

HOOFDSTUK 1

BASISCONCEPTEN C++

Helga Naessens

Inhoud

- **C++ as a better C?**
- Nieuwe datatypes
- Referentietype
- Functie-templates
- Console invoer en uitvoer
- Namespaces
- Werken met bestanden
- Dynamisch geheugenbeheer

C++ as a better C?

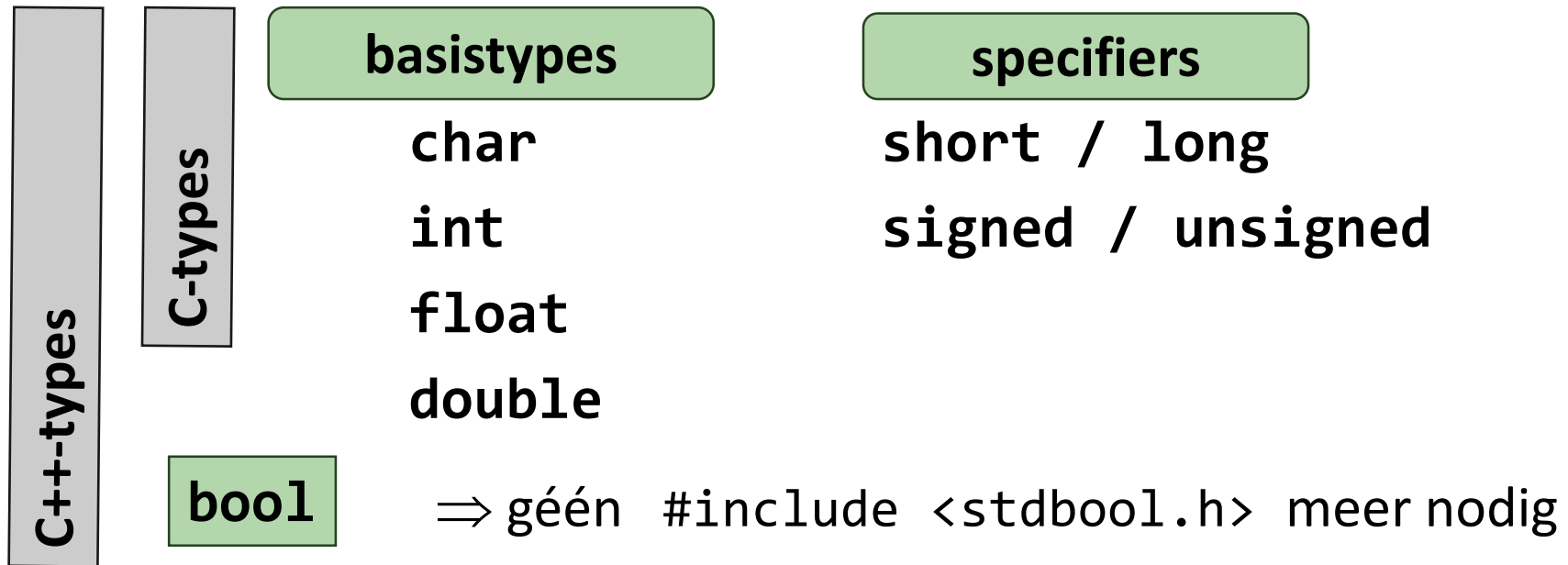
Uitbreidingen:

- datatype bool
- standaardstring
- overloading (functies en operatoren)
- default parameters
- klassen
- templates
- ...

Inhoud

- C++ as a better C?
- **Nieuwe datatypes**
- Referentietype
- Functie-templates
- Console invoer en uitvoer
- Namespaces
- Werken met bestanden
- Dynamisch geheugenbeheer

Fundamentele datatypes



- waarden: true en false
- automatische conversie `bool` → `int` (`true` → 1, `false` → 0)

variabelen mogen op willekeurige plaats in blok gedeclareerd worden
(geen voorwaartse referenties)

Conversies

- Ook in C++ impliciete (= automatische) conversies bij toekenningsoopdrachten en rekenkundige opdrachten (zelfs voor diegenen die informatie kunnen verliezen)
- Expliciete conversie (casten):
 - Operator-syntax: (type)uitdrukking
 - Funtieoproep-syntax: type(uitdrukking)

```
double d; int i;  
i = (int)(d*2+3);  
i = int(d*2+3);
```

Het type `std::string` (standaardstrings)

- Bovenaan: `#include <string>`
- **Declaratie en initialisatie**

```
std::string s1;    //s1 bevat lege string ""  
s1 = "Hij zei: \"Hallo\"\\n\";\";  
std::string s2("test");  
std::string s3 = s2; //s3 is kopie van s2  
std::string s4(s3);  //copyconstructor  
std::string s5(10, 'c');
```

Merk op: **constructor** oproepen = **géén new** gebruiken!!!

