Informatika 3.
4. előadás: Osztályok

Kovács Kristóf

Budapesti Műszaki és Gazdaságtudományi Egyetem

2024-03-19

Mese

Néhány programozási módszer:

- Idők kezdetén való programozás
- Struktúrált
- Moduláris
- Funkcionális
- Objektum-orientált
- ...

Mese

Néhány programozási módszer:

- Idők kezdetén való programozás
- Struktúrált
- Moduláris
- Funkcionális
- Objektum-orientált
- ...

Idők kezdetén:

- Ránézésre semmi szerkezet
- Csak az érti aki írta
- Jóformán lecserélhetetlenné teszi a programozót

Struktúrált programozás

- Struktúrált adatok, műveletek
- Áttekinthetőbb, hordozhatóbb
- Nem túl hatékon, nehezen cserélhető

Struktúrált programozás

- Struktúrált adatok, műveletek
- Áttekinthetőbb, hordozhatóbb
- Nem túl hatékon, nehezen cserélhető

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

Moduláris programozás

- Egy modul egy önálló egység
- Interfészekkel kapcsolódik más modulokhoz
- Önállóan fordítható, tesztelhető
- A modulok nem tudnak egymás működéséről
- Az adattípusok megjelennek az interfészeken
- Nehezen cserélhető egy adattípus

Moduláris programozás

- Egy modul egy önálló egység
- Interfészekkel kapcsolódik más modulokhoz
- Önállóan fordítható, tesztelhető
- A modulok nem tudnak egymás működéséről
- Az adattípusok megjelennek az interfészeken
- Nehezen cserélhető egy adattípus

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

Objektum-orientált megoldás

```
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) {
        ...
    }
};
```

Objektum-orientált programozás

- Egységbe zárás
- Felbontás egyszerűbb feladatokra
- Megtervezhető a program anélkül, hogy végig kellene gondolni a megírásához szükséges algoritmusokat
- Interfészek kötik össze az absztrakt részeket
- Objektum:

Objektum-orientált programozás

- Egységbe zárás
- Felbontás egyszerűbb feladatokra
- Megtervezhető a program anélkül, hogy végig kellene gondolni a megírásához szükséges algoritmusokat
- Interfészek kötik össze az absztrakt részeket
- Objektum:
  - Könnyen módosítható
  - Általánosítható
  - Tárolja a saját állapotát
  - Elrejti a belső adatszerkezetét, a műveleteinek algoritmusait
  - Minél absztraktabb annál jobb, a használathoz ne kelljen ismerni a működését
  - Pl: Komplex szám, Láncolt lista, Bináris fa

Referencia (kitérő)

Csere függvény jelenlegi tudásunkkal

```
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 (kitérő)

```
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;
}
```

Referenciával

```
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

- Referencia szerinti értékátadásnál nem másolat a paraméter változó, hanem ténylegesen az a változó amivel meg volt hívva a függvény.

Referencia

- Referencia szerinti értékátadásnál nem másolat a paraméter változó, hanem ténylegesen az a változó amivel meg volt hívva a függvény.
- Lehetséges nem paraméterként is referenciát használni:

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

Referencia

- Referencia szerinti értékátadásnál nem másolat a paraméter változó, hanem ténylegesen az a változó amivel meg volt hívva a függvény.
- Lehetséges nem paraméterként is referenciát használni:

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

- Referencia "mutatója" nem módosítható, létrehozáskor meg kell adni melyik változóra referencia és ez többé nem módosítható.

Komplex szám jelenleg

Próbáljunk most a jelenlegi tudásunkkal írni egy Komplex adattípust.

Komplex szám jelenleg

Próbáljunk most a jelenlegi tudásunkkal írni egy Komplex adattípust.

```
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;
}
```

Komplex szám jelenleg

Ez eddig nem rossz, még írjuk meg a szorzást, abszolút értéket és szép kiírását.

Komplex szám jelenleg

Ez eddig nem rossz, még írjuk meg a szorzást, abszolút értéket és szép kiírását.

```
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;
}
```

Komplex szám jelenleg, 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;
}
```

Komplex szám jelenleg, 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;
}
```

A probléma az átláthatósággal van.

