Contenidos

Clases

Definition

class tipo de datos compuesto que puede almacenar datos y funciones.

Las clases son una forma de agrupar datos y funciones de manera de crear tipos de datos que *encapsulan* responsabilidades.

Clases

Definition

class tipo de datos compuesto que puede almacenar datos y funciones.

Las clases son una forma de agrupar datos y funciones de manera de crear tipos de datos que *encapsulan* responsabilidades.
Ejemplo:

```
class Vector2
{
public:
  double x, y;
};
```

Este código define una clase `Vector2` la cual tiene dos variables miembro, `x` y `y`, del tipo `double` y que son accesibles de forma pública.

Clases

```
int main()
{
  Vector2 r;
  r.x = 3.0;
  r.y = 10.0;
  std::cout << "x: " << r.x << " y: " << r.y;
}
```

Acá creamos una instancia de un *objecto* del tipo `Vector2`. Usamos el operador (`.`) para acceder a las variables públicas del vector, `r`.

Clases

```
int main()
{
  Vector2 r;
  r.x = 3.0;
  r.y = 10.0;
  std::cout << "x: " << r.x << " y: " << r.y;
}
```

Acá creamos una instancia de un *objecto* del tipo `Vector2`. Usamos el operador (`.`) para acceder a las variables públicas del vector, `r`.

Definiciones

objects	una instancia de una clase (ej, plano de una casa)
instanciar	crear una instancia de una clase.
variable miembro	una variable que está contenida en una clase.

Métodos

Al igual que las variables miembro, una clase puede tener funciones miembro.

Definition

función miembro (o *método*) una función que está contenida en una clase.

Métodos

Al igual que las variables miembro, una clase puede tener funciones miembro.

Definition

función miembro (o *método*) una función que está contenida en una clase.

Ejemplo:

```
class Vector2
{
public:
  double length()
  { return sqrt(x * x + y * y); }

  double x, y;
};
int main()
{
  Vector2 r;
  r.x = 3;  r.y = 4;
  std::cout << "|r| = " << r.length() << "\n";
}
```

Output: `|r| = 5`

Clases

Ejemplo Completo

```
class Vector2
{
public:
  double length()
  { return std::sqrt(x * x + y * y); }

  void sub(const Vector2 toSub)
  { x -= toSub.x; y -= toSub.y; }

  void mul(const double k)
  { x *= k; y *= k; }

  double dot(const Vector2 b)
  { return x * b.x + y * b.y; }

  void printCoords()
  { std::cout<< x << " "<< y << "\n";}

  double x, y;
};
```

Clases

Ejemplo Completo

```
class Vector2
{
public:
  double length()
  { return std::sqrt(x * x + y * y); }

  void sub(const Vector2 toSub)
  { x -= toSub.x; y -= toSub.y; }

  void mul(const double k)
  { x *= k; y *= k; }

  double dot(const Vector2 b)
  { return x * b.x + y * b.y; }

  void printCoords()
  { std::cout<< x << " "<< y << "\n";}

  double x, y;
};
```

```
int main()
{
  Vector2 r1, r2;
  r1.x = 2; r1.y = 1;
  r2.x = 4; r2.y = 10;

  std::cout << "r1: ";
  r1.printCoords();
  std::cout << "r2: ";
  r2.printCoords();
  std::cout << "Performing
       r2.sub(r1)\n";
  r2.sub(r1);
  std::cout << "r2: ";
  r2.printCoords();

  return 0;
}
```

Output:
```
r1: 2 1
r2: 4 10
Performing r2.sub(r1)
r2: 2 9
```

Acceso

El kewyword `public` especifica el tipo de acceso a la variable o método.

Definition

Tipo de acceso le dice al compilador quien (ej, qué partes del código) tienen acceso a los miembros de la clase.

Existen tres tipos de acceso:

Acceso

El kewyword `public` especifica el tipo de acceso a la variable o método.

Definition

Tipo de acceso le dice al compilador quien (ej, qué partes del código) tienen acceso a los miembros de la clase.

