Gertjan Laan

AAN DE SLAG MET C++

Boom

Zevende druk

Aan de slag met C++

Gertjan Laan

Boom

inclusief website!

Met behulp van onderstaande unieke activeringscode krijg je toegang tot de website www.aandeslagmetcpp.nl voor extra materiaal. Deze code is persoonsgebonden en gekoppeld aan de 7e druk. Na activering van de code is de website twee jaar toegankelijk. De code kan tot zes maanden na het verschijnen van een volgende druk geactiveerd worden. De code is eenmalig te gebruiken.

Opmaak binnenwerk: Holland Graphics, Amsterdam
Basisontwerp omslag: Dog & Pony, Amsterdam
Omslagontwerp: Coco Bookmedia, Amersfoort
Beeld omslag: Olga Salt/Shutterstock
Beeld hoofdstukopener: Nickolay Grigoriev/Shutterstock

ISBN 9789024438617
ISBN e-book 9789024438624
NUR 173

www.aandeslagmetcpp.nl
www.boomhogeronderwijs.nl

Inhoudsopgave

Voorwoord bij de zevende druk

Deze zevende editie van *Aan de slag met C++* is geheel herzien.
Aanleiding voor de herziening is de voortdurende ontwikkeling van de taal C++. Sinds 2011 vinden om de drie jaar updates van C++ plaats. Een aantal van deze wijzigingen, voor zover relevant voor de in dit boek besproken onderwerpen, zijn in deze nieuwe editie opgenomen.
Ik heb in deze editie het gebruik van *using namespace std* vermeden, maar benoem waar nodig de voorzieningen uit de standaard library expliciet, zoals in *std::cout* of eventueel *using std::cout*.
In het algemeen heb ik uniforme initialisatie toegepast; dit maakt het onderscheid tussen initialisatie en assignment gemakkelijker.
De grootste wijzigingen zijn te vinden in hoofdstuk 12, dat in zijn geheel is herschreven en waarin, naast de standaardalgoritmen, ook aandacht is voor ranges en views uit de nieuwe std::ranges library.
Net als in de vorige editie eindigt elk hoofdstuk met een samenvatting, vragen en programmeeropgaven. De code van veel voorbeelden uit het boek, de antwoorden op de vragen en uitwerkingen van de meeste opgaven zijn te vinden op www.aandeslagmetcpp.nl. Verder is hier het online boek te raadplegen met de extra hoofdstukken 13, 14 en 15. De bijlagen zijn ook op de website te vinden.
De code van alle genummerde voorbeelden, evenals die van de uitwerkingen van de opgaven, is getest met de g++-compiler, versie 10.1 met compileroptie std=c++20.
Op www.cppreference.com kun je veel informatie vinden over C++ en de standaardlibrary.

Gertjan Laan
Zaandam, voorjaar 2021
www.gertjanlaan.nl


1

Introductie 1

1.1 Inleiding

Deze editie is geschreven met een lezer in gedachten die enigszins bekend is met elementaire begrippen en principes van veelgebruikte programmeertalen. Die kennis heb je bijvoorbeeld opgedaan door code te schrijven in Java, HTML en JavaScript, of Python. Hoewel veel programmeertalen in de basis op elkaar lijken, kunnen ze in die basis ook op subtiele wijze van elkaar verschillen. Subtiliteiten in een programmeertaal kunnen essentieel blijken. Ik geef daarom in dit hoofdstuk een overzicht van de elementaire voorzieningen en eigenaardigheden van C++. Daarbij zal ik steeds proberen de eigenschappen te demonstreren aan de hand van een geschikt, werkend voorbeeld.

1.2 De preprocessor, compiler en linker

Elk C++-programma dat je intikt, de broncode, moet eerst worden vertaald voor het kan worden uitgevoerd. Voor elk platform, zoals Linux, Android, iOS of macOS X en Windows, zijn compilers in omloop die ervoor zorgen dat de vertaalde versie van je programma geschikt is voor dat platform. Het hele vertaalproces verloopt in fasen waarbij achtereenvolgens de volgende onderdelen een rol spelen: de preprocessor, de compiler en de linker.
De preprocessor leest de broncode en gaat al lezend op zoek naar preprocessoropdrachten, de zogeheten *preprocessor directives*. Een preprocessor directive kun je herkennen aan het hekje `#` dat ervoor staat. Een voorbeeld van een preprocessor directive is:

