Dit document bevat informatie over *make* bij het eerstejaars college *Programmeermethoden*, Universiteit Leiden, najaar 2010, zie

`www.liacs.nl/home/kosters/pm/`

Met dank aan allen die aan deze tekst hebben bijgedragen.
Walter A. Kosters, Leiden, 23 augustus 2010.

Meerdere files compileren

Tot nu toe hebben we vooral/alleen gekeken naar korte programma's die allemaal in één bestand ingetypt werden en als zodanig ook werden gecompileerd. Wanneer je met grotere opdrachten bezig gaat zul je zien dat het handig is om een applicatie uit meerdere bestanden op te bouwen. In dit hoofdstuk geven we een korte introductie hoe je dit het best kunt aanpakken.

1 Waarom meerdere files?

Voordat we uitleggen hoe je een applicatie kunt verdelen over meerdere files, is het belangrijk om eerst goed te weten waarom je dat zou doen. Een aantal belangrijke redenen is:

Scheiding van interface en implementatie
Deze belangrijke overweging zegt eigenlijk niets anders dan: "Je hoeft niet te weten wat erin zit, zolang je maar weet hoe je het moet gebruiken".
Stel bijvoorbeeld dat je een programma hebt geschreven dat het spel Boter, Kaas en Eieren (BKE) speelt. Je hebt een heel efficiënte BKE-computerspeler geïmplementeerd, met daarin een klasse, zeg, `BKE`, met duidelijke member-functies zoals `vind_beste_zet ( )`, `doe_zet ( )`, `print_bord ( )`, `onderzoek_alle_zetten ( )`, `undo_zet ( )`, etc. Al deze member-functies bestaan zelf misschien wel uit heel veel regels ingewikkelde C++ programma-code en roepen allerhande functies aan uit bibliotheken, etcetera.
Stel nu ook dat iemand (jijzelf of iemand anders) gebruik wil maken van jouw mooi ontworpen klasse `BKE`. Dan is deze persoon helemaal niet geïnteresseerd in de precieze implementatie van al die member-functies. Het enige wat belangrijk is, is dat bekend is welke member-functies er zijn, en hoe hun declaratie eruitziet. Het is daarom een goede gewoonte om de definitie van je klasse in een zogenaamde `include` file te zetten (deze worden ook wel *header* files genoemd; gewoonlijk hebben ze `.h` of `.hh` als extensie) en de werkelijke implementatie in een normale `.cc` of `.cpp` file (de *source*-file). Men hoeft dan alleen maar naar de `include` file te kijken om te zien wat jij hebt geschreven en hoe het moet worden gebruikt.

Overzichtelijkheid
Maar zelfs als je de scheiding tussen implementatie en header file buiten beschouwing laat is het overzichtelijker om verschillende onderdelen van een groot programma onder te brengen in verschillende bestanden. Als je bijvoorbeeld een klasse `document` hebt die onder andere bestaat uit objecten van klassen `tekst` en `tekening`, dan verdient het aanbeveling om alle programma code die te maken heeft met de tekst in een apart bestand onder te brengen (of eigenlijk 2 bestanden met bijvoorbeeld namen `tekst.cc` voor de implementatie en `tekst.h`

voor de include file) en hetzelfde te doen voor de `tekening` code. Je kunt dan bijvoorbeeld snel de bij een bepaald onderdeel horende functies vinden.

Modulariteit
Dit is een heel belangrijk aspect van programmeren dat aangeeft hoe gemakkelijk het is om *modules* (onderdelen) van je programma te wijzigen of volledig te vervangen. Als je een mooi modulair opgebouwd programma hebt kun je eenvoudig een module vervangen door een andere module, zonder dat dat gevolgen heeft voor de overige onderdelen. Een voorwaarde is dan natuurlijk wel dat de nieuwe functies op dezelfde wijze kunnen worden gebruikt als de oude (dat wil zeggen dezelfde interface implementeren, maar misschien efficiënter of beter).
Het voordeel is dan gelijk dat als twee verschillende programma's gebruik willen maken van de `BKE` klasse, ze allebei alleen maar `#include "bke.h"` hoeven toe te voegen aan hun code om van de definitie gebruik te maken. Het enige alternatief zou zijn dat je in *beide* bestanden de definitie van klasse `BKE` zou moeten opnemen. Stel nu dat 100 programma's gebruik maken van deze klasse en stel nu ook dat je op een gegeven moment besluit om de klasse aan te passen (bijvoorbeeld uit te breiden) — je zou dan alle 100 bestanden moeten wijzigen en dat is niet de bedoeling. Ook kunnen anderen gemakkelijk jouw code gebruiken door slechts één bestand te includen.

