Inhoud

Voorwoord

Als je iemand wilt onderrichten, wees dan kort.

– Cicero

C++ lijkt haast een nieuwe taal. Ik kan mijn ideeën duidelijker, eenvoudiger en directer uitdrukken dan in C++98. Bovendien worden de resulterende programma's beter gecontroleerd door de compiler en sneller uitgevoerd.

Dit boek geeft een overzicht van C++ zoals gedefinieerd door C++17, de huidige ISO C++-standaard, en geïmplementeerd door de belangrijkste C++-leveranciers. Daarnaast is er sprake van concepten en modules, zoals gedefinieerd in de ISO Technical Specifications en in huidig gebruik, maar niet gepland voor opname in de standaard tot C++20.

Net als andere moderne talen is C++ uitgebreid, met een groot aantal bibliotheken die nodig zijn voor effectief gebruik. Dit boek is bedoeld om de ervaren programmeur een idee te geven van modern C++. Het behandelt de belangrijkste taalkenmerken en componenten in de standaardbibliotheek. Dit boek is in een paar uur te lezen, maar er komt uiteraard veel meer kijken bij het schrijven van goede C++ dan u in één dag kunt leren. Het doel is hier echter niet meesterschap, maar om een overzicht en sleutelvoorbeelden te geven waarmee een programmeur aan de slag kan.

We gaan ervan uit dat u al ervaring hebt met programmeren. Als dat niet het geval is, neem dan voordat u verder gaat een goed leerboek ter hand. Misschien hebt u eerder geprogrammeerd, maar de taal die u gebruikte of de applicaties die u hebt geschreven kunnen totaal verschillen van de C++-manier die we hier presenteren.

Zie het boek als een rondleiding door een stad als Kopenhagen of New York. In een paar uur krijgt u wat grote attracties te zien en een paar achtergrondverhalen of suggesties over wat u verder kunt gaan doen. Na zo'n rondleiding kent u de stad nog niet. U begrijpt nog lang niet alles wat u gezien en gehoord hebt. Want wat weet u van de geschreven en ongeschreven regels die het leven in de stad beheersen? Om een stad echt te leren kennen, moet je er vaak jarenlang wonen. Met een beetje geluk hebt u misschien enig inzicht gekregen in wat de stad speciaal maakt en voor u mogelijk van belang is. Na de rondleiding kan de echte verkenning beginnen.

Deze rondleiding presenteert de belangrijkste functionaliteiten van C++ die programmeerstijlen zoals objectgeoriënteerde en generieke programmering

ondersteunen. Ons doel is niet een gedetailleerd naslagwerk met een opsomming van alle functionaliteit van de taal. In de beste leerboektraditie probeer ik een functionaliteit uit te leggen voordat ik deze gebruik, maar dat is niet altijd mogelijk en niet iedereen leest de tekst regel voor regel. Daarom wordt de lezer aangemoedigd de verwijzingen en de inhoudsopgave te gebruiken.

Op dezelfde manier presenteren we in deze rondleiding de standaardbibliotheken niet uitputtend, maar in voorbeelden. We behandelen geen bibliotheken die buiten de ISO-norm vallen. De lezer kan ondersteunend materiaal zo nodig zelf doornemen. [Stroustrup, 2013] en [Stroustrup, 2014] zijn voorbeelden van dergelijk materiaal, maar er is een enorme hoeveelheid materiaal (van uiteenlopende kwaliteit) te vinden op internet, bijvoorbeeld [Cppreference]. Wanneer ik het bijvoorbeeld heb over een standaardbibliotheekfunctie of -klasse, kan de definitie daarvan eenvoudig worden opgezocht; door de documentatie te bekijken kunt u veel gerelateerde mogelijkheden vinden.

Deze rondleiding presenteert C++ als een geïntegreerd geheel, niet als een 'lagentaart'. Daarom worden taalkenmerken niet opgegeven als aanwezig in C, of onderdeel van C++98, nieuw in C++11, C++14 of C++17. Dergelijke informatie is te vinden in hoofdstuk 16 (Geschiedenis en compatibiliteit). Ik concentreer me op de grondbeginselen en probeer beknopt te zijn, maar ik kon de verleiding niet helemaal weerstaan om nieuwe functionaliteit uitgebreid te behandelen. Dit lijkt ook tegemoet te komen aan de nieuwsgierigheid van veel lezers die al een oudere versie van C++ kennen.

