EDAF30 – Programmering i C++

11. Lågnivå-detaljer

Sven Gestegård Robertz
Datavetenskap, LTH

2016

Innehåll

Heltalstyper

- heltal med tecken

Typ	Storlek	Talområde (minst)
signed char	8 bitar	$[-127, 127]$*
short	minst 16 bitar	$[-2^{15}+1,\ 2^{15}-1]$
int	minst 16 bitar, oftast 32	$[-2^{15}+1,\ 2^{15}-1]$
long	minst 32 bitar	$[-2^{31}+1,\ 2^{31}-1]$
long long	minst 64 bitar	$[-2^{63}+1,\ 2^{63}-1]$

*typiskt $[-128, 127]$, etc.

- **unsigned**(icke-negativa) heltal
  - samma storlek som motsvarande **signed** typ
  - **unsigned char**: $[0, 255]$, **unsigned short**: $[0,\ 2^{16}-1]$. etc.
- specialfall
  - **char** (får *representeras* som **signed char** *eller* **unsigned char**)
  - Använd **char** endast för att lagra tecken
  - Använd **signed char** eller **unsigned char** för talvärden
- Storlekar enligt standarden:
  char $\leq$ **short** $\leq$ **int** $\leq$ **long** $\leq$ **long long**

Heltalstyper

Test med sizeof

```
#include <iostream>
using namespace std;
int main () {
   cout << "sizeof(char)= \t" << sizeof(char)<<endl;
   cout << "sizeof(short)= \t" << sizeof(short) <<endl;
   cout << "sizeof(int) = \t" << sizeof(int) <<endl;
   cout << "sizeof(long)= \t" << sizeof(long)<<endl;
}
```

```
sizeof(char)=   1
sizeof(short)=  2
sizeof(int) =   4
sizeof(long)=   8
```


Heltalstyper – Test av talområdet via casting
eller: *se upp med typecasts* från **signed** till **unsigned**-typer

```
int main () {
 cout << "(char) -1 = " << (int)(char) -1 << endl;
 cout << "(unsigned char) -1 = " << (int)(unsigned char) -1 << endl;
 cout << "(short int) -1 = " << (short int) -1 << endl;
 cout << "(unsigned short int) -1 = "<<(unsigned short int)-1<<endl;
 cout << "(int) -1 = " << (int) -1 << endl;
 cout << "(unsigned int) -1 = " << (unsigned int) -1 << endl;
 cout << "(long) -1 = " << (long) -1 << endl;
 cout << "(unsigned long) -1 = " << (unsigned long) -1 << endl;

}
```

```
(char) -1 = -1
(unsigned char) -1 = 255
(short int) -1 = -1
(unsigned short int) -1 = 65535
(int) -1 = -1
(unsigned int) -1 = 4294967295
(long) -1 = -1
(unsigned long) -1 = 18446744073709551615
```


Heltalstyper
Storlekarna specificeras i <climits>

```
CHAR_BIT    Number of bits in a char object (byte)  (>=8)
SCHAR_MIN   Minimum value for an object of type signed char
SCHAR_MAX   Maximum value for an object of type signed char
UCHAR_MAX   Maximum value for an object of type unsigned char
CHAR_MIN    Minimum value for an object of type char
            (either SCHAR_MIN or 0)
CHAR_MAX    Maximum value for an object of type char
            (either SCHAR_MAX or UCHAR_MAX)
SHRT_MIN    Minimum value for an object of type short int
SHRT_MAX    Maximum value for an object of type short int
USHRT_MAX   Maximum value for an object of type unsigned short int
INT_MIN     Minimum value for an object of type int
INT_MAX     Maximum value for an object of type int
UINT_MAX    Maximum value for an object of type unsigned int
LONG_MIN    Minimum value for an object of type long int
LONG_MAX    Maximum value for an object of type long int
ULONG_MAX   Maximum value for an object of type unsigned long int
LLONG_MIN   Minimum value for an object of type long long int
LLONG_MAX   Maximum value for an object of type long long int
ULLONG_MAX  Maximum value for an object of type unsigned long long
```

Heltalstyper
Storlekarna specificeras i <climits>

```
#include <iostream>
#include <climits>
int main()
{
    std::cout << CHAR_MIN << ", " << CHAR_MAX << ", ";
    std::cout << UCHAR_MAX << std::endl;
    std::cout << SHRT_MIN << ", " << SHRT_MAX << ", ";
    std::cout << USHRT_MAX << std::endl;
    std::cout << INT_MIN << ", " << INT_MAX << ", ";
    std::cout << UINT_MAX << std::endl;
    std::cout << LONG_MIN << ", " << LONG_MAX << ", ";
    std::cout << ULONG_MAX << std::endl;
    std::cout << LLONG_MIN << ", " << LLONG_MAX << ", ";
    std::cout << ULLONG_MAX << std::endl;

}
```

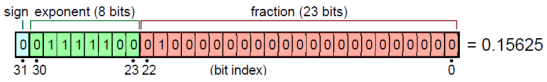
```
128, 127, 255
-32768, 32767, 65535
-2147483648, 2147483647, 4294967295
-9223372036854775808, 9223372036854775807, 18446744073709551615
-9223372036854775808, 9223372036854775807, 18446744073709551615
```


