C++, Övning 1

Jonas Sjöbergh, jsh@nada.kth.se

hur man kompilerar och kör
make
preprocessor
minnesallokering, pekare
grundläggande C++, funktioner m.m.
ett exempel

Ett enkelt program i C++, hello.cpp

```
#include <iostream>
int main() {
  std::cout << "Hello World\n";
  return 0;
}
```

Då program körs anropas main (endast en main per program får länkas in).

```
> g++ -o hello hello.cpp
> hello
Hello World
>
```

`-o` anger vad man vill att det körbara programmet ska heta. Annars blir namnet `a.out`

Det finns många flaggor till g++, här är några

-g debug-information, bra när man letar efter fel
-Wall g++ varnar för saker som är misstänkta
-ansi kräver att man följer standarden
-pedantic klagar mycket (bra!)
-c ingen länkning
-O optimera (programmet blir snabbare)
-v kolla vilken version av kompilatorn man kör nu

Större program delas lämpligen upp i flera filer. Normalt lägger man **deklarationer** i header-filer (oftast ".h") och **definitioner** i källkodsfiler (oftast ".cpp").

```
> g++ -c studenter.cpp
> g++ -c klasser.cpp
> g++ -c main.cpp
> ls
klasser.cpp klasser.o main.cpp main.o studenter.cpp studenter.o
> g++ -o mitt_program klasser.o studenter.o main.o
> mitt_program
```

Man kan också göra

```
> g++ -o mitt_program klasser.cpp studenter.cpp main.cpp
> mitt_program
```

Det första alternativet är bättre. När man ändrar i en fil behöver man bara kompilera om den och sen länka om. Sparar tid.

Makefile håller reda på vilka filer som behöver kompileras om. Mycket användbart.

```
FLAGS= -g -Wall -ansi -pedantic
# OBS!! alla indrag måste vara en TAB

mitt_program: main.o studenter.o klasser.o
    g++ -o mitt_program main.o studenter.o klasser.o

main.o: main.cpp studenter.h klasser.h
    g++ $(FLAGS) -c main.cpp
studenter.o: studenter.cpp studenter.h klasser.h
    g++ $(FLAGS) -c studenter.cpp
klasser.o: klasser.cpp klasser.h
    g++ $(FLAGS) -c klasser.cpp
```

kör ''make'', vilket kör första regeln (mitt_program)

När man har många filer är det jobbigt att lista alla beroenden m.m. för hand. Det kan make göra åt oss.

```
FLAGS= -g -Wall -ansi -pedantic
FILES= main.o studenter.o klasser.o

mitt_program: $(FILES)
    g++ $(FLAGS) -o $@ $^

%.o:%.cpp
    g++ $(FLAGS) -c $<

depend:
    makedepend -- $(FLAGS) -- $(wildcard *.cpp)
```

Kör ''make depend'' (första gången) och sen ''make''

Preprocessorn kan lite av varje:

```
#include <iostream>
#include "studenter.h" // klistra in innehållet här

#define MAX 10000 // använd const int MAX = 10000; istället
#define SQR(X) (X)*(X) // använd inline och template istället

#ifndef STUDENTER_H
#define STUDENTER_H
... // detta kommer klistras in högst 1 gång
#endif

#if 0
... // en sorts kommentarer,
... // där man kan ha kommentarer i kommentarer
#endif
```

Funktioner

```
int foo(int x); // deklaration
int bar(int x) { return foo(x);} // definition
int foo(int x) { return x+1;}

int foobar(int x = 0) {foo(x); return bar(x);}
foobar(2); // ok
foobar(); // ok, x blir 0

double foo(double x) { ... } // överlagring av foo

inline int snabb(int x) { ... }
// inline betyder ungefär klistra in kroppen istället för att anropa
```

enum, en sorts heltalstyp

```
enum color {white, red, blue};
// white = 0, red = 1, blue = 2
void foo(color c) {
  switch(c) {
   case white: bar(); break;
   case red: bar2(); break;
   case blue: bar2(); break;
  }
}
foo(white); // ok
foo(2); // inte ok
```

typedef ger nya namn till datatyper

```
typedef char * CharPointer;
CharPointer p = "hej";
```