In een referentiehandboek of standaardwerk van de programmeertaal kunnen we eenvoudig opzoeken wat er kan worden gedaan, maar programmeurs zijn vaak meer geïnteresseerd in hoe ze de taal goed leren gebruiken. Dit aspect komt deels aan bod in de selectie behandelde onderwerpen, deels in de tekst, en met name in de afsluitende adviesrubrieken. Meer adviezen voor *goed modern* C++ zijn te vinden in de C ++ Core Guidelines [Stroustrup, 2015]. Deze richtlijnen zijn een goede bron voor verdere verkenning van de ideeën die in dit boek worden gepresenteerd. U zult een opmerkelijke gelijkenis zien in de formulering en zelfs de nummering van de adviezen tussen de Core Guidelines en dit boek. Dat komt onder andere doordat de eerste editie van *A Tour of C ++* een belangrijke bron was voor de eerste Core Guidelines.

Hoofdstuk 1

De grondbeginselen

We gaan allereerst alle taalpuristen om zeep helpen.

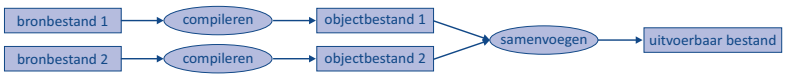
– Henry VI, deel II

1.1 Introductie

In dit hoofdstuk behandelen we de notatie van C++, het geheugen- en rekenmodel en de basismechanismen voor het ordenen van code in een programma. Dit zijn de taalfaciliteiten die de stijlen ondersteunen die we het vaakst in C tegenkomen en soms procedureel programmeren worden genoemd.

1.2 Programma's

C++ is een gecompileerde taal. De brontekst wordt daarbij door een compiler verwerkt tot objectbestanden die op hun beurt door een linker worden samengevoegd tot een uitvoerbaar programma. Een C++-programma bestaat meestal uit veel broncodebestanden (meestal eenvoudigweg bronbestanden of source files genoemd).



Een uitvoerbaar programma is bestemd voor een specifieke hardware-systeemcombinatie; het is bijvoorbeeld niet overdraagbaar van een Mac naar een Windows-pc. Wanneer we het hebben over de overdraagbaarheid van C++-programma's, bedoelen we meestal overdraagbaarheid van broncode; dat wil zeggen, de broncode kan met succes worden gecompileerd en op verschillende systemen worden uitgevoerd.

De ISO C++-standaard definieert twee soorten entiteiten:

- *In de taal zelf ingebakken functionaliteit (core language features)*, zoals ingebouwde typen (zoals `char` en `int`) en loops (zoals `for`- en `while`-statements)
- *Componenten uit de standaardbibliotheek*, zoals containers (`vector` en `map`) en I/O-bewerkingen (`<<` and `getline()`)

De componenten uit de standaardbibliotheek zijn gewone C++-code die door elke C++-implementatie worden aangeboden. Dat wil zeggen, de C++-standaardbibliotheek kan in C++ zelf worden geïmplementeerd en wordt dat ook, met zeer beperkt gebruik van machinecode voor zaken zoals thread context switching. Dit betekent dat C++ voldoende expressief en efficiënt is voor de meest veeleisende systeemprogrammeertaken.

C++ is een statisch getypeerde taal. Het type van elke entiteit (object, waarde, naam of expressie) moet op het moment van gebruik bekend zijn bij de compiler. Het type van een object bepaalt de set bewerkingen die daarop van toepassing zijn.

1.2.1 Hello, World!

Het kleinste C++-programma is

```
int main() { } //      het kleinste C++-programma
```

Dit definieert een functie met de naam `main`, die geen argumenten heeft en verder niets doet.

Accolades `{}` zorgen voor groepering in C++. Hier geven ze het begin en het einde van een functie aan. De dubbele slash `//` geeft het begin van commentaar aan dat zich uitstrekt tot het einde van de regel. Commentaar is voor de menselijke lezer; de compiler negeert het.

Elk C++-programma moet exact één globale functie hebben met de naam `main()`. Een programma start met de uitvoering van die functie. De waarde van de integer `int` die door `main()` wordt geretourneerd, is de geretourneerde waarde van het programma naar 'het systeem'. Als er geen waarde wordt geretourneerd, ontvangt het systeem een waarde die aangeeft dat het programma met succes is voltooid. Een waarde vanuit `main()` die ongelijk is aan 0 geeft een fout aan. Niet elk besturingssysteem en elke uitvoeringsomgeving gebruikt die returnwaarde: op Linux/Unix gebaseerde omgevingen wel, maar op Windows-systemen zelden.