Merk op: function/constructor **overloading** is **toegestaan** in C++

- **Concatenatie van strings**

```
std::string s = "dag";
```

```
s = s + " jan";
```

```
s += '!';
```

```
std::string t = "dag" + s + "jan"; //OK
```

```
t = "dag" + " jan" + s; //FOUT
```

- **Vergelijken van strings**

```
std::string s, t; ...
```

```
if (s < "tomaat") ... //komt s alfabetisch voor "tomaat"?
```

```
if (s >= t) ... //is s alfabetisch groter dan t?
```


```
if (s == "stop") ... //bevat s het woord "stop"?
```


- **Lengte van een string bepalen**

```
std::string s = "dag";  
int l = s.size(); //3  
l = s.length(); //3
```

- **String ontleden**

string_vb.cpp

```
std::string s = "hallo";  
for (int i = 0 ; i < s.size() ; i++)  
    printf("%c",s[i]); //of: cout << s[i]; (zie later)  
s[1] = 'e';  
for (char c : s)   
    printf("%c",c); //of: cout << c; (zie later)
```

Oefening

Schrijf een functie `omgekeerde(s)` die de omgekeerde string van de gegeven string `s` teruggeeft.

Bv: als `s = "voorbeeld"` wordt de string `"dleebroov"` teruggegeven.

- Enkele lidfuncties uit de klasse `string` (zie API)

- `size_t find(string/[const] char [*] s, int pos=0)`

Geeft de positie in de string waar de deelstring `s` voor het eerst optreedt, vanaf de positie `pos`.

Geeft `string::npos` terug als `s` niet gevonden wordt.

Merk op: laatste parameter = **default parameter**

⇒ bij oproep functie is 2^{de} argument optioneel

(hier: 2^{de} argument is dan 0)

Intermezzo: meer info omtrent **default parameters**

```
void schrijf_lijn(char, int);
```

```
void schrijf_lijn(char); //print 80 keer meegegeven char
```

```
void schrijf_lijn(); //print 80 keer '-'
```

⇒ hier wordt gebruik gemaakt van **function overloading**

⇒ korter: gebruik default parameters:

declaratie functie: `void schrijf_lijn(char = '-', int = 80);`

opmerkingen:

- default parameters moeten **achteraan** staan

```
void schrijf_lijn(char = '-', int);
```

- Bij definitie functie: default waarden niet meer herhalen!!

Voorbeeld: **vb_default_parameters.cpp**

- `void insert(size_t pos, string/const char* s)`
Voegt de string *s* in, op positie *pos*.
- `string substr(size_t pos=0, size_t len=string::npos)`
Geeft de deelstring opgebouwd uit de eerste *len* tekens, vanaf positie *pos*.
- `void erase(size_t pos=0, size_t len=string::npos)`
Verwijdert in de string *len* tekens, vanaf positie *pos*.
- `void replace(size_t pos, size_t len, string/const char* s)`
Vervangt in de string *len* tekens, vanaf positie *pos*, door de deelstring *s*.
- ... (<http://www.cplusplus.com/reference/string/string/>)

Voorbeeld

```
std::string s = " Een voorbeeld ";  
int pos = s.find(" ");  
while (pos != std::string::npos) {  
    s.replace(pos,2," ");  
    pos = s.find(" ");  
}  
if (s[0]==' ')  
    s = s.substr(1, s.size()-1);  
  
//inhoud s???
```

Inhoud

- C++ as a better C?
- Nieuwe datatypes
- **Referentietype**
- Functie-templates
- Console invoer en uitvoer
- Namespaces
- Werken met bestanden
- Dynamisch geheugenbeheer

Referentietype

- Declaratie: `type &`
- Verplichte initialisatie bij declaratie (geen NULL referentie)
- Bevat automatisch het adres van een andere variabele (zet géén & voor deze variabele)
- Wordt automatisch gederefereerd (gebruik géén *)
- Voorbeeld

```
int a; int &x = a; //géén & vóór a
```

```
x = 5; x++; //géén * vóór x
```

```
printf("%d %d", a, x); // 6 6
```

```
int &y; // Fout: initialisatie ontbreekt
```