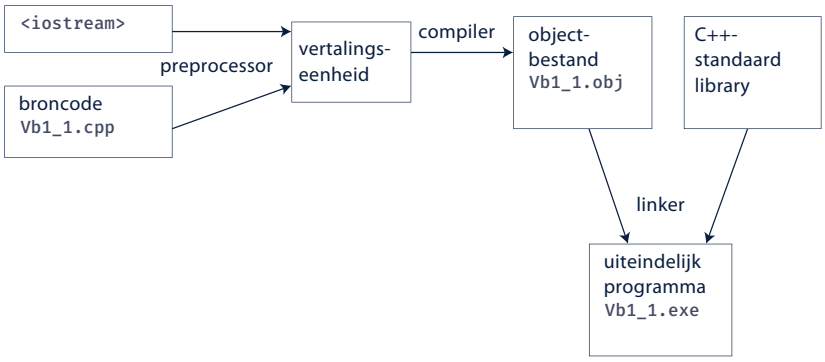
```
#include <iostream>
```

`<iostream>` is de naam van een zogeheten headerbestand (*header file*), dat onderdeel is van het C++-systeem. De complete inhoud van dit bestand moet door de preprocessor worden ingevoegd (geïncluded) in de broncode. In het headerbestand staan definities die noodzakelijk zijn om de compiler zijn werk goed te laten doen. Deze definities behelzen bijvoorbeeld constanten, prototypen en klassen. Hoe die definities eruitzien, wordt in de rest van dit boek duidelijk. Het resultaat van het werk van de preprocessor is een zogeheten *translation unit* (vertalingseenheid).

Na de preprocessor is het de beurt aan de compiler om de translation unit te vertalen. De compiler heeft twee belangrijke taken:

1. Het programma controleren op fouten en daar melding van maken.
2. Als er geen fouten gevonden zijn, het programma vertalen naar machinecode, dat wil zeggen: vertalen naar code die hoort bij de microprocessor die zich in de computer of het apparaat bevindt.

Dit hele proces wordt ook wel aangeduid met *compile-time*. Gedurende compile-time kunnen veel fouten tegen de grammaticale regels van de taal worden gevonden. Compile-time staat tegenover *runtime* (zie verderop).
Laten we even aannemen dat het programma taalkundig correct is, zodat er een vertaling tot stand komt. De vertaling wordt eventueel opgeslagen in een tussenbestand waarvan de naam het achtervoegsel `.obj` of `.o` heeft. Deze code heet wel *objectcode*, of doelcode (als tegenhanger van broncode).
De rol van de compiler is nu uitgespeeld. Vervolgens komt de linker aan de beurt. Het woord *linker* komt van het Engelse *to link*, aan elkaar koppelen.
C++ wordt geleverd met honderden voorgedefinieerde functies, klaar voor gebruik. Deze functies zitten opgeborgen in bestanden die *libraries* heten. Het woord *library* betekent letterlijk bibliotheek, maar het is duidelijk dat het hier niet om boeken gaat. De overeenkomst met een echte bibliotheek is dat er een voorraad functies is (in plaats van boeken), waar elk C++-programma naar believen kopieën van kan maken. Alle bibliotheken die standaard bij C++ worden geleverd heten de *C++ standard library*, of C++-standaardbibliotheek.
Een belangrijke taak van de linker is het uit de bibliotheek halen van een kopie van de functies die het vertaalde C++-programma nodig heeft en deze toe te voegen aan het C++-programma. Het resultaat daarvan wordt weggeschreven naar een uitvoerbaar bestand, onder Windows vaak met de extensie `.exe`. Dat uitvoerbare bestand is het bestand waar het allemaal om draait: dit bevat in machinecode alle opdrachten en informatie die het betreffende apparaat en het bijbehorende besturingssysteem nodig hebben om datgene uit te voeren wat je in de broncode van het C++-programma hebt ingetikt. Het proces van het uitvoeren (runnen) van een programma wordt ook wel aangeduid met *runtime*. Ook in runtime kunnen zich fouten voordoen, bijvoorbeeld omdat een berekening in de code verkeerd is opgeschreven, of omdat een bestand waaruit gelezen moet worden niet kan worden gevonden.
In figuur 1.1 is alles nog eens in beeld gebracht.



