C++ 1 Les 5

Copy and Move Constructors,
Copy and Move Assignment Operators,
Rule of Five

Leerdoelen

Aan het eind van deze workshop kun je de volgende functies correct schrijven:

- copy constructor
- copy assignment operator
- move constructor
- move assignment operator

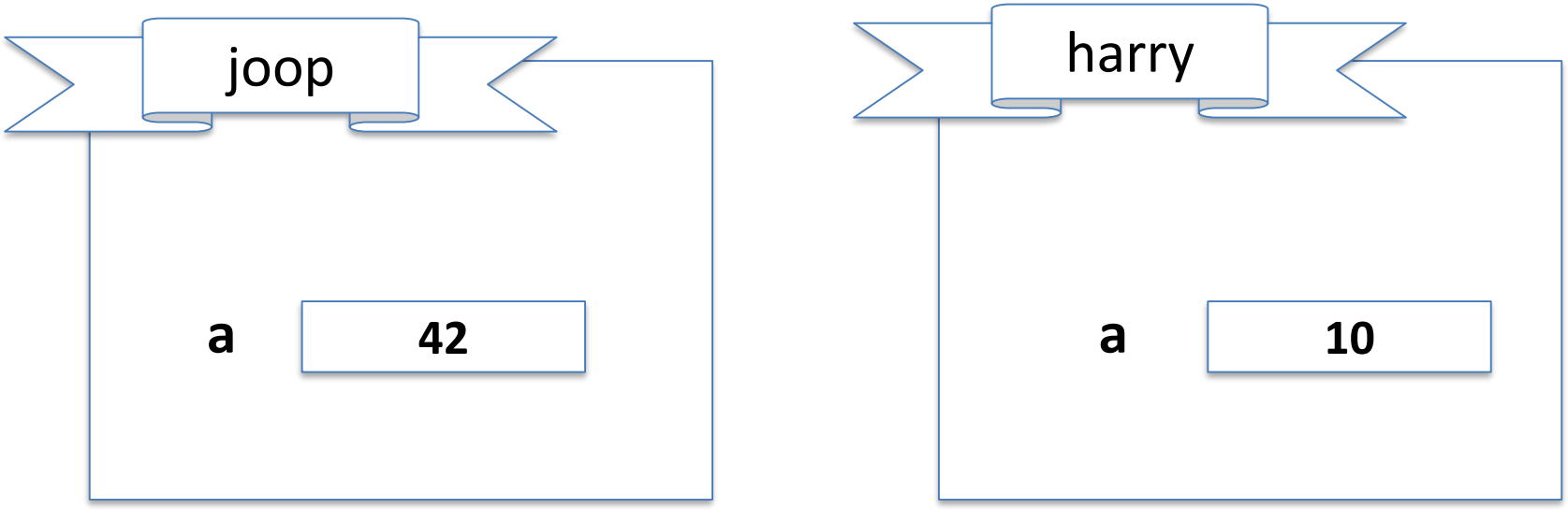
- En ken je de "Rule of Five"

objecten kopiëren (1)

```
class A
{
public:
    A() : a {0} {}
    A(int val) : a {val} {}

    int get_a() const
       { return a; }

    void set_a(int val)
       { a = val; }

private:
    int a;
};
```

```
A maria;
cout << maria.get_a() << '\n'; // 0

A joop {42};
A harry {joop};
cout << harry.get_a() << '\n'; // 42
harry.set_a(10);
cout << harry.get_a() << '\n'; // 10
cout << joop.get_a() << '\n'; // 42

// Dit gaat goed...
```