Pekare

En pekare är en adress i minnet.

```
int *ptr = 0;
ptr = new int; // allokera en int att peka på
// ptr innehåller adressen till en int vi allokerat

int *ptr2 = 0;
int x = 3;
ptr2 = &x; // peka på x
// ptr2 innehåller adressen till x:s plats i minnet

*ptr2 = 4; // nu är x == 4, * betyder följ pekaren
```

En array använder pekare implicit

```
int foo[4]; // 4 integer
int bar = {1, 27, 2}; // tre integer, med dessa värden
int *ptr = new int[12]; // en array av 12 integer, allokerad

bar[1]; // 27, första position har index 0

bar[i] är samma sak som *(bar + i) så 1[bar] kan man också skriva...
```

Automatiskt och dynamiskt minne

Dynamiskt minne:
Större än stacken, nödvändigt för stora objekt
Storlek (t.ex. på array) kan bestämmas vid runtime
Objekt kan leva utanför det scope de skapades i
Måste frigöras när man är klar med det

Automatiskt minne:
Städas upp automatiskt, när scope tar slut
Snabb allokering
Mindre risk för slarvfel, används så ofta det går

```
int *p = new int[100];
...
delete p; // borde vara delete [] p

int *p = new int[100];
int *p2 = new int[200];
p = p2; // minnesläcka, vi kan inte längre frigöra våra 100 integer
delete [] p;
int x = p2[10]; // bug! vi har redan gjort delete på minnet

int *foo() {
  int x;
  ...
  return &x; // x lokal!
}
int * p, p2; // p är int*,  p2 är int
int *p, *p2; // båda är int*
```

Kommentarer

Kommentera er kod. Kommentarer är till dels för andra dels för ens egen skull. Pröva att läsa några andra studenters kod om ni tycker det verkar onödigt.

All er kod i kursen förväntas vara kommenterad.

Kommentarer i header-filer bör förklara formen på indata och utdata eller vad en klass är tänkt att göra.

Kommentarer i källkodsfiler bör förklara implementationsdetaljer som ej framgår av programkoden.

```
#include "student.h"
#include "lecture.h"
int main() {
  Student s1("Anna"); Student s2("Per");
  s1.learn("C++ is related to C.");
  s2.learn("g++ is a useful compiler.");

  Lecture l(10); l.addStudent(s1); l.addStudent(s2);
  l.teach("DDD is a C++-debugger.");
  l.query(0);
  l.query(1);
  return 0;
}

Anna says: C++ is related to C.
        DDD is a C++-debugger.
Per says: g++ is a useful compiler.
        DDD is a C++-debugger.
```

```
#ifndef STUDENT_H
#define STUDENT_H
#include <string>

class Student {  // deklarera klassen Student
  public:
    Student(std::string Name = "");
    ~Student();
    void learn(std::string);
    void speak();
  private:
    std::string name;
    std::string knowledge;
};
#endif
```

```
#ifndef LECTURE_H
#define LECTURE_H
#include <iostream>
#include <string>
#include "student.h"
class Lecture {   // deklarera klassen Lecture
  public:
    Lecture(int);
    ~Lecture();
    void query(int);
    void addStudent(Student &);
    void teach(std::string);
  private:
    int capacity;
    int size;
    Student *members;
};
#endif
```

```
#include <iostream>
#include <string>
#include "student.h"

Student::Student(std::string Name) : name(Name), knowledge("") {}
Student::~Student() {}

void Student::learn(std::string info) { knowledge += info;}

void Student::speak() {
  std::cout << name << " says: "
            << knowledge << std::endl;
}
```

```
#include "lecture.h"
Lecture::Lecture(int cap) : capacity(cap), size(0) {
  members = new Student[cap]; }
Lecture::~Lecture() { delete [] members;}
void Lecture::addStudent(Student &s) {
  if(0 <= size && size < capacity) {
    members[size++] = s; } }  // OBS! kopia av s
void Lecture::query(int i) {
  if(i >= 0 && i < size) {
    members[i].speak(); } }
void Lecture::teach(std::string info) {
  for(int i = 0; i < size; i++) {
    members[i].learn("\n\t"+info); } }
// eftersom vi har new i konstruktorn borde vi egentligen skriva
// operator=() och copy-konstruktor också, i verkligheten
```