Slide 1

2D1387, Programsystemkonstruktion med C++

Johnny Bigert, johnny@nada.kth.se

Kursens hemsida: http://www.nada.kth.se/kurser/kth/2D1387

Slide 2

Varför vill man lära sig C++?

- Objektorienterat
- Tunt lager ovanpå C, vilket gör att det är
- Snabbt
- C++-programmerare eftertraktade i näringslivet
- Använt och testat sedan 1984, många verktyg

Slide 3

Kursen i stort

- Visar hela språket
- Mycket att lära på kort tid ger *mycket* labbtid och självstudier
- Sju föreläsningar på totalt 14 timmar
- Tre omfattande labbar på totalt 26 timmar
- Tre övningstillfällen på totalt 6 timmar

Nada ger inte rena språkkurser; C++ är ett undantag eftersom det är så mycket mer än bara språkdefinitionen.

Slide 4

Examination

För att examineras krävs att man godkänt redovisar tre datorlaborationer. Man skall dessutom skriva en tentamen. Varje labb ger **tre bonuspoäng till tentan** om de redovisas i tid.

Labb 1 - Grundläggande C++ Senast redovisad på labbtillfället onsdagen 15 nov (v.46)

Labb 2 - Kalender Senast redovisad på labbtillfället onsdagen 29 nov (v.48)

Labb 3 - Äventyrsspel Senast redovisad på labbtillfället onsdagen 13 dec (v.50)

Tentamen Onsdagen 20 dec (v.51) klockan 14.00 i V01, 11 och 21.

Slide 5

Del 1 – grundläggande C++

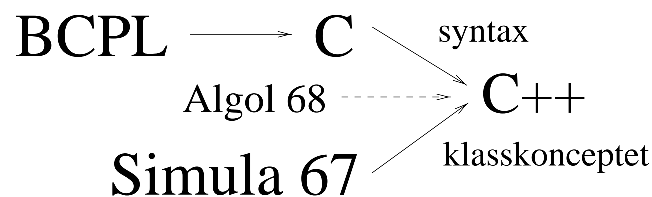
Ämnesområden denna föreläsning:

- Datatyper
- Funktioner och funktionsanrop
- Sammansatta datatyper
- Pekare, aritmetik och referenser
- Minneshantering, preprocessorn

Slide 6

Kort historik

- **Bjarne Stroustrup** skapade "*C with Classes*" 1980.
- År 1983 användes det för första gången av en extern forskargrupp.
- Namnet kommer från C och ++-operatorn och betyder att C++ är en (ut)ökning och förbättring av C.



Slide 7

Datatyper

I programspråket C++ finns följande grundläggande datatyper:

- `bool` – sanningsvärde, `true` eller `false`
- `char` – heltal, 8 bitar. Används ofta för att lagra bokstäver
- `int` – heltal, oftast 32 bitar
- `long` – heltal, större eller lika med `int`
- `float` – flyttal
- `double` – flyttal med dubbel precision
- pekare – håller adressen till en av ovanstående typer eller annan, mer komplicerad typ.

Slide 8

Exempel på användning av datatyper

```
bool b = true;       // sanningsvariabel
int i = 85;          // i tilldelas värdet 85
int j = i;           // j blir 85
char a = 'x';        // a blir 120, dvs asciivärdet för 'x'
float f = 3.1;       // flyttal
double d = 3.141593; // dubbel precision
```

Slide 9

Funktioner och funktionsanrop

- För att dela upp sitt program i logiska delar använder man **funktioner**
- Vid **anropet** till funktionen skickar man med **argument**
- Alla anrop är **värdeanrop** (*call by value*) vilket innebär att det är kopior av argumenten som används under körningen av funktionen. Undantag: referenser och vektorer.
- Tillbaka får man ett **returvärde**

Slide 10

En funktion bör ha en **deklaration**. En funktion som används har precis en **definition**.

```
double maximum(double, double);        // deklaration
int main() {
    double d = maximum(1.7, 3.2);      // anrop
    cout << "d = " << d << endl;       // mata ut d = 3.2
    return 0;
}
double maximum(double d1, double d2) { // definition
    if(d1 > d2)
        return d1;                     // returvärde
    else
        return d2;                     // returvärde
}
```