Nem ránézésre olvasható a kód.

Komplex szám jelenleg, 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;
}
```

A probléma az átláthatósággal van.

Nem ránézésre olvasható a kód.

Azt szeretnénk, hogy olyan legyen komplex számokkal számolni, mintha **float**-ok lennének.

Komplex szám osztállyal

```
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();
};
```

Új kulcsszavak

Új kulcsszavak:

- **class**-al tudunk létrehozni új osztályt.
- **private** adattagok/metódusok csak az osztálynak érhetõek el (belsõ, elrejtett mûködés)
- **public** adattagok/metódusok a külvilág számára is elérhetõek (ezekrõl késõbb részletesebben lesz szó)

Komplex szám osztállyal 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;
}
```

Komplex szám osztállyal 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;
}
```

Átláthatóbb egy fokkal, mint a **struct**-os megoldást

Komplex szám osztállyal 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;
}
```

Átláthatóbb egy fokkal, mint a **struct**-os megoldást

De azért még nem az igazi.

Komplex szám osztállyal 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;
}
```

Átláthatóbb egy fokkal, mint a **struct**-os megoldást

De azért még nem az igazi.

Kicsit késõbb képesek leszünk átdefiniálni az operátorokat, mint a + és *, valamint megtanítani a **cout**-ot, hogy hogyan kell kiírni általunk definiált objektumokat.

Adattagok, konstruktorok

- Adattagokat és metódusokat a . (pont) operátorral érünk el.

Adattagok, konstruktorok

- Adattagokat és metódusokat a . (pont) operátorral érünk el.
- Ha egy metódusnak nincsenek argumentumai akkor is ki kell tenni a zárójeleket híváskor.

Adattagok, konstruktorok

- Adattagokat és metódusokat a . (pont) operátorral érünk el.
- Ha egy metódusnak nincsenek argumentumai akkor is ki kell tenni a zárójeleket híváskor.
- Amikor új objektumot készítünk, azt a megírt konstruktorokkal tehetjük.

Adattagok, konstruktorok

- Adattagokat és metódusokat a . (pont) operátorral érünk el.
- Ha egy metódusnak nincsenek argumentumai akkor is ki kell tenni a zárójeleket híváskor.
- Amikor új objektumot készítünk, azt a megírt konstruktorokkal tehetjük.
- Ez a konstruktor például két **float**-ot vár.

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

Adattagok, konstruktorok

- Adattagokat és metódusokat a . (pont) operátorral érünk el.
- Ha egy metódusnak nincsenek argumentumai akkor is ki kell tenni a zárójeleket híváskor.
- Amikor új objektumot készítünk, azt a megírt konstruktorokkal tehetjük.
- Ez a konstruktor például két **float**-ot vár.

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

- Ez a default konstruktor (ha csak létrehozunk egy objektumot, de nem inicializáljuk, nem adunk neki értékeket).

```
Complex();
```

Adattagok, konstruktorok

- Adattagokat és metódusokat a . (pont) operátorral érünk el.
- Ha egy metódusnak nincsenek argumentumai akkor is ki kell tenni a zárójeleket híváskor.
- Amikor új objektumot készítünk, azt a megírt konstruktorokkal tehetjük.
- Ez a konstruktor például két **float**-ot vár.

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

- Ez a default konstruktor (ha csak létrehozunk egy objektumot, de nem inicializáljuk, nem adunk neki értékeket).

```
Complex();
```

- Ez pedig a copy konstruktor, ez hívódik meg, ha lemásolunk egy ilyen objektumot (például amikor átadjuk egy függvénynek).

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

Adattagok, konstruktorok

- Adattagokat és metódusokat a . (pont) operátorral érünk el.
- Ha egy metódusnak nincsenek argumentumai akkor is ki kell tenni a zárójeleket híváskor.
- Amikor új objektumot készítünk, azt a megírt konstruktorokkal tehetjük.
- Ez a konstruktor például két **float**-ot vár.

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

- Ez a default konstruktor (ha csak létrehozunk egy objektumot, de nem inicializáljuk, nem adunk neki értékeket).

