Introducción a las clases

Sofía Beatriz Pérez
Daniel Agustín Rosso

sperez@iua.edu.ar
drosso@iua.edu.ar

Centro Regional Univesitario Córdoba
Instituto Univeristario Areonáutico

Clase número 9 - Ciclo lectivo 2022

Agenda

Disclaimer

Los siguientes slides tienen el objetivo de dar soporte al dictado de la asignatura. De ninguna manera pueden sustituir los apuntes tomados en clases y/o la asistencia a las mismas.
Es importante mencionar que todos este material se encuentra en un proceso de mejora continua.
Si encuentra bugs, errores de ortografía o redacción, por favor repórtelo a sperez@iua.edu.ar y/o drosso@iua.edu.ar. También puede abrir issues en el repositorio de este link: ▸ infoI_IUA_GitLab

Hablemos de paradigmas...

La mayor parte de los lenguajes de programación de alto nivel, pueden clasificarse en una de tres categorías principales:

- Lenguaje de procedimientos
- Puros orientados a objetos
- Híbridos

El paradigma orientado a objetos está basado en tres pilares fundamentales:

- Encapsulación
- Herencia
- Polimorfismo

La principal característica distintiva de C++ frente a C es su soporte a la programación orientada a objetos.

Tipos de datos abstractos I

En búsqueda de una definición...

```
1  int a;
2  int b;
3  double c;
4  double d;
5  double e;
6  a=10;
7  b=a*4;
8  e=c%d; //Tiene sentido?
```

Tipos de datos abstractos II

En terminología de la computación, la combinación de datos y sus operaciones asociadas se define como **tipo de dato**.

Tipo de dato abstracto

Es simplemente un tipo definido por el usuario que define tanto un tipo de datos como el conjunto de operaciones que pueden ejecutarse con ellos.

En C++ un tipo de datos abstracto se lo conoce como una **clase**.

Construcción de clases I

Dentro de una clase se definen datos y funciones. En general, una clase consta de dos partes: una sección de declaración donde declaran los tipos de datos y las funciones para la clase. Y otra de implementación para definir las funciones cuyo prototipo se ha declarado en la sección anterior.

```
1 //Seccion de declaracion
2 class nombre_clase
3 {
4     miembros de datos   //Variables
5     miembros de funcion //Prototipos
6 };
7 //seccion de implementacion
```

Construcción de clases II

Tanto las variables como las funciones listadas en la sección de declaración, se conocen como **miembros de la clase**.

```
1  #include <iostream>
2  using namespace std;
3  //SECCION DE DECLARACION DE CLASE
4  class Fecha
5  {
6      private:
7          int mes;
8          int dia;
9          int anio;
10     public:
11         void setFecha (int, int, int);
12         void printFecha(void);
13 };
```

Construcción de clases III

```
14
15  //SECCION DE IMPLEMENTACION DE CLASE
16  void Fecha::setFecha(int d, int m, int a)
17  {
18      mes = m;
19      dia = d,
20      anio = a;
21  }
22
23  void Fecha::printFecha()
24  {
25      cout << "La fecha es " << dia << "/" << mes
26      << "/" << anio<< endl;
27  }
```

Construcción de clases IV

Las palabras reservadas public y private son especificadores de acceso:

Private

Significa ue sólo se puede tener acceso a los miembros de la clase que están debajo de esta palabra utilizando únicamente las funciones de la clase. También es posible utilizando funciones amigas, pero no está dentro del alcance de esta materia.

Public

Significa que estos métodos de clase pueden ser llamadas por los objetos y funciones que no estén en la clase.

Construcción de clases V

La sección de implementación de una clase es donde se describen las funciones miembro declaradas en la sección de declaración. De forma genérica:

```
1 tipoDevuelto nombreClase::nombreFuncion(params)
2 {
3     cuerpo de la funcion;
4 }
```

Es importante aclarar que mediante el uso de estas secciones sólo declaran a la clase; no se crea ninguna variable de este tipo de clase.

Instancia y clase

Clase

Una clase es un tipo de datos definido por el programador del cual pueden crearse objetos.

Objetos

Los objetos se crean de las clases; tienen la misma relación con las clases que las variables con los tipos de datos primitivos de C/C++. Los objetos también son conocidos como instancia de una clase y el proceso de crear un objeto nuevo se lo conoce como **instanciación** del objeto.

Cada vez que un objeto nuevo es instanciado, se crea un conjunto nuevo de datos pertenecientes al objeto.

El método constructor I

Una método constructor, es cualquier método que tenga el mismo nombre que su clase. Pueden definirse múltiplos constructores para cada clase en tanto sean distinguibles por el número y tipo de sus parámetros.
El método constructor es invocado **automáticamente** cuando se crea un objeto y, en general, el propósito es inicializar los miembros de datos de un objeto nuevo.

