Programmeermethoden

Files & functies

Walter Kosters

week 4: 26–30 september 2011

http://www.liacs.nl/home/kosters/pm/

En dan nu eerst: 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 in een file namen, getallen en email-adressen vervangt, te vergelijken met dat wat een *compiler* of *interpreter* doet.

```
#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 ( );
```

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, gevolgd door het "einde-file (EOF) symbool". Op dat laatste kun je testen met de methode `eof ( )`.

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

```
kar = invoer.get ( );    // eerst get-ten!!!
while ( ! invoer.eof ( ) ) {
  uitvoer.put (kar);
  kar = invoer.get ( ); // en hier de 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 is dit het UNIX-commando `cp`.)

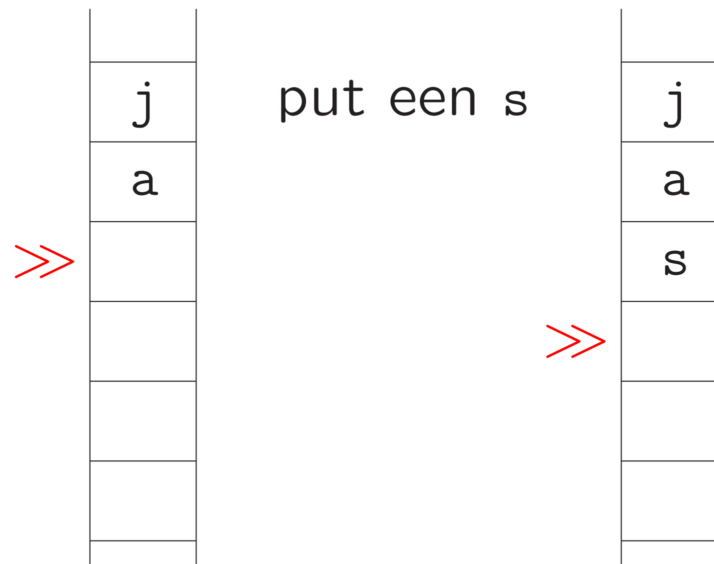
Wijzig stap voor stap zo'n kopieerprogramma:

```
kar = invoer.get ( );
while ( ! invoer.eof ( ) ) {
  if ( kar != '\n' )     // wijzig dit voor de
    uitvoer.put (kar);  // 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 videobanden. Voorlopig kun je alleen maar 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 ( )); // !!!
if ( invoer.fail ( ) ) {
  cout << filenaam << " niet te openen" << endl;
  return 1;  // of exit (1);
}//if
```

PS En files doorgeven als parameter:

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

Voor de tweede programmeeropgave moet je dus een C++-programma schrijven dat in een file iets doet:

```
abC␣DeF
ab-123.4500def
Ab42@Leuk456hoor.nl%
```

moet worden (waarbij ␣ een spatie voorstelt):

```
abC␣ddd
ab-321.0054def
aa24@xxxx654xxxxxxx%
```

www.liacs.nl/home/kosters/pm/op2pm.php

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

Elk C++-programma bestaat uit een verzameling functies. Executie begint bij de functie `main`.

Sommige functies rekenen iets uit (zoals `int`-functies: geef het kwadraat van x), andere verrichten een taak (een `void`-functies: afwassen).

Functies hebben allerlei (soorten) parameters.

Functies mogen alleen functies aanroepen die eerder gedefinieerd zijn.

Een eenvoudige `void`-functie:

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

En een eenvoudige `int`-functie:

```
int inhoud (int l, int b, int h) {
  return l * b * h;
}//inhoud
```

Met aanroepen:

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

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 l bij b bij h blok
double inhoud (double l, double b, double h) {
  double temp;
  temp = l * b * h;
  return temp;
}//inhoud
```

Hier zijn `l`, `b`, `h` en `temp` locale variabelen, waarbij `l`, `b` en `h` (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 `l`, `b` en `h` wel call by value parameters. Bij een aanroep als `t = inhoud (b,5,x);` zijn `b`, `5` en `x` de actuele parameters (of variabelen).

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

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

Dus 1963 niet, 2011 niet, 2012 wel, 2100 niet, en 2000 wel . . .

De grootste gemeenschappelijke/gemene deler (ggd) van twee positieve gehele getallen ($\geq$ 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 doen?

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.

```
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.

In C, dat alleen call by value heeft, moet je `wissel` als volgt schrijven (zie later):

```
void wissel (int *a, int *b) {
  int hulp = *a;  // *a is de int waar a naar wijst
  *a = *b;
  *b = hulp;
}//wissel
```

Voorbeeldaanroep, waarbij `&a` het adres van `a` betekent:
`a = 8; k = 2; wissel (&a,&k);`

NB De functie mag weer `wissel` heten, omdat de types van de parameters anders zijn; dit fenomeen heet overloading.

Merk op dat `a = b; b = a;` niet werkt! Blijkbaar heb je een hulpvariabele nodig.

Of toch niet:

```
void wisseltruc (int & a, int & b) {
  a = a + b;  // a = a_oud + b_oud
  b = a - b;  // b = a_oud
  a = a - b;  // a = b_oud
}//wisseltruc
```

De aanroep `wisseltruc (x,x)` maakt helaas `x` gelijk aan `0`. En deze getructe functie mag overigens geen `wissel` heten.

```
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.

```
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.

- werk aan de tweede programmeeropgave — de deadline is op vrijdag 14 oktober 2011
- Den Haag: deadline maandag 24 oktober 2011
  en geen (werk)college op maandag 3 oktober 2011!
- 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
- `http://www.liacs.nl/home/kosters/pm/`