Compilatie-snelheid
Een laatste voordeel van het verdelen van een programma over verschillende bestanden is dat dit de compilatie-snelheid ten goede komt. Stel dat je applicatie uit 10 bestanden bestaat waarvan er 9 foutloos zijn en 1 een bug bevat. Die 9 goede bestanden kunnen dan eenmalig gecompileerd worden en hoeven daarna nooit meer opnieuw. Die ene file met de fout kan vervolgens gewijzigd worden (waarschijnlijk zelfs meerdere keren) en iedere keer dat er opnieuw gecompileerd wordt hoef je alleen die ene file te compileren en dat gaat veel sneller dan wanneer alles steeds van begin af aan gecompileerd moet worden.

2 Header files en implementaties

We hebben hierboven gepleit voor het scheiden van interface en implementatie, maar hoe doen we dat dan precies? Een simpele vuistregel is dat je in je header alleen maar *declaraties* opneemt, en geen *definities* en geen *implementaties*. Je kunt er bijvoorbeeld wel goed een definitie van een klasse opnemen. De volgende code zou goed in een header file voor een BKE bord passen:

```
class bke_bord {
  public:
    void print ( );
    void maak_leeg ( );  // misschien beter als constructor?
  public:
    int bord[3][3];
};  // bke_bord
```

Voorbeeldprogramma 1: Boter, Kaas en Eieren, het bord

We hebben hier een klasse `bke_bord` gedefinieerd die slechts 2 member-functies heeft (het voorbeeld is nogal simplistisch), namelijk `print` (om het bord af te drukken) en de welbekende

`maak_leeg ( )` om het bord “leeg” te maken (dat wil zeggen elk veld te vullen met 0). De klasse heeft eveneens 1 member-variabele, namelijk het 3×3 bord. We hebben echter nog niets gezegd over hoe de `print` functie geïmplementeerd moet worden. Dat bewaren we voor de source-file. Die ziet er bijvoorbeeld als volgt uit:

```
#include <iostream>
#include "bke_bord.h"   // lees de header file

void bke_bord::print ( ) {
  int i, j;
  for ( i = 0; i < 3; i++ ) {
    for ( j = 0; j < 3; j++ )
      cout << bord [i][j] << " ";
    cout << endl;
  }  // for
}  // print

void bke_bord::maak_leeg ( ) {
  int i, j;
  for ( i = 0; i < 3; i++ )
    for ( j = 0; j < 3; j++ )
      bord[i][j] = 0;
}  // maak_leeg
```

In deze laatste file zouden we ook onze `main` functie op kunnen nemen (het mag ook in een andere file), om er een werkend programma van te maken. Bijvoorbeeld:

```
int main ( ) {
  bke_bord b;
  b.maak_leeg ( );
  b.print ( );
  return 0;
}  // main
```

Let er op dat in de implementatie file een `#include` plaatsvindt van de header file. Vanaf dat moment kunnen we de definities uit de header file gewoon gebruiken in de implementatie.
Probeer zelf ook eens wat header files te schrijven en die te gebruiken in je implementaties. Begin simpel en probeer langzaam ingewikkelder headers te schrijven. (Wat moet je bijvoorbeeld doen als je in een header file voor een klasse, de definitie van een andere klasse nodig hebt?)