Existen tres tipos de acceso:

`publico` los miembros de la clase son visibles para todos

Acceso

El kewyword `public` especifica el tipo de acceso a la variable o método.

Definition

Tipo de acceso le dice al compilador quien (ej, qué partes del código) tienen acceso a los miembros de la clase.

Existen tres tipos de acceso:

`publico` los miembros de la clase son visibles para todos

`privado` los miembros de la clase solamente son visibles para otros miembros de la misma clase u otra clase del tipo *friend*.

Acceso

El kewyword `public` especifica el tipo de acceso a la variable o método.

Definition

Tipo de acceso le dice al compilador quien (ej, qué partes del código) tienen acceso a los miembros de la clase.

Existen tres tipos de acceso:

`publico` los miembros de la clase son visibles para todos

`privado` los miembros de la clase solamente son visibles para otros miembros de la misma clase u otra clase del tipo *friend*.

`protegido` los miembros de la clase son visibles para otros miembros de la misma clase o para clases derivadas.

Acceso

El kewyword `public` especifica el tipo de acceso a la variable o método.

Definition

Tipo de acceso le dice al compilador quien (ej, qué partes del código) tienen acceso a los miembros de la clase.

Existen tres tipos de acceso:

- `publico` los miembros de la clase son visibles para todos
- `privado` los miembros de la clase solamente son visibles para otros miembros de la misma clase u otra clase del tipo *friend*.
- `protegido` los miembros de la clase son visibles para otros miembros de la misma clase o para clases derivadas.

Por defecto las clases se definen con accesso `privado` para todos los miembros.

Encapsulación

El objetivo de un buen diseño orientado a objetos es dividir el problema en trozos de programa que pueden ser desarrollados de manera independiente. Esto permite lograr *modularidad* ya que los datos de la clase pueden no estar visibles pero si las funciones que las utilizan. Esto se denomina *encapsulación*.

Encapsulación

El objetivo de un buen diseño orientado a objetos es dividir el problema en trozos de programa que pueden ser desarrollados de manera independiente. Esto permite lograr *modularidad* ya que los datos de la clase pueden no estar visibles pero si las funciones que las utilizan. Esto se denomina *encapsulación*.

```
class Vector2
{
public:
  double getX()
  { return x; }
  double getY()
  { return y; }
private:
  double x, y;
};
```

Encapsulación

El objetivo de un buen diseño orientado a objetos es dividir el problema en trozos de programa que pueden ser desarrollados de manera independiente. Esto permite lograr *modularidad* ya que los datos de la clase pueden no estar visibles pero si las funciones que las utilizan. Esto se denomina *encapsulación*.

```
class Vector2
{
public:
  double getX()
  { return x; }
  double getY()
  { return y; }
private:
  double x, y;
};
```

```
class Vector2
{
public:
  double getX()
  { return coords[0]; }
  double getY()
  { return coords[1]; }
private:
  double coords[2];
};
```

Interfaz común, diferente implementación.

Encapsulación

La encapsulación separa los detalles de la implementación de la forma en que se usa la clase. Esto tiene las siguientes ventajas:

Encapsulación

La encapsulación separa los detalles de la implementación de la forma en que se usa la clase. Esto tiene las siguientes ventajas:

- Entrega mayor flexibilidad al momento de realizar mejoras ya que no es necesario modificar otras partes del código

Encapsulación

La encapsulación separa los detalles de la implementación de la forma en que se usa la clase. Esto tiene las siguientes ventajas:

- Entrega mayor flexibilidad al momento de realizar mejoras ya que no es necesario modificar otras partes del código
- Hace el código mas mantenibless.

Encapsulación

La encapsulación separa los detalles de la implementación de la forma en que se usa la clase. Esto tiene las siguientes ventajas:

- Entrega mayor flexibilidad al momento de realizar mejoras ya que no es necesario modificar otras partes del código
- Hace el código mas mantenibless.
- Se facilita el desarrollo de funcionalidades complejas, ya que en un principio solo es necesario definir las interfaces y luego se desarrollan las implementaciones.