El método constructor, en general:

- Debe tener el mismo nombre que la clase a la que pertenece
- No debe devolver ningún tipo de datos (ni siquiera void)

El método constructor II

```
1 nombreClase::nombreClase(lista de parametros)
2 {
3   cuerpo de la funcion;
4 }
```

Si no se declara ningún constructor, el compilador crea un constructor por omisión.

El método constructor III

```
1  #include <iostream>
2  using namespace std;
3  class Fecha
4  {
5      private:
6          int mes;
7          int dia;
8          int anio;
9      public:
10         void setFecha (int, int, int);
11         void printFecha(void);
12         Fecha(int, int, int);
13 };
14
15
```

El método constructor IV

```
16
17  void Fecha::setFecha(int d, int m, int a)
18  {
19      mes = m;
20      dia = d,
21      anio = a;
22  }
23
24  void Fecha::printFecha()
25  {
26      cout << "La fecha es " << dia << "/" << mes << "/" <
27  }
28
29
30
```

El método constructor V

```
31  Fecha::Fecha(int d, int m, int a)
32  {
33      mes = m;
34      dia = d,
35      anio = a;
36      cout << "Se creo un objeto nuevo. "<< endl;
37      cout << "Con datos: " << dia <<
38      "/" << mes << "/" << anio<< endl;
39  }
40
41
42
43
44
45
```

El método constructor VI

```
46
47  int main()
48  {
49      Fecha f1(30,10,2022);
50      f1.printFecha();
51      Fecha f2(10,7,2022);
52      f2.printFecha();
53      return 0;
54  }
```

El método destructor I

La contraparte del método constructor es el **destructor**. Estos métodos tienen el mismo nombre que la clase pero precedido por una tilde ( ).

Los constructores son llamados de manera automática siempre que un objeto deja de existir. Se utilizan para "limpiar" los efectos indeseables que pudieran ser dejados por el objeto.
A diferencia de los constructores, sólo puede existir uno por clase.

El método destructor II

```
1  #include <iostream>
2  using namespace std;
3  class Fecha
4  {
5      private:
6          int mes;
7          int dia;
8          int anio;
9      public:
10         void setFecha (int, int, int);
11         void printFecha(void);
12         Fecha(int, int, int);
13         ~Fecha();
14 };
15
```

El método destructor III

```
16
17
18  void Fecha::setFecha(int d, int m, int a)
19  {
20      mes = m;
21      dia = d,
22      anio = a;
23  }
24
25  void Fecha::printFecha()
26  {
27      cout << "La fecha es " << dia << "/" << mes << "/" <
28  }
29
30
```

El método destructor IV

```
31
32  Fecha::Fecha(int d, int m, int a)
33  {
34      mes = m;
35      dia = d,
36      anio = a;
37      cout << "Se creo un objeto nuevo. "<< endl;
38      cout << "Con datos: " << dia <<
39      "/" << mes << "/" << anio<< endl;
40  }
41
42  Fecha::~Fecha()
43  {
44      cout << "Ejecutando el destructor "<< endl;
45  }
```

El método destructor V

```
46
47
48
49  int main()
50  {
51      Fecha f1(30,10,2022);
52      f1.printFecha();
53      Fecha f2(10,7,2022);
54      f2.printFecha();
55      return 0;
56  }
```

Pasando objetos a funciones I

Los objetos pueden ser pasados a las funciones de la misma manera que cualquier otro tipo de variables. Es por esto, que los objetos son pasados a las funciones **por valor**.

Pero si es por valor... ¿qué pasa con los constructores y destructores?
Analicemos el siguiente ejemplo:

Pasando objetos a funciones II

```
1  #include <iostream>
2  using namespace std;
3  class my_class
4  {
5      private:
6          int data;
7      public:
8          void setData (int);
9          void printData(void);
10         ~my_class();
11         my_class(int);
12 };
13
14
15
```

Pasando objetos a funciones III

```
16  my_class::my_class(int d)
17  {
18      data=d;
19      cout <<"Construyendo"<< endl;
20  }
21
22
23  my_class::  ~my_class()
24  {
25      cout <<"Destruyendo"<< endl;
26  }
27
28
29
30
```

Pasando objetos a funciones IV

```
31  void my_class::setData(int d)
32  {
33      data=d;
34  }
35
36  void my_class::printData(void)
37  {
38      cout <<"Data vale"<< endl;
39      cout <<data<< endl;
40  }
41
42
43  void function(my_class);
44
45
```

Pasando objetos a funciones V

```
46
47  int main()
48  {
49      my_class a(10);
50      a.printData();
51      function(a);
52      a.printData();
53  }
54
55
56
57
58
59
60
```

Pasando objetos a funciones VI

```
61  void function(my_class obj)
62  {
63      cout <<"En la funcion"<< endl;
64      obj.printData();
65      obj.setData(13);
66      obj.printData();
67  }
```

Pasando objetos a funciones VII

- Hay una sola llamada a la función constructor. Este evento sucede cuando el objeto "a" es creado
- Hay dos llamadas a la función destructor
- Cuando un objeto se pasa a una función **una copia** del mismo es enviada a la función. Para realizar esto, se llama al constructor de copia. Cuando este no está definido, el compilador implementa uno por default.

