Programación Avanzada

Dr. Alfonso Alba Cadena
fac@galia.fc.uaslp.mx

Facultad de Ciencias
UASLP

Introducción

- La programación orientada a objetos (POO) surje a partir de la necesidad de diseñar y mantener software que es cada vez más complejo.

- Este curso sienta las bases de la POO, las cuales son necesarias para el desarrollo de aplicaciones modernas, orientadas tanto a computadoras personales, como a dispositivos móviles e internet.

Propiedades de la POO

- Enfatiza el uso de unidades de programación autosuficientes y reutilizables (clases).

- Se enfoca más en los datos que en los procedimientos o funciones.

- Proporciona herramientas de encapsulamiento, que permiten restringir el acceso a datos sensibles.

- Proporciona métodos que permiten la transferencia de información entre las distintas unidades que componen un programa.

Lenguaje utilizado en el curso

- El curso se basa en el lenguaje **C++** , el cual es uno de los lenguajes de uso general más populares en la actualidad.

- Otros lenguajes similares son C# y Java.

- Existen múltiples compiladores de libre distribución para diferentes plataformas:
  - GNU C++ (Unix / Linux / OSX)
  - MinGW (Windows) - basado en GNU
  - Microsoft Visual C++ Express (Windows)

Contenido del curso

1. Repaso de apuntadores
2. Estructuras de datos estáticas
3. Introducción a la POO
4. Sobrecarga de funciones y operadores
5. Herencia
6. Polimorfismo
7. Flujos de entrada y salida

Unidad 0

Repaso de apuntadores

Almacenamiento en memoria

- La memoria puede verse como un conjunto de celdas numeradas y ordenadas, cada una de las cuales puede almacenar un byte (8 bits) de información. El número correpondiente a cada celda se conoce como su *dirección*.

- Cuando se crea una variable, el programa reserva el número suficiente de celdas de memoria (bytes) requeridos para almacenar la variable, y se encarga de que los espacios reservados no se traslapen.

Direcciones de memoria

- En C++ es posible obtener la dirección de memoria donde se encuentra almacenada una variable mediante el operador de referencia `&`

- Ejemplo:

```
int main() {
    int a = 10, b = 20;
    cout << &a << endl;
    cout << &b << endl;
    return 0;
}
```

Tamaño de una variable

- Podemos obtener también la cantidad de memoria que ocupa una variable (en bytes) mediante el operador `sizeof`:

```
int main() {
    int a = 10;
    cout << "Valor de a: " << a << endl;
    cout << "Direccion de a: " << &a << endl;
    cout << "Tamano de a: " << sizeof(a) << endl;
    return 0;
}
```

Apuntadores

- Un apuntador es una variable que contiene una dirección de memoria (donde posiblemente se almacene el valor de otra variable).

- Para crear apuntadores se utiliza el operador * como se muestra a continuación:

```
int main() {
    int a = 10;
    int *p;     // p es un "apuntador a int"
    p = &a;
    cout << "Valor de p: " << p << endl;
    cout << "Valor en p: " << *p << endl;
    cout << "Direcci\'on de p: " << &p << endl;
    cout << "Tamamo de p: " << sizeof(p) << endl;
    return 0;
}
```

Almacenamiento de arreglos

- Cuando se declara un arreglo, se reserva un solo bloque contiguo de memoria para almacenar todos sus elementos, y éstos se almacenan en la memoria en el mismo orden que ocupan en el arreglo:

```
int main() {
    int i, a[10];
    for (i = 0; i < 10; i++) {
        cout << &a[i] << endl;
    }
    return 0;
```

- En el ejemplo anterior, se puede verificar que `&a[i]` es igual a `&a[0] + i * sizeof(int)`.

Arreglos y apuntadores

- En C++, el nombre de un arreglo es también un apuntador al primer elemento del mismo.

- De hecho, el lenguaje C++ no puede distinguir entre un arreglo y un apuntador: ambos son completamente intercambiables.

```
int main() {
    int i, a[10], *p;
    for (i = 0; i < 10; i++) { a[i] = i; }
    p = a;
    for (i = 0; i < 10; i++) {
        cout << p[i] << endl;
    }
    return 0;
}
```

Paso de arreglos a funciones

- Ya que un apuntador puede ser visto como un arreglo, una manera común de pasar arreglos como parámetros de una función consiste en pasar simplemente un apuntador al primer elemento del arreglo, y el número de elementos:

```
void suma_vectores(float *r, float *a, float *b, int n) {
    int i;
    for (i = 0; i < n; i++) {
        r[i] = a[i] + b[i];
    }
}
```

- Para el caso de arreglos bidimensionales y multidimensionales, lo más conveniente es utilizar arreglos unidimensionales en orden lexicográfico, de manera que la función pueda manejar arreglos de cualquier tamaño.

Aritmética de apuntadores

- Por lo general, un apuntador siempre está asociado con el tipo de dato al que apunta (e.g., apuntador a int, apuntador a float, etc).*

- Esto hace posible definir la operación `p+k` donde `p` es un apuntador y `k` es un entero. El resultado de esta operación es

```
p + k * sizeof(tipo)
```

  donde `tipo` es el tipo de datos asociado a `p`.

- Notar que si `p` es un arreglo, entonces `&p[k]` es equivalente a `p+k`, y `p[k]` es equivalente a `*(p+k)`

*La excepción son los apuntadores a void.

Ejemplo: operaciones elementales

```
void MulRen(float *m, int n, int r, float k) {
    float *p = m + r * n;
    for (int j = 0; j < n; j++) { p[j] *= k; }
}

void SumRen(float *m, int n, int r1, int r2, float k) {
    float *p1 = m + r1 * n;
    float *p2 = m + r2 * n;
    for (int j = 0; j < n; j++) { p1[j] += p2[j] * k; }
}

void InterRen(float *m, int n, int r1, int r2) {
    float t;
    float *p1 = m + r1 * n;
    float *p2 = m + r2 * n;
    for (int j = 0; j < n; j++) {
        t = p1[j];  p1[j] = p2[j];  p2[j] = t;
    }
}
```

Asignación dinámica de memoria

- Para cada función, el programa define una cierta cantidad de memoria (conocida como pila o stack) para almacenar las variables locales.