Flyttalstyper

`float` (oftast 32 bitar) `double` (oftast 64 bitar) `long double`

IEEE 754 32-bitars flyttal: $1 + 8 + 23$ bitar
(tecken + exponent + mantissa)

$$\begin{aligned} \text{värde} &= -1^{sign} \cdot (1.b_{22}b_{21}\ldots b_0)_2 \cdot 2^{(b_{30}b_{20}\ldots b_{23})_2 - 127} \\ &= -1^{sign} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} \cdot 2^{-i}\right) \cdot 2^{exponent-127} \end{aligned}$$



exponent $= \texttt{0x7c} = 124$

$$\text{värde} = +1.01_2 \cdot 2^{124-127} = 1.25 \cdot 2^{-3} = 1.25 \cdot 0.125$$

Flyttalstyper

Test med sizeof

```
#include <iostream>
using namespace std;
int main () {
   cout << "sizeof(float)="
        << sizeof(float)<<endl;
   cout << "sizeof(double)="
        << sizeof(double) << endl;
   cout << "sizeof(long double)="
        << sizeof(long double) << endl;
}
```

```
sizeof(float)=4
sizeof(double)=8
sizeof(long double)=12
```

I <cfloat> definieras minsta värde, största värde, epsilon, mm.

C-strängar
Detaljer och varningar

- en c-sträng är en *null-terminerad* **char**`[]`
- har inget skyddsnät
- kräver funktioner ur C-standardbiblioteket för
  - jämförelse,
  - att ta reda på längden,
  - kopiering, etc.

Inmatning
Varnande exempel

Den lästa strängen får inte plats i x

Satserna

```
char z[] {"zzzz"};
char y[] {"yyyy"};
char x[5];

stringstream sin{"aaaaaaaaaaaaaaaaa bbbbbb"};
sin >> x;

cout << x << " : " << y << " : " << z << endl;
```

Ger vid körning (på min dator):

```
aaaaaaaaaaaaaaaaa : aa : zzzz
```

- C-strängar ger inget skyddsnät
- läsningen till x har skrivit över (en del av) y
- getline() är säkrare

C-strängar – Funktioner

In- och utmatning av tecken

```
#include <cstring>  // Bibliotek att inkludera

strcpy(s,t)     // Kopierar t till s
strncpy(s,t,n)  // Variant med max n tecken

strcat(s,t)     // Lägger till t till slutet av s
strncat(s,t,n)  // Variant med max n tecken

strlen(s)       // Returnerar längden av s

strcmp(s,t)     // Jämför s och t
strncmp(s,t,n)  // Variant med max n tecken
// s<t, s==t, s>t ger resp. <0, =0, >0
```

Osäkra, undvik!

Initiering och tilldelning

Initiering

```
char s[4]= "abc"; // OK
char s[] = "abc"; // OK
char s[3]= "abc"; // Inte OK (pga \0)
```

Tilldelning

```
s = "def";       //Felaktigt!
s[0]='d'; s[1]='e'; s[2]='f'; // OK

// Bättre variant:
strncpy(s, "def", 4);  // kräver #include <cstring>
```


Strängkopiering
Varnande exempel

Satserna

```
char s[20];

strncpy(s, "abc", 4);
cout << s << endl;

strncpy(s, "kalle anka", 20);
cout << s << endl;

strncpy(s, "def", 3);
cout << s << endl;
```

ger utskriften

```
abc
kalle anka
defle anka
```

Satserna

```
int data[] {558646598, 65, 66};
char x[16];
char t[30] {"test"};

strncpy(x,"abcdefghijklmnop",16);
strcpy(t,x);
cout << t << endl;
```

ger utskriften

```
abcdefghijklmnopFEL!A
```

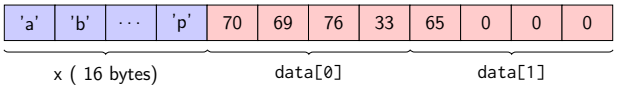
Tänk på att

- `strncpy` måste ha plats till avslutande `\0`.
- `strcpy` kopierar tills den hittar ett `\0` i `src`.

Strängkopiering
Varnande exempel: förklaring

```
int data[]{558646598, 65, 66};
char x[16];
```

- `int data[]` tolkas som `char[]`.
- representation i minnet



x (16 bytes) data[0] data[1]

Hexadecimal representation:

$$558646598_{10} = 214c4546_{16}$$
$$65_{10} = 41_{16}$$

Byte-ordning: *little-endian*

hex	ASCII)	dec
46	F	70
45	E	69
4c	L	76
21	!	33
41	A	65
0	\0	0
0	\0	0
0	\0	0
...	...	...

union

I en "vanlig" **struct** (**class**) allokeras utrymme motsvarande *summan* av de ingående delarna

```
struct DataS {
    int nr;
    double v;
    char txt[6];
};
```

Alla medlemmar i en **struct** ligger efter varandra i minnet.

I en **union** allokeras utrymme motsvarande *maxstorleken* av de ingående delarna

```
union DataU {
    int nr;
    double v;
    char txt[6];
};
```

Alla medlemmar i en **union** har *samma address*: bara en medlem åt gången kan användas.

union

Exempel på användning av `DataU`

```
union DataU {
    int nr;
    double v;
    char txt[6];
};
```

```
DataU a;

a.nr = 57;
cout << a.nr << endl;          57

a.v = 12.345;
cout << a.v << endl;           12.345

strcpy(a.txt, "Tjo");
cout << a.txt << endl;         Tjo
```

programmerarens ansvar att "rätt" medlem används

union
Varnande exempel

```
using std::cout;
using std::endl;

union Foo{
  int i;
  float f;
  double d;
  char c[10];
};
int main()
{
  Foo f;

  f.i = 12;
  cout << f.i << ", " << f.f << ", " << f.d << ", " << f.c << endl;

  strcpy(f.c, "Hej, du");
  cout << f.i << ", " << f.f << ", " << f.d << ", " << f.c << endl;
