Programmeermethoden

Functies & files

Walter Kosters en Jonathan Vis

week 4: 25–29 september 2023

www.liacs.leidenuniv.nl/~kosterswa/pm/

ma	di	wo	donderdag	vrijdag
			11:00–12:45 Gorlaeus zaal 1 college 4 iedereen	11:00–12:45 Snellius 303/6/7 werkcollege 4 (*), vragenuurtje Wiskundigen
			13:15–. . . Snellius 302/3/6/7 werkcollege 4 (*), vragenuur Informatici	

(*) beter bekend als het functie-practicum

Vragenuren op maandagmiddagen op afspraak.

Een functie is een zwarte doos (black box) waar informatie in gaat en informatie uit komt.

Elk C++-programma bestaat uit een stel functies, onder elkaar. Executie begint bij de functie `main`.

Sommige functies rekenen iets uit (zoals `int`-functies: geef het kwadraat van x terug), andere verrichten een taak (`void`-functies: druk een tabel af op het scherm, of afwassen; geef `void` = leeg = niets terug, doe alleen wat).

Functies hebben allerlei (soorten) parameters, die ze ook kunnen aanpassen.

Functies mogen in C++ alleen functies aanroepen die eerder gedefinieerd zijn.

Een eenvoudige `void`-functie:

```
void tekstOpScherm ( ) { // heel kort infoblokje
  cout << "Sterke tekst." << endl;
}//tekstOpScherm
```

En een eenvoudige `int`-functie:

```
int inhoud (int lengte, int breedte, int hoogte) {
  return lengte * breedte * hoogte;
}//inhoud
```

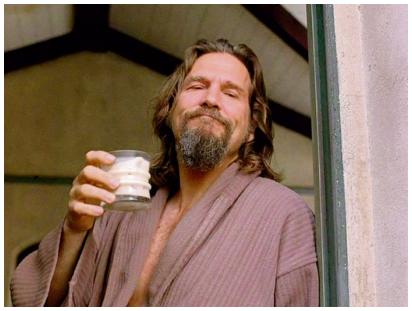
Met aanroepen:

```
tekstOpScherm ( );
cout << inhoud (16,37,42) << endl;
```

Verwissel = swap

Een functie die zijn parameters omwisselt:

```
void wissel (int & x, int & y) { // call by reference: &
  int hulp = x;
  x = y;
  y = hulp;
}//wissel
```



Met aanroep:

```
int a = 33, x = 88;
cout << a << " en " << x << endl; // 33 en 88
wissel (a,x);
cout << a << " en " << x << endl; // 88 en 33
```

Hoe werkt het functie-mechanisme?

Bij aanroep “spring” je naar de desbetreffende functie, en als die klaar is, wanneer je een `return` of de laatste `}` tegenkomt, “spring” je weer terug, en wel naar het “return-adres”. Parameters worden netjes doorgegeven.

Soms helpt het als je niet aan het “springen” denkt, maar meer denkt in termen van “deze functie verricht die taak”.

Eigenlijk komen functie-aanroepen op een stapel gevuld met “uitgestelde verplichtingen”.

```
// bereken inhoud van lengte bij breedte bij hoogte blok
double inhoud (double lengte, double breedte,
               double hoogte) {
  double temp;
  temp = lengte * breedte * hoogte;
  return temp;
}//inhoud
```

Hier zijn `lengte`, `breedte`, `hoogte` en `temp` locale variabelen, waarbij `lengte`, `breedte` en `hoogte` (de formele parameters) als startwaarde de waarde van de actuele parameters krijgen; ze worden *wel* geinitialiseerd, in tegenstelling tot `temp`. Hun scope — waar ze leven — is de functie `inhoud`. Men noemt `lengte`, `breedte` en `hoogte` wel call by value parameters. Bij een aanroep als `t = inhoud (breedte,5,x);` zijn `breedte`, `5` en `x` de actuele parameters (of variabelen).

De volgende functie bepaalt of `jaar` een schrikkeljaar is:

```
// is jaar een schrikkeljaar?
bool schrikkel (int jaar) {
   return ( jaar % 4 == 0
            && ( jaar % 400 == 0 || jaar % 100 != 0 ) );
}//schrikkel
```

Dus 1963 niet, 2023 niet, 2024 wel, 2000 wel, en 2100 niet . . .