- En algunas ocasiones, esta memoria no será suficiente. Por ejemplo:

```
int main() {
    int arreglote[1024 * 1024];
    return 0;
}
```

Asignación dinámica de memoria

- La solución consiste en asignar memoria al arreglo de manera dinámica mediante el operador `new`, el cual devuelve un apuntador a la memoria reservada.

```
int main() {
    int i, n = 1024 * 1024;
    int *a = new int[n];
    for (i = 0; i < n; i++) { a[i] = i; }
    delete[] a;
    return 0;
}
```

- Es importante siempre liberar la memoria reservada mediante el operador `delete[]` una vez que ya no se va a utilizar.

El apuntador `NULL`

- El lenguaje C++ define una constante especial llamada `NULL`, el cual es un apuntador que no apunta a ningún lugar válido en la memoria y tiene como valor numérico el cero.

- Se recomienda siempre inicializar los apuntadores con una dirección válida, o bien con `NULL`, para evitar sobreescribir información de manera accidental.

- El operador `new` devuelve `NULL` si no es capaz de reservar la cantidad de memoria solicitada. Esto permite detectar la falta de memoria en un programa.

Práctica 1

- Escriba un programa que realice lo siguiente:
  1. Declare un arreglo bidimensional `m` de 10 x 10 enteros.
  2. Llene el arreglo con números aleatorios entre 0 y 10.
  3. Imprima las direcciones de memoria de cada uno de los elementos y verifique que están organizadas en orden lexicográfico; es decir, `&m[i][j] == m + 10 * i + j`.
  4. Declare un apuntador `p` a int y asígnele la dirección del primer elemento de `m`. Ahora `p` actúa como un arreglo unidimensional que hace referencia a los mismos elementos que `m`, pero en orden lexicográfico. Utilice `p` para imprimir el contenido del arreglo.

Unidad II

Clases

Introducción

- En muchas aplicaciones, la organización de la información juega un papel crucial, en particular si la cantidad de datos es considerable.

- Los arreglos proporcionan una manera de almacenar datos de forma ordenada, siempre y cuando éstos sean todos del mismo tipo.

- También existen las *clases*, las cuales permiten organizar datos, posiblemente de distintos tipos, en una sola unidad.

- En los arreglos, los datos individuales se distinguen por medio de un subíndice. En las clases, se distinguen por medio de un nombre de campo.

Definición de clases

- La manera más común de definir una clase es la siguiente:

```
class nombre_de_clase {
public:
    tipo_1 nombre_de_campo_1;
    tipo_2 nombre_de_campo_2;
    ...
    tipo_n nombre_de_campo_n;
};
```

- La instrucción anterior define un nuevo tipo de datos llamado `nombre_de_clase`, a partir del cual se pueden declarar variables (instancias).

Ejemplo de clase

- La siguiente clase agrupa las coordenadas (x, y) de un punto en el plano cartesiano:

```
class punto {
public:
    int x;
    int y;
};
```

- Podemos declarar variables (instancias) de la clase `punto` (cada una de las cuales contiene componentes `x` y `y`) de la forma siguiente:

```
int main() {
    punto p;
    return 0;
}
```

Acceso a miembros

- El acceso a los miembros o campos de una instancia se realiza mediante el operador punto (.) :

```
int main() {
    punto p;
    p.x = 10;
    p.y = -5;

    cout << "p = (" << p.x << ", " << p.y << ")" << endl;
    return 0;
}
```

Otro ejemplo

- Los miembros de una clase no necesariamente deben ser del mismo tipo:

```
class alumno {
public:
    char nombre[100];
    int generacion;
    float promedio;
};
```

Inicialización de instancias

- En muchos casos, las instancias de una clase pueden inicializarse al momento de declararlas:

```
int main() {
    punto p = { 10, -5 };

    cout << "p = (" << p.x << ", " << p.y << ")" << endl;
    return 0;
}
```

Ejercicios

1. Defina una clase para almacenar las partes real e imaginaria de un número complejo.

2. Declare una instancia z de la clase anterior e inicialícela con cero. Pida al usuario un ángulo ϕ en radianes y haga $z = e^{i\phi}$.

3. Imprima el valor de z en el formato $a+bi$, donde a es la parte real y b la imaginaria. Utilice estructuras de decisión (`if`) para imprimir correctamente los casos donde a o b puedan ser negativos o cero.

Apuntadores a clases

- De manera similar a los arreglos, los miembros de una instancia de una clase residen en un bloque contiguo de memoria, pero a diferencia de los arreglos, es posible que algunas celdas de memoria queden vacías (este fenómeno se conoce como alineamiento).

- El tamaño total (en bytes) de una clase es la suma de los tamaños de sus campos de datos.

- Otras diferencias importantes
  - El nombre de una instancia no representa un apuntador a la misma. Es necesario precederlo con el operador & para obtener un apuntador al primer elemento.
  - Las operaciones aritméticas con apuntadores a clases utilizan como tamaño de paso el tamaño total de la estructura.