- Wordt dikwijls als formele parameter gebruikt
- Voorbeeld

```
void swap(int &, int &);

int main() {
    int a = 5, b = 6;
    swap(a, b);    // verwissel a en b
}

void swap(int &x, int &y) {
    int h = x;
    x = y;
    y = h;
}
```


Oefening:

Schrijf een procedure $\text{vkw}(a, b, c, \text{aantal}, w1, w2)$ die de vierkantswortels van de gegeven vergelijking $ax^2 + bx + c$ bepaalt (a , b en c reële getallen).

Het aantal wortels wordt opgeslagen in aantal , de eventuele wortel(s) in $w1$ en $w2$.

Schrijf daarna een hoofdprogramma dat de vierkantswortels van $7x^2 - 8x + 16$ bepaalt.

- Referentietype wordt ook gebruikt om kopie te vermijden.

Voorbeeld:

```
double afstand(punt p1, punt p2) { //kopie => minder goed  
    return sqrt((p1.x-p2.x)*... + ...; }
```

of

```
double afstand(const punt *p1, const punt *p2) { //C-stijl  
    return sqrt((p1->x-p2->x)*... + ...; }
```

of

```
double afstand(const punt &p1, const punt &p2) { //C++-stijl  
    return sqrt((p1.x-p2.x)*... + ...; }
```

Pointerparameters in C versus parameters in C++

C

- array:
`[const] type * t`
gebruik: `...t[i]...`
- (invoer-)uitvoerparameter:
`type *p`
gebruik: `...*p...`
- kopie vermijden:
`const type *p`
gebruik: `...*p...`

C++

- array:
`[const] type * t`
gebruik: `...t[i]...`
- (invoer-)uitvoerparameter:
`type &p`
gebruik: `...p...`
- kopie vermijden:
`const type &p`
gebruik: `...p...`

Inhoud

- C++ as a better C?
- Nieuwe datatypes
- Referentietype
- **Functie-templates**
- Console invoer en uitvoer
- Namespaces
- Werken met bestanden
- Dynamisch geheugenbeheer

Functie-templates

Beschouw volgende procedure:

```
void wissel(int &var1, int &var2) {  
    int temp = var1;  
    var1 = var2;  
    var2 = temp;  
}
```

- ⇒ zou handig zijn om procedure te kunnen declareren en definiëren die voor alle type objecten werkt (nu enkel voor type `int`)
- ⇒ hoe oplossen in C?
- ⇒ in C++: gebruik **functie-templates**

`template <typename T>`  template prefix

```
void wissel( T & a, T & b) {
```

```
    T hulp = a;
```

```
    a = b;
```

```
    b = hulp;
```

```
}
```

implementatie met
gebruik van type T

meestal wordt T als naam
gebruikt, maar een andere
naam mag ook

```
int main() {
```

```
    double d1 = 3.6, d2 = 5.4;
```

```
    wissel(d1,d2);
```

```
    punt p1 = {1,2}, p2 = {3,4};
```

```
    wissel(p1,p2);
```

```
}
```

enkel types die alle operatoren/functies
ondersteunen die op T opgeroepen worden,
mogen in T gesubstitueerd worden

Opmerkingen

- Elke functie/procedure die templates gebruikt, moet voorafgegaan worden door de template prefix (zowel bij declaratie als bij definitie)
- In de template prefix mag ook het keyword `class` gebruikt worden (i.p.v. het keyword `typename`)
- Meerdere `typename` parameters zijn toegestaan (op voorwaarde dat ze allemaal gebruikt worden in de implementatie)

Typename types mogen gecombineerd worden met “echte” types:

```
template <typename T1, typename T2>  
T1 functie_naam(int i, const T1 &t1, const T2 &t2);
```

- Functies/procedures met templates mogen overladen worden door functies/procedures met enkel “echte” types (zie [template_vb.cpp](#))

Functie-templates: aanbevolen aanpak

1. Ontwikkel functie met een specifiek type
2. Test deze functie grondig
3. Converteer vervolgens naar template door de type namen te vervangen door de typename parameter

Voordelen:

- Laat grondige testing toe
- Nadruk op het algoritme zelf en niet op de syntax van de template