3 Meerdere header en meerdere source files

We weten nu hoe we header en implementatie van elkaar scheiden. De volgende stap betreft het samenstellen van één executeerbaar programma uit meerdere header- en source-files. Met andere woorden, we willen (zoals in Hoofdstuk 1 al werd gesuggereerd) meerdere source bestanden apart compileren en deze gecompileerde programma’s uiteindelijk aan elkaar plakken voor het uiteindelijke programma. Dit aan elkaar plakken wordt *linken* genoemd.

Het zal duidelijk zijn dat we slechts één `main ( )` mogen gebruiken in ons programma. De overige files bevatten alleen de implementatie van specifieke functies en klassen en dergelijke. Deze files kunnen ieder wel gecompileerd worden, maar natuurlijk niet naar een executable. In plaats daarvan worden ze "zover gecompileerd als mogelijk is en zodanig dat ze later gemakkelijk aan elkaar gelinked kunnen worden". De precieze terminologie is: zulke files worden gecompileerd naar *object* files. De extensie van object files is `.o` met g++ en `.obj` onder Visual C++.
Laten we een voorbeeld bekijken. Stel we hebben een klasse `bke_bord` en een hieraan gerelateerde klasse `array_van_borden` (die alleen maar een array van 10 borden bevat). Deze klasse kan handig zijn wanneer we een spel "simultaan-BKE" willen implementeren. De bijbehorende klassen kennen definities die in header files staan. De header file `bke_bord.h` is bijvoorbeeld:

```
class bke_bord {
  public:
    void print ( );     // druk het bord af
    void maak_leeg ( ); // maak het bord leeg
  public:
    int bord[3][3];
};  // bke_bord
```

Met andere woorden: dit is precies de `bke_bord` klasse uit Hoofdstuk 2. De headerfile `array_van_borden.h` wordt gegeven door:

```
#include "bke_bord.h" // nodig voor declaratie van bke_bord

class array_van_borden {
  public:
    void print_bord (int t);       // print het t-de bord
    void maak_bord_leeg (int t);   // maak het t-de bord leeg
    bke_bord bord_verzameling[10]; // de daadwerkelijke borden
};  // array_van_borden
```

De implementatie van deze files bestaat uit de twee bijbehorende source bestanden. Het bestand `bke_bord.cc` (of `.cpp`) hebben we hierboven eerder gezien en blijft onveranderd. Het source bestand `array_van_borden.cc` is niet heel anders en ziet er bijvoorbeeld zo uit:

```
#include "array_van_borden.h"

void array_van_borden::print_bord (int t) {
  bord_verzameling[t].print ( );
}  // print_bord

void array_van_borden::maak_bord_leeg (int t) {
  bord_verzameling[t].maak_leeg ( );
}  // maak_bord_leeg
```

Let erop dat we alleen de file `array_van_borden.h` "includen" en niet `bke_bord.h`, terwijl we wel gebruik maken van de functies van deze klasse. De reden waarom we deze header file

niet hoeven te includen is dat deze `include` al gebeurt in `array_van_borden.h`. (Sommige mensen includen overigens alleen in `.cc`-files.)
Tenslotte hebben we nog een derde source-file, en dat is het bestand dat `main ( )` bevat. Let op: het programma hieronder doet eigenlijk nog niets — het declareert een variabele van klasse `array_van_borden` en verder niets. Laten we zeggen dat het bestand met `main ( )` de naam `prog.cc` heeft en er als volgt uitziet:

```
#include "array_van_borden.h"   // voor de definitie van array_van_borden

int main ( ) {
  array_van_borden mijn_borden; // declareer de borden
  ...
}  // main
```

4 Linking

Nu hebben we vijf bestanden, `prog.cc`, `bke_bord.cc`, `bke_bord.h`, `array_van_borden.cc` en `array_van_borden.h`, die tezamen het programma vormen. Hoe combineren we ze nu precies? Vooraf moeten we één ding duidelijk maken: header files hoeven niet apart gecompileerd te worden. De reden is dat ze ge-`include` worden in de source bestanden. Dat betekent dat ze in de allereerste compilerstap letterlijk in de programma-tekst van de source bestanden worden gekopieerd (en daarna *met* de implementatie worden gecompileerd). Met andere woorden, alleen de source bestanden hoeven te worden gecompileerd. Op de verschillende operating systemen werkt het linken met wisselend gemak.