Acceso a miembros a través de apuntadores

- Considere una instancia `p` de la clase `punto` definida anteriormente. Entonces:
  - `&p.x` y `&p.y` son las direcciones de memoria de los miembros `x` y `y` de `p`, respectivamente.
  - `&p` es la dirección de memoria del primer elemento de `p` (en este caso, `&p == &p.x`).
- Considere el apuntador definido como

```
punto *q = &p;
```

  - `q` es un apuntador a una instancia de clase `punto`
  - `*q` es la instancia (de clase `punto`) a la que apunta `q`
  - Para acceder a los campos individuales a través de `q` puede escribirse `(*q).x` y `(*q).y`
  - Equivalentemente, puede utilizarse el operador `->` - por ejemplo: `q->x` y `q->y`

Paso de instancias como parámetros

- Las instancias de una clase pueden pasarse como parámetros a una función de múltiples formas:
  - Por valor: El contenido de la instancia se copia en una variable local de la función. Dentro de la función se trabaja con la copia, por lo que la instancia original no se modifica. Ejemplo:
    ```
    float Magnitud(punto p) { return sqrt(p.x * p.x + p.y * p.y); }
    ```
  - Por referencia: El parámetro es una referencia a la instancia que se pasa a la función, por lo que la instancia original puede ser modificada. Ejemplo:
    ```
    float Magnitud(punto &p) { return sqrt(p.x * p.x + p.y * p.y); }
    ```
  - Por apuntador: Se le pasa a la función un apuntador a la instancia y se accede a la misma a través del apuntador. Por lo tanto, cualquier modificación se aplica directamente a la instancia original. Ejemplo
    ```
    float Magnitud(punto *p) { return sqrt(p->x * p->x + p->y * p->y); }
    ```

Ejemplo

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

using namespace std;

class polinomio {
public:
    double *coef;
    int grado;
};

bool inicializa(polinomio *p, int g) {
    int i;
    if (p == NULL) return false;
    p->coef = new double[g+1];
    if (p->coef == NULL) return false;
    p->grado = g;
    for (i = 0; i <= p->grado; i++) p->coef[i] = 0;
    return true;
}

double evalua(polinomio p, double x) {
    int i;
    double suma = 0, potencia = 1;
```

```
    if (p.coef == NULL) return 0;
    for (i = 0; i <= p.grado; i++) {
        suma += p.coef[i] * potencia;
        potencia *= x;
    }
    return suma;
}

int main() {
    polinomio p;
    double x = -1;

    inicializa(&p, 3);
    p.coef[3] = 1;
    p.coef[0] = -1;

    cout << "x\t\tp(x)\n------------------------" << endl;
    cout << setprecision(3) << fixed;
    while (x <= 1) {
        cout << x << "\t\t" << evalua(p, x) << endl;
        x += 0.2;
    }
    delete[] p.coef;
    return 0;
}
```

Arreglos y clases

- Como se vió en el ejemplo anterior, un arreglo puede ser elemento de una clase. Si el tamaño del arreglo no es fijo (varía de instancia a instancia), lo más conveniente es declararlo como un apuntador y reservar memoria para el arreglo de forma dinámica.

- También es posible declarar arreglos de instancias de una clase. Por ejemplo:

```
int i;
punto p[10];
for (i = 0; i < 10; i++) {
    p[i].x = rand() % 100;
    p[i].y = rand() % 100;
}
```

Asignación dinámica

- También es posible reservar memoria de manera dinámica para almacenar instancias o arreglos de instancias de alguna clase:

```
punto *p, *q;
p = new punto;       // reserva espacio para una instancia
q = new punto[10];   // reserva espacio para un arreglo de 10 instancias
...
delete p;            // libera el espacio reservado para una instancia
delete[] q;          // libera el espacio reservado para el arreglo
```

Clases anidadas

- Una clase puede tener como miembros a instancias de otra clase, de manera que puedan construirse clases muy complejas pero bien organizadas.

```
class poligono {
    punto *v;   // arreglo de vertices de un poligono
    int nv;     // numero de vertices en el poligono
};

bool inicializa(poligono *p, int nv) {
    int i;
    p->v = new punto[nv];
    if (p->v == NULL) return false;
    p->nv = nv;
    for (i = 0; i < nv; i++) {
        p->v[i].x = cos(2 * M_PI * i / nv);
        p->v[i].y = sin(2 * M_PI * i / nv);
    }
    return true;
}
```

Métodos de una clase

- Además de agrupar la información relevante sobre un cierto tipo de objetos, una clase puede incluir también las funciones necesarias para procesar esa información.

- Por ejemplo, suponga que se desea implementar una clase para manejar números complejos, que incluya funciones para calcular el módulo y argumento de un número. Esta clase puede declararse como sigue:

```
class Complejo {
public:
    float re, im;   // partes real e imaginaria

    float Modulo();     // funcion para obtener el modulo
    float Argumento();  // funcion para obtener el argumento
};
```

Implementación de métodos

- La implementación de estas funciones (llamadas *métodos*) suele realizarse fuera de la declaración de la clase. Por ejemplo:

```
float Complejo::Modulo() {
    return sqrt(re * re + im * im);
}
```

- Note que el acceso a los miembros de la clase (`re` e `im`) desde dentro de un método no se hace a través de una instancia, ya que el método se implementa de forma general para todas las instancias de la clase.

- También es posible, aunque poco recomendable, implementar métodos directamente en la defínción de la clase:

```
class Complejo {
public:
    float re, im;   // partes real e imaginaria
    float Modulo() { return sqrt(re * re + im * im); }
};
```

Objetos

- Dado que ahora las clases no solo contienen información, sino también el código necesario para procesar esa información, a las instancias de una clase ya no se les puede llamar variables.

- A las instancias de una clase se les conoce comúnmente como *objetos*.

Llamadas a métodos desde un objeto

- Para utilizar alguno de los métodos de una clase, es necesario acceder a ellos a través de un objeto de esa clase.

