Informatics 3.
Lecture 4: Classes

Kristóf Kovács

Budapest University of Technology and Economics

2024-03-21

Story

Some programming method:

- Programming at the beginning of time
- Structured
- Modular
- Functional
- Object-oriented
- ...

Story

Some programming method:

- Programming at the beginning of time
- Structured
- Modular
- Functional
- Object-oriented
- ...

At the beginning of time:

- No visible structure
- Only the author understands it
- Makes the author irreplaceable

Structured programming

- Structured data, operations
- Readable, portable
- Not efficient enough, hard to extend

Structured programming

- Structured data, operations
- Readable, portable
- Not efficient enough, hard to extend


```
AsteroidVector asteroidDestroyed(Asteroid ast) {
  int n = ast.size; // better name: numberOfNewAsteroids
  AsteroidVector newAsteroids = createAsteroidVector(n);
  int i;
  for(i = 0; i < n; i++) {
    newAsteroids.push(instantiateFragmentAsteroid(ast.pos, n));
    ...
  }
  return newAsteroids;
}
```

Modular programming

- Independent modules
- Modules connected through interfaces
- Independently compilable, testable
- The modules directly don't know about each other
- The data types show up on the interfaces
- Hard to replace a data type

Modular programming

- Independent modules
- Modules connected through interfaces
- Independently compilable, testable
- The modules directly don't know about each other
- The data types show up on the interfaces
- Hard to replace a data type

```
AsteroidHandler.c
GameHandler.c
PlayerController.c
Physics.c
Renderer.c
...
```

Object-oriented solution

```
class Asteroid: public Renderable, public Destroyable... {
private:
   int size;
   Position pos;
   ...
public:
   AsteroidContainer destroy() {
      AsteroidContainer newAsteroids = new AsteroidVector(size);
      for(int i = 0; i < size; i++) {
          Asteroid a = AsteroidFactory.createFragmentAsteroid(
                                pos, size);
          ...
      }
      return newAsteroids;
   }
   void render(Renderer rend) {
      ...
   }
};
```

Object-oriented programming

- Independence is part of the language
- Breakdown into simpler tasks
- The program can be designed without deeply going into the algorithms that will need to be used
- Abstract parts connected through interfaces
- Object:

Object-oriented programming

- Independence is part of the language
- Breakdown into simpler tasks
- The program can be designed without deeply going into the algorithms that will need to be used
- Abstract parts connected through interfaces
- Object:
  - Easily modifiable
  - Genaralizable
  - Stores its own state
  - Hides its own data structure and its algorithms
  - The more abstract the better, using it shouldn't require knowledge of its workings
  - PI: Complex number, Linked list, Binary tree

Reference

Swap function with our current knowledge

```
void csere(int* a, int* b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main(void) {
  int x = 2;
  int y = 10;

  csere(&x, &y);
  cout << "x: " << x << endl;
  cout << "y: " << y << endl;

  return 0;
}
```

Reference

With reference

```
void csere(int* a, int* b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main(void) {
  int x = 2;
  int y = 10;

  csere(&x, &y);
  cout << "x: " << x << endl;
  cout << "y: " << y << endl;

  return 0;
}
```

```
void csere(int& a, int& b) {
  int temp = a;
  a = b;
  b = temp;
}

int main(void) {
  int x = 2;
  int y = 10;

  csere(x, y);
  cout << "x: " << x << endl;
  cout << "y: " << y << endl;

  return 0;
}
```

Referencia

- Using a reference to transfer values to a function uses the actual variable, as if we used a pointer.

Referencia

- Using a reference to transfer values to a function uses the actual variable, as if we used a pointer.
- You can use references outside of parameters too:

```
int a = 5;
int& b = a;
```

Referencia

- Using a reference to transfer values to a function uses the actual variable, as if we used a pointer.
- You can use references outside of parameters too:

```
int a = 5;
int& b = a;
```

- The "pointer" of a reference cannot be modified. When created it has to be set to a variable and this can no longer be changed.

Complex number with structures

Let's try to write a complex number data type with our current knowledge.

Complex number with structures

Let's try to write a complex number data type with our current knowledge.

```
typedef struct comp {
  float re;
  float im;
} Complex;

Complex add(Complex first, Complex other) {
  Complex ret;
  ret.re = first.re + other.re;
  ret.im = first.im + other.im;
  return ret;
}
```

Complex number with structures

Not too bad yet. Let's try to implement multiplication, absolute value and printing as well.

