructora y destructora (tenga en cuenta que la clase cola no necesita un destructor).

```
class cola {
    int c[100];
    int ppio,fin;
public :
    cola (void);          //constructor
    ~cola (void);         //destructor
    void meter (int i);
    int sacar (void);
}

//Función constructora
cola::cola (void)
{
 ppio=fin=0;
 cout <<"cola inicializada\n";
}

//Función destructora
cola::~cola (void)
{
 cout <<"cola destruida\n";
}
```

Funciones *inline*

Una función de línea es una función que se «expande» en el punto donde se llama en vez de ser realmente llamada. Esto lo haremos cuando dicha función sea muy corta y se use mucho. Nos ahorraremos el tiempo que se pierde haciendo el pase de parámetros de una llamada a función.

Hay dos formas de declarar una función como **inline**:

1ª) Precediendo la definición de una función con la palabra clave **inline**.

inline <declaración de función>

2ª) Para las funciones miembro de una clase, se puede hacer definiendo el código de la función dentro de la definición de la clase.

www.profmatiasgarcia.com.ar

```
class  <nombre_de_clase>
{
  <declaración_de_función> {<código de la función>}
}
```

Funciones *friend*

Es posible que una función que no es miembro de una clase tenga acceso a la parte privada de esa clase declarándola como **friend** (amiga) de la clase.

El formato de la declaración de funciones friend es el siguiente:

```
class <nombre de la clase>
{
 public :
    friend <declaración de función>
}
```

Calificación de variables miembro

A veces es necesario distinguir entre variables miembro de una clase y otro tipo de variables. Esto se puede realizar mediante el operador de resolución de ámbito "::".

Ejemplo:

```
class X
{
   int m;
public :
   void Setm (int);
   void Getm (void) {cout <<m;}
};

void main( )
{
 X x;
 x.Setm (5);
 x.Getm ( );
}
```

www.profmatiasgarcia.com.ar

```
void X::Setm (int m)
{
 X::m=m;                  //para distinguir el parámetro m, de miembro m de la clase X
}
```

Variables de clase

Entre los datos que pueden ser declarados en una clase se pueden hacer dos distinciones:

- Las **variables de instancia** representan campos con denominación común para todos los objetos de la clase, pero con un contenido particular para cada uno de ellos.

- Una **variable de clase** es un campo con idéntico nombre e idéntico contenido para todos los objetos de una clase. Es más, la modificación del contenido de ese campo en un objeto, afectará a todos los demás objetos. En realidad una variable de clase no es un campo que se halle en todos los objetos de una clase y que tenga el mismo contenido para todos ellos, sino que es un mismo espacio de memoria que es compartido por todos los objetos de una clase.

La forma de declarar una variable de clase en C++ es declarando un campo miembro como **static**.

```
class  <nombre_clase>
{
  .....
 static <tipo> <nombre_variable>
  .....
}
```

Vectores de objetos

Se pueden crear arrays de objetos de la misma manera que se crean arrays de cualquier otro tipo de datos. Por ejemplo la siguiente línea declara un vector de 10 objetos de una clase llamada X, que deberá haber sido declarada con anterioridad.

```
X vector[10];
```

En este ejemplo, en caso de existir un constructor para la clase X, éste se ejecutará 10 veces, una para cada objeto del vector. De igual manera, cuando se salga del ámbito de utilización del vector de objetos declarado, en caso de existir una función destructora de la clase, ésta se ejecutará para cada uno de los objetos del vector.

www.profmatiasgarcia.com.ar

Ejemplo para pasar parámetros al constructor:

```
class X {
      int             n;
      static int suma;              X v[3] = {1,2,3}

 X(int i)                            X::suma=6
 {
  n=i;
  suma+=i;
 }
}
X::suma=0;
```

Punteros a objetos

De forma similar en C++ puede hacerse referencia a un objeto ya sea directamente, o bien usando un puntero a ese objeto.