```
int main() {
    Complejo c;
    c.re = 5;
    c.im = 2;
    cout << "|" << c.re << "+" << c.im << "i| = " << c.Modulo() << endl;
    return 0;
}
```

Llamadas a métodos desde otros métodos

- Así como es posible acceder a los miembros de una clase desde uno de sus métodos, también es posible llamar a otros métodos. Por ejemplo:

```
class Complejo {
public:
    float re, im;   // partes real e imaginaria
    float Modulo();
    void Normaliza();
};

float Complejo::Modulo() {
    return sqrt(re * re + im * im);
}

void Complejo::Normaliza() {
    float m = Modulo();
    re /= m;
    im /= m;
}
```

Tipos de acceso

- Por default, todos los miembros de una clase (tanto datos como métodos) son privados. Eso significa que solamente puede accederse a ellos desde dentro de los mismos métodos de la clase.

- En contraste, los miembros públicos de cualquier clase pueden accederse desde cualquier punto del programa.

- En la definición de una clase, uno puede cambiar el tipo de acceso mediante las palabras clave `public:` y `private:` - todos los miembros declarados a continuación tendrán el tipo de acceso especificado.

Ejemplo de acceso

```
class Paciente {
    int edad;
    float peso;
    float estatura;
public:
    float IndiceMasaCorporal() {
        return peso / (estatura * estatura);
    }
};

int main() {
    Paciente p;
    p.peso = 70;         // error de acceso
    p.estatura = 1.7;    // error de acceso
    cout << p.IndiceMasaCorporal() << endl;
    return 0;
}
```

Acceso a miembros privados

- Una de las principales ventajas del acceso privado consiste en evitar la modificación de datos sensibles que pueda ocasionar problemas en el programa.

- Para acceder y modificar datos privados suelen implementarse métodos públicos comúnmente llamados *set* (modificar) y *get* (obtener). Considere el siguiente ejemplo:

```
class Paciente {
    float peso;
    float estatura;
public:
    float getPeso() { return peso; }
    void setPeso(float x) { peso = x; }
    float getEstatura() { return estatura; }
    void setEstatura(float x) { estatura = x; }
};
```

Acceso con restricciones

- Una de las ventajas de las funciones tipo *set* y *get* radica en que pueden añadirse restricciones que determinen bajo qué condiciones puede realizarse el acceso. Por ejemplo:

```
class Polinomio {
    int grado;
    float *coef;    // arreglo para los coeficientes
public:
    float getCoef(int k) {
        if ((k >= 0) && (k <= grado)) return coef[k];
        else return 0;
    }

    void setCoef(int k, float c) {
        if ((k >= 0) && (k <= grado)) coef[k] = c;
    }
};
```

Constructores

- En cualquier lenguaje de programación, en particular C y C++, siempre es importante inicializar las variables de manera adecuada.

- En la programación orientada a objetos, existen métodos especiales, llamados *constructores* que se ejecutan automáticamente a la hora de declarar un objeto de una cierta clase, y se encargan de inicializar los miembros de datos.

- En el lenguaje C++, los constructores tienen el mismo nombre que la clase a la que corresponden, pueden recibir argumentos, y no devuelven ningún tipo de datos (ni siquiera `void`). Además, suelen declararse con acceso público.

Ejemplo de constructores

```
class Polinomio {
    int grado;
    float *coef;    // arreglo para los coeficientes
public:
    // constructores
    Polinomio() { grado = 0; coef = NULL; }
    Polinomio(int g);
    int getGrado() { return grado; }
};

Polinomio::Polinomio(int g) {
    coef = new float[g];
    if (coef != NULL) grado = g;
    else grado = 0;
}

int main() {
    Polinomio p(3);     // crea un polinomio de grado 3
    cout << p.getGrado() << endl;
    return 0;
}
```

Destructores

- En algunos casos, es necesario revertir el proceso de inicialización realizado por un constructor, o realizar otras tareas de limpieza una vez que un objeto deja de ser utilizado.

- Lo anterior puede realizarse de manera mediante un método llamado *destructor* el cual se llama de manera automática cuando un objeto deja de existir en la memoria.

- En C++, los destructores llevan el mismo nombre de la clase a la que pertenecen, precedido por una tilde ($\sim$). No reciben argumentos ni devuelven resultados.

Ejemplo de destructores

```
class Polinomio {
    int grado;
    float *coef;    // arreglo para los coeficientes
public:
    // constructores y destructor
    Polinomio() { grado = 0; coef = NULL; }
    Polinomio(int g);
    ~Polinomio();
};

Polinomio::Polinomio(int g) {
    coef = new float[g];
    if (coef != NULL) grado = g;
    else grado = 0;
}

Polinomio::~Polinomio() {
    if (coef != NULL) delete[] coef;
    coef = NULL;
    grado = 0;
}
```

Argumentos por defecto en constructores

- Cualquier función o método de C/C++ puede incluír valores por defecto para uno o más de sus argumentos.

- Esto es particularmente útil para los constructores. Ejemplo:

```
class Punto {
public:
    int x, y;
    Punto() { x = 0; y = 0; }
    Punto(int xx, int yy) { x = xx; y = yy; }
};
```

puede reemplazarse simplemente por

```
class Punto {
public:
    int x, y;
    Punto(int xx = 0, int yy = 0) { x = xx; y = yy; }
};
```

Objetos como miembros de una clase

- Considere las siguientes clases

```
class Punto {
public:
    int x, y;
    Punto(int xx = 0, int yy = 0) { x = xx; y = yy; }
};

class Linea {
public:
    Punto a, b;
    Linea(int x1, int y1, int x2, int y2);
};
```

- El constructor de la clase `Linea` debe inicializar dos objetos de clase `Punto`. Esto puede hacerse llamando a los constructores de los miembros de la forma siguiente:

```
Linea::Linea(int x1, int y1, int x2, int y2)
    : a(x1, y1), b(x2, y2) {
    // codigo adicional puede incluirse aqui
}
```

Miembros estáticos

- En algunos casos, puede ser de utilidad compartir datos entre todas las instancias de una clase. En este caso, los datos pueden declararse con el calificador `static`.

- Si un miembro es declarado como `static`, entonces existirá solo una copia para todos los objetos de esa clase, o incluso cuando no existan objetos de la clase.

- Si un miembro se declara como estático dentro de una clase, debe entonces definirse (y posiblemente inicializarse) también fuera de la declaración de la clase.

- Un método declarado como `static` podrá acceder solamente a datos estáticos de la clase a la que pertenece, y no requiere ser llamado a través de una instancia de la clase.

Ejemplo de miembros estáticos

```
class CuentaInstancias {
    static int num;
public:
    CuentaInstancias() { num++; }
    ~CuentaInstancias() { num--; }
    static int getNum() { return num; }
};

int CuentaInstancias::num = 0;

void Prueba() {
    CuentaInstancias a;
    cout << CuentaInstancias::getNum() << endl;
}

int main() {
    CuentaInstancias *p;
    cout << CuentaInstancias::getNum() << endl;
    Prueba();
    p = new CuentaInstancias;
    cout << CuentaInstancias::getNum() << endl;
    Prueba();
    delete p;
    cout << CuentaInstancias::getNum() << endl;
    return 0;
}
```

El apuntador `this`

- Todo objeto tiene acceso a su propia dirección de memoria a través de un apuntador llamado `this`. El apuntador `this` no es un miembro de la clase, sino un argumento implícito que se pasa a todos los métodos de la clase a la hora de llamarlos desde un objeto específico.

- Ejemplo:

```
class ClaseX {
    int x;
public:
    ClaseX(int xx = 0) { x = xx; }
    void Print() { cout << this->x << endl; }
};

int main() {
    ClaseX obj;
    obj.Print();
    return 0;
}
```

- Los métodos declarados como `static` no tienen acceso al apuntador `this`.

Llamadas en cascada

- Uno de los usos mas interesantes del apuntador `this` consiste en implementar métodos que pueden llamarse en cascada.

- Para lograr esto, los métodos deben devolver una referencia o apuntador al objeto desde el cual son llamados. Por ejemplo:

```
class Punto {
    int x, y;
public:
    Punto(int xx = 0, int yy = 0) { x = xx; y = yy; }

    Punto *setX(int xx) { x = xx; return this; }
    Punto *setY(int yy) { y = yy; return this; }
    Punto *print() {
        cout << "(" << x << ", " << y << ")" << endl;
        return this;
    }
};

int main() {
    Punto p;
    p.setX(3).setY(5).print();
    return 0;
}
```

Unidad III

Sobrecarga de funciones y operadores

Introducción

- Se le llama *sobrecarga de funciones* al hecho de declarar dos o más funciónes con el mismo nombre y con el mismo alcance.

- Considere el siguiente ejemplo:

```
void imprime(int x) { cout << "Esto es un int: " << x << endl; }

void imprime(float x) { cout << "Esto es un float: " << x << endl; }

void imprime(char *x) { cout << "Esto es un char *: " << x << endl; }

int main() {
    imprime("hola");
    imprime(10);
    imprime(3.1416);
    return 0;
}
```

Reglas de sobrecarga

- Las distintas declaraciones de una función sobrecargada deben diferir en el número y/o tipo de argumentos que reciben.

- En los siguientes casos, puede haber ambigüedad al llamar a una función sobrecargada, ocasionando un error de compilación:
  - Cuando las funciones solo difieren en el tipo de datos que devuelven. Ejemplo: `int f()` y `char f()`.
  - Cuando una de las funciones recibe como parámetro un arreglo de un cierto tipo, mientras que otra recibe un apuntador al mismo tipo. Ejemplo: `void f(int *x)` y `void f(int x[10])`.
  - Cuando una de las funciones recibe un parámetro de un cierto tipo, mientras la otra recibe una referencia del mismo tipo. Ejemplo: `void f(int x)` y `void f(int &x)`.

Sobrecarga de métodos

- También es posible sobrecargar los métodos de una clase.

```
class Matriz {
    int ren;
    int col;
    float *datos;
public:
    ...
    void Multiplica(Matrix &m, float x);
    void Multiplica(float x, Matrix &m);
    void Multiplica(Matrix &m1, Matrix &m2);
};
```

Operadores y sobrecarga

- En los lenguajes C y C++, muchos de los operadores actúan de manera diferente según el tipo de datos de sus operandos.

- Por ejemplo, el operador de división devuelve el cociente de una división cuando los operandos son enteros, pero si alguno de los operandos es de punto flotante, el resultado es la división exacta con la misma precisión que el operando de mayor precisión.

- Otro ejemplo son los operadores << y >>, los cuales se usan para realizar corrimientos binarios en números enteros, o bien, como operadores de inserción y extracción de flujo (e.g., con `cout` y `cin`).

- De cierta forma, puede decirse que estos operadores están sobrecargados para actuar de manera diferente dependiendo del tipo de sus operandos (argumentos).

Ejemplo de operadores sobrecargados

```
#include <iostream>

using namespace std;

#define DIV(x) cout << x << "\t" << sizeof(x) << endl

int main() {
    int x;
    DIV(10/3);
    DIV(10.0/3.0);
    DIV(10.0f/3.0f);
    DIV(10.0f/3.0);

    cout << endl << "8 << 2 = " << (8 << 2) << endl;
    cout << "Dame un entero x: ";
    cin >> x;
    cout << "x >> 1 = " << (x >> 1) << endl;
    return 0;
}
```

Sobrecarga de operadores

- En C++, también es posible sobrecargar muchos de los operadores para definir su acción con operandos cuyo tipo de datos es no nativo.

- La sobrecarga de un operador se realiza definiendo una función de manera habitual. El nombre de la función es `operator` seguido del operador que se desea sobrecargar (por ejemplo, `operator +`). La función puede tomar uno o dos argumentos, dependiendo de si se trata de un operador unario o binario. En el caso de operadores binarios, el primer argumento representa el operando del lado izquierdo, y el segundo el del lado derecho.

- Existen algunos operadores en C++ que no pueden sobrecargarse. Estos son:

```
.        .*        ::        ?:        sizeof
```

Sobrecarga de operadores

```
class Complejo {
public:
    float re, im;
    Complejo(float a = 0, float b = 0) { re = a; im = b; }
};

Complejo operator +(Complejo p, Complejo q) {
    return Complejo(p.re + q.re, p.im + q.im);
}

Complejo operator~(Complejo p) {
    return Complejo(p.re, -p.im);
}

int main() {
    Complejo u(1,2), v(3,4), w;
    w = ~u + v;
    cout << w.re << ", " << w.im << endl;
    return 0;
}
```

Operadores como miembros de una clase

- También es posible declarar operadores como miembros de una clase. En este caso, el primer operando se convierte en la instancia desde la cual se manda llamar al operador, por lo que solo será necesario pasar como argumento el segundo operando en el caso de los operadores binarios.

- El uso de operadores como miembros de una clase tiene la ventaja de permitir que el operador acceda a los miembros privados de la clase.

- Los operadores que se sobrecarguen como miembros de una clase no pueden ser miembros estáticos.

Operadores como miembros de una clase

```
#include <iostream>

using namespace std;

class Complejo {
    float re, im;
public:
    Complejo(float a = 0, float b = 0) { re = a; im = b; }

    Complejo operator +(Complejo q) { return Complejo(re+q.re, im+q.im); }
    Complejo operator ~() { return Complejo(re, -im); }

    void imprime() { cout << "(" << re << ", " << im << ")" << endl; }
};

int main() {
    Complejo u(1,2), v(3,4), w;
    w = ~u + v;
    w.imprime();
    return 0;
}
```

Operadores de asignación

- El operador de asignación = se sobrecarga por defecto para las nuevas clases definidas por el programador.

- La acción por defecto de este operador consiste en copiar los valores de los campos de un objeto a otro.

- En algunos casos, el operador puede sobrecargarse para permitir la asignación entre objetos de distintas clases (e.g., asignar un valor real a un objeto de clase `Complejo`).

- En algunos casos, la acción por defecto no es apropiada; por ejemplo, cuando uno o más campos de la clase son apuntadores a arreglos independientes para cada instancia de la clase (e.g., clases `Matriz` y `Polinomio`). En estos casos, puede redefinirse la acción del operador de asignación para que en lugar de simplemente copiar los valores de los campos que son apuntadores, reserve adecuadamente memoria para el objeto destino.

Ejemplo: operador de asignación

```
class Complejo {
public:
    float re, im;
    Complejo(float r=0, float i=0) { re = r; im = i; }

    // operadores de asignacion
    Complejo operator =(float r) { re = r; im = 0; }
    Complejo operator +=(Complejo c) { re += c.re; im += c.im; }
    Complejo operator +=(float r) { re += r; }
};

int main() {
    Complejo a(1,2), b = 7;
    b += a;
    a += -2;
    return 0;
}
```

Operadores de inserción de flujo

- El operador de inserción de flujo << se utiliza con objetos de salida de flujo como `cout`, cuya clase es `ostream`, para escribir información en la pantalla o en un archivo.

- Es posible sobrecargar este operador para imprimir información de objetos de otras clases.

```
ostream &operator <<(ostream &os, Complejo c) {
    os << c.re;
    if (c.im >= 0) os << "+" << c.im << "i";
    else os << c.im << "i";
    return os;  // permitir encadenamiento
}

int main() {
    Complejo a(1,2), b = 7;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    return 0;
}
```

Consideraciones para objetos con un manejo dinámico de memoria

- Para aquellas clases que reservan memoria dinámica y la liberan en el constructor, uno debe tener cuidado al pasar objetos por valor, dado que los miembros que contienen apuntadores serán copiados a un objeto local de la función, y su destructor se ejecutará una vez que la función termine.

- Una manera de evitar lo anterior consiste en pasar los objetos por referencia.

Ejemplo

```
class Matrix {
public:
    int rows ,cols;
    float *data;

    Matrix(int r = 0, int c = 0) { Create(r,c); }
    ~Matrix() { delete[] data; }
    void Create(int r, int c);
    Matrix &operator =(Matrix m);
};

void Matrix::Create(int r, int c) {
    if (data) delete[] data;
    data = new float[r * c];
    rows = r; cols = c;
}

Matrix &Matrix::operator =(Matrix m) {
    Create(m.rows, m.cols);
    memcpy(data, m.data, rows * cols * sizeof(float));
    return *this;
}

int main() {
    Matrix m1(2,2), m2;
    m2 = m1;
    // el arreglo m1.data ha sido liberado prematuramente
}
```

Unidad IV

Herencia

Introducción

- En ocasiones, puede ser necesario aumentar o modificar la funcionalidad de una clase, de manera que la nueva implementación se enfoque en casos mas específicos que la implementación de la clase original.

- Ejemplo: Una imagen digital es un arreglo bidimensional de puntos, llamados *pixeles*, donde el valor de cada elemento corresponde con la intensidad o color del punto correspondiente. De esta manera, una imagen puede representarse simplemente como una matriz; sin embargo, algunas de las operaciones que suelen realizarse con imágenes son muy distintas a las que se realizan con matrices.

- En los lenguajes orientados a objetos, lo anterior puede realizarse a través de un mecanismo llamado *herencia de clases*, a partir del cual se puede definir una nueva clase a partir de otra ya existente. Mediante este mecanismo, la nueva clase *hereda* la funcionalidad de la primera, y además es posible agregar nueva funcionalidad o modificar la existente.

Definición de una clase heredada

- Considere la clase `Matriz` y suponga que deseamos heredar su funcionalidad para definir una nueva clase `Imagen`. Esto puede realizarse de la manera siguiente:

```
class Imagen : public Matriz {
public:
    void AjustarBrillo(float brillo);
    void AjustarContraste(float contraste);
    ...
};
```

- En ese caso, la clase `Imagen` será una clase *heredada* o *derivada* de la clase `Matriz`.

- La clase `Matriz` será la *clase base* de `Imagen`.

Clases base y clases derivadas

- Una clase derivada hereda todos los miembros de datos y métodos de la clase base. Sin embargo, los métodos propios de la clase derivada no tendrán acceso a los miembros privados de la clase base.

- Los miembros públicos de una clase base serán también públicos en cualquier clase derivada de ésta.

- Además, pueden agregarse nuevos miembros (datos y métodos) a la clase derivada. Estos no formarán parte de la clase base.

Reimplementación de métodos

- También es posible redefinir los métodos de una clase base en una clase derivada para modificar su funcionalidad o adaptarla a casos específicos.

```
class Vehiculo {
public:
    int NumRuedas() { return 0; }
};

class Coche : public Vehiculo {
public:
    int NumRuedas() { return 4; }
};

class Bicicleta : public Vehiculo {
public:
    int NumRuedas() { return 2; }
};

int main() {
    Coche c;
    Bicicleta b;
    cout << "Coche tiene " << c.NumRuedas() << " ruedas" << endl;
    cout << "Bicicleta tiene " << b.NumRuedas() << " ruedas" << endl;
    return 0;
}
```

Llamadas a métodos de la clase base

- Cuando se llama a un método a través de un objeto, siempre se llama a la versión más actualizada del método.
- Es posible también llamar a un método de una clase base mediante el operador `::`

```
class X {
public: void f() { cout << "X" << endl; }
};

class Y : public X {
};

class Z : public Y {
public: void f() { cout << "Z" << endl; }
};

class W : public Z {
public: void f() { X::f(); }
};

int main() {
    X x;    x.f();
    Y y;    y.f();
    Z z;    z.f();
    W w;    w.f();
    return 0;
}
```

Tipo de acceso protegido

- Los miembros privados de una clase base no pueden ser accedidos desde ninguno de los métodos de las clases derivadas.
- Existe otro tipo de acceso que permite que miembros de una clase base sean privados, pero aún accesibles desde las clases heredadas. Este tipo de acceso se llama *protegido* y se especifica mediante la palabra `protected`.

```
class Base {
public:     int x;
private:    int y;
protected:  int z;
};

class Derivada : public Base {
public:
    int X() { return x; }
    int Y() { return y; }   // Error: y es miembro privado de Base
    int Z() { return z; }
};

int main() {
    Derivada d;
    cout << d.x << endl;
    cout << d.y << endl;    // Error: y es miembro privado de Base
    cout << d.z << endl;    // Error: z es miembro protegido de Derivada
    return 0;
}
```

Herencia privada y protegida

- La herencia privada y protegida se realiza de la manera siguiente:

```
class B                     { /* ... */ };
class D_priv : private   B  { /* ... */ };
class D_prot : protected B  { /* ... */ };
class D_pub  : public    B  { /* ... */ };
```

- Las diferencias son las siguientes:
  - **Herencia privada:** Los miembros públicos y protegidos de `B` son privados en `D_priv`.
  - **Herencia protegida:** Los miembros públicos y protegidos de `B` son protegidos en `D_prot`.
  - **Herencia pública:** Los miembros públicos de `B` son públicos en `D_pub` y los miembros protegidos de `B` son protegidos en `D_pub`.

Constructores

- Al declarar un objeto de una clase derivada, el constructor de la clase base siempre debe llamarse antes del constructor de la clase derivada.

- Si el constructor de la clase base no toma parámetros, esta llamada se realiza de forma automática.

```
class Base {
public:
    Base() { cout << "Constructor de Base" << endl; }
};

class Derivada : public Base {
public:
    Derivada() { cout << "Constructor de Derivada" << endl; }
};

int main() {
    Derivada d;
    return 0;
}
```

Constructores

- Si el constructor de la clase base requiere parámetros, la llamada a este constructor debe hacerse de forma explícita en la declaración del constructor de la clase derivada:

```
class Base {
public:
    Base(int x) { cout << "Desde Base: " << x << endl; }
};

class Derivada : public Base {
public:
    Derivada(int x) : Base(x) { cout << "Desde Derivada: " << x << endl;
};

int main() {
    Derivada d(5);
    return 0;
}
```

Destructores

- El destructor de una clase base siempre se llama automáticamente después de ejecutarse el destructor de la clase derivada.

```
class Base {
public:
    ~Base() { cout << "Destructor de Base" << endl; }
};

class Derivada : public Base {
public:
    ~Derivada() { cout << "Destructor de Derivada" << endl; }
};

int main() {
    Derivada d;
    return 0;
}
```

Apuntadores a objetos de una clase y sus descendientes

- Una de las características más importantes de la programación orientada a objetos es que un apuntador a una clase derivada puede interpretarse automáticamente como un apuntador a la clase base, dado que la clase derivada contiene todos los miembros de la clase base.

```
class Base {
public:
    int x;
    Base(int _x = 0) : x(_x) {};
};

class Derivada : public Base {
public:
    int y;
    Derivada(int _x = 0) : Base(_x), y(_x) {};
};

int main() {
    Derivada *pd = new Derivada(5);
    Base *pb = pd;
    cout << pb->x << endl;
    cout << pb->y << endl;    // Error: y no pertenece a Base
    return 0;
}
```

Apuntadores a objetos de una clase y sus descendientes

- Es importante tomar en cuenta que al interpretar un apuntador a una clase derivada como un apuntador a la clase base, no podrán accederse a los miembros exclusivos de la clase derivada, y al llamar a un método a través del apuntador, se llamará al método correspondiente en la clase base, aunque éste haya sido reimplementado en la clase derivada.