De grootste gemeenschappelijke/gemene deler (ggd) van twee positieve gehele getallen (≥ 0, niet beide 0) wordt met het algoritme van Euclides als volgt berekend:

```
int ggd (int x, int y) {
  int rest;
  while ( y != 0 ) {
    rest = x % y;  x = y;  y = rest;
  }//while
  return x;
}//ggd
```

Voorbeeldaanroepen:

```
  cout << ggd (15,21) << endl;
  z = ggd (z,7);  // z van type int
```

Een functie kan maar één waarde retourneren = teruggeven. (Of zelfs geen, bij een `void`-functie.)

Hoe kun je dan twee of meer waarden genereren?

Antwoord: met "call by reference", let op de &.

Overigens: een `void`-functie hoeft geen `return`-statement te hebben, maar het mag wel. Er staat dan *geen* waarde achter, dus gewoon `return;` stopt zo'n functie.

```
// vereenvoudig breuk teller/noemer zoveel mogelijk
// aanname: teller >= 0, noemer > 0
void vereenvoudig (int & teller, int & noemer) {
  int deler = ggd (teller,noemer);
  if ( deler > 1 ) { // test hoeft niet
    teller = teller / deler;
    noemer = noemer / deler;
  }//if
}//vereenvoudig
```

Voorbeeldaanroep:

```
  int tel = 15, noem = 21;
  vereenvoudig (tel,noem);
  cout << tel << " " << noem << endl;
```

Boven iedere functie hoort duidelijk commentaar:

```
// vereenvoudig breuk teller/noemer zoveel mogelijk
// aanname: teller >= 0, noemer > 0
void vereenvoudig (int & teller, int & noemer) {
  ...
}//vereenvoudig
```

Tip: maak een zin waarin de functienaam en de namen van de parameters voorkomen.
En: wat geldt vooraf, en wat na afloop?

Naast call by value, waar de *waarde* van de variabele aan een "lokale kopie" wordt doorgegeven, bestaat ook call by reference, waar de variabele zelf, of preciezer: diens *adres*, wordt doorgegeven.

```
// wissel inhoud van a en b
void wissel (int & a, int & b) {
  int hulp = a;
  a = b;
  b = hulp;
}//wissel
```

Voorbeeldaanroep: `a = 8; k = 2; wissel (a,k);`

De & (ampersand) geeft aan dat het een call by reference variabele betreft.

Voorbeeld — call by value

```
void hoogop (int x) { x = x + 10; cout << x; }//hoogop

void maaknul (int t) { t = 0; cout << t; }//maaknul
```

```
int x, m, q;
x = 7; hoogop (x); cout << x;          17 7
m = 3; hoogop (m+8); cout << m;        21 3
q = 5; maaknul (q); cout << q;         0 5
maaknul (42);                          0
```

Er wordt alleen een *waarde* doorgegeven, en wel van de *actuele* parameter aan de *formele* parameter; er wordt dus een "lokale kopie" gemaakt, wat tijd en ruimte kost.

Voorbeeld — call by reference

```
void hoogop (int & x) { x = x + 10; cout << x; }//hoogop

void maaknul (int & y) { y = 0; cout << y; }//maaknul

  int x, m, q;
  x = 7; hoogop (x); cout << x;          17 17
  m = 3; hoogop (m+8); // VERBODEN!!!
  q = 5; maaknul (q); cout << q;         0 0
  maaknul (42);  // VERBODEN!!!
```

Er wordt nu een *adres* (een *pointer*) doorgegeven. De *actuele* parameter kan nu wel veranderen. De *actuele* parameter mag geen "rare" expressie als `m+8` of `42` zijn. Er wordt alleen een *adres* gekopieerd.

En dan nu: files.

Input en output voor programma's staan vaak in files, bijvoorbeeld `iets.cc`, `uitvoer.txt`, `cin` (toetsenbord) en `cout` (beeldscherm).

Voor de tweede programmeeropgave moet je een C++-programma schrijven dat een file codeert of decodeert, en het Collatz-vermoeden enigszins controleert voor getallen uit de file.

```
#include <fstream>
...
ifstream invoer ("jefile.txt", ios::in);
ofstream uitvoer ("./C++/iets.cc", ios::out);
char letter;                // zelfs / bij Windows!
...
letter = invoer.get ( );
uitvoer.put (letter);
uitvoer << "Hitchcock";
...
invoer.close ( );
uitvoer.close ( );      // niet vergeten!
```

Hier is `invoer` de *variabele* die de file voorstelt, die in het echt `jefile.txt` heet.

