

C++ 1 Les 4

Constructors, Destructor,
Initialization, RAI, Inheritance

Leerdoelen

Aan het eind van deze workshop kun je:

- correcte **constructors** schrijven die **initialization lists** gebruiken
- **destructors** op een veilige manier gebruiken
- Het **RAll-idioom** toepassen
- **inheritance** correct aangeven

DEEL 1

CONSTRUCTORS & DESTRUCTORS

De Basics

- Constructor wordt altijd aangeroepen zodra het object wordt geïntantieerd.
 - Global: Bij starten programma (in volgorde van schrijven)
 - Stack: Bij aanvang functie (in volgorde van schrijven)
 - Heap: Bij new (onmiddellijk)
- Destructor wordt aangeroepen wanneer de instantie wordt weggegooid.
 - Global: Bij verlaten programma (omgekeerde volgorde)
 - Stack: Bij verlaten functie (omgekeerde volgorde)
 - Heap: Bij delete en delete[] (onmiddellijk)

Syntax

- Constructor: **Klassenaam()**
- Destructor: **~Klassenaam()**

```
class A
{
    A(); // constructor declaration
    ~A(); // destructor declaration
}

A::A() // constructor implementation
{
}

A::~~A() // destructor implementation
{
}
```

constructie/destructie in de **heap**

```
Ding* obj {new Ding}; // moderne notatie
```

```
Ding* obj = new Ding; // oude notatie
```

```
Ding* obj = new Ding(); // mag ook
```

```
// opruimen:
```

```
delete obj;
```

constructie/destructie op de **stack**

```
void func()  
{  
    Ding obj; // wow, da's compact!  
}
```

- opruimen gaat **automatisch** wanneer obj out-of-scope gaat, dus bij de '}'
- lijkt dus sprekend op ingebouwde types (char, int, float, double, bool, etc.)
- let op: er staat een **complete instantie** op de stack, geen pointer!

even opletten...

```
void func()  
{  
    Ding* obj; // wat gebeurt hier?  
}
```

- hier wordt een locale pointer-variabele op de stack gezet, **niet** een object!

constructie/destructie van **global**

Ding obj;

// constructor: bij starten programma

```
int main()  
{  
    ...  
}
```

// destructor: bij einde programma

DEEL 2

INITIALISATIE

initialisatie (algemeen)

```
int a = 1; // oud en vertrouwd, maar...
```

```
int a {1}; // modern C++, beter!
```

```
// constructor-aanroep met parameters
```

```
string text {"voorbeeld"};
```

```
vector<int> lijst {42, 17, -6, 99};
```

```
// default constructor-aanroep
```

```
string str {}; // expliciet
```

```
string str; // simpeler, "true C++ style"
```

member initialization (1)

```
class A
{
public:
    A();
    A(int e, int f, int g);

private:
    int i, j, k;
};
```

```
A::A()
{
    i = 1;
    j = 2;
    k = 3;
}
```

Dubbel
Werk!

```
A::A(int e, int f, int g)
    : i{e}, j{f}, k{g}
{
}
```

```
A::A() : A {1, 2, 3}
{
}
```

delegating constructor

member initialization (2)

```
class B : public A
{
public:
    B();

private:
    int a;
    int b;
};
```

de volgorde is dus:

1. superclass constructor
2. members in volgorde van declaratie

De volgorde in de member initializer list doet er dus niet toe!

TIP: gebruik dezelfde volgorde in de member initializer list!

```
B::B() : a{1}, b{k}, A{6, 7, 8}
{
}
```

b wordt 8



DEEL 3

RESOURCE ACQUISITION IS INITIALISATION

RAII: Wat is het?