Complex number with structures

Not too bad yet. Let's try to implement multiplication, absolute value and printing as well.

```
Complex times(Complex first, Complex other) {
  Complex ret;
  ret.re = first.re * other.re - first.im * other.im;
  ret.im = first.re * other.im + first.im * other.re;
  return ret;
}

float abs(Complex first) {
  return sqrt(first.re * first.re + first.im * first.im);
}

void print(Complex first) {
  cout << first.re << " + " << first.im << "i" << endl;
}
```

Complex number with structures, main

```
int main(void) {
  Complex a;
  a.re = 0; a.im = 0;
  Complex b;
  b.re = 1; b.im = 2;
  Complex c = times(a, b);

  print(a);
  print(b);
  print(c);

  Complex d = add(b, c);
  print(d);

  cout << abs(b) << endl;

  return 0;
}
```

Complex number with structures, main

```
int main(void) {
  Complex a;
  a.re = 0; a.im = 0;
  Complex b;
  b.re = 1; b.im = 2;
  Complex c = times(a, b);

  print(a);
  print(b);
  print(c);

  Complex d = add(b, c);
  print(d);

  cout << abs(b) << endl;

  return 0;
}
```

The problem is with the readability.

Can't be understood at a glance.

Complex number with structures, main

```
int main(void) {
  Complex a;
  a.re = 0; a.im = 0;
  Complex b;
  b.re = 1; b.im = 2;
  Complex c = times(a, b);

  print(a);
  print(b);
  print(c);

  Complex d = add(b, c);
  print(d);

  cout << abs(b) << endl;

  return 0;
}
```

The problem is with the readability.

Can't be understood at a glance.

We want to make complex numbers work as if they were for example **floats**.

Complex number with classes

```
class Complex {
private:
  float re;
  float im;
public:
  Complex();
  Complex(const Complex& other);
  Complex(float r);
  Complex(float r, float i);

  Complex add(Complex& other);
  Complex times(Complex& other);
  float abs();

  void print();

  ~Complex();
};
```

New keywords

New keywords:

- **class** is used to create a new class.
- **private** data members/methods set to private can only be accessed by their own class (hidden data types/algorithms)
- **public** data members/methods set to public can be accessed from outside

Complex number with classes, main

```
int main(void) {
  Complex a;
  Complex b = Complex(1,2);
  Complex c = a.times(b);

  a.print();
  b.print();
  c.print();

  (b.add(c)).print();

  cout << b.abs() << endl;

  return 0;
}
```

Complex number with classes, main

```
int main(void) {
  Complex a;
  Complex b = Complex(1,2);
  Complex c = a.times(b);

  a.print();
  b.print();
  c.print();

  (b.add(c)).print();

  cout << b.abs() << endl;

  return 0;
}
```

A bit more readable than the solution with **structs**.

Complex number with classes, main

```
int main(void) {
  Complex a;
  Complex b = Complex(1,2);
  Complex c = a.times(b);

  a.print();
  b.print();
  c.print();

  (b.add(c)).print();

  cout << b.abs() << endl;

  return 0;
}
```

A bit more readable than the solution with **structs**.

Not perfect yet.

Complex number with classes, main

```
int main(void) {
  Complex a;
  Complex b = Complex(1,2);
  Complex c = a.times(b);

  a.print();
  b.print();
  c.print();

  (b.add(c)).print();

  cout << b.abs() << endl;

  return 0;
}
```

A bit more readable than the solution with **structs**.

Not perfect yet.

Later we will be able to redefine operators like + and *. We'll also be able to "teach" **cout** how to print a given object.

Data members, constructors

- Data members and methods are accessed through the . (dot) operator.

Data members, constructors

- Data members and methods are accessed through the . (dot) operator.
- When a method has no parameters the parentheses still need to be used.

Data members, constructors

- Data members and methods are accessed through the . (dot) operator.
- When a method has no parameters the parentheses still need to be used.
- When creating a new object, we do it with constructors.