4.1 Visual C++

Bij het programmeren op een Windows-systeem met behulp van Visual Studio is het werken met meerdere files erg eenvoudig. Bij het toevoegen van een header- of sourcebestand kiest men voor de optie: project⇒add to project. Het bestand zal, als het ge-include wordt in het hoofdbestand of één van de al ge-include bestanden automatisch worden meegenomen bij het compileerproces.
Denk er wel aan dat je voortaan de project file opent (`.dsw`) om verder te programmeren aan je project!!

4.2 Unix en de Makefile

Bij het werken op de Unix systemen is het noodzakelijk de verschillende compileerstappen via de commandprompt uit te voeren. De manier waarop we iets compileren naar een object (`.o`) file is met behulp van de compiler optie `-c`. Bijvoorbeeld als volgt:

```
g++ -Wall -c prog.cc
```

Dit vertelt de compiler dat hij niet hoeft te proberen er een executeerbaar programma van te maken. Het enige dat nodig is is een `.o` file die later makkelijk gelinked kan worden om het eindprogramma te vormen. Laten we dus eerst maar alle `.cc` bestanden compileren naar de corresponderende `.o` bestanden.

```
g++ -Wall -c prog.cc
g++ -Wall -c bke_bord.cc
g++ -Wall -c array_van_borden.cc
```

We hebben nu 3 `.o` bestanden gemaakt. Hoe worden deze gecompileerde bestanden vervolgens gelinked tot een werkend programma? Hiervoor kunnen we gewoon weer `g++` gebruiken (maar nu zonder `-c` optie). Als we onze executable `doe_sim_bke` willen noemen, dan stellen we deze als volgt samen uit de drie object bestanden (de `-o` geeft aan dat de "output" `doe_sim_bke` moet heten):

```
g++ -Wall prog.o bke_bord.o array_van_borden.o -o doe_sim_bke
```

In principe hebben we nu veruit de belangrijkste aspecten behandeld van het samenstellen van een programma uit meerdere `.h` en `.cc` files. We zullen nu gaan kijken hoe we zoveel mogelijk van dit proces kunnen automatiseren, zodat we niet steeds alle commando's (zoals "`g++ ...`") opnieuw in hoeven te typen.
Stel dat je applicatie bestaat uit 50 files (dit is geen uitzondering in echte projecten), dan wordt deze handmatige compilatie snel vervelend. Een ander nadeel daarvan is dat het soms lastig is om te bepalen welke files je moet hercompileren wanneer je een van de bestanden wijzigt. Zelfs in het extreem eenvoudige voorbeeld uit de vorige paragraaf moet je even goed kijken voor dat duidelijk is. Een wijziging in `bke_bord.cc` heeft enkel gevolgen voor dit bestand zelf en zal alleen hoeven leiden tot hercompilatie van dit bestand, maar hoe zit het wanneer je een wijziging aanbrengt in `bke_bord.h` (bijvoorbeeld wanneer de naam van de methode "`print ( )`" wordt veranderd in "`druk_af ( )`")? Het zal duidelijk zijn dat `bke_bord.cc` dan opnieuw gecompileerd (en zelfs zelf gewijzigd) moet worden, maar hoe zit het met de andere bestanden? Je moet goed kijken om te zien dat de methode `print ( )` ook wordt aangeroepen in de file `array_van_borden ( )` en dat deze dus ook gecompileerd moet worden.
Als er meer bestanden zijn en de functies en klassen ingewikkelder worden, is dit alleen nog maar lastiger. Het zou mooi zijn als we konden aangeven welke `.o` files van welke bestanden afhangen.
Onder UNIX wordt gebruik gemaakt van de/een *Makefile* om beide problemen op te lossen. Een Makefile geeft aan wat van wat afhangt en hoe je alles moet compileren. In Visual C++ wordt zo'n file automatisch gemaakt — deze kan je overigens wel bewerken! Een geavanceerde Makefile kan erg gecompliceerd zijn. In dit vak behandelen we alleen de meest elementaire aspecten ervan.
Een voorbeeld van een eenvoudige Makefile vind je hieronder:

```
# Alles wat begint met '#' is commentaar (analoog aan '//' in C++).

# De volgende regel geeft aan wat er allemaal 'gemaakt' moet worden.
# In dit geval is dat slechts 1 applicatie, maar het mogen ook
# meerdere executables zijn - deze scheidt je dan met een spatie. De
# 'all:' regel geeft de ultieme 'targets' van deze Makefile

all: doe_sim_bke

# Om het programma 'doe_sim_bke' te maken hebben we nodig:
# prog.o, bke_bord.o en array_van_borden.o. Dat geven we aan met de
```

```
# 'target'-naam gevolgd door een ':', en de namen van de bestanden
# die nodig zijn om deze file te maken. De regel eronder geeft aan
# hoe, als we die bestanden eenmaal hebben, de 'target' van deze
# regel gebouwd kan worden (gebruik 'tabs' om in te springen!).

doe_sim_bke: prog.o bke_bord.o array_van_borden.o
     g++ -Wall prog.o bke_bord.o array_van_borden.o -o doe_sim_bke
# de regel hierboven begint dus met een tab!!!

# prog.o hangt alleen af van prog.cc en array_van_borden.h

prog.o: prog.cc array_van_borden.h
     g++ -Wall -c prog.cc

# bke_bord.o hangt alleen af van bke_bord.cc en bke_bord.h

bke_bord.o: bke_bord.cc bke_bord.h
     g++ -Wall -c bke_bord.cc

# array_van_borden.o hangt af van:
#   1. array_van_borden.cc
#   2. array_van_borden.h, EN
#   3. bke_bord.h (!)

array_van_borden.o: array_van_borden.cc array_van_borden.h bke_bord.h
     g++ -Wall -c array_van_borden.cc
```