eindsituatie



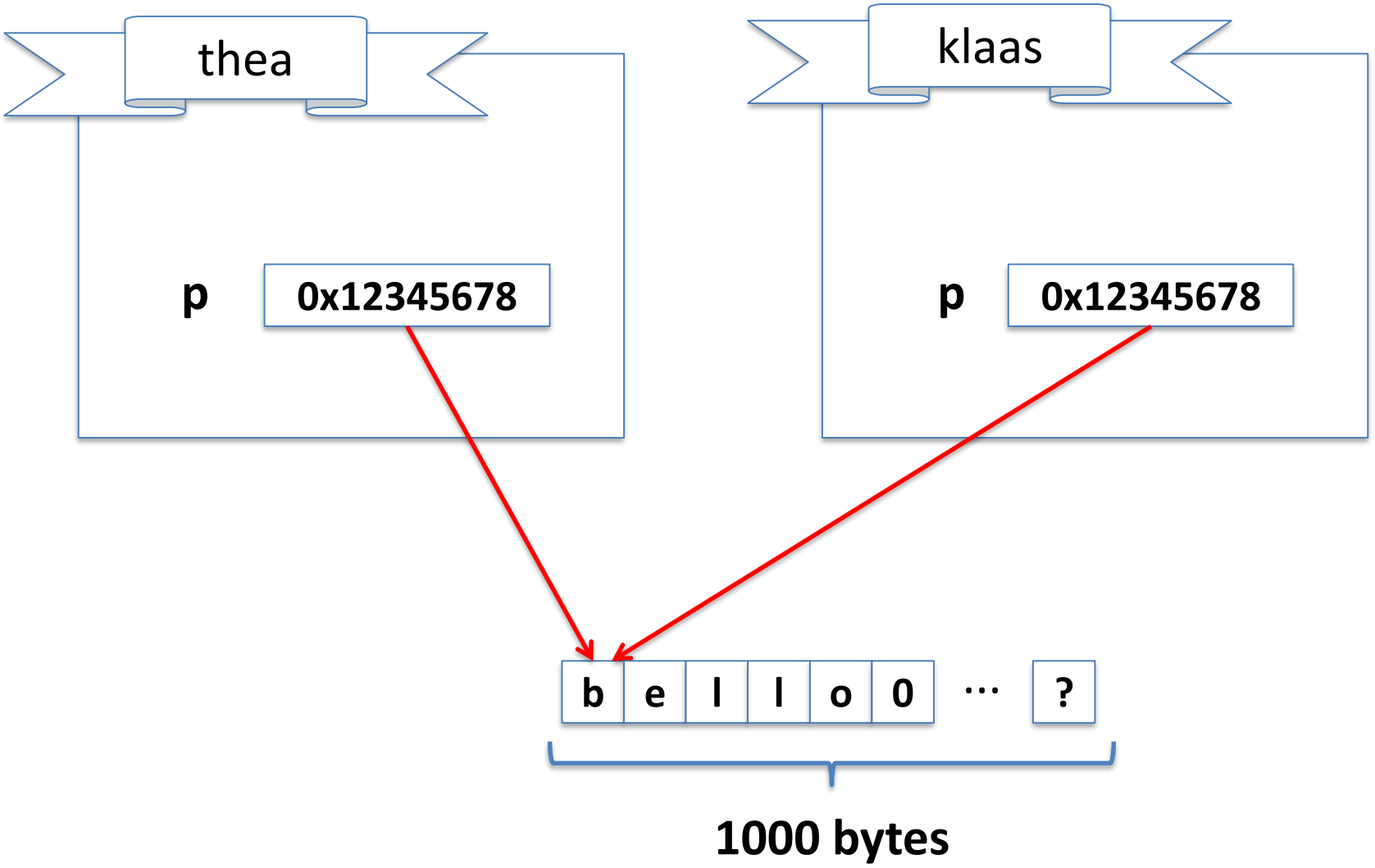
objecten kopiëren (2)

```
class B
{
public:
    B() {
      p = new  char[1000];
    }

    ~B() { delete[] p; }

    char* get_p() const
       { return p; }

private:
    char* p;
};
```

```
int main() {
B thea;
char* str {thea.get_p()};
const char* text {"hello"};
std::strcpy(thea.get_p(), text);
cout << thea.get_p() << '\n'; // hello

B klaas {thea};
cout << klaas.get_p() << '\n'; // hello
char* klaas_text {klaas.get_p()};
klaas_text[0] = 'b';
cout << klaas.get_p() << '\n'; // bello
cout << thea.get_p() << '\n'; // bello
}
```

OEPS! 3 problemen: 1000 bytes gelekt,
1 dubbele delete, en geen kopie!

eindsituatie (vóór destructie)



standaard: **shallow copy**

- als je schrijft:

```
B klaas {thea};
```

- wordt een speciale constructor aangeroepen die de compiler al voor je heeft gebouwd:

```
B(const B& other); // copy constructor
```

- en die doet een byte-voor-byte copy...

