

H1: Basisconcepten C++

Uitbreidingen t.o.v C

- * datatype bool
- * standaardstring
- * OOP
- * overloading
- * default param.
- * templates
- * exceptions

fund. datatypes

- * enkel bool extra
- * true en false
- * ↓ ↓
- * 1 0

Conversies

- * ook impl. casten
- * extra functieoproep-syntaxis
type(uitdrukking)
- "

(Std::) string

- * s1 == leeg
- * s2("test") == test
- * s3 = "ja" == ja
- * copy-constr.
- * find → zic API

- * strings optellen
en vergelijken:
+, +=, ==, <(=), >(=)
- * lengte → .length(), .size()
- * ontleiden → for (each) loop

call by reference

- * decl: type &
- * altijd initialiseren
- * geen * toevoegen

default parameters

- * declaratie
- * return type naam(param1, param2 = waarde);
- * altijd als laatste achteraan!

functie templates

- * template <typename T>
- * T var;
- * bij def. en decl.
- * meerdere typenames OK
- * mix'en met "echte" OK

namespaces

- * using namespace XXX;
- XXX:: weglaten

Tot einde bestand

- * while (!inv.fail()) {
- } if (inv.eof()) {
- }

Opmerking

- * streams mogen niet gekopieerd worden! ⇒ &

uitvoer

- * <iostream>
- * cout << ... < endl;
- * << oct of dec of hex

invoer

- * cin >> → tot whitespace
- * getline (cin, string);
- * char → cin.get()
- * getchar()

werken met bestanden

- * <fstream>
- * ifstream → invoer
- * ofstream → uitvoer
- * void open (naam, modus)
- * modus: in → inv.
↓ out → uitv.
ios: app → beide
- * void close()

dynamisch geheugenbeheer

- | | |
|-------------------|--------------|
| * array | * variabele |
| new type [uitdr]; | new type; |
| delete [] naam; | delete naam; |

H2: Mieuw sinds C++(11) (1)

Functies als parameters

- * <functional>
- * function < Returntype (param. lyst) naam >

Lambda functies

- * [] (param. lyst) → Return. { statements }
optioneel ↓

* captures

[] niks, [a,b], [&] alles met ref.,

[=] alles met waarde ...

unique_ptr

- * vervangt Null en 0
- * is een int *;

Smart pointers

- * <memory>
- * unique en smart

- * swap en move
- * g kopie mogelijk
- * make_unique<type>();
- * reset()
→ get = vry

Shared_ptr

- * make_shared<type>();
- * wordt pas vrijgegeven
als iedereen ze vrij
gelt

H3: Collections

Basisprincipes

- * collection is soort klasse
- * methodes → API
- * elementen → zelfde type

Sequences

- * groeitabel → string, vector
- * gelinkte lijst → list, forward-list

Groeitabel

- * indien cap vol → nieuwe maken met grotere cap en oude erin kopiëren

Container adaptors

- * beperken de vorige containers

Verzameling

- * elk element uniek
- * toegroegen, opzoeken, overlopen
- * families → ek OGP
Tree- en Hash-

multi-verzameling

- * duplicaten worden bewaard
- * (multi)map → <
- * zie API

Iteratoren

- * [typename] collectiontype:: [const-] iterator naam;
- * .begin() of .end()
- * ++(--)it of it++ -
- * advance(it, n)
- * *it → waarde

vector

- * <vector>
- * vector<type> v1 → Ø
- * v2(6) → 6 × Ø of ""
- * v3(6, "test") → 6 × "test"

- * copy-constructor
- * achteraan toevoegen → push-back(el)
- * .capacity() → capaciteit

Associatiële containers

- * set, multi-set
- * unordered_set en unordered-multi-set
- * map ...

set en unordered-set

- * tempi. param. bij set moet s implementeren
- * zie API

map vs vector

- * vector houdt alles 0...size()-1 bij
- * indexering map moeilijk

H4: OOP in C++

Klassen in C++

- * class Klasse {
 - public:
 - void naam(int i=1);
 - private:
 - ...
- * Klasse::naam(int waarde) { ... }
- * ~~void~~
- * getters → const toevoegen

operator overloading

- * return-type operator([param]) [const];
- * diepe kopie → = overschrijven
- * t++ → operator++()
- * t+t → operator++(int)
- * operator[] (int);
 - int &
- * friend ostream& operator<<(ostream& os, ...)