Encapsulación

La encapsulación separa los detalles de la implementación de la forma en que se usa la clase. Esto tiene las siguientes ventajas:

- Entrega mayor flexibilidad al momento de realizar mejoras ya que no es necesario modificar otras partes del código
- Hace el código mas mantenibless.
- Se facilita el desarrollo de funcionalidades complejas, ya que en un principio solo es necesario definir las interfaces y luego se desarrollan las implementaciones.

Do

Las variables miembrodebieran ser `privadas`. El acceso a los valores deberia ser a través de métodos de acceso y seteo (GET/SET).

Métodos de Acceso

Definiciones

getter un método que accede al valor de una variable interna.

setter un método que setea el valor de una variable interna.

Métodos de Acceso

Definiciones

getter un método que accede al valor de una variable interna.

setter un método que setea el valor de una variable interna.

```
void setX(const double newX)
{ x = newX; }
void setY(const double newY)
{ y = newY; }
```

Constructores

Una clase podría inicializar variables o alojar memoria dinámica una vez que es instanciada. Esto se realiza en el constructor de la clase.

Definition

constructor es una función miembro de la clase que tiene el mismo nombre de la clase y que no retorna ningún tipo de dato.

Constructores

Una clase podría inicializar variables o alojar memoria dinámica una vez que es instanciada. Esto se realiza en el constructor de la clase.

Definition

constructor es una función miembro de la clase que tiene el mismo nombre de la clase y que no retorna ningún tipo de dato.

```
class Vector2
{
public:
  Vector2(const double x0,
    const double y0)
  {
    x = x0; y = y0;
  }
  /*..and the rest..*/
private:
  double x, y;
};
```

```
int main()
{
  Vector2 r1(3, 14);
  r1.printCoords();
  return 0;
}
```

Output: `3 10`

Constructores por defecto

Definition

constructor por defecto — un constructor que no toma ningún argumento y que se define automáticamente.

Al momento de definir un constructor, el constructor por defecto no está disponible!

Constructores por defecto

Definition

constructor por defecto un constructor que no toma ningún argumento y que se define automáticamente.

Al momento de definir un constructor, el constructor por defecto no está disponible!

```
int main()
{
  Vector2 r1; // error 'Vector2': no appropriate
  return 0;         // default constructor available
}
```

Sobrecarga de constructores

Es posible tener más de un constructor, a esto se le llama *sobrecarga de constructor*. Por ejemplo:

```
class Vector2
{
public:
  Vector2()
  {
    x = 0.0; y = 0.0;
  }
  Vector2(const double x0,
    const double y0)
  {
    x = x0;  y = y0;
  }
private:
  double x, y;
};
```

```
int main()
{
  Vector2 r1(3, 14);
  Vector2 r2; // No brackets!
  std::cout << "r1: "
  r1.printCoords();
  std::cout << "r2: "
  r2.printCoords();
  return 0;
}
```

Output:
```
r1: 3 10
r2: 0 0
```

Destructores

Tal como se vió cuando usamos memoria dinámica, una clase podría requerir liberar recursos una vez que ya no son necesarios. Para esto se utiliza un destructor.

Definition

destructor Una función miembro que tiene el mismo nombre que la clase pero usa el prefijo ~, no tiene argumentos ni tipo de salida.

Destructores

Tal como se vió cuando usamos memoria dinámica, una clase podría requerir liberar recursos una vez que ya no son necesarios. Para esto se utiliza un destructor.

Definition

destructor Una función miembro que tiene el mismo nombre que la clase pero usa el prefijo ~, no tiene argumentos ni tipo de salida.

```
class ClassicalSpinString {
public:
  ClassicalSpinString(const int length)
  {
    vectors = new Vector2[length];
  }
  ~ClassicalSpinString()
  {
    delete[] vectors; // Have to delete to avoid leak!
  }
private:
  Vector2 * vectors;
};
```

Destructores

Un destructor se llama cuando el alcance del objeto ha terminado.