Inhoud

- C++ as a better C?
- Nieuwe datatypes
- Referentietype
- Functie-templates
- **Console invoer en uitvoer**
- Namespaces
- Werken met bestanden
- Dynamisch geheugenbeheer

Vanaf nu zullen we géén gebruik meer maken van `scanf`, `fgets` en `printf`

Uitvoer

- bovenaan: `#include <iostream>`
- via `std::cout` (console output) en `<<` (concateneerbaar)
- rechteroperand kan o.a. zijn:
 - variabele/constante van willekeurig type (ook string)
 - `std::endl`
 - controle-informatie via manipulatoren, bv:

`std::oct``std::dec``std::hex`

→ druk alle volgende operandi octaal/decimaal/hexadecimaal af (tot een andere manipulator gebruikt wordt)
- voorbeeld: `std::cout<<"dag"<<s<<std::endl<<"getal="<<g;`

Invoer

- bovenaan: `#include<iostream>`
- via `std::cin` (console output) en `>>` (concateneerbaar)
- rechteroperand is variabele van willekeurig type

```
#include <iostream>
```

io_vb.cpp

```
int main() {
```

```
    std::cout << "Geef twee getallen in :\n";
```

```
    int g1, g2;
```

```
    std::cin >> g1 >> g2;
```

```
    std::cout << "getal1 dec = " << g1 << std::endl
```

```
                << "getal2 hex = " << std::hex << g2;
```

```
    return 0;
```

```
}
```

Inlezen van een string

- Manier 1:

```
std::string s;  
std::cin >> s;
```

Voorbeeld: **string_inlezen_vb1.cpp**

⇒ whitespaces worden overgeslagen, leest **slechts 1 woord**
(= alle karakters tot aan de volgende whitespace) in!!


- Manier 2:

```
std::string s;  
getline(std::cin, s);
```

Voorbeeld: **string_inlezen_vb2.cpp**

⇒ (de rest van) **1 lijn** wordt ingelezen (t.e.m newline-karakter,
maar dit karakter wordt niet bewaard in s)

- **Let op** voor getline onmiddellijk na >>:

 std::cin >> getal; getline(std::cin, s);

Kan er nog iets ingegeven worden?

Voorbeeld: **string_inlezen_vb3.cpp**

Inlezen van een char

- Manier 1:

```
char a;  
std::cin >> a;
```

Voorbeeld: **char_inlezen_vb1.cpp**

⇒ er kunnen **geen whitespace karakters** ingelezen worden!!

- Manier 2:

```
char a;  
a = std::cin.get(); //of: a = getchar();
```

Voorbeeld: **char_inlezen_vb2.cpp**

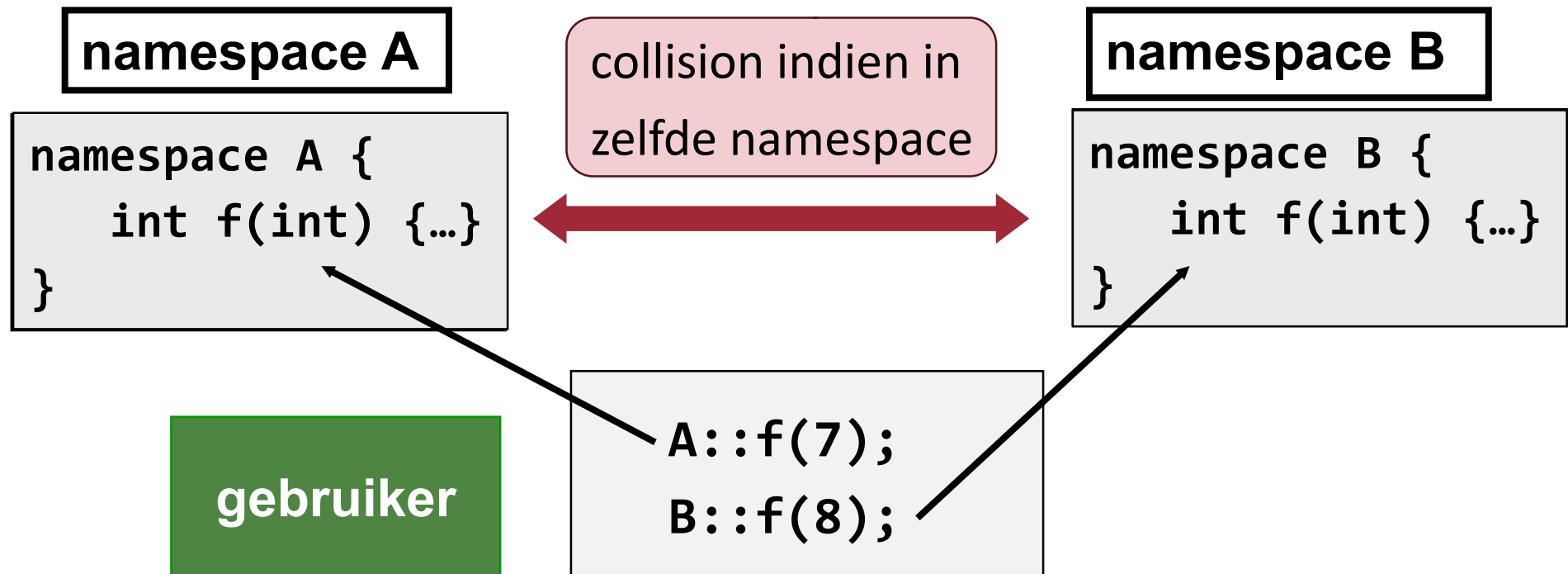
⇒ er kunnen **wel whitespace karakters** ingelezen worden!!