Constructor - destructor

- * klassenaam(parameterlijst)
- * ~klassenaam()

Initialiser list

A(const B& b1, int i1) : b(b1), i(i1) {}

Friend Fct's

- * toegang tot private attr.

Klasse templates

- * template <typename T>
- * geen <T> bij constructor en destructor
- * T klasse<T> :: ...

H5: Overerving

privaat

- * \bar{g} public impliceert private
- * public gedeelte basiskl. \rightarrow private
in afgeleide klasse
- * \bar{g} is-een relatie

constructoren

- * indien \bar{g} gedeclareerd
 \rightarrow die van basisklasse
- * via using kunnen we
overnemen van basiskl.
- * using erft alle const.
 \bar{g} specifieke

Polymorfisme

- * gebruik pointers
- * constructor \rightarrow pointers!
- * bij gelegenheid basis- en afg.
zal enkel de info r. d.
basiskl. behouden worden
- + unique-ptr < \Rightarrow ...>

Abstracte klasse

- * minstens één lid \bar{f} (niet virtueel)
- \rightarrow \bar{g} body
- * \bar{g} constructor

public

- * is-een relatie

protected

- * publieke leden worden "protected"
- \rightarrow zelden gebruikt

multiple inheritance

- * meerdere basisklassen
- * aanduiden met :: van wie welke
attributen / methodes overgenomen worden

Dynamic binding

- * methodes virtual maken in basiskl.
- * methodes in afg. kl. overschrijven
- * virtual \rightarrow traage uitvoering

Virtueel Destruktor

- * altijd op examen doen!

H6: Nieuw sinds C++ 11 (2)

automatische type-afleiding

- * enkel bij declaratie en initialisatie
- * nuttig bij complexe STL iteratoren

initialisatie syntax

- * uniforme initialisatie met accolades
- * int b{2} ipv int b = 2;
- * A a{3} ipv A a(3);
- * in klasse initialisatie:
int x=7;
int y{2};

The Big Three

- * operator =
- * copy constr.
- * destr.

The Big Five

- * operator =
- * copy const.
- * destr.
- * move const.
- * move oper.

defaulted en deleted

- * default → expl. default impl.
- * deleted → omgekeerd van default

move constructor/operator

- * constr. A b(move(a));
- * oper. c = move(b);
- * zelf schrijven
 - pointer: ondiepe kopie, originele pte → nullptr
 - prim. type: kopie + origineel → 0
- * move constr.
A(A&&);
- * move op.
A& operator=(A&&);

Rule of Five

- * sinds C++ 11
- * indien één v.d. Big Five gevraagd → allemaal maken

Rule of zero

- * als we kunnen vermijden default operaties te schrijven → doen

HT: Exception Handling

Opwerpen

- * throw
- * We kunnen met alles gooien!

Oprangen

- * catch()
- * C-string: (const char *s)
- * alles: (...)
 - altijd als laatste
- * niet per se variabele
 - (const char *)

Gebruik bestaande exceptionklasse

```
#include <stdexcept>  
* catch (const out_of_range& e)  
{ cout << e.what() << endl; }
```

Classes

- * logic_error
- * runtime_error

Zelfgemaakte exceptionklasse

- * class naam: public runtime_error {
 - public:
 - naam(): runtime_error("oops") {}
 - naam(const string &s): runtime_error(s) {}
- * best van runtime_error afleiden
→ exception heeft geen string