Övning 1, Ögrupp 4,
Programsystemkonstruktion med C++,
2003−09−11

Ronnie Johansson, rjo@nada.kth.se

Vi kommer att titta på:

- Kompilering och länkning
- make och Makefile
- Preprocessordirektiv
- main()−funktionen
- Funktioner
- Datatyper
- Automatiska och dynamiska objekt
- Pekare och arrayer
- Pekararitmetik och vanliga slarvfel
- Referenser
- Kommentarer

Kurskatalog: `/info/cprog03/`

(student.h)

```
#ifndef _STUDENT_
#define _STUDENT_
#include <iostream>
#include <string>
class Student{
	private:
		string knowledge_;
	public:
		Student();
		~Student(){};
		void learn(string info);
		string speak();
};
#endif
```

(student.cpp)

```
#include ”student.h”
Student::Student(){
	knowledge_=""; //medlemsvariabler initieras här
}

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

string Student::speak(){
	return knowledge_;
}
```

(main.cpp)

```
#include ”student.h”
int main(){
	Student s;
	s.learn("bla bla bla");
	cout << s.speak() << endl;
	return 0;
}
```

Första exemplet

− Först deklarerar vi klassen sedan definierar vi dess funktioner.
  - Private är default.
  - Student:: anger scope.
  - ~Student() är destruktor.

− Global main−funktion.
  - Globala funktioner och variabler är tillåtna.

− Deklarationer i .h definitioner i .cpp.

− Macro (#...) körs av preprocessorn.

```
>g++ −o ex1 main.cpp student.cpp
>ex1
>bla bla bla
```

Kompilator & länkare

Filer kan kompileras var för sig till objektsfiler (.o−filer):

```
>g++ −c student.cpp
>g++ −c main.cpp
>ls
... main.o          student.o
```

och länkas (kräver main()−funktion):

```
>g++ −o ex1 main.o student.o
```

```
>g++ −g −o ex1 main.o student.o
```

(för att använda med debuggern)

Projekt

− Om vi ändrar main.cpp räcker det att kompilera den.
− Men om vi ändrar i student.cpp/h måste vi eventuellt kompilera båda (main.cpp beroende av student.h).

Makefile

```
# variabler
CC = g++
FLAGS = −g −I~/cppkurs/student
OBJ = main.o student.o

# regler
ex1: $(OBJ)
    $(CC) $(FLAGS) −o ex1 $(OBJ)
#glöm inte TAB!

main.o: main.cpp student.h
    $(CC) $(FLAGS) −c main.cpp

student.o: student.cpp student.h
    $(CC) $(FLAGS) −c student.cpp

clean:
    rm *.o ex1
```

(http://www.gnu.org/manual/make−3.79.1/html_chapter/make_toc.html)

...Makefile

```
CC = g++
FLAGS = −g −I~/cppkurs/student
OBJ = main.o student.o
SRC = main.cpp student.cpp

ex1: $(OBJ)
    $(CC) $(FLAGS) −o ex1 $(OBJ)

%.o: %.cpp
    $(CC) $(FLAGS) −c $*.cpp

depend:
    makedepend −− $(FLAGS) −− $(SRC)

clean:
    rm *.o ex1
```

Preprocessordirektiv

#include slår ihop filer innan kompilering.

#ifndef ... #define ... #endif garanterar att en fil bara inkluderas en gång.

#define MAX 10000 //kan vara praktiskt

#define SQR(A) A*A //farligt!!
SQR(x+1) −> x+1*x+1=2x+1
#define SQR(A) (A)*(A)

mainsyntax

Java

```
public class Main {
    public static void main(String args[]) {
        int no_of_args = args.length;
        for (int i = 0; i < no_of_args; i++)
            System.out.println(args[i]);
    }
}
```

> java Main abc 123
abc
123

C++

```
int main(int argn, char* argc[]){
    for(int i=0; i<argn; i++){
        cout<<argc[i]<<endl;
    }
    return 0;
}
```

> a.out abc 123
a.out
abc
123

Skillnader mellan C++ och Java

- första argumentet, argc[0], är programmet självt
- args.length finns ej, argn har längden.
- main()−funktionen ligger ej i en klass

Funktioner

En funktion...
...behöver inte definieras där den deklareras.
Men den behöver deklareras innan den används:

```
int foo(int x);
int bar(int x){
    return foo(x);
}

int foo(int x){
    return x;
}
```

...kan överlagras:

```
    int foo(int x){...}
    int foo(float x){...}
```

...kan ha argument med default−värde:

```
    foo(int x=0); // ok
    foo(int x = 0, float y, char c = 'a');
                    // ej ok
```

...kan vara inline:

```
    inline foo(int x){...}
int min( int v1, int v2 )
{
    return ( v1 < v2 ? v1 : v2);
}
```

Fördelar med en funktion min() istället för att skriva den korta koden direkt:

- Ger mer läsbar kod
- Ett ställe att ändra på
- Alla anrop till min fungerar likadant

Tyvärr går det slött att använda funktionen: argumenten måste kopieras och sparas undan, och programkörningen måste hoppa till den plats där funktionen finns i minnet.