Figuur 1.1

1.3 Eerste voorbeeld

Laten we eens kijken naar een eenvoudig, maar goed werkend C++-programma:

Voorbeeld 1.1

```
#include <iostream>          // preprocessor directive

int main()                   // start van de functie main()
{
  int a;                     // declaratie van variabele a
  int b;                     // declaratie van variabele b
  int antwoord;              // declaratie van variabele antwoord
  a = 17;                    // a krijgt de waarde 17
  b = 24;                    // b krijgt de waarde 24
  antwoord = a + b;          // antwoord krijgt de waarde van a + b

  std::cout << "Het resultaat is: "; // tekst naar uitvoerscherm
  std::cout << antwoord;     // antwoord naar uitvoerscherm
  std::cin.get();            // wacht op indrukken van de entertoets
  return 0;                  // zie toelichting
}
```

De uitvoer van dit programma ziet er zo uit:

```
Het resultaat is: 41
```

1.3.1 Toelichting bij het eerste voorbeeld

Het belangrijkste gedeelte van voorbeeld 1.1 is het gedeelte dat begint met de regel `int main()` en dat helemaal onderaan eindigt met de sluitaccolade. Dit gedeelte is een functie die `main()` heet. Elk C++-programma moet een functie hebben met de naam `main()`. Als het programma wordt uitgevoerd, komt altijd als eerste de functie `main()` aan bod. Dit wil zeggen dat de opdrachten die tussen de openings- en sluitaccolades achter `main()` staan, als eerste worden uitgevoerd. Deze opdrachten heten ook wel *statements*. Zoals je ziet, eindigt elk statement met een puntkomma.
Als eerste worden in de functie `main()` drie variabelen gedefinieerd: `a`, `b` en `antwoord`. C++ kent veel verschillende typen voor variabelen, deze drie zijn van het type `int`. Het woord `int` is een afkorting van *integer*, wat *geheel getal* betekent. Het benoemen van het type en de naam van een variabele heet ook wel het declareren van die variabele.
In de volgende drie regels krijgen de variabelen een waarde. Dit soort opdrachten heten *assignment-statements* of toekenningsopdrachten. Het teken = heet de *assignment-operator* of toekenningsoperator. Met behulp van de assignment-operator geef je een waarde aan een variabele. De variabele staat altijd aan de linkerkant van het teken =. Zo'n variabele wordt ook wel een *modifiable lvalue* genoemd. De letter l van lvalue staat voor *left*.
De in- en uitvoer van gegevens wordt in C++ geregeld via *streams*, ofwel stromen van informatie. Er is bijvoorbeeld een stream van je programma naar het beeldscherm. In voorbeeld 1.1 staan twee opdrachten die met `std::cout` (*cout* spreek je uit als: cie-out) beginnen. Deze regels zorgen voor de uitvoer naar het beeldscherm. Het symbool <<, dat uit twee kleinerdantekens bestaat, moet je opvatten als één symbool. Dit symbool heet de *insertion operator* of uitvoeroperator. Met behulp van de uitvoeroperator stuur je iets naar `std::cout`.
Het een-na-laatste statement van het programma is:

```
std::cin.get();
```

Deze opdracht wacht tot je op de entertoets drukt. Sommige ontwikkelomgevingen hebben de gewoonte om het uitvoerscherm in een flits te tonen, te kort om iets van de uitvoer te kunnen zien. Door aan het eind van je programma de opdracht `std::cin.get()` neer te zetten, blijft het uitvoerscherm zichtbaar tot je op Enter drukt. Om ruimte te sparen zal ik in de rest van de voorbeelden in dit boek deze opdracht niet steeds neerzetten, zie ook voorbeeld 1.1a in paragraaf 1.3.2.
In C++ kan een functie een waarde afleveren en de betekenis van `return 0` is dat de functie `main()` de waarde `0` aflevert. In principe kun je deze waarde opvragen vanuit het besturingssysteem. Na afloop van het programma kun je aan de waarde `0` zien dat het programma succesvol is geëindigd. Eventueel kun je het programma een andere waarde dan `0` laten afleveren als er een fout optreedt. Die waarde geeft dan aan dat er iets mis is. Als je als programmeur een lijstje levert