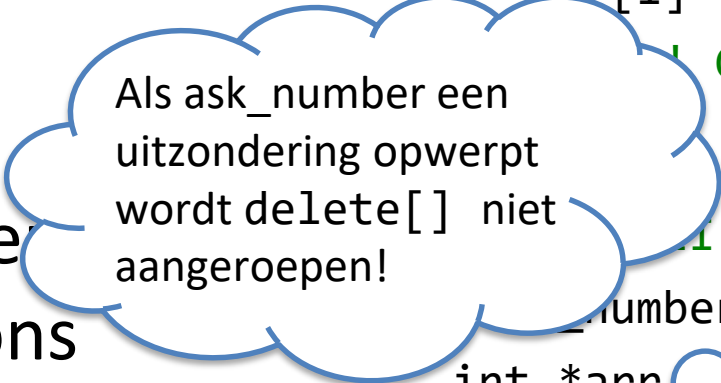
- Een *idioom* in C++:
 - De constructor
 - reserveert geheugen met `new`
 - of beheert een resource (file, netwerkverbinding)
 - De destructor
 - geeft het geheugen vrij met `delete`
 - of sluit file of netwerkverbinding

RAII: Wat lost het op?

- Geen geheugenlekken door vergeten deletes

// zonder RAII

```
void ask_numbers(int max) {  
    int *arr = new int[max];  
    for(int i {0}; i < max; ++i)  
        arr[i] = ask_number();  
    // Geen delete.
```



Als ask_number een uitzondering opwerpt wordt delete[] niet aangeroepen!

- Geen geheugenlekken door exceptions (komen nog aan bod)

```
void ask_numbers(int max) {  
    int *arr = new int[max];  
    for(int i {0}; i < max; ++i)  
        arr[i] = ask_number();  
    delete[] arr;  
}
```


RAII: Een voorbeeld

- Destructor zal automatisch worden aangeroepen bij verlaten scope
- Werkt ook bij het optreden van uitzonderingen

```
class array {  
public:  
    int *elem;  
    array(int size) :  
        elem{ new int[size] } {}  
    ~array() { delete[] elem; }  
};  
  
void ask_numbers(int max) {  
    array arr { max };  
    for(int i {0}; i < max; ++i)  
        arr.elem[i] = ask_number();  
    // delete niet nodig  
}
```

DEEL 4

INHERITANCE

inheritance: syntax

```
class Derived : public Base
{
    ...
};
```

- iets anders dan **public** is ook mogelijk, maar dat voert te ver voor deze cursus; meer info: zie boek "The C++ Programming Language", 4th Ed., page 605.

multiple inheritance

```
class Derived
    : public Base, public OtherBase
{
    ...
};
```

method override

- om een methode te kunnen overriden in een subclass, **moet** hij **virtual** gedeclareerd zijn:

```
class Animal
{
public:
    virtual void eat();
    ...
};
```

```
class Cow: public Animal
{
public:
    void eat() override;
    ...
};
```

virtual destructor

```
class A {  
public:  
    virtual ~A();  
};
```

TIP: maak **altijd** de destructor virtual
wanneer de class minstens één virtual method bevat

Heeft te maken met polymorfisme.
Later hierover meer.

abstracte classes

- een class is automatisch abstract wanneer minstens één method abstract is
- abstracte method heet: pure virtual

```
class AbstractBase
{
public:
    virtual void abstract_func() = 0;
    ..
};
```

interfaces?

- C++ kent geen interfaces als apart type!
 - gebruik **abstracte classes**
 - samen met **multiple inheritance**

Virtual Functions in Constructor?

```
class A
{
public:
    A() { fun(); }

    virtual void fun() { nummer = 1; }
    int getn() { return nummer; }

protected:
    int nummer;
};

class B : public A
{
public:
    B() : A() {}

    void fun() override { nummer = 2; }
};
```

```
int main()
{
    B b;
    int n = b.getn();
}
```

Wat is de
waarde
van n?

Virtual Functions **nooit** in Constructor!

Er kan vanuit de constructor van member variables gebruik gemaakt worden *die nog niet bestaan*.

Om die problemen te voorkomen gaat C++ altijd naar de base versie van de virtual functie en niet naar de derived.

Het compileert dus wel, maar je moet het **nooit** doen.

