

C++-programmets beståndsdelar

- Ett C++-program är uppdelat i headerfiler (fil.h) och implementationsfiler (fil.cpp)
- Programmet måste innehålla åtminstone funktionen `int main()`
- `main()` startar programmet

C++-programmets beståndsdelar

Ett första litet C++-program

```
#include <iostream>    // behövs för utskrifter
int main()             // main är obligatorisk i C++
{
    // utskrift av "Hello world!" och radbrytning
    std::cout << "Hello world!" << std::endl;

    return 0;         // avsluta programmet med värde 0
}
```

Byggstenar

Inbyggda datatyper
Funktionsanrop,
argument och
returvärden
Styrstrukturer

Grundläggande datatyper

I C++ finns många inbyggda datatyper. De flesta av dem representerar heltal:

- `bool` – sanningsvärde, `true` eller `false`
- `char` – oftast 8 bitar (ettor och nollor). Används ofta för att lagra bokstäver
- `short` – oftast 16 bitar
- `int` – oftast 32 bitar, datorns ordstorlek
- `long int` – större eller lika med `int`

Grundläggande datatyper

Övriga datatyper representerar flyttal...

- `float` – flyttal, 16 bitar
- `double` – flyttal med dubbel precision, dvs 32 bitar
- `long double` – 64 bitars flyttal

...och minnesplatser:

- Pekare – Håller adressen till valfri datatyp, oftast 32 bitar (för att adressera datorns hela minne), även funktioner

Exempel på datatyper

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

signed int k = 17;     // heltal med tecken (default)
unsigned char c = 4;  // positivt heltal

int *q = &i;          // pekare till ett heltal
int *p = 0;           // pekare till null
```

Funktioner och funktionsanrop

- För att dela upp sitt program i logiska delar använder man **funktioner**
- All kod i C++ ligger i funktioner
- Vid **anrop** till funktionen skickar man med **argument**
- Anrop i C++ är antingen **värdeanrop** (*call by value*) där argumentens värden kopieras eller **referensanrop** (mer om det senare)
- Tillbaka får man ett returvärde

Funktioner och funktionsanrop

- 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
    std::cout << "d = " << d << std::endl; // d = 3.2
    return 0;
}

double maximum(double d1, double d2) // definition
{
    if(d1 > d2)
        return d1; // returvärde
    else
        return d2; // returvärde
}
```

Funktioners egenskaper

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

Funktioners egenskaper

```
int f(char c, int i = 7); // ett förvalt argument
f('a'); // anrop med förvalt arg = 7
f('a', 3); // anrop med annat arg = 3

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; }
double x = h(3.14); // anrop till inlinefunktion
double y = 3.14; // anrop ersatt mha inline
```

Styrstrukturer

Översikt:

```
int i = 7;

if(i) {} else {} // if-else
i = a > b ? a : b; // villkorsoperatör
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;
}
```

Styrstrukturer – for

```
for(int i = 0; i < 10; i++) // 10 varv
{
    // i är synlig här
}
// i är inte synlig här

const int size = 100;
int j;
for(j = 0; j < size; j++) // 100 varv
    std::cout << "varv " << j << std::endl;
```

Styrstrukturer – while

```
int i = 0;
while(i < 100)    // 100 varv
{
    std::cout << "varv " << i << std::endl;
    i++;
}
```

Styrstrukturer – do while

```
bool done = false;
do
{
    done = do_something();
    /* ... */
} while(!done);
```

Styrstrukturer – switch case

```
// skriv ut meny
std::cout << "1. Öppna" << std::endl;
std::cout << "2. Spara" << std::endl;
std::cout << "3. Avsluta" << std::endl;

// hämta tal från tangentbordet
int value;
std::cin >> value;

switch(value)
{
case 1:    open(); break;
case 2:    save(); break;
case 3:    quit(); break;
default:   std::cout << "ogiltig inmatning" << std::endl;
}
```

Sammansatta datatyper

Vektorer
Strukturer
Klasser
Unioner

Sammansatta datatyper

Man kan skapa vektorer av en given datatyp

```
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'}; // '\0' har värdet 0
char t[4] = "bar";           // "bar" är en sträng
char s[] = "bar";           // s har 4 element pga '\0'
```

Förklaring av strängar →

Sammansatta datatyper

- Det finns inga inbyggda **strängar** i C/C++ (C++:s standardbibliotek innehåller dock en strängklass, `std::string`)
- Strängar i C/C++ representeras av en vektor av `char`, avslutad med bokstaven `'\0'` (nolla) som har värdet noll och kallas **NULL**.
- Vill man använda en sträng läsare man vektorn tills man stöter på bokstaven `'\0'`.