Een file is een "object" van "klasse" `ios`. Ook `cin` en `cout` zijn van deze klasse. Met objecten kun je bepaalde dingen doen: "memberfuncties" (= "methoden") aanroepen, zoals `get`. Je zegt dan de naam van het object, dan een punt, en dan de naam van de methode.

Voorbeelden:

```
letter = invoer.get ( );
cout.put (letter);
```

Eigenlijk is een invoerfile van klasse (= type) `ifstream`, en een uitvoerfile van klasse `ofstream`. Beide stammen af van `ios`. En `get` en `put` zijn (member)functies.

Een tekstfile, zoals een C++-programma, bestaat uit regels, gescheiden door regelovergangen (bij UNIX LF, bij Windows CR-LF). Meestal staat aan het eind ook een regelovergang, soms gevolgd door het "einde-file (EOF) symbool". Daarop kun je testen met de methode `eof ( )`.

Zo kopiëren we een file `invoer` naar een file `uitvoer`:

```
kar = invoer.get ( );   // eerst een maal get-ten!!!
while ( ! invoer.eof ( ) ) {
  uitvoer.put (kar);
  kar = invoer.get ( ); // en hier alle volgende ...
}//while
```

Het lijkt alsof er één `get` meer wordt gedaan dan `put`'s, maar de `close` zet als het ware de als laatste gelezen EOF. (Eigenlijk staat hier het UNIX-commando `cp`.)

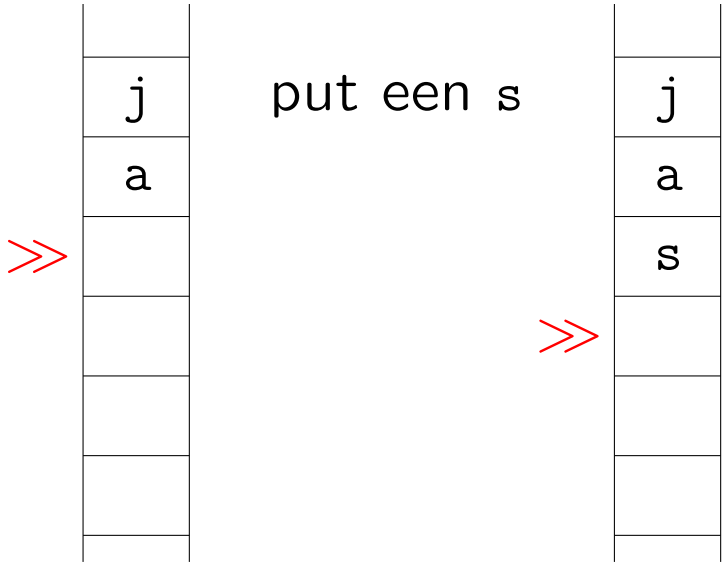
Wijzig stap voor stap zo'n kopieerprogramma:

```
kar = invoer.get ( );
while ( ! invoer.eof ( ) ) {
                              // wijzig dit
  if ( kar != '\n' )          // stap voor stap
    uitvoer.put (kar);        // voor de tweede
                              // programmeeropgave
  kar = invoer.get ( );
}//while
```

Dit kopieert, maar sloopt alle regelovergangen weg.

Meer `get`'s zijn niet nodig!

Eigenlijk werken files met filepointers, net als bij oude video-banden. Voorlopig kun je alleen vooruit spoelen.
Een `put` zet een karakter neer en schuift de filepointer één op, `get` pakt een karakter en schuift de filepointer ook op.



Iets algemener:

```
string filenaam;  // gebruik <string>
ifstream invoer;  // gebruik <fstream>
...
cin >> filenaam;
invoer.open (filenaam.c_str ( )); // in C++11 hoeft
if ( invoer.fail ( ) ) {          // ".c_str ( )" niet
  cout << filenaam << " niet te openen" << endl;
  return 1;  // of  exit (1);  of ...
}//if
```

PS En files doorgeven als parameter:

```
void doewat (ifstream & invoer, ofstream & uitvoer) ...
```

Tweede prog'opgave: DeCoderen

Voor de tweede programmeeropgave moet je een C++-programma schrijven dat een gegeven file codeert of decodeert met run-length encoding.

```
Eet␣meer␣␣␣␣␣␣␣␣␣␣zeeegels
ABC11123ddd\efG\\\1
```

moet worden:

```
Eet␣me2r␣10ze3gels
ABC\13\2\3d3\\efG\\3\1
```

Hier is ␣ een spatie.