```
class Base {
public:
    void f() { cout << "Soy la clase Base" << endl; }
};

class Derivada : public Base {
public:
    void f() { cout << "Soy la clase Derivada" << endl; }
};

int main() {
    Derivada *pd = new Derivada;
    Base *pb = pd;
    pd->f();
    pb->f();
    delete pd;
    return 0;
}
```

Unidad V

Polimorfismo

Introducción

- Se llama *polimorfismo* al arte de definir una clase que puede tomar múltiples formas (mediante herencia), de manera que los objetos de esta clase se comporten de manera diferente, dependiendo de la forma particular que cada uno tome.

- En C/C++, el polimorfismo se logra mediante los siguientes mecanismos:
  - Sobrecarga de operadores
  - Métodos virtuales
  - Plantillas

Motivación

- Considere una clase base `Animal` y dos clases (formas) derivadas de ella: `Perro` y `Gato`. Suponga que las tres clases tienen un método llamado `Habla()` el cual imprime el sonido que hace un animal de una clase particular.

- Considere ahora un apuntador `p` a `Animal`. Sabemos que en realidad, `p` puede apuntar a un objeto de clase `Animal` o cualquiera de sus clases derivadas (e.g., `Gato`). Por lo tanto, la llamada

  ```
  p->Habla();
  ```

  es ambigüa, ya que no queda claro si debe llamarse al método `Animal::Habla()` o al método `Gato::Habla()`.

Funciones virtuales

- Considere nuevamente un apuntador `p` a una clase base al que se le asigna la dirección de memoria de un objeto de una clase derivada.

- Por defecto, las llamadas a métodos a través de `p`, e.g., `p->metodo()`, se harán a los métodos de la clase base (determinada por el tipo de apuntador).

- Si se desea que las llamadas a través de `p` se realicen a los métodos implementados para la clase derivada (la verdadera clase del objeto al que `p` apunta), es necesario declarar los métodos como virtuales desde la definición de la clase base.

- Un método se declara como virtual anteponiendo la palabra `virtual` en la definición del método dentro de la clase.

Ejemplo: funciones virtuales

```
class Animal {
public:
    virtual void Habla() { cout << "?" << endl; }
};

class Perro : public Animal {
public:
    void Habla() { cout << "Guau" << endl; }
};

class Gato : public Animal {
public:
    void Habla() { cout << "Miau" << endl; }
};

int main() {
    Animal *a1 = new Perro;
    Animal *a2 = new Gato;
    a1->Habla();
    a2->Habla();
    delete a1;
    delete a2;
    return 0;
}
```

Destructores virtuales

- Considere una clase base `A` y una clase `B` heredada de `A`. Suponga que ambas clases tienen constructores y destructores. Observe el problema que se genera en el siguiente ejemplo:

```
int main() {
    A *p = new B;
    delete p;
    return 0;
}
```

- En la primer línea de la función `main()` se llama al constructor de `B` (lo cual requiere llamar primero al constructor de `A`) para crear un nuevo objeto de clase `B`.

- En la segunda línea, se llama al destructor del objeto al que apunta `p`; sin embargo, `p` es un apuntador a `A`, por lo que se estará llamando solamente al destructor de `A`, sin llamar primero al destructor de `B`.

- Para evitar lo anterior, es necesario declarar el destructor de `A` como virtual.

Ejemplo de destructores virtuales

```
class Base {
public:
    Base() { cout << "Constructor de Base" << endl; }
    virtual ~Base() { cout << "Destructor de Base" << endl; }
};

class Derivada : public Base {
protected:
    int *arreglo;
public:
    Derivada(int n) : arreglo(NULL) {
        arreglo = new int[n];
        cout << "Constructor de Derivada" << endl;
    }
    ~Derivada() {
        if (arreglo) delete[] arreglo;
        cout << "Destructor de Derivada" << endl;
    }
};

int main() {
    Base *pb = new Derivada(10);
    delete pb;
    return 0;
}
```

Consideraciones sobre funciones virtuales

- Las funciones virtuales deben declararse como tales en la clase base, no en las clases derivadas.

- La virtualidad se hereda de manera automática; es decir, una función declarada como virtual en una clase base, será virtual para todas las clases derivadas de ella.

- En C++ los constructores no pueden declararse como virtuales. (Porqué?)

Funciones virtuales puras

- En muchos casos, uno diseña una clase base que solamente servirá para implementar cierta funcionalidad común de las clases derivadas, pero en la práctica esta clase base nunca será instanciada.

- Es común que este tipo de clases base contengan métodos virtuales que no realizan ninguna tarea; simplemente sirven como prestanombres para poder llamar los métodos de las clases derivadas desde un apuntador a la clase base.

- Otra alternativa es agregar métodos virtuales (a la clase base) sin ningún tipo de implementación. Estos métodos se conocen como *funciones virtuales puras* o *puramente virtuales*, y se declaran asignándoles el valor cero. Por ejemplo:

```
class Animal {
public:
    virtual void Habla() = 0;
};
```

Clases base abstractas

- Una clase que contiene por lo menos una función virtual pura se conoce como *clase abstracta*.

- Una clase abstracta no puede instanciarse (no pueden crearse objetos de esa clase), pero pueden crearse apuntadores a una clase abstracta para acceder a los métodos de las clases derivadas mediante polimorfismo.

Ejemplo: clase abstracta

```
class Animal {
public:
    virtual void Habla() = 0;
};

class Perro : public Animal {
public:
    void Habla() { cout << "Guau" << endl; }
};

class Gato : public Animal {
public:
    void Habla() { cout << "Miau" << endl; }
};

int main() {
    Animal *a1 = new Perro;
    Animal *a2 = new Gato;
    a1->Habla();
    a2->Habla();
    delete a1;
    delete a2;
    return 0;
}
```