Inhoud

- C++ as a better C?
- Nieuwe datatypes
- Referentietype
- Functie-templates
- Console invoer en uitvoer
- **Namespaces**
- Werken met bestanden
- Dynamisch geheugenbeheer

Namespaces

- om collisions te vermijden
- aangeven naar welke namespace gerefereerd wordt
- cf. “package”-mechanisme in Java



- Door middel van de opdracht

`using XXX::identifier;`

kan men er voor zorgen dat bij het gebruik van *identifier*

`XXX::` mag weggelaten worden

Voorbeeld: `using A::f; using std::string;`

- Door middel van de opdracht

`using namespace XXX;`

kan men er voor zorgen dat bij alle identifiers uit de namespace `XXX`

`XXX::` mag weggelaten worden

Voorbeeld: `using namespace std;`

- Voorbeeld: `nmspc.cpp` en `namespace_vb.cpp`

Inhoud

- C++ as a better C?
- Nieuwe datatypes
- Referentietype
- Functie-templates
- Console invoer en uitvoer
- Namespaces
- **Werken met bestanden**
- Dynamisch geheugenbeheer

Werken met bestanden: inleiding

- **Invoerbestand**: bestand waaruit je gegevens leest
- **Uitvoerbestand**: bestand waarnaar je gegevens schrijft
- We zullen enkel werken met **sequentiële** tekstbestanden

Declaratie invoerbestand/uitvoerbestand

```
#include <fstream>

using namespace std;

int main() {

    ifstream inv;      //invoerbestand

    ofstream uitv1,uitv2; //uitvoerbestand

    fstream inuit;

        //bestand voor invoer én uitvoer

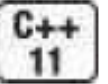
    ...

}
```

Openen bestand

```
void open(const char *fname [, openmode mode]);
```

```
void open(const string &fname [, openmode mode]);
```



ios::in	Open for input
ios::out	Open for output
ios::ate	Position after open: end of file
ios::app	Go to end before every write
ios::trunc	If the file already existed it is erased

```
uitv1.open("example.txt", ios::out | ios::app );
```

- Opmerkingen *openmode*:
 - out: enkel voor ofstream of fstream object.
 - in: enkel voor ifstream of fstream object.
 - trunc: enkel te gebruiken in combinatie met out
 - app: file wordt steeds geopend in out mode (zelfs indien out niet gespecificeerd werd), niet gebruiken in combinatie met trunc
 - ate: kan voor elk type file stream object en in combinatie met elke andere mode gebruikt worden

Let op: By default a file opened in out mode is truncated! To preserve the contents of a file opened with out, either we must also specify app or we must also specify in.

- Default *openmode* (indien geen mode gespecificeerd werd)
 - `ofstream: out`
 - `ifstream: in`
 - `fstream: in en out`
- Controleer na het openen van een (invoer)bestand of het openen gelukt is! Bestaat het bestand wel?
 - Als openen mislukt is, staat `failbit` op `true`
 - Controle:

```
if (inv.is_open()) ...  
if (!inv.fail()) ...  
if (inv) ...
```

Voorbeeld:

```
#include <fstream>
#include <iostream>
using namespace std;

int main() {
    ifstream inv; inv.open("b1.txt");
    if (inv.is_open()) cout << "openen gelukt";
    ofstream uit1, uit2;
    uit2.open("c:\\b2.txt");
    string s = "b3.txt";
    uit2.open(s, ios::app);
    return 0;
}
```

default ios::out
⇒ bestand wordt gewist!