```
Complex();
```

- Ez pedig a copy konstruktor, ez hívódik meg, ha lemásolunk egy ilyen objektumot (például amikor átadjuk egy függvénynek).

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

- A konstruktoroknak nincs visszatérési értéke.

Komplex szám osztállyal metódusok

```
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;
}
```

Adattagok és konstruktorokról még

- Az adattagokat simán a nevükkel elérhetjük.

Adattagok és konstruktorokról még

- Az adattagokat simán a nevükkel elérhetjük.
- De referálhatunk az adott objektumunkra a **this** kulcsszóval, ez egy pointer az adott objektumra (pl az az objektum amin meg van hívva a metódus, vagy amit épp konstruálunk egy konstruktorral).

Adattagok és konstruktorokról még

- Az adattagokat simán a nevükkel elérhetjük.
- De referálhatunk az adott objektumunkra a **this** kulcsszóval, ez egy pointer az adott objektumra (pl az az objektum amin meg van hívva a metódus, vagy amit épp konstruálunk egy konstruktorral).
- A **const** kulcsszóról késõbb lesz szó.

Adattagok és konstruktorokról még

- Az adattagokat simán a nevükkel elérhetjük.
- De referálhatunk az adott objektumunkra a **this** kulcsszóval, ez egy pointer az adott objektumra (pl az az objektum amin meg van hívva a metódus, vagy amit épp konstruálunk egy konstruktorral).
- A **const** kulcsszóról késõbb lesz szó.
- Annak a fontosságáról, hogy miért referenciaként adjuk át a copy konstruktorban a lemásolandó objektumot érdemes elgondolkodni.

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

Adattagok és konstruktorokról még

- Az adattagokat simán a nevükkel elérhetjük.
- De referálhatunk az adott objektumunkra a **this** kulcsszóval, ez egy pointer az adott objektumra (pl az az objektum amin meg van hívva a metódus, vagy amit épp konstruálunk egy konstruktorral).
- A **const** kulcsszóról késõbb lesz szó.
- Annak a fontosságáról, hogy miért referenciaként adjuk át a copy konstruktorban a lemásolandó objektumot érdemes elgondolkodni.

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

- Ha nem referenciaként adnánk át, akkor végtelen rekurzióként meghívná önmagát, hisz le kéne másolnia a stack-re a kapott Complex objektumot és azt ezzel a metódussal tudja csak.

Komplex szám osztállyal metódusok

```
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() {
}
```

Komplex szám osztállyal metódusok

- A :: operátorral tudunk osztályok névterébe belépni ezért kellett a függvénynevek elé a **Complex::add**, mert simán **add** függvény nincs. Az **add** amit meg szeretnénk itt írni az a **Complex** osztályban található metódus.

Komplex szám osztállyal metódusok

- A **::** operátorral tudunk osztályok névterébe belépni ezért kellett a függvénynevek elé a **Complex::add**, mert simán **add** függvény nincs. Az **add** amit meg szeretnénk itt írni az a **Complex** osztályban található metódus.
- A **~Complex()** a destruktor. Ez akkor hívódik meg, amikor egy ilyen objektum megszûnik. Legyen ez azért mert automatikusan megszûnik, mert a blokkjának vége, vagy mert **delete**-el felszabadítottuk.

Komplex szám osztállyal metódusok

- A :: operátorral tudunk osztályok névterébe belépni ezért kellett a függvénynevek elé a **Complex::add**, mert simán **add** függvény nincs. Az **add** amit meg szeretnénk itt írni az a **Complex** osztályban található metódus.
- A **~Complex()** a destruktor. Ez akkor hívódik meg, amikor egy ilyen objektum megszûnik. Legyen ez azért mert automatikusan megszûnik, mert a blokkjának vége, vagy mert **delete**-el felszabadítottuk.
- A destruktornak akkor van jelentõssége, ha dinamikusan foglalt adatszerkezetekkel dolgozik az osztályunk (például láncolt lista).

Amit szeretnénk (legközelebb)

```
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;
}
```

kisZH kérdések

- Mutass példát referenciára
- Mire való a **this** kulcsszó?
- Mi a default konstruktor?
- Mi a copy konstruktor?
- Mi a destruktor?