Para acceder a un elemento de un objeto usando el objeto real, se usa el operador punto (.). Para acceder a un elemento específico de un objeto cuando se usa un puntero al objeto, se debe usar el operador flecha (->).

Un puntero a un objeto se declara con la misma sintaxis que con cualquier otro tipo de dato.

```
<Nombree_Clase>  *<Nombre_Variable>;
```

A un puntero se le puede asignar un objeto de dos maneras:

- asignándole la dirección de un objeto existente.
- localizando espacio en memoria mediante el operador **new**.

Ejemplo (primer caso):

```
class P_ejemplo {
      int  num;
public:
      void est_num(int val) {num=val;}
      void mostrar_num();
};
```

www.profmatiasgarcia.com.ar

```
void P_ejemplo :: mostrar_num()
{
 cout << num << "\n";
}

main(void)
{
  P_ejemplo  ob, *p;          //declarar un objeto y un puntero a él

  ob.est_num(1);              //acceso a ob directamente
  ob.mostrar_num();
  p=&ob;                      //asignar a p la dirección de ob
  p->mostrar_num();           // acceder a ob usando un puntero
  return 0;
}
```

La asignación de espacio de memoria mediante el operador **new** tiene la característica de que al mismo tiempo que se asigna memoria, se ejecuta la función constructor de la clase.

De similar manera, cuando se ejecuta el operador **delete** para un objeto, además de liberar el espacio ocupado, se ejecuta la función destructora de la clase del puntero.

Ejemplo:

```
main(void)
{
 int *p;
 p=new int;          //asigna memoria para un entero
 if (!p) {
    cout << "fallo en la asignación";
    return 1;
 }
 *p=20;               //asigna a esa memoria el valor 20
 cout << *p;
 delete p;           //libera la memoria
 return 0;
```

La Herencia

La *herencia* es el proceso por el cual un objeto puede adquirir las propiedades de otro objeto. En C++, la herencia se soporta permitiendo a una clase incorporar otra clase dentro de su declaración.

www.profmatiasgarcia.com.ar

Las clases que heredan propiedades se llaman **clase derivada**, mientras que la clase de la que se heredan se denomina **clase base**.

La forma de declarar una clase derivada es:

class <nombre_clase_derivada> : <acceso> <nombre_clase_base>

<acceso> puede ser public o private:

- **public:** los miembros public siguen siendo public, y los private siguen siendo private en la clase derivada.
- **private:** los miembros public y private son todos private en la clase derivada.

Ejemplo:

```
class X {
   char nombre[80];
 public :
   void mostrar(void);
   void nombrar(char *nom);
 };

 class Y : public X {          //clase derivada de X
    int edad;
 public:
    void mostrarY(void);
    void poneredad(int edad);
 };
```

Uso de la palabra protected

Se puede conceder a la clase derivada acceso a los elementos **private** de una clase base haciéndo a estos protected.

Ejemplo:

```
class X {
    protected:
        int i;
        int j;
    public:
        void obt_ij(void)
        void poner_ij(void);
```

www.profmatiasgarcia.com.ar

```
class y : public X {
      int k;
  public:
      int obj_k(void);
      void hacer_k(void);
};
```

Da a Y acceso a i y j aunque permanezcan inaccesibles para el resto del programa.

Resumiendo: Un miembro de una clase puede ser private, protected o public.

- Si es **private** su nombre sólo puede ser usado por funciones miembro y friend de la clase en la cual es declarado.
- Si es **protected,** su nombre sólo puede ser usado por funciones miembro y friend de la clase en la cual es declarado y por funciones miembro y friend de las clases derivadas de esta clase.
- Si es **public,** su nombre puede ser usado por cualquier función.