Indexoperatören

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

```
int c[2] = {7, 8};
std::cout << "c[0] = " << c[0] << std::endl; // c[0] är 7
std::cout << "c[1] = " << c[1] << std::endl; // c[1] är 8
c[0] = 5; // c[0] är 5

// exempel på läsning av en sträng
char str[4] = "foo";
int i = 0;
while(str[i] != '\0') // loopa tills str är slut
{
    std::cout << str[i]; // skriv ut värdet
    i++; // öka värdet på i
}
```

class - en kort introduktion

- För att kapsla in data tillsammans med funktioner använder man datatypen `class`
- **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 fnc(double d) // funktion fnc som tar double och
    { // lägger till ett
        return d + 1;
    }
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` och `public` i `struct`.

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

Uppräkningar

Typen `enum` gör uppräknningar mer läsbara

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

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

Uppräkningar

Utdata blir

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

Pekare, minne och referenser

- Pekararitmetik
- Vektorer
- Referenser
- Ekvivalens
- Statiskt och globalt minne
- Dynamiskt minne

Pekare

- En pekare är en adress till en plats i minnet
- Alla variabler och funktioner i ett program har en adress och kan pekars på
- Pekare ger upphov till *många* fel, och dessa kan vara svåra att finna

```
int i = 7;           // i är 7
int *ip = 0;        // pekare till en int

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

Pekare

Ytterligare 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'
```

Pekare

Exempel på när det kan gå fel med pekare

```
char *s = name();
std::cout << "my name is " << s // skriver ut skräp
          << std::endl;

char *name()
{
    char *str = "alice"; // fel: lokalt minne är
    return s;           // ogiltig då funktionen returnerat
}
```

Pekare, vektorer och minne

- En vektor konverteras till en pekare vid användning
- Man kan addera en pekare med ett heltal, s.k. **pekararitmetik**
- Man kan subtrahera pekare (avstånd) men inte addera

Pekare, vektorer och minne

Exempel på hur pekare och vektorer opererar på datorns minne

```
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}
```

Pekare, vektorer och minne

- Objekt tar upp olika storlek i minnet.
- Pekararitmetik använder storleken för att bestämma avstånd mellan pekare

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

j = sizeof a[2];    // j är 4
j = sizeof(int);   // j är 4
j = sizeof a;      // j är 20

int *p = a + 1;    // stegar 1 vektorposition = 4 bytes
```

Minneshantering

- Lokala objekt **allokeras på stacken** och har kort livslängd
- Objekt med längre livslängd måste allokeras **dynamiskt** på heapen (free store)
- Dynamisk allokering görs med **new** och **delete**
- Statiska och globala objekt finns under programmets hela körtid

Minneshantering

- Minne (objekt) som allokerats med **new** ska deallokeras med **delete**
- Minne (vektorer av objekt) som allokerats med **new []** ska deallokeras med **delete []**

```
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: odefinierat beteende!
    delete [] aa;  // ok: destruktör för 5 element
} /* vid funktionens slut frigörs a automatiskt */
```

Minneshantering

- C++ har ingen automatisk minneshantering (*garbage collection*)
- Alla dynamiska objekt som inte lämnas tillbaka läcker minne
- Speciellt viktigt är det att hålla ordning på minnet i objekt som skapas och destrueras ofta som t.ex. strängar.

Undvik vanliga problem

- Undvik pekare!
 - Det är lätt att referera otillåtet minne (t.ex. genom att avreferera `NULL`)
 - Det är lätt att referera ogiltigt minne (redan frigjort med `delete`)
- Skapa istället objekt på stacken
- Använd referenser! Dessa är garanterade att alltid referera ett giltigt objekt

Byggstenar

Typdefinitioner
Styrstrukturer
Operatorer
Preprocessorn

Operatorer

Operatorerna i C++ har olika prioritetsordning (*precedence*) och associativitet:

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

Preprocessorn

- Körs innan programmet ses av kompilatorn
- Används för att inkludera/exkludera källkod
- Kan användas för att definiera konstanter och makron
- C++ har många tekniker för att undvika preprocessorn, såsom `inline`, `template` och `const`

Preprocessorn

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

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

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
std::cout << "You won't see this one" << std::endl;
#endif

#endif // matchar #ifndef MYFILE_H
```