Bovenstaande Makefile geeft als het ware het "recept" voor het bouwen van de applicatie `doe_sim_bke`. Als je nu `make` intypt op de command line, dan wordt de Makefile gelezen en het recept gevolgd om de applicatie te bouwen. De `make` tool gaat bijvoorbeeld eerst `prog.cc`, `array_van_borden.cc` en `bke_bord.cc` compileren (want volgens het recept zijn die nodig voor `doe_sim_bke`). De `make` tool begrijpt de afhankelijkheden die je in het recept hebt aangegeven. Als je bijvoorbeeld een wijziging aanbrengt in `bke_bord.h`, dan zorgt hij er voor dat automatisch de regels voor `bke_bord.o` en `array_van_borden.o` worden uitgevoerd. En omdat deze `.o` files wijzigen en het recept aangeeft dat de applicatie `doe_sim_bke` daarvan afhankelijk is, dan zal vervolgens deze regel ook uitgevoerd worden. Het resultaat is dat je een gestructureerde hercompilatie krijgt, waarbij de afhankelijkheden automatisch in het oog worden gehouden.
Een laatste opmerking over Makefiles. Het is vaak handig om in een directory met schone lei te beginnen. Dat wil zeggen dat je dan af wil van allerlei executables en object files die daarin misschien al aanwezig zijn. Een handige manier om dat te doen in Makefiles is het opnemen van een `clean` regel (onderaan) in de Makefile. De clean regel ziet er dan bijvoorbeeld uit zoals hieronder:

```
clean:
     rm -f *.o doe_sim_bke
```

Wat hier staat is dat er een target `clean` is die nergens van afhankelijk is. Wat er moet gebeuren om deze target te "halen" is het verwijderen ("removen") van alle files die eindigen op `.o` en ook

de file `doe_sim_bke`. Je kunt deze target expliciet laten maken door het volgende commando:

```
make clean
```

Hiermee zijn we aan het eind gekomen van dit hoofdstuk. Het verdient ten zeerste aanbeveling om eens met de stof uit dit hoofdstuk te oefenen. Schrijf zelf een paar Makefiles en kijk eens naar manieren om je programma op een nette wijze onder te brengen in meerdere bestanden.