Herencia múltiple

Es posible que una clase herede atributos de dos o más clases. Para realizar esto, se usa una lista de herencia, separada por comas, en la lista de las clases base de la clase derivada. La forma general es:

```
class nombre_clase_derivada : lista de clases base
{
 ..............
};
```

Por ejemplo en este programa **Z** hereda de **X** y de **Y**.

```
class X {
  protected:
      int a:
  public:
      void hacer_a(int i);
};

class Y {
   protected:
       int b;
   public:
```

www.profmatiasgarcia.com.ar

```
        void hacer_b(int i);
};

class Z : public X, public Y {
  public:
      int hacer_ab(void);
 };
```

Constructores y destructores en clases derivadas

Es posible que una clase base y una clase derivada tengan cada una su función de construcción. Cuando una clase derivada contiene un constructor, se ejecuta el constructor base antes de ejecutar el constructor de la clase derivada.

```
class Base {
   public:
      Base ( ) { cout <<"\n La clase Base ha sido creada";}
};

class Derivada : public Base {
    public:
       Derivada ( ) { cout << " La clase Derivada ha sido creada"; }
};

main ( )
{
 Derivada der;
 return 0;
}
```

```
La clase Base ha sido creada
La clase Derivada ha sido creada
```

La función de destrucción de una clase derivada se ejecuta antes que la de la clase base. Esto es así ya que la destrucción de la clase base implica la destrucción de la clase derivada, el destructor de la clase derivada debe ser ejecutado antes de ser destruido.

Cuando la herencia es múltiple los constructores de las clases base se invocan por órden, de izquierda a derecha y siempre antes que el de la clase derivada. Mientras que los destructores se invocan de derecha a izquierda, y siempre después que el destructor de la clase derivada.

www.profmatiasgarcia.com.ar

Clase base virtual

Cuando dos objetos o más se derivan de una clase base común, se puede impedir que estén presentes múltiples copias de la clase base en un objeto derivado de esos objetos, declarando la clase base como virtual cuando se hereda.

```
#include <iostream>

class base {
  public:
     int i;
};

class d1 : virtual public base {                    // d1 hereda base como virtual
  public:
     int j;
 };

class d2 : virtual public base {                    // d2 hereda base como virtual
  public:
     int k;
 };

class d3 : public d1, public d2 {                   // d3 hereda d1 y d2 pero solo hay una
  public:                                           // copia de base en d3
     int m;
 };

main (void)
{
  d3 d;
  d.i=10;
  d.j=20;
  d.k=30;
  d.m=40;
}
```

En **d3** sólo hay una copia de **base** y **d.i=10** es perfectamente válido y no es ambiguo.

www.profmatiasgarcia.com.ar

Polimorfismo

El fin del polimorfismo es permitir el uso de un nombre para especificar un nombre de acción general. Se ejecuta una parte específica de la clase general dependiendo del tipo de dato con el que está tratando.

Sobrecarga de funciones

Una de las maneras que tiene C++ de llegar al polimorfismo es a través de la *sobrecarga de funciones*. En C++, dos o más funciones pueden compartir un mismo nombre siempre y cuando difieran en la declaración de sus parámetros.
Las funciones que comparten el mismo nombre se dice que están *sobrecargadas*.

```
int Cuadrado (int i);
long Cuadrado (long l);
double Cuadrado (double x);

void main ( )
{
 int i=1;
 long l=3;
 double x=0.45;
 cout << "El cuadrado de un entero es: << Cuadrado (i) <<"\n";
 cout << "El cuadrado de un long es: << Cuadrado (l) << "\n";
 cout << "El cuadrado de un double es: << Cuadrado (x) << "\n";
}

int Cuadrado (int i)
{
  return i*i;
}

long Cuadrado (long l)
{
  return l*l;
}

double Cuadrado (double x)
{
  return x*x;
}
```

www.profmatiasgarcia.com.ar

La palabra reservada *this*

Cada vez que se invoca una función miembro, se pasa automáticamente un puntero al objeto que la invoca. Se puede acceder a este puntero usando **this.** El puntero **this** es un parámetro implícito para todas las funciones miembro.

```
class c1 {
  int i;
  ......
  ......
}
```

Una función miembro puede asignarle a i el valor 10 usando esta sentencia: i=10;
En realidad, esta sentencia es la foma corta de la sentencia this->i=10;

Sobrecarga de operadores

Para sobrecargar un operador se debe definir que operación significa con relación a la clase a la que se aplica. Para hacer esto, hay que crear una función operador que defina su acción.