Nyckelordet inline gör att funktionsanropet kan ersättas vid kompileringstillfället med koden som den innehåller.

```
inline int min( int v1, int v2 )
```

Mest användbart med korta kodsnuttar som används mycket ofta.

Datatyper

bool
char
int
short
long
float
double
long double
void

```
enum Sex{unknown, male, female}

typedef char* p_char
```

−Testa sizeof(T)!

System

GUI samt tråd− och processhantering finns ej i språket.
Man får använda OS−specifika APIn.

Garbage collect finns inte. Man måste själv se till att minne inte läcker.

Automatiska eller dynamiska objekt

Automatiskt allokerade objekt

Ex.vis: void foo(Student s)

```
{
    int i;
}
```

- − Städas automatiskt upp när scopet är slut (t ex när funktion returnerar).
- − Snabb allokering på stacken
- Mindre risk för slarvfel (minneshanteringen sker ju automatiskt)

Dynamiskt allokerade objekt:

Ex.vis

```
{
    int *arr = new int[N]; // allockerar minne för N stycken int
}
```

- − Free store större än stacken.
- Arrayers längd kan bestämmas i runtime
- − Objektet kan leva utanför scope där det skapas.
- − Länkad lista (implementeras ofta med pekare)

Pekare * & array []

- En pekare är en address i minnet.
  - int *ptr;
    // plats för en adress
  - int *ptr = new int;
    //allokerar en int i minnet.
  - ptr är adressen.
  - *ptr är värdet som ptr pekar på
  - Student *ptr = &stud;
    (*ptr).learn(”weird eh?”);
    ptr−>learn(”weird eh?”);

- − En array använder pekare.
  - argc är &argc[0].
  - argc[i] är *(argc+i).
    // pekararitmetik
  - int arr[4];
    // 4 oinitierade int.
  - int arr[] = {1,2,3,4};
    //initierad array.
  - int *arr = new int[N];
    //dynamisk array.
  - char *str = "hej"; //{’h’,’e’,’j’,’\0’}

Vanliga slarvfel med pekare

```
void foo(int* p){
    delete p;
}
int *p1 = new int[2];
int *p2 = new int[2];
p2 = p1;    //minnesläcka
foo(p2);    //p1 och p2 förstörda
int x = p1[0]; //oops!

int* bar() {
    int x=1;
    return &x; //oops!
}


int* p1, p2; //p2 är vanlig int
```

Referenser &

- Som i Java!
- − Alias för ett objekt.
- Som en pekare vars adress inte får ändras och objektet måste existera.

```
{
    int i = 1;
    int &j = i; // j har typen int, inte int*
    j++; // i ökas till 2
    int *k = &i;  // i,j,*k har samma värde
}

void swap( int &v1, int &v2 ) {
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

Statiska variabler

```
foo() {
    static int N = 0;
    N++;
    ...
}
```

(lecture.h)

```
#ifndef _LECTURE_
#define _ LECTURE _
#include <iostream>
#include <string>
#include "student.h"

class Lecture{
        private:
                int capacity_, size_;
                Student *members_; //pointer to array of Students
        public:
                Lecture(int cap);
                ~Lecture();
                string query(int i);
                void addStudent(Student s);
                void teach(string info);
};

#endif
```

(lecture.cpp)

```
#include "lecture.h"

Lecture::Lecture(int cap){
    capacity_= cap;
    members_ = new Student[cap]; //dynamic allocation
    size_= 0;
}

Lecture::~Lecture(){
    delete [] members_; //clean−up memory
}

void Lecture::addStudent(Student s){ //copies Student
    if(size_<capacity_){
        members_[size_++] = s;
    }
}

string Lecture::query(int i){
    if(i<capacity_){   //check range
        return members_[n].speak();
    }
    return "";
}

void Lecture::teach(string info){
    for(int i=0; i<size_; i++){
        members_[i].learn("\n\t"+info);
    }
}
```

Kommentering

Funktionskommentar förklarar vad funktionen returnerar.

```
/* list_empty
 *
 * list_empty returns true if the list is empty
 * and false otherwise.
 *
 * Def:
 *     list_empty (l)  ==  (list_size (l) == 0)
 *
 */
bool list_empty (List   *l);
```

Procedurkommentar förklarar vad proceduren gör och vad den påverkar (variabler, filer ...).

```
/* getline
 *
 * getline reads one line from stdin and stores it in
 * the given buffer. The newline is stored.
 * The given buffer must be big enough to store
 * the trailing '\0' (i.e. it must be at least one bigger than
 * nmax).
 *
 */

void
getline (
    char   buf[],  /* IN: where to store line        */
    int    nmax,   /* IN: max #chars to read         */
    int   *nread);  /* OUT: #chars read, 0 means EOF  */
```