- dus: **pointers gaan naar hetzelfde wijzen!**

copy constructor

- zelf schrijven, indien nodig

```
B::B(const B& other) {
   // eigen buffer maken
   p = new char[1000];

   // alle bytes van de ander kopiëren
   std::memcpy(p, other.p, 1000);
}
```

probleem opgelost...?

- en dit dan:

```
B thea;
B klaas;
klaas = thea; // wat gebeurt hier?
```

- eerst een default constructie...
- ...en dan een toekenning!

wat is in C++ een toekenning?

- een toekenning is de aanroep van een speciale functie, die de compiler voor je maakt:

```
B& B::operator=(const B& other);
```

- dit heet: de **copy assignment operator**

- hm, lijkt veel op een copy constructor
- ook zelf maken dus, indien nodig!

zelf copy assignment operator maken (poging 1)

```
void B::operator=(const B& other) {
  // eigen buffer maken
  p = new char[1000];

  // alle bytes van de ander kopiëren
  std::memcpy(p, other.p, 1000);
}
```

vergeten we niet iets...?

- als van een object de copy assignment operator wordt aangeroepen, **bestaat dat object dus al**
- we moeten dus **eerst de boel opruimen**, voordat we iets nieuws beginnen!

zelf copy assignment operator maken (poging 2)

```
void B::operator=(const B& other) {
  // cleanup
  delete[] p;

  // eigen buffer maken
  p = new char[1000];

  // alle bytes van de ander kopiëren
  std::memcpy(p, other.p, 1000);
}
```

bijna goed...

- wat gebeurt er als we dit doen:

```
B thea; // default constructie
B* p1 {&thea}; // pointer naar thea
B* p2 {&thea}; // nog een ptr naar thea
*p1 = *p2; // oeps!
```

- toekenning aan zichzelf...

wat gebeurt er dan?

```
void B::operator=(const B& other) {
  // cleanup
  delete[] p; // oeps

  // eigen buffer maken
  p = new char[1000];

  // alle bytes van de ander kopiëren
  std::memcpy(p, other.p, 1000); // rommel
}
```

zelf copy assignment operator maken (poging 3)

```
void B::operator=(const B& other) {
  if (this != &other) {
    // cleanup
    delete[] p;

    // eigen buffer maken
    p = new char[1000];

    // alle bytes van de ander kopiëren
    std::memcpy(p, other.p, 1000);
  }
}
```

nog één ding...

- in C++ mag je dit schrijven:

```
int a {3};
int b, c;
c = b = a; // chaining
```

- moet ook met objecten kunnen
- daarom moet `operator=()` het object zelf returnen

zelf copy assignment operator maken (zo moet het)

```
B& B::operator=(const B& other) {
  if (this != &other) {
    // cleanup
    delete[] p;

    // eigen buffer maken
    p = new char[1000];

    // alle bytes van de ander kopiëren
    std::memcpy(p, other.p, 1000);
  }
  return *this; // om chaining mogelijk te maken
}
```

kopiëren vermijden (1)

- wat gebeurt hier?

```
B func() {
  B ding;
  // ... doe iets met ding

  return ding;
}
```

- `ding` wordt **gekopieerd** naar de stack
- best zonde, is eigenlijk niet nodig ☹

Vanaf C++11: **move**

- door **move** te doen i.p.v. **copy** ga je resources hergebruiken ("ingewanden stelen")
- het object waarvan je steelt gaat immers direct daarna toch dood ☺

- om move mogelijk te maken:
  - schrijf een **move constructor**
  - schrijf een **move assignment operator**

move constructor

```
B::B(B&& other)
{
  // steal resources
  p = other.p;

  // reset other instance
  other.p = nullptr;
}
```

move assignment operator

```
B& B::operator=(B&& other)
{
  if (this != &other) {
    // cleanup
    delete[] p;

    // steal resources
    p = other.p;

    // reset other instance
    other.p = nullptr;
  }
  return *this;
}
```

Rule of Five

Als je zelf één van de volgende aan het schrijven bent:

- destructor
- copy constructor
- copy assignment operator
- move constructor
- move assignment operator

...dan heb je ze waarschijnlijk alle vijf nodig!