Funktioners egenskaper i C++

- Funktioner kan ta ett förvalt argument (*default argument*)
- Funktioner med samma namn och olika argument kallas **överlagringar**
- Funktioner kan deklareras `inline`, vilket är ett förslag till kompilatorn att ersätta funktionsanropet med programkod

Slide 11

```
int f(char c, int i = 7);       // ett förvalt argument
int g(int);                     // g tar int
int g(double);                  // överlagring
int g(A);                       // överlagring

int h(double d);                // deklaration
inline int h(double d)          // inlinedefinition
{ return d; }
```

Uppräkningar

Typen `enum` gör uppräkningar mer läsbara:

Slide 12

```
enum weekday {Mon, Tue, Wed, Thu, Fri, Sat, Sun, daycount};
enum wingdings {foo = 11, bar = 3, baz};
weekday today = Mon;

cout << "Today is day number " << today + 1 << endl;
cout << "There are " << daycount
     << " days of the week" << endl;
cout << "bar = " << bar << ", "
     << "baz = " << baz << endl;
```

Utdata blir

Slide 13

```
Today is day number 1
There are 7 days of the week
bar = 3, baz = 4
```

Sammansatta datatyper

Man kan skapa vektorer av en given datatyp.

Slide 14

```
int a[7];                    // alla element oinitierade
int b[]   = {1, 2, 3};       // b har 3 element
int c[2]  = {7, 8};

char r[]  = {'b', 'a', 'r', '\0'};
char s[]  = "bar";           // s har 4 element pga '\0'
char t[4] = "bar";
```

Slide 15

Indexoperatorn

Åtkomst i vektorer sker genom att använda `[]`-operatorn.
Elementen i en vektor är indexerade från noll.

```
int c[2] = {7, 8};

cout << "c[0] = " << c[0] << endl;    // c[0] är 7
cout << "c[1] = " << c[1] << endl;    // c[1] är 8
c[0] = 5;                             // c[0] blir 5
```

Slide 16

Klasser

För att kapsla in data med funktioner använder man datatypen `class` precis som i Java. **Åtkomsttypen** avgör vem som kan se in i instanser av klassen.

```
class Foo
{
   int i;              // åtkomst private är default
public:
   int j;              // alla kommer åt j
   int fnk(double d)   // publik funktion som tar double
   { return d; }       // ...och returnerar heltalsdelen
private:
   int k;              // endast medlemsfunktioner når k
};
```

`struct` — en variant av `class`

Den enda skillnaden mellan `class` och `struct` är att åtkomsttypen är private i `class` medan den är public i `struct`.

Slide 17

```
struct Bar { int i; long j; };
Bar a = {7, 4711};      // lista av initierare
Bar b;                  // medlemarna oinitierade
b.i = 7;                // explicit initiering
b.j = 4711;
```

Union

Det händer att man har behov av en typ som kan hålla olika sorters data i sig, man använder då `union`. Storleken på den sammansatta typen blir storleken av den största datatypen.

Slide 18

```
union Baz
{
  int    i;
  long   l;
  double d;
  int    a[7];
} b;              // skapa objekt b av typ union Baz
b.a[3] = 5;       // tilldela en medlem
b.i = 7;          // skriv över och tilldela annan medlem
```

Pekare

- En pekare är en adress till en plats i minnet
- Man kan peka till alla typer av data i ett program har en plats i minnet
- Pekare ger upphov till *många* fel, och dessa kan vara svåra att finna

Slide 19

```
int i = 7;    // i är 7
int *ip;      // pekare till en int

ip = &i;      // ip innehåller adressen till i
*ip = 8;      // avreferera pekaren och tilldela
              // i är nu 8
```

Ytterligare ett exempel på pekare

```
char c;
char *s = "foobar";
char *t;

c = s[3];        // c är nu 'b'
t = s;           // t pekar på 'f'
t = &s[0];       // t pekar på 'f'
t = &s[4];       // t pekar på 'a'
```

Slide 20

Slide 21

Ett exempel på när det går fel med pekare

```
int *i = int_pointer();
cout << "i = " << *i << endl;  // skriver ut ett skräptal

int *int_pointer()
{
    int i = 7;      // fel: lokal variabel är inte längre
    return &i;      // giltig då funktionen returnerat
}
```