Arreglo de objetos I

En C++ es posible crear arreglos de objetos. La sintaxis a utilizar es exactamente la misma que para cualquier otro tipo de variable.

```
1  #include <iostream>
2  using namespace std;
3  class my_class
4  {
5      private:
6          int data;
7      public:
8          void setData (int);
9          void printData(void);
10 };
11
```

Arreglo de objetos II

```
12  void my_class::setData(int d)
13  {
14      data=d;
15  }
16
17  void my_class::printData(void)
18  {
19      cout <<"Data vale "<<data<< endl;
20  }
21
22
23
24
25
26
```

Arreglo de objetos III

```
27  int main(void)
28  {
29      my_class a[10];
30      for(int ii=0;ii<10;ii++)
31      {
32          a[ii].setData(ii);
33      }
34      for(int ii=0;ii<10;ii++)
35      {
36          a[ii].printData();
37      }
38      return(0);
39  }
```

Punteros a objetos I

Al igual que ocurre con otro tipo de variables, es posible tener punteros a objetos. Cuando se accede a una clase utilizando un puntero a ella, se utiliza el operador $(->)$ en lugar del operador $(.)$

```
1 #include <iostream>
2 using namespace std;
3 class my_class
4 {    private:
5          int data;
6      public:
7          void setData (int);
8          void printData(void);
9 };
```

Punteros a objetos II

```
10  void my_class::setData(int d)
11  {
12      data=d;
13  }
14  void my_class::printData(void)
15  {
16      cout <<"Data vale "<<data<< endl;
17  }
18
19
20
21
22
23
24
```

Punteros a objetos III

```
25  int main(void)
26  {
27      my_class a;
28      my_class *p;
29      a.setData(1234);
30      a.printData();
31      p=&a;
32      p->printData();
33      p->setData(12);
34      p->printData();
35
36      return(0);
37  }
```

Aritmética de punteros aplicada a arreglos de objetos I

Como se ha estudiado, cuando un puntero es incrementado, este apunta al siguiente elemento de su tipo. Es decir que toda la aritmética de punteros es relativa al tipo base del puntero (es relativo también al tipo de dato al cual apunta).

```
1 #include <iostream>
2 using namespace std;
3 class my_class
4 {    private:
5          int data;
6      public:
7          void setData (int);
8          void printData(void);
9 };
```

Aritmética de punteros aplicada a arreglos de objetos II

```
10
11  void my_class::setData(int d)
12  {
13      data=d;
14  }
15
16  void my_class::printData(void)
17  {
18      cout <<"Data vale "<<data<< endl;
19  }
20
21
22
23
24
```

Aritmética de punteros aplicada a arreglos de objetos III

```
25  int main(void)
26  {
27      my_class a[10];
28      my_class *p;
29      for(int ii=0;ii<10;ii++)
30      {
31          a[ii].setData(ii);
32      }
33      p=a;
34      for(int ii=0;ii<10;ii++)
35      {
36          (p+ii)->printData();
37      }
38      return(0);
39  }
```

El puntero this I

Cuando una función miembro de una clase es invocada, automáticamente es pasado a ella un argumento implícito que apunta al objecto que está invocando al método. Este puntero es denominado **puntero this**.

El puntero this II

```
1  #include <iostream>
2  using namespace std;
3  class my_class
4  {
5      private:
6          int data;
7      public:
8          void setData (int);
9          void printData(void);
10         ~my_class();
11         my_class(int);
12 };
13
14
15
```

El puntero this III

```
16  my_class::my_class(int d)
17  {
18      this->datadata=d;
19      cout <<"Construyendo"<< endl;
20  }
21
22
23  my_class::  ~my_class()
24  {
25      cout <<"Destruyendo"<< endl;
26  }
27
28
29
30
```

El puntero this IV

```
31  void my_class::setData(int d)
32  {
33      this->data=d;
34  }
35
36  void my_class::printData(void)
37  {
38      cout <<"Data vale"<< endl;
39      cout <<this->data<< endl;
40  }
41
42
43
44
45
```

El puntero this V

```
46
47  int main()
48  {
49      my_class a(10);
50      a.printData();
51      a.setData(12);
52      a.printData();
53  }
```

Hasta aquí parece no tener demasiado sentido la utilización del puntero this. El mismo cobra vital importancia cuando se utilizan sobrecarga de operadores

¡Muchas gracias!

Consultas:

sperez@iua.edu.ar
drosso@iua.edu.ar