Data members, constructors

- Data members and methods are accessed through the . (dot) operator.
- When a method has no parameters the parentheses still need to be used.
- When creating a new object, we do it with constructors.
- This constructor takes two **float**s.

```
Complex(float r, float i);
```

Data members, constructors

- Data members and methods are accessed through the . (dot) operator.
- When a method has no parameters the parentheses still need to be used.
- When creating a new object, we do it with constructors.
- This constructor takes two **float**s.

```
Complex(float r, float i);
```

- This is the **default constructor**. This is called when we create an object without inicializing it.

```
Complex();
```

Data members, constructors

- Data members and methods are accessed through the . (dot) operator.
- When a method has no parameters the parentheses still need to be used.
- When creating a new object, we do it with constructors.
- This constructor takes two **float**s.

```
Complex(float r, float i);
```

- This is the **default constructor**. This is called when we create an object without inicializing it.

```
Complex();
```

- This is the **copy constructor**. It's called when we copy an object of this type (for example when passed to a function).

```
Complex(const Complex& other);
```

Data members, constructors

- Data members and methods are accessed through the . (dot) operator.
- When a method has no parameters the parentheses still need to be used.
- When creating a new object, we do it with constructors.
- This constructor takes two **float**s.

```
Complex(float r, float i);
```

- This is the **default constructor**. This is called when we create an object without inicializing it.

```
Complex();
```

- This is the **copy constructor**. It's called when we copy an object of this type (for example when passed to a function).

```
Complex(const Complex& other);
```

- Constructors have no return values.

Complex number with classes, methods

```
Complex::Complex() {
  re = 0;
  im = 0;
}

Complex::Complex(const Complex& other) {
  this->re = other.re;
  this->im = other.im;
}

Complex::Complex(float r) {
  re = r;
  im = 0;
}

Complex::Complex(float r, float i) {
  re = r;
  im = i;
}
```

Data members and constructors

- Data members can be accessed through their name.

Data members and constructors

- Data members can be accessed through their name.
- But we can also refer to the given object with the keyword **this**, it is a pointer to the given object (the object on which the method was called, or the object that's being constructed).

Data members and constructors

- Data members can be accessed through their name.
- But we can also refer to the given object with the keyword **this**, it is a pointer to the given object (the object on which the method was called, or the object that's being constructed).
- The **const** keyword will be discussed later.

Data members and constructors

- Data members can be accessed through their name.
- But we can also refer to the given object with the keyword **this**, it is a pointer to the given object (the object on which the method was called, or the object that's being constructed).
- The **const** keyword will be discussed later.
- It's important to notice that the copy constructor uses a reference to the other object:

```
Complex(const Complex& other);
```

Data members and constructors

- Data members can be accessed through their name.
- But we can also refer to the given object with the keyword **this**, it is a pointer to the given object (the object on which the method was called, or the object that's being constructed).
- The **const** keyword will be discussed later.
- It's important to notice that the copy constructor uses a reference to the other object:

```
Complex(const Complex& other);
```

- If it wasn't a reference then it would lead to an infinite recursion. Copying an object would call the copy constructor which would need to copy the given object calling another copy constructor and so on.

Complex number with classes, methods

```
Complex Complex::add(Complex& other) {
  return Complex(this->re + other.re, this->im + other.im);
}

Complex Complex::times(Complex& other) {
  float real = this->re * other.re - this->im * other.im;
  float imag = this->re * other.im + this->im * other.re;
  return Complex(real, imag);
}

float Complex::abs() {
  return sqrt(this->re * this->re + this->im * this->im);
}

void Complex::print() {
  cout << re << " + " << im << "i" << endl;
}
Complex::~Complex() {
}
```

Namespaces, destructor

- The :: operator lets us enter a class' namespace. This is why we wrote **Complex::add**, because the **add** function doesn't exist. The **add** function we're thinking of is actually a method of the **Complex** class.

Namespaces, destructor

- The **::** operator lets us enter a class' namespace. This is why we wrote **Complex::add**, because the **add** function doesn't exist. The **add** function we're thinking of is actually a method of the **Complex** class.
- The **~Complex()** is a destructor. This is called when the object is deleted. This can be automatic, when the block where the object was created is at its end or manual with the **delete** keyword when we created the object dynamically.

Namespaces, destructor

- The **::** operator lets us enter a class' namespace. This is why we wrote **Complex::add**, because the **add** function doesn't exist. The **add** function we're thinking of is actually a method of the **Complex** class.
- The **~Complex()** is a destructor. This is called when the object is deleted. This can be automatic, when the block where the object was created is at its end or manual with the **delete** keyword when we created the object dynamically.
- The destructor is important when the class handles dynamically allocated data. For example a linked list.

What's next

```
int main(void) {
  Complex a;
  Complex b = Complex(1,2);
  Complex c = a * b;

  cout << a << endl;
  cout << b << endl;
  cout << c << endl;

  cout << b + c << endl;

  cout << b.abs() << endl;

  return 0;
}
```

Pop quiz questions

- Show an example of a reference
- What is the **this** keyword?
- What is the default constructor?
- What is the copy constructor?
- What is the destructor?