Vóór C++11:
uit2.open(**s.c_str()**, ios::app);

Initialisatie bij declaratie

```
ifstream inv("b1.txt");  
if (inv.is_open())  
    cout << "openen gelukt";  
ofstream uit1("c:\\b2.txt");  
ofstream uit2(s, ios::app);
```

LET OP: onderstaande opdracht is **FOUT**

```
ifstream inv.open("b1.txt");
```

Lezen/schrijven van/naar een bestand

```
int getal;  
inv >> getal;  
  
char ch;  
  
ch = inv.get();  
  
string lijn;  
  
getline(inv, lijn);  
  
uitv1 << getal << " " << ch  
        << " " << lijn << endl;
```

Sluiten van een bestand

- Methode **close**:

```
inv.close();
```

```
uitv1.close();
```

```
uitv2.close();
```

- Als een fstream object reeds geopend is, kan het niet opnieuw geopend worden (failbit wordt true).
Hiertoe moet het eerst gesloten worden.
- Als een fstream object out of scope gaat, wordt het bestand waaraan het object gekoppeld is automatisch gesloten.

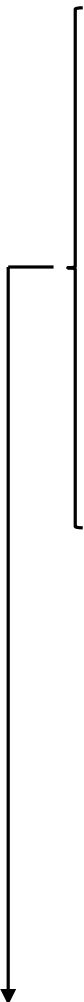
Oefeningen

- Oefening 1: Laat de gebruiker 5 lijnen tekst ingeven en voeg deze toe achteraan het bestand "tekst.txt".
- Oefening 2: Wat doet volgend programmafragment?

```
fstream inuit("g.dat", ios::in | ios::out);  
int getal;  
while (inuit >> getal)  
    inuit << getal/2.0 << endl;
```

⇒ Tegelijkertijd lezen van en schrijven naar een bestand (type `fstream`) lukt niet zomaar!

Testen op het einde van een invoerbestand



```
... //lees iets in uit het bestand (1)
while (!inv.fail()) {
    ... //doe iets met wat je ingelezen hebt
    ... //lees iets in uit het bestand (2)
}
if (inv.eof())
    ... //OK: bestand werd volledig gelezen
else
    ... //FOUT: bestand bevat foutieve gegevens
```

The diagram shows a vertical line on the left side of the code block. A horizontal line branches off from this vertical line at the level of the first line of code, and an arrow points down from this horizontal line to a box at the bottom of the slide.

Of: while (inv >> ... / getline(inv,s)) zonder (1) en (2)

Voorbeeld

Gevraagd

Schrijf een functie `aantalpositief(s)`, waarbij `s` een gegeven string is met de *naam* van een bestand, dat een onbepaald aantal gehele getallen bevat.

De functie bepaalt hoeveel strikt *positieve* getallen dit bestand bevat. Het resultaat van de functie is -1 indien het bestand niet kan geopend worden of fouten bevat.

Oplossing

`bestand_vb.cpp`

Opmerking

- streams mogen **niet** gekopieerd worden
- voorbeeld

```
void lees_en_schrijf_getal(ifstream &inv) {  
    int getal;  
    inv >> getal; cout << getal;  
}  
  
int main() {  
    ifstream inv("test.txt");  
    lees_en_schrijf_getal(inv);  
    lees_en_schrijf_getal(inv);  
}
```



zonder & =
compileerfout

Inhoud

- C++ as a better C?
- Nieuwe datatypes
- Referentietype
- Functie-templates
- Console invoer en uitvoer
- Namespaces
- Werken met bestanden
- **Dynamisch geheugenbeheer**

Dynamisch geheugenbeheer

array

variabele

creatie

```
new type[uitdr];
```

```
new type;
```

vrijgeven

```
delete[] naam;
```

```
delete naam;
```

```
int *r = new int[n];  
...  
delete[] r;
```

```
int *a = new int;  
...  
delete a;
```

gebruik in C++ enkel new en delete
(géén malloc, calloc, free, ...)