van de mogelijke foutcodes en hun betekenis, kan de gebruiker van je programma nagaan wat er mis is.
Volgens standaard-C++ is het niet noodzakelijk `main()` af te sluiten met `return 0`. Om ruimte te sparen in de voorbeelden zal ik deze opdracht dan ook weglaten, zie ook voorbeeld 1.1a in paragraaf 1.3.2.
Alle namen in C++ zijn opgeborgen in een zogeheten namespace. Een namespace is een verzameling namen die zelf weer een naam heeft. Dit is een manier om naamconflicten te vermijden, die zich snel kunnen voordoen als je met meerdere programmeurs aan een groot project werkt. Een namespace is wat dat betreft vergelijkbaar met een directory. De namespace `std` is een belangrijke namespace. De in voorbeeld 1.1 gebruikte namen `cin` en `cout` maken deel uit van deze namespace, en daarom heten ze voluit `std::cin` en `std::cout`. Meer over namespaces kun je lezen paragraaf 1.4.
Alles wat in de broncode achter twee slashes op dezelfde regel staat, is commentaar of een toelichting voor de menselijke lezer. Het commentaar wordt door de compiler overgeslagen en heeft dus geen invloed op het vertaalde resultaat. Als je meer dan een regel commentaar hebt, kun je dit beginnen met slash sterretje (`/*`) en eindigen met sterretje slash (`*/`). Bijvoorbeeld:

```
/* Dit is ook
   commentaar
   dat 3 regels beslaat */
```

1.3.2 Declaratie van variabelen

In C++ moet je variabelen declareren voordat je ze kunt gebruiken. In plaats van

```
int a;
int b;
int antwoord;
```

mag je ook schrijven:

```
int a, b, antwoord;
```

De typeaanduiding `int` geldt voor alle variabelen die erachter komen, tot aan de eerstvolgende puntkomma. De variabelen moeten van elkaar gescheiden worden door een komma. Als je zo veel variabelen hebt dat ze niet meer op een regel passen, kun je gewoon op de volgende regel doorgaan. De compiler begrijpt ook een schrijfwijze als:

```
int a, b,
    antwoord;
```

Een wat kortere versie van voorbeeld 1.1 zie je in onderstaande code:

Voorbeeld 1.1a

```
#include <iostream>

int main()
{
  int a, b, antwoord;          // declaratie van drie variabelen in een keer
  a = 17;
  b = 24;
  antwoord = a + b;

  std::cout << "Het resultaat is: " << '\n';
  std::cout << antwoord;
}
```

De uitvoer van dit programma is:

```
Het resultaat is:
41
```

In dit geval komt de uitvoer op twee verschillende regels, omdat `'\n'` ervoor zorgt dat de cursor in de uitvoer naar de volgende regel gaat. `'\n'` is een zogeheten *escape character*, zie ook paragraaf 1.8.2.
De naam van een variabele heet ook wel een *identifier*. Niet elke identifier is geschikt als naam voor een variabele. Namen van variabelen in C++ moeten voldoen aan de volgende regels:

- Een naam mag je opbouwen uit kleine letters, hoofdletters, cijfers en de *underscore* _.
- Een naam mag nooit beginnen met een cijfer.
- Een naam mag niet een gereserveerd woord (*reserved word*) of een voorgedefinieerde identifier (*predefined identifier*) zijn.

Een gereserveerd woord is een woord dat in de taal C++ een speciale betekenis heeft. Een voorgedefinieerde identifier is een naam die al eens gebruikt is, maar niet tot de taal zelf behoort. Gereserveerde woorden heten ook wel *keywords*. Een lijst van keywords vind je in bijlage D.
Namen die aan de genoemde regels voldoen zijn bijvoorbeeld:

```
X       x               a23
Hoogte  ANNA            _a23
Hoogte  breedte1        x_1
x1      breedte_1       x_2
x2      Dit_is_een_naam aantalPersonen
```

```
2a                      // begint met een cijfer
Dit is geen naam        // bevat spaties
d'66                    // bevat apostrof
4711                    // begint met een cijfer
C&A                     // bevat & (ampersand)
C++                     // bevat plustekens
```

C++ is *case sensitive* en maakt dus verschil tussen hoofdletters en kleine letters: `hoogte` is een andere variabele dan `Hoogte` (en `HOOGTE` is weer een andere). Je mag namen zo lang maken als je wilt, maar er zijn compilers die onderscheid maken tussen (bijvoorbeeld) alleen de eerste 32 tekens.
Voor de zelfgekozen namen van variabelen in dit boek houd ik mij aan de volgende regels:

- een identifier begint met een kleine letter;
- als een identifier uit twee of meer woorden bestaat, zet je een underscore tussen de woorden. Bijvoorbeeld: `aantal_euros`, `nieuw_saldo`, `percentage_eerste_schijf`;
- identifiers geven bij voorkeur zo precies mogelijk aan wat de betekenis van de variabele is. De naam `temperatuur` of `tijd` is veel duidelijker dan de naam `t`. En `nieuw_saldo` en `oud_saldo` zijn duidelijker dan `saldo1` en `saldo2`.

Er zijn andere goede regels denkbaar dan deze. Zo zijn er veel programmeurs die juist wel underscores gebruiken, `nieuwSaldo` wordt dan `nieuw_saldo`. In elk geval raad ik elke beginnende C++-programmeur sterk aan een keuze te maken en de regels consequent in eigen programma's toe te passen. Hierdoor worden ze – voor jezelf en voor anderen – leesbaarder en begrijpelijker.

1.4 Namespaces

Een namespace is enigszins vergelijkbaar met een folder. Zoals een folder bestanden bevat, bevat een namespace namen. In C++ hebben veel zaken een naam: variabelen, constanten, functies, klassen.
Een van de plezierige dingen van folders is dat je er daardoor niet voor hoeft te zorgen dat alle namen uniek zijn: in twee verschillende folders kun je heel goed twee (verschillende) bestanden hebben met dezelfde naam, bijvoorbeeld `test.cpp`.
Je kunt ze van elkaar onderscheiden door de folder te noemen waar ze in zitten:

`folder1/test.cpp` en `folder2/test.cpp`

of in Windows:

`folder1\test.cpp` en `folder2\test.cpp`

Omdat veel programmeerproblemen overeenkomsten bevatten, hebben programmeurs de neiging vaak dezelfde (Engelse) namen te kiezen: zoals `string`, `list`, `vector`, `count`. Als je tijdens het schrijven van een programma gebruikmaakt van het werk van anderen, bijvoorbeeld van een van de bibliotheken die standaard bij C++ geleverd worden, waar honderden of duizenden namen in zitten, kan het lastig zijn steeds te moeten nagaan of een naam die je zelf bedenkt misschien al in gebruik is. Als dat zo is, klaagt de compiler dat de naam al bestaat: je hebt een *name clash* of *naming collision*: een naamconflict.
De oplossing is de namen onder te brengen in verschillende namespaces. Programmeur a gebruikt bijvoorbeeld `namespace a`, en programmeur b gebruikt `namespace b`. Beide programmeurs zorgen ervoor dat bij definitie van namen deze in hun eigen namespace komen. Als programmeur a een functie `count()` maakt, is de volledige naam (*fully qualified name*) van die functie `a::count()`. En als b ook een functie `count()` maakt, heeft die als volledige naam `b::count()`. Op die manier kun je vrij eenvoudig functies uit verschillende namespaces uit elkaar houden, of ze nu dezelfde naam hebben of niet.
Een belangrijke namespace is `std`, wat een afkorting is van *standard*, deze namespace is onderdeel van de standard library van C++.
De namespace `std` bevat veel namen die je vaak gebruikt, bijvoorbeeld `cout`, `cin` en `string`. Hun volledige naam luidt dan ook: `std::cout`, `std::cin` en `std:string`. Dat betekent dat je in een programma vaak `std::` moet tikken, zoals in voorbeeld 1.1 drie keer. Het intikken van `std::` is op den duur nogal vervelend, zeker als je dat tientallen of honderden keren moet doen. Om dat te voorkomen, kun je vanaf C++17 een `using`-statement gebruiken, zoals in dit voorbeeld:

```
int main() {
    using std::cout, std::cin, std::string;
    ...
}
```

Het statement `using std::cout` wil zeggen dat als je daarna `cout` gebruikt, je dit moet opvatten als `std::cout`. Iets dergelijks geldt voor `cin` en `string`.
Voorbeeld 1.1a komt er met een using statement als volgt uit te zien:

Voorbeeld 1.1b

```
#include <iostream>

int main() {
  using std::cout;          // using statement voor std::cout
  int a, b, antwoord;       // declaratie van drie variabelen in een keer
  a = 17;
  b = 24;
  antwoord = a + b;
```

```
  cout << "Het resultaat is: " << '\n';  // cout zonder std::
  cout << antwoord;
}
```

1.4.1 Over using namespace std

In veel kleinere programma's, en dus in veel leerboeken of in online uitleg, vind je vaak de volgende opdracht bovenaan het programma:

```
using namespace std;

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

Het voordeel hiervan is dat je nu *alle* namen uit `std` kunt gebruiken zonder `std::` te typen. Het grote bezwaar hiervan is dat er erg veel namen in `std` zitten, wat heel makkelijk tot een naamconflict kan leiden. Het idee van namespaces was nu juist om dat te voorkomen.
In principe is het dus beter om `using namespace std` niet te gebruiken. Nu kan dat in een basaal programma van tien regels in een leerboek niet zoveel kwaad: alles is goed te overzien, en een eventuele name clash is snel opgelost. Maar beginnende programmeurs worden gevorderde programmeurs, het aantal regels code kan toenemen tot duizenden, en daarmee de onoverzichtelijkheid over welke namen wel of niet in gebruik zijn. Naamconflicten kun je beter voorkomen. Om die reden zal ik in dit boek `using namespace std` niet gebruiken, maar steeds bij een naam expliciet aangeven uit welke namespace hij komt, zoals in `std::cout`, of met `using` aangeven welke namen uit de namespace gebruikt worden, zoals in `using std::cout;`.

1.5 Getaltypen

Naast het type `int` kent C++ nog drie typen voor gehele getallen: `short`, `long` en `long long`. Op veel systemen wordt een getal van het type `short` in twee bytes opgeslagen en heeft dan een bereik van -32768 tot en met 32767. Een `int` wordt doorgaans in vier bytes opgeslagen en heeft een bereik van -2147483648 tot en met 2147483647. Het bereik van `short` is dus veel kleiner dan dat van `int`, maar daar staat tegenover dat het geheugengebruik slechts de helft is. Op veel systemen wordt een `long` in acht bytes opgeslagen, waardoor het bereik van `long` aanzienlijk groter is dan dat van `int`. Je zou verwachten dat een `long long` dan in zestien bytes wordt opgeslagen, maar dat hoeft niet het geval te zijn. De standaard eist slechts dat voor het bereik geldt:

```
short ≤ int ≤ long ≤ long long
```

Het is dus van belang na te gaan welke waarden worden gehanteerd door de compiler die je gebruikt.
De integer-getaltypen kennen ook een `unsigned`-variant: gehele getallen zonder teken, dus met uitsluitend niet-negatieve waarden. Je krijgt dergelijke typen door het woord `unsigned` voor het getaltype te zetten. Een dergelijk woord heet wel een *type modifier* of kortweg *modifier*. Bijvoorbeeld:

```
unsigned short x;
unsigned int xx;
unsigned long xxx;
unsigned long long xxxx;
```

Als het bereik van een `short` loopt van -32768 tot en met 32767, dan omvat het bereik van `unsigned short` de getallen van 0 tot en met 65535. Wanneer je het negatieve getal -1 opbergt in een `unsigned short`, wordt de waarde omgezet in 65535. Het getal -2 wordt omgezet in 65534 et cetera.
Voor de andere `unsigned`-typen geldt iets dergelijks. Zie voor een overzicht van het bereik van verschillende getaltypen de tabel aan het eind van deze paragraaf.
Voor gebroken getallen (getallen met een decimale punt) kent C++ de typen `float`, `double` en `long double`.

Naam integer-type	Mogelijk aantal bytes	Kleinste waarde	Grootste waarde	
`short`	2	`-32768`	`32767`	
`unsigned short`	2	`0`	`65535`	
`int`	4	`-2147483648`	`2147483647`	
`unsigned int`	4	`0`	`4294967295`	
`long`	8	`-9223372036854775808`	`9223372036854775807`	
`unsigned long`	8	`0`	`18446744073709551615`	
`long long`	8	`-9223372036854775808`	`9223372036854775807`	
`unsigned long long`	8	`0`	`18446744073709551615`	
floating point-type				**Precisie**
`float`	4	`3.4e-38`	`3.4e+38`	7 cijfers
`double`	8	`1.7e-308`	`1.7e+308`	15 cijfers
`long double`	10	`3.4e-4932`	`1.1e+4932`	19 cijfers

In veel C++-implementaties kun je in een `float` getallen opbergen van 3.4 10^{-38} tot 3.4 10^{+38}, zowel positief als negatief. Een `float` wordt weergegeven met een precisie van zeven cijfers. Dat wil zeggen dat de overige cijfers achter de decimale punt worden weggelaten door afronding.
Het type `double` is met 15 cijfers een stuk nauwkeuriger dan `float`.
Een notatie als 2.3 10^{+8} met een macht van 10 heet ook wel de wetenschappelijke notatie (Engels: *scientific notation*). In C++ schrijf je in plaats van de 10 de letter `e` of `E`. Dus 2.3 10^{+8} wordt bijvoorbeeld 2.3e+8 of 2.3E8 en 3.15 10^{-12} wordt 3.15e-12.
In deze tabel staat een overzicht van het aantal bytes en het bereik van getaltypen zoals die in een specifieke C++-implementatie kunnen voorkomen. Bij het floating-pointtype zijn in de tabel bij de kleinste en grootste waarde alleen positieve getallen aangegeven, maar dergelijke grenzen gelden ook in het negatieve gebied. Houd er rekening mee dat de implementatie die je gebruikt kan afwijken van de gegevens in deze tabel. Met de operator `sizeof` kun je het aantal bytes van elk type opvragen, zie voorbeeld 1.2.

Voorbeeld 1.2 **Aantal bytes opvragen met sizeof**