Doorgaans produceert een programma enige uitvoer. Hier is een programma dat `Hello, World!` afdrukt:

```
#include <iostream>

int main()
{
      std::cout << "Hello, World!\n";
}
```

De regel `#include <iostream>` instrueert de compiler om de declaraties van de voorzieningen van de standaard I/O-kanalen in de broncode in te voegen, zoals te vinden in `iostream`. Zonder deze declaraties snijdt de expressie

```
std::cout << "Hello, World!\n"
```

geen hout. De operator `<<` (‘schrijf naar’) schrijft het tweede argument naar het eerste. In dit geval wordt de string `"Hello, World!\n"` geschreven naar de standaarduitvoerstream `std::cout`. Een string is een reeks tekens die wordt omgeven door dubbele aanhalingstekens. In een string verwijst een backslash `\` die gevolgd wordt door een ander teken naar een ‘speciaal teken’. In dit geval is `\n` het nieuweregelteken, zodat de geschreven tekens `Hello, World!` worden gevolgd door een nieuwe regel.

`std::` geeft aan dat de naam `cout` te vinden is in de naamruimte (*namespace*) van de standaardbibliotheek (§3.4). Meestal laat ik `std::` achterwege bij het bespreken van standaardfuncties. In §3.4 gaan we zien hoe we namen uit een namespace ‘zichtbaar’ maken zonder dat we die expliciet benoemen.

Feitelijk wordt alle uitvoerbare code in functies geplaatst en direct of indirect aangeroepen vanuit `main()`. Bijvoorbeeld:

```
#include <iostream>        // include ('importeer') de declaraties voor de bibliotheek I/O-stream

using namespace std;       // maak namen van std zichtbaar zonder std:: (§3.4)

double square(double x)    // kwadrateer een drijvendekommagetal met dubbele precisie
{
    return x*x;
}

void print_square(double x)
{
    cout << "the square of" << x << "is" << square(x) << "\n";
}

int main()
{
    print_square(1.234);   // uitvoer naar het scherm: the square of 1.234 is 1.52276
}
```

Het ‘returntype’ `void` geeft aan dat de functie geen waarde retourneert.

1.3 Functies

Dé manier om iets gedaan te krijgen in een C++-programma is door een functie aan te roepen. Door een functie te definiëren geeft u aan hoe een bewerking moet worden uitgevoerd. Een functie kan alleen worden aangeroepen als deze al eerder is gedeclareerd.

Een functiedeclaratie geeft de naam van de functie, het type van de geretourneerde waarde (indien aanwezig) en het aantal en de typen parameters die in een aanroep moeten worden meegenomen. Bijvoorbeeld:

```
Elem* next_elem();      // geen parameter; retourneer een pointer naar Elem (een Elem*)
void exit(int);         // parameter int; retourneer niets
double sqrt(double);    // parameter double; retourneer een double
```

In een functiedeclaratie komt het returntype voor de naam van de functie en de parametertypen komen na de naam tussen haakjes.

De semantiek van het doorgeven van argumenten is identiek aan de semantiek van initialisatie (§3.6.1). Dat wil zeggen dat de argumenttypen worden gecontroleerd en impliciete conversies van het argumenttype plaatsvinden wanneer dat nodig is (§1.4). Bijvoorbeeld:

```
double s2 = sqrt(2);          // roep sqrt() aan met het argument double{2}
double s3 = sqrt("three");    // fout: sqrt() vereist een argument van het type double
```

De waarde van een dergelijke compilatiecontrole en typeconversie mag niet worden onderschat.

Een functiedeclaratie kan parameternamen bevatten. Dit kan de lezer van een programma helpen, maar omdat een declaratie geen functiedefinitie is, negeert de compiler eenvoudig dergelijke namen. Bijvoorbeeld:

```
double sqrt(double d);  // retourneer de wortel van d
double square(double);  // retourneer het kwadraat van het argument
```

Het type van een functie bestaat uit het returntype en de reeks parametertypen. Bijvoorbeeld:

```
double get(const vector<double>& vec, int index);     // type: double(const vector<double>&,int)
```

Een functie kan lid (*member*) zijn van een klasse (*class*) (§2.3, §4.2.1). Voor zo'n *lidfunctie* maakt de naam van zijn klasse ook deel uit van het functietype. Bijvoorbeeld:

```
char& String::operator[](int index);                // type: char& String::(int)
```