Slide 22

Pekare, vektorer och `sizeof`

- En vektor är **ekvivalent** med en pekare
- Man kan addera en pekare med ett heltal, s.k. **pekararitmetik**
- Man kan subtrahera pekare men inte addera

```
int a[] = {0, 1, 2, 3, 4};
int *p;
int j;

p = a + 1;          // p pekar på 1
j = a[2];           // j är 2
j = p[2];           // j är 3
*(a+1) = 5;         // a är {0,5,2,3,4}
j = sizeof a[2];    // j är 4
j = sizeof(int);    // j är 4
```

Referenser

- **Referenser** är ett sätt att undvika syntaxen hos pekare
- En referens refererar alltid till ett objekt eller en variabel
- Objekt och referenser är syntaktiskt ekvivalenta

Slide 23

```
class A { public: int i; } a;

A *ap;              // ap är pekare till A
A &arf;             // fel: arf måste referera objekt
A &ar = a;          // ok: ar refererar a

ap = &a;            // ap är adressen till a
int i = ap->i;      // avreferera pekaren till a
ap = &ar;           // ap är adressen till a
i = ar.i;           // ar används istället för a
```

Typer och typdefinitioner

- Definitioner av typer kan vara svåra att förstå i C++
- Man kan använda `typedef` för att döpa om en typ

Slide 24

```
int ipart(double);      // deklarera funktion

typedef char str[7];    // str är en vektor av 7 char
typedef int (*funct)(double); // funct är funktionspekare

str x = "foo";          // x är en vektor av 7 char
funct f = ipart;        // f pekar till funktionen ipart
int i = f(3.14);        // f anropar funktionen ipart

int *api[5];            // vektor av pekare till int
int (*pai)[5];          // pekare till vektor av int
```

Slide 25

Styrstrukturer

Styrstrukturerna i C++ är desamma som i Java.

```
int i = 7, j;

if(i) {} else {}                // if-else
i = a > b ? a : b;              // villkorsoperatorn
for(i = 0; i < 17; i++) {}      // loop med for
while(i) {}                     // loop med while
do {} while(i);                 // loop med do-while

switch(i) {                     // switch-case
case 1: f(); break;
case 2: g(); break;
default: h(); break;
}
```

Slide 26

Operatorer

Operatorerna är desamma som i Java (med några undantag) och har samma prioritetsordning (*precedence*) och associativitet:

Funktionsanrop m.m.	`() [] :: -> . ->* .*`
Unära operatorer	`! ~ ++ -- + - * &`
Aritmetiska operatorer	`+ - * / %`
Jämförelseoperatorer	`< <= >= > == !=`
Bitvisa operatorer	`& ^ | ~ << >>`
Logiska operatorer	`&& ||`
Jämförelseoperatorn	`?:`
Tilldelning	`+= -= &= ~= <<=` etc.
Kommaoperatorn	`,`

Minneshantering

- Lokala objekt **allokerar** eget minne på stacken
- Objekt som skall behållas måste allokeras **dynamiskt** på heapen
- Dynamisk allokering görs med `new[]` och `delete[]`

Slide 27

```
void foo()
{
    A a;                   // a allokerad på stacken
    A *ap = new A();       // dynamiskt allokerad
    A *aa = new A[5]();    // vektor med 5 A-objekt
    delete ap;             // frigör allokerat minne
    delete aa;             // fel: destruktor för a[0]
    delete [] aa;          // ok: destruktor för 5 element
} /* vid funktionens slut frigörs a automatiskt */
```

Statiska objekt i funktioner

- Statiska objekt allokerar eget minne
- Värdet på objekten behålls och ändras inte mellan anropen

Slide 28

```
void foo()
{
    static bool initialized = false;
    if (!initialized)
    { /* do initialization */ }
    ...
}
```

Slide 29

Preprocessorn

- Körs innan programmet ses av kompilatorn
- Används för att inkludera/exkludera källkod
- Definierar konstanter och makron

```
#include <iostream>   // inkluderar filen iostream
#include "myfile.h"   // letar i lokal katalog först

#ifdef MSDOS          // inkludera endast om MSDOS-miljö
#include <conio>
#endif
```

Slide 30

Preprocessorn

Exempel på konstanter och makron:

```
#ifndef MYFILE_H         // kontrollera så att filen
#define MYFILE_H         // inte inkluderas 2 ggr

#define PI 3.14159265    // konstant
#define max(x, y) ((x) > (y) ? (x) : (y))  // makro
#if 0                    // kompileras inte
cout << "You won't see this one" << endl;
#endif

#endif                   // matchar #ifndef MYFILE_H
```