Forma general:

```
tipo nombreclase :: operator  # (lista de argumentos)
{

}
```

- es el operador que se quiere sobrecargar.
tipo - es el tipo de valor devuelto por la operación especificada, es con frecuencia, del mismo tipo que la clase para la cual se ha sobrecargado el operador.

El objeto causante de una llamada a un operador sobrecargado, siempre es el situado a la izquierda del operador.

En una expresión a+b donde a y b son objetos, a es el causante de la llamada (pasa su valor a través de this), mientras que el valor de b es recibido a través del paso de parámetros.

En general, cuando se usa una función miembro, no se necesitan parámetros al sobrecargar una operación monaria y solo se necesita un parámetro cuando se sobrecarga un operador binario.

www.profmatiasgarcia.com.ar

Funciones operadoras amigas

Es posible que una función operador sea amiga de una clase en vez de miembro, las funciones amigas no tienen el operador implícito **this** por tanto cuando se usa una función amiga para sobrecargar un operador, se pasan los dos operandos cuando se está sobrecargando un operador binario y un solo operando cuando el operador es unario.

```
punto a;
punto b:

b=a+5;
```

Esta operación debe retornar un punto cuyas coordenadas serán las de *a* incrementadas en 5.

```
punto punto :: operator + (int i)
{
 punto temp;
 temp.x=p.x+i;
 temp.y=p.y+i;
 temp.z=p.z+i;
 return temp;
}
```

Esta función resolvería expresiones como *a+5* pero no funcionaría si la expresión estuviese invertida, es decir, *5+a.* Para resolver este problema se usa la sobrecarga de operadores mediante funciones **friend,** ya que podemos escpecificar el orden de todos los parámetros, porque no existen parámetros implícitos.

El problema del doble orden de los operadores de una suma, solo puede ser resuelto escribiendo dos funciones friend.

```
punto operator + (punto p, int i)
{
 punto temp;
 temp.x=p.x+i;
 temp.y=p.y+i;
 temp.z=p.z+i;
 return temp;
}
```

www.profmatiasgarcia.com.ar

```
punto operator + (int i, punto p)
{
 punto temp;
 temp.x=p.x+i;
 temp.y=p.y+i;
 temp.z=p.z+i;
 return temp;
}
```

Punteros a clases derivadas

En C++ el polimorfismo se admite tanto en el momento de la compilación, como en el de la ejecución. La sobrecarga de operadores y funciones es un ejemplo de polimorfismo en el momento de la compilación. Para conseguir el polimorfismo en el momento de la ejecución C++ permite usar clases derivadas y funciones virtuales.

En C++ los punteros a clases base y a clases derivadas están relacionados. Mediante un puntero de una clase base es posible acceder a una clase derivada, aunque no es posible a la inversa. Por ejemplo:

```
clase_B  *p;                    clase_B   tipo base
clase_B  B_ob;                  clase_D   derivado de clase_B
clase_D  D_ob;

p=&B_ob;        // p apunta a un objeto del tipo clase_B
p=&D_ob;        // p apunta a un objeto del tipo clase_D
```

Usando **p,** se puede acceder a todos los elementos de **D_ob** heredados de **B_ob**. Sin embargo, los elementos específicos de la **D_ob** no pueden ser referenciados usando **p**, a menos que se utilice una refundición de tipos.

```
class X {
  protected:
     char nombre[30];
  public:
     void mostrar (void);
     void nombrar (char *nom);
};
```

www.profmatiasgarcia.com.ar

```
class Y : public X {
     int edad;
  public:
     void mostrarY (void);
     void poneredad (int ed);
};

main ( )
{
  X  x;
  Y  y;
  X  *px;
  Y  *py;
  ((Y *) px)-> poneredad (16)          // Accedemos a los no miembros de X
  ((Y *) px)-> mostrarY ( )            // refundiendo el tipo del puntero
```

Funciones virtuales

Una función virtual es una función que se declara en la clase base como **virtual** y se redefine en una o más clases derivadas.