Destructores

Un destructor se llama cuando el alcance del objeto ha terminado.

```
int main()
{
  bool calcSpinStringProperties = true;
  // do stuff...
  if(calcSpinStringProperties)
  {
    SpinString spins;
    // do stuff with spins...
    // ...report results.
  } // <- Here spins is destructed
  return 0;
}
```

Destructores

Un destructor se llama cuando el alcance del objeto ha terminado.

```
int main()
{
  bool calcSpinStringProperties = true;
  // do stuff...
  if(calcSpinStringProperties)
  {
    SpinString spins;
    // do stuff with spins...
    // ...report results.
  } // <- Here spins is destructed
  return 0;
}
```

Do

Aunque parezca innecesario, siempre es bueno usar destructores para limpiar memoria no utilizada.

Punteros a clases

```
int main()
{
  Vector2 * r1 = new Vector2;
  (*r1).printCoords();

  delete r1; // Clean up
}
```

Punteros a clases

```
int main()
{
  Vector2 * r1 = new Vector2;
  (*r1).printCoords();

  delete r1; // Clean up
}
```

La sintaxis `(*name).member` se vuelve un poco engorrosa, por lo tanto C++ provee una sintaxis alternativa (->):

```
int main()
{
  Vector2 * r1 = new Vector2;
  r1->printCoords();

  delete r1; // Clean up
}
```

Sobrecarga de operadores

C++ provee operadores para algunos tipos de datos, por ejemplo.:

```
int a = 5, b = 10;
int c = a + b;

bool haveMyCake = true, eatIt = true;
bool haveMyCakeAndEatIt = haveMyCake && eatIt;
```

Qué pasa si queremos definir operadores para nuestras clases?

Sobrecarga de operadores

C++ provee operadores para algunos tipos de datos, por ejemplo.:

```
int a = 5, b = 10;
int c = a + b;

bool haveMyCake = true, eatIt = true;
bool haveMyCakeAndEatIt = haveMyCake && eatIt;
```

Qué pasa si queremos definir operadores para nuestras clases?

```
Vector2 r1(3, 14), r2(23, 1);
Vector2 r12 = r2 - r1;
```

Sobrecarga de operadores

Qué debiera suceder al aplicar el operador menos a un objeto de la clase `Vector2`?

- Tomar otro objeto de la clase `Vector2` como parámetro.

Sobrecarga de operadores

Qué debiera suceder al aplicar el operador menos a un objeto de la clase `Vector2`?

- Tomar otro objeto de la clase `Vector2` como parámetro.
- Retornar un objeto de la clase `Vector2`.

Sobrecarga de operadores

Qué debiera suceder al aplicar el operador menos a un objeto de la clase `Vector2`?

- Tomar otro objeto de la clase `Vector2` como parámetro.
- Retornar un objeto de la clase `Vector2`.

```
class Vector2
{
public:
  Vector2 operator -(Vector2 toSub)
  {
    return Vector2(x - toSub.getX(), y - toSub.getY());
  }
/* ..other stuff */
};
```

Sobrecarga de operadores

Qué sucede al utilizar el operador? : `Vector r12 = r2 - r1;`

Sobrecarga de operadores

Qué sucede al utilizar el operador? : `Vector r12 = r2 - r1;`

1. El compilador ve que se usó el `operador -` en un objeto que define este operador. En la práctica es igual que llamar a un método de la siguiente forma:

```
r2.operator -(r1);
```

Sobrecarga de operadores

Qué sucede al utilizar el operador? : `Vector r12 = r2 - r1;`

1. El compilador ve que se usó el `operador -` en un objeto que define este operador. En la práctica es igual que llamar a un método de la siguiente forma:

```
r2.operator -(r1);
```

2. El operador − sobre el objeto `r2` usa al objeto `r1` como parámetro.

Sobrecarga de operadores

Qué sucede al utilizar el operador? : `Vector r12 = r2 - r1;`

1. El compilador ve que se usó el `operador -` en un objeto que define este operador. En la práctica es igual que llamar a un método de la siguiente forma:

```
r2.operator -(r1);
```

2. El operador − sobre el objeto `r2` usa al objeto `r1` como parámetro.
3. La salida del método/operador es otro objeto de la clase `Vector2` y se copia en `r12`.