```
#include <iostream>

int main()
{
  using std::cout;
  cout << "short: "       << sizeof(short) << " bytes" << '\n';
  cout << "int: "         << sizeof(int) << " bytes" << '\n';
  cout << "long: "        << sizeof(long) << " bytes" << '\n';
  cout << "long long: "   << sizeof(long long) << " bytes"
       << '\n';
  cout << "float: "       << sizeof(float) << " bytes" << '\n';
  cout << "double: "      << sizeof(double) << " bytes" << '\n';
  cout << "long double: " << sizeof(long double)
       << " bytes" << '\n';
}
```

De C++-implementatie die ik momenteel gebruik (g++-10) geeft als uitvoer:

```
short: 2 bytes
int: 4 bytes
long: 8 bytes
long long: 8 bytes
float: 4 bytes
double: 8 bytes
long double: 16 bytes
```

Deze geheel herziene zevende druk van *Aan de slag met C++* is een inleiding in de objectgeoriënteerde programmeertaal C++ volgens de laatste standaarden C++11, C++14 en C++20.

In een heldere, aansprekende stijl zet de auteur eerst de basisprincipes van de taal uiteen. Vervolgens komen onder meer de volgende onderwerpen aan de orde: het ontwerpen van klassen, het werken met objecten, overerving, dynamisch geheugen, templates, polymorfie, streams en exception handling. Daar waar nodig verhelderen UML-klassendiagrammen de broncode.
Verder biedt dit boek een nauwkeurige inleiding in alle belangrijke begrippen uit de standaardbibliotheek (STL) zoals vector, list, stack, queue, deque, iterator, algoritmen, functieobjecten en lambdafuncties.

In de zevende druk is het gebruik van het globale statement 'using namespace std;' zoveel mogelijk vermeden. Uniforme initialisatie wordt vrijwel overal toegepast.

Via de website **www.aandeslagmetcpp.nl** is aanvullend materiaal beschikbaar. De broncode van alle voorbeelden uit het boek is hier te downloaden. Verder vind je hier het online boek met extra hoofdstukken ter verdieping en diverse bijlagen. Daarnaast zijn de antwoorden op de vragen en uitwerkingen van de opdrachten die aan het eind van elk hoofdstuk staan op de website te raadplegen.

Aan de slag met C++ is geschikt voor programmeercursussen in het informatica onderwijs. Het boek kan ook ingezet worden voor zelfstudiedoeleinden.

Gertjan Laan (www.gertjanlaan.nl) was als docent werkzaam in het hoger onderwijs. Hij gaf jaren les in wiskunde en programmeertalen, waaronder C++ en Java. Hij is auteur van diverse succesvolle boeken, waaronder *Aan de slag met Java en JavaFX*.

ISBN 978-90-244-3861-7

www.aandeslagmetcpp.nl
www.boomhogeronderwijs.nl