Las funciones virtuales son especiales, porque cuando se accede a una usando un puntero de la clase base a un objeto de una clase derivada, C++ determinada en tiempo de ejecución a que función llamar en función del tipo de objeto apuntado.

Una función virtual se declara como virtual dentro de la clase base usando la palabra clave **virtual.** Sin embargo, cuando se redefine una función virtual en una clase derivada no se necesita repetir la palabar clave **virtual** (aunque no es un error hacerlo).

```
class Base {
  public:
     virtual void quien ( ) { cout << "Base\n"; };
};

class primera_d : public Base {
  public:
     void quien ( ) { cout << "Primera derivación\n"; };
};

class segunda_d : public Base {

};
```

www.profmatiasgarcia.com.ar

```
main ( )
{
 Base  obj_base;
 Base  * p;
 primera_d   obj_primera;
 segunda_d  obj_segunda;
 p=&obj_base;
 p->quien ( );                // Se accede al quien de Base
 p=&obj_primera;
 p->quien ( );                // Se accede al quien de primera
 p=&obj_segunda;
 p->quien ( );                // Se accede al quien de base
}
```

Funciones virtuales puras y tipos abstractos

Una función virtual pura es una función que se declara en una clase base y que no tiene definición relativa a la base, por lo que todos los tipos derivados se ven obligados a definir su propia versión de esa función.

Para declarar una función virtual pura se utiliza la forma general:

```
virtual  tipo  nombre_funcion (lista parametros) = 0;
```

Si una clase tiene por lo menos una función virtual pura, entonces se dice que esa clase es abstracta. Las clases abstractas tienen una característica importante: no puede haber objetos de esa clase, porque una o más de sus funciones carecen de definición. Es posible declarar punteros de una clase abstracta.

```
class figura {
  protected:
      double x,y;
  public:
      void pon_dim (double i, double j) { x=i ; y=j; };
      virtual void mostrar_area ( ) = 0;
};
```

www.profmatiasgarcia.com.ar

```
class triangulo : public figura {
 public:
      void mostrar_area ( )
      {
       cout << "Triangulo de altura" << x << " y base" << y;
      };
};

class cuadrado : public figura {
 public:
      void mostrar_area ( )
      {
       cout << "Cuadrado de dimensión << x << "," << y;
      };
};

main ( )
{
 figura       *p;
 triangulo   trian;
 cuadrado  cuad;
 p=&trian;
 p->pon_dim (10.0, 5.0);
 p->mostrar_area ( );
 p=&cuad;
 p->pon_dim (10.0, 5.0);
 p->mostrar_area ( );
}
```

Webgrafía y Licencia

- Textos tomados, corregidos y modificados de diferentes páginas de Internet, tutoriales y documentos.

- Este documento se encuentra bajo Licencia Creative Commons Attribution – NonCommercial - ShareAlike 4.0 International (CC BY-NC-SA 4.0), por la cual se permite su exhibición, distribución, copia y posibilita hacer obras derivadas a partir de la misma, siempre y cuando se cite la autoría del **Prof. Matías García** y sólo podrá distribuir la obra derivada resultante bajo una licencia idéntica a ésta.

- Autor:

Matías E. García
Prof. & Tec. en Informática Aplicada
www.profmatiasgarcia.com.ar
info@profmatiasgarcia.com.ar

creative commons

www.profmatiasgarcia.com.ar