We willen dat onze code begrijpelijk is, want dat is de eerste stap op weg naar onderhoudbaarheid. De eerste stap naar begrijpelijkheid is het opdelen van rekentaken in zinvolle blokken (weergegeven als functies en klassen) en goede naamgeving. Zulke functies leveren immers de basiswoordenschat voor bereke-

ningen, net zoals (ingebouwde en zelfgedefinieerde) typen de basiswoordenschat voor data verschaffen. De standaard C++-algoritmen (zoals `find`, `sort` en `iota`) bieden een goede start (hoofdstuk 12). Vervolgens kunnen we functies samenstellen voor algemene of gespecialiseerde taken in grotere berekeningen.

Het aantal fouten in de code hangt sterk samen met de hoeveelheid code en de complexiteit daarvan. Beide problemen kunnen worden aangepakt met behulp van meer en kortere functies. Door een functie voor een specifieke taak te schrijven, besparen we ons de moeite die code in het midden van andere code in te voeren. Een nieuwe functie dwingt ons een naam te geven die beschrijft wat hij doet en zijn afhankelijkheden te documenteren.

Als er twee functies worden gedefinieerd met dezelfde naam maar met verschillende parametertypen, kiest de compiler de geschiktste functie voor elke aanroep. Bijvoorbeeld:

```
void print(int);     // neemt een integer als argument
void print(double);  // neemt een double als argument
void print(string);  // neemt een string als argument

void user()
{
     print(42);              // roept print(int) aan
     print(9.65);            // roept print(double) aan
     print("Barcelona");     // roept print(string) aan
}
```

Als twee alternatieve functies kunnen worden aangeroepen maar geen van beide beter is dan de andere, wordt de aanroep als ambigu gezien en geeft de compiler een fout. Bijvoorbeeld:

```
void print(int,double);
void print(double,int);

void user2()
{
     print(0,0);     // fout: ambigu
}
```

Het definiëren van meerdere functies met dezelfde naam staat bekend als functieoverlading (*function overloading*) en is een van de essentiële onderdelen van generiek programmeren (§7.2). Wanneer een functie overladen is, moet elke functie met dezelfde naam dezelfde semantiek implementeren. De `print()`-functies zijn hiervan een voorbeeld; elke `print()` drukt zijn argument af.

1.4 Rekenen met typen en variabelen

Elke naam en elke expressie heeft een type dat bepaalt welke bewerkingen erop kunnen worden uitgevoerd. Bijvoorbeeld de declaratie

```
int inch;
```

geeft aan dat inch van het type `int` is; dat wil zeggen, inch is een integervariabele.

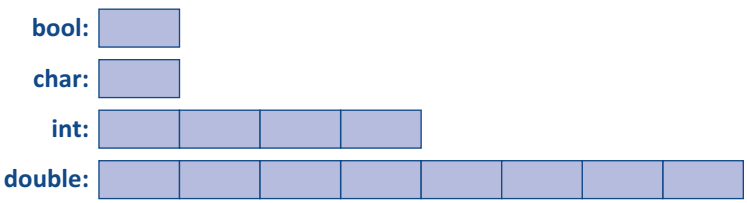
Een *declaratie* is een statement dat een entiteit in het programma introduceert. Ze specificeert een type voor de entiteit:

- Een *type* definieert een set mogelijke waarden en bewerkingen (voor een object).
- Een *object* is een stuk geheugen dat een waarde van een bepaald type bevat.
- Een *waarde* is een verzameling bits die wordt geïnterpreteerd volgens een type.
- Een *variabele* is een object met een naam.

C++ bezit een kleine dierentuin met basistypen, maar aangezien ik geen zoöloog ben, zal ik ze niet allemaal opsommen. U vindt ze allemaal in referentiebronnen, zoals [Stroustrup, 2013] of de [Cppreference] op het web. Voorbeelden zijn:

```
bool      // Boolean, mogelijke waarden zijn true en false
char      // teken, bijvoorbeeld 'a', 'z' en '9'
int       // integer (geheel getal), bijvoorbeeld -273, 42 en 1066
double    // gebroken getal met dubbele precisie, bijvoorbeeld -273.15, 3.14 en 6.626e-34
unsigned  // positief geheel getal, bijvoorbeeld 0, 1 en 999 (voor bitgewijze logische bewerkingen)
```

Elk basistype staat in direct verband met de mogelijkheden van de hardware en heeft een vaste grootte die het bereik van de waarden bepaalt die daarin kunnen worden opgeslagen:



Een `char`-variabele heeft de natuurlijke grootte voor een teken op een bepaalde machine (meestal een 8-bits byte); de afmetingen van andere typen zijn veelvouden van de grootte van een `char`. De grootte van een type wordt door de implementatie gedefinieerd (dit kan per machine verschillen) en kan worden opgevraagd met de operator `sizeof`; bijvoorbeeld `sizeof(char)` is gelijk aan `1` en `sizeof(int)` is vaak `4`.

Getallen kunnen decimalen hebben of geheel zijn.

- Niet-gehele getallen zijn te herkennen aan het decimaalteken (zoals `3.14`) of door een exponent (zoals `3e-2`).
- Integers (gehele getallen) worden standaard in het decimale stelsel weergegeven (`42` betekent tweeënveertig). Een voorvoegsel `0b` staat voor een binaire integer (base 2, bijvoorbeeld 0b10101010). Een voorvoegsel `0x` staat voor een hexadecimale integer (base 16, bijvoorbeeld `0xBAD1234`). Een voorvoegsel `0` staat voor een octale integer (base 8, bijvoorbeeld `0334`).

Om lange waarden beter leesbaar te maken, kunnen we een enkel aanhalingsteken (`'`) gebruiken als een separator voor getallen. Bijvoorbeeld π is ongeveer `3.14159'26535'89793'23846'26433'83279'50288` of als u de voorkeur geeft aan hexadecimale notatie `0x3.243F'6A88'85A3'08D3`.

1.4.1 Rekenen

De rekenkundige operatoren kunnen worden gebruikt voor aangewezen combinaties van fundamentele typen:

```
x+y     // som
+x      // unair plus
x-y     // verschil
-x      // unair min
x*y     // vermenigvuldiging
x/y     // deling
x%y     // rest (modulo) voor integers
```

Zo ook de vergelijkingsoperatoren:

```
x==y    // gelijk
x!=y    // niet gelijk
x<y     // minder dan
x>y     // groter dan
x<=y    // minder dan of gelijk aan
x>=y    // groter dan of gelijk aan
```

Er zijn bovendien logische operatoren:

```
x&y      // bitgewijs en
x|y      // bitgewijze of
x^y      // bitgewijze exclusief of
~x       // bitgewijs complement
x&&y     // logisch en
x||y     // logisch of
!x       // logisch niet (ontkenning)
```

Een bitgewijze logische operator levert een resultaat op van het operandtype waarvoor de bewerking op elke bit is uitgevoerd. De logische operatoren `&&` en `||` retourneren eenvoudig `true` of `false`, afhankelijk van de waarden van hun operanden.

In toewijzingen en rekenkundige bewerkingen voert C++ alle zinvolle conversies uit tussen de basistypen, zodat ze vrij kunnen worden gecombineerd:

```
void some_function()    // functie die niets retourneert
{
     double d = 2.2;    // initialiseer double
     int i = 7;         // initialiseer integer
     d = d+i;           // wijs som toe aan d
     i = d*i;           // wijs product toe aan i; let op: double d*i wordt afgekapt tot int
}
```

De conversies die in expressies worden gebruikt, worden de gebruikelijke rekenkundige conversies genoemd en moeten ervoor zorgen dat expressies worden berekend met de hoogste precisie van de operanden. Bijvoorbeeld, een optelling van een `double` en een `int` wordt berekend met behulp van drijvendekommaberekeningen met dubbele precisie.

Let op: `=` is de toewijzingsoperator en `==` test op gelijkheid.

Naast de conventionele rekenkundige en logische operatoren biedt C++ nog andere bewerkingen voor het wijzigen van een variabele:

```
x+=y     // x = x+y
++x      // met 1 ophogen: x = x+1
x-=y     // x = x-y
--x      // met 1 verlagen: x = x-1
x*=y     // vermenigvuldigen: x = x*y
x/=y     // delen: x = x/y
x%=y     // x = x%y
```

Deze operatoren zijn beknopt en gemakkelijk en worden vaak gebruikt.

De volgorde van evaluatie van expressies is van links naar rechts, behalve voor toewijzingen die van rechts naar links zijn. De volgorde van evaluatie van functieargumenten is helaas niet gespecificeerd.

1.4.2 Initialisatie

Voordat een object kan worden gebruikt, moet het een waarde krijgen. C++ biedt allerlei notaties voor initialisatie, zoals de = die hiervoor is genoemd, en een universele vorm op basis van in accolades gevatte initialisatielijsten (*initializer lists*):

```
double d1 = 2.3;                // initialiseer d1 als 2.3
double d2 {2.3};                // initialiseer d2 als 2.3
double d3 = {2.3};              // initialiseer d3 als 2.3 (de = is facultatief bij { ... })
complex<double> z = 1;          // een complex drijvendekommagetal met dubbele precisie
complex<double> z2 {d1,d2};
complex<double> z3 = {d1,d2};   // de = is facultatief bij { ... }
vector<int> v {1,2,3,4,5,6};    // een vector van integers
```

De vorm met = is traditioneel en gaat terug naar C, maar gebruik bij twijfel de algemene vorm met een {}-lijst. Op die manier hebt u in ieder geval geen last van conversies waarbij informatie verloren gaat:

```
int i1 = 7.8;       // i1 wordt 7 (verrast?)
int i2 {7.8};       // fout: conversie van floating-point naar integer
```

Helaas zijn conversies waarbij informatie verloren gaat, zoals de *narrowing conversies* van `double` naar `int` en `int` naar `char`, toegestaan en worden ze impliciet toegepast. De problemen die worden veroorzaakt door impliciete narrowing conversies zijn de prijs die we betalen voor C-compatibiliteit (§16.3).

Een constante (§1.6) moet geïnitialiseerd worden en een variabele mag alleen in uiterst zeldzame gevallen niet-geïnitialiseerd blijven. Introduceer geen naam voordat u er een geschikte waarde voor hebt. Zelfgedefinieerde typen (zoals `string`, `vector`, `Matrix`, `Motor_controller` en `Orc_warrior`) kunnen impliciet geïnitialiseerd worden gedefinieerd (§4.2.1).

Wanneer u een variabele definieert, hoeft u het type ervan niet expliciet te vermelden wanneer dit uit de geïnitialiseerde waarde kan worden afgeleid:

```
auto b = true;      // bool
auto ch = 'x';      // char
```

```
auto i = 123;        // int
auto d = 1.2;        // double
auto z = sqrt(y);    // z is van het type dat door sqrt(y) wordt geretourneerd
auto bb {true};      // bb is bool
```

Met `auto` gebruiken we meestal `=` omdat er geen sprake is van een mogelijk lastige typeconversie, maar als u de initialisatie met `{}` consistent wilt blijven gebruiken, doet u dat dan gerust.

We gebruiken `auto` waar we geen specifieke reden hebben om het type expliciet te vermelden. Die 'specifieke redenen' kunnen zijn:

- De definitie is bepalend voor een grote scope waarin we het type duidelijk zichtbaar willen maken voor lezers van onze code.
- We willen expliciet zijn over het bereik of de precisie van een variabele (bijvoorbeeld `double` in plaats van `float`).

Met behulp van `auto` vermijden we redundantie en lange typenamen. Dit is vooral belangrijk bij generiek programmeren, waarbij de programmeur het exacte type van een object moeilijk kan weten en de typenamen behoorlijk lang kunnen zijn (§12.2).

1.5 Scope en levensduur

Met een declaratie wordt de naam geïntroduceerd in een bereik (*scope*):

- *Lokale scope:* een naam die wordt gedeclareerd in een functie (§1.3) of lambda (§6.3.2) wordt een *lokale naam* genoemd. Zijn scope strekt zich uit van het punt van declaratie tot het einde van het blok waarin de declaratie voorkomt. Een *blok* wordt begrensd door accolades. Functieparameternamen zijn lokale namen.
- *Klassescope:* een naam wordt een *lidnaam* (of een *klasselidnaam*) genoemd als deze is gedefinieerd in een klasse (§2.2, §2.3, hoofdstuk 4), buiten enige functie (§1.3), lambda (§6.3.2) of `enum class` (§2.5). Zijn scope strekt zich uit vanaf de openingsaccolade van de omsluitende klassedeclaratie tot het einde van die declaratie.
- *Namespace scope:* een naam wordt een *namespace membername* genoemd als deze is gedefinieerd in een namespace (§3.4) buiten een functie, lambda (§6.3.2), class (§2.2, §2.3, hoofdstuk 4) of `enum class` (§2.5). De scope strekt zich uit vanaf het punt van declaratie tot het einde van de namespace.