Klopt Collatz voor 6171?

www.liacs.leidenuniv.nl/~kosterswa/pm/op2pm.php

Tweede programmeeropgave

Programmeermethoden 2023
Tweede programmeeropgave: DeCoderen

De *tweede* programmeeropgave van het vak **Programmeermethoden** in het najaar van 2023 heet *DeCoderen*; zie ook het **vierde werkcollege**, **vijfde werkcollege** (de betreffende WWW-bladzijde bevat handige tips) en **zesde werkcollege**, en lees geregeld deze pagina op WWW.

We gaan een file coderen en decoderen met behulp van een eenvoudig coderingsschema. En we controleren het Collatz-vermoeden voor getallen in de invoerfile.

De compressietechnieken **GIF** en **JPEG** dreigen steeds duurder te worden. We gaan het nu dus maar zelf doen. Voor de tweede programmeeropgave moet een programma worden geschreven dat een file kan coderen en decoderen.
Aan de gebruiker wordt gevraagd of het om coderen of decoderen gaat en hoe de originele (bestaande) file en de "doelfile" heten. De veranderde invoerfile komt in deze doelfile terecht; de invoerfile zelf blijft onveranderd.
Stel eenvoudige vragen om deze gegevens van de gebruiker te weten te komen.
Het programma leest dan eenmalig de opgegeven invoerfile, en schrijft de uitvoer symbool voor symbool weg naar de uitvoerfile. Elk symbool uit de invoerfile mag en moet precies één maal (met `invoer.get (...)`) gelezen worden.

Het coderen geschiedt *regel voor regel*, en gaat als volgt. Iedere directe opeenvolging van *k* (groter dan of gelijk aan 2) dezelfde karakters binnen een regel wordt vervangen door dat karakter onmiddellijk gevolgd door het getal *k*. Een enkel karakter blijft gewoon zichzelf, evenals regelovergangen. Zo wordt de te coderen regel `Eet meer zeeegels` gecodeerd als `Eet me2r 10ze3gels` (in de originele zin staan blijkbaar tien spaties tussen meer en zeeegels).
Om later te kunnen decoderen hadden we moeten aannemen dat er geen cijfers in de regel staan, immers wat zou anders de codering `e434` betekenen — vier e's en vier 3-en of 434 e's? Om dat probleem op te lossen worden gecodeerde cijfers voorafgegaan door een \ (backslash). De backslash zelf wordt met twee backslashes gecodeerd. Zo moet `ABC11123ddd\efG\\1` gecodeerd worden als `ABC\13\2\3d3\\efG\\3\1`. Deze codering heet officieel **run-length encoding**.

Na het coderen wordt op het scherm afgedrukt hoeveel karakters de originele file en de gecodeeerde file bevatten, en wat de "compressie-ratio" is: de verhouding tussen deze twee getallen, als geheeltallig percentage. Er moet op de gebruikelijk manier worden afgerond; het percentage kan groter dan 100 zijn. Bij het decoderen hoeft dit niet.

Voor elk geheel getal > 0 uit de invoerfile moet worden gekeken of het **Collatz-vermoeden** waar is voor dat getal; zie ook sheet 17 van het **derde college**. Op het scherm wordt afgedrukt wat het aantal iteraties is bij 1 te komen, of het nummer van de iteratie waarvan het resultaat boven `INT_MAX` (gebruik `include <climits>`) uitkomt. Als dit laatste gebeurt, wordt dit erbij vermeld.
Elke directe opeenvolging van cijfers in de invoerfile wordt als een geheel getal opgevat. Neem aan dat ze alle kleiner dan of gelijk aan `INT_MAX` zijn. Zo bevat `123abcd-"qqq 5"+++uvw-77.88ddd//vb5656` de gehele getallen 123, 5, 77, 88 en 5656. Bij het decoderen hoeft dit niet te worden gecontroleerd.

Ter verdere inspiratie, zie het **vijfde werkcollege**, en een tweetal voorbeelden:

- Eenvoudige testinvoer: **simpel2023.txt** met bijbehorende uitvoerfile **simpel2023uit.txt**. Grootte invoerfile 704 karakters, uitvoerfile 499 karakters, compressie-ratio 71%; 20 regels.
- Moeilijke testinvoer: **moeilijk2023.txt** met bijbehorende uitvoerfile **moeilijk2023uit.txt**. Grootte invoerfile 348 karakters, uitvoerfile 373 karakters, compressie-ratio 107%; 15 regels. En 6171 vergt 261 iteraties, en 270271 gaat bij de 121ste door `INT_MAX`.

Let op: deze files kopiëren door met rechter muisknop op de links te klikken, anders (met markeer-copy-paste) gaan spaties/tabs wellicht fout!

Opmerkingen

- We nemen aan dat de gebruiker zo vriendelijk is verder geen fouten te maken bij het invoeren van gegevens. Als een getal gevraagd wordt, geeft hij/zij een getal.
- Gebruik de regelstructuur: elke regelovergang in een bestand bestaat uit een LineFeed (`\n`) (in UNIX) of een CarriageReturn gevolgd door een LineFeed (`\r\n`) (in Windows). Normaal gesproken gaat dit "vanzelf" goed. We nemen aan dat er voor het EndOfFile-symbool (wat dat ook moge zijn) een regelovergang staat.
- Alleen voor de namen van de files mag een array (of string) gebruikt worden; voor het lezen en verwerken van de tekst is slechts het huidige karakter en enige kennis over de voorgaande karakters nodig — zie boven. Alleen de headerfiles `iostream` en `fstream` mogen gebruikt worden (en `string` voor de filenamen; denk in dat geval wellicht aan het gebruik van `c_str` en `climits` voor `INT_MAX`). Uit een file mag alleen met `invoer.get (...)` gelezen worden, vergelijk Hoofdstuk 3.7 uit het dictaat, **gedeelte "aantekeningen bij de hoorcolleges"**. Binnen de hoofdloop van het programma staat bij voorkeur maar één keer een `get`-opdracht, vergelijk het voorbeeldprogramma uit dit hoofdstuk (daar staat twee keer `get`, één maal vóór de loop, uiteraard). Karakters mogen niet worden teruggezet in de oorspronkelijke file.
  Schrijf zelf functies die testen of een karakter een cijfer is, etcetera. Er mogen geen andere functies dan die uit `fstream` gebruikt worden, en wellicht `c_str`.
- Denk aan het infoblokje dat aan begin op het scherm verschijnt. Gebruik enkele geschikte functies, bijvoorbeeld voor infoblokje, inlezen gegevens van de gebruiker, aantal cijfers van een getal, Collatz-controle, coderen en decoderen van een file; zie de tips bij het **vijfde werkcollege**. Globale variabelen zijn streng verboden. Ruwe indicatie voor de lengte van het C++-programma: circa 250 regels.

Uiterste inleverdatum: **maandag 16 oktober 2023, 18:00 uur**.

Manier van inleveren (één exemplaar per koppel, dat — ter herinnering — uit maximaal twee personen bestaat) is als volgt:

1. Digitaal de C++-code inleveren via Brightspace > Course Tools > Assignments. Stuur geen executable's, LaTeX-files of PDF-files, lever alleen één C++-file digitaal in!
2. En *ook* een papieren versie van het verslag (inclusief de C++-code) deponeren in de speciaal daarvoor bestemde doos "Programmeermethoden" bij kamer 159 van het Snellius-gebouw.
   Overal duidelijk datum en namen van de (maximaal twee) makers vermelden, in het bijzonder als commentaar in de eerste regels van de C++-code. Lees bij het **zesde werkcollege** hoe het verslag eruit moet zien en wat er in moet staan.

Te gebruiken compiler: als hij maar C++ vertaalt; het programma moet in principe zowel op een Linux-machine (met g++) als onder Windows met WSL draaien. Test dus in principe op beide systemen! Normering: verslag 1; layout 1; commentaar 1; overzichtelijkheid/modulariteit 2; werking 5. Eventuele aanvullingen en verbeteringen: lees deze WWW-bladzijde: `www.liacs.leidenuniv.nl/~kosterswa/pm/op2pm.php`.

www.liacs.leidenuniv.nl/~kosterswa/pm/op2pm.php

- werk aan de tweede programmeeropgave — de deadline is op maandag 16 oktober 2023, 18:00 uur
- lees Savitch Hoofdstuk 3 en 4, en 12.1/2
- lees dictaat Hoofdstuk 3.6, 3.7 en 4.1
- maak opgaven 11/17 uit het opgavendictaat
- doe nu het `vierde werkcollege`: het functie-practicum
- `www.liacs.leidenuniv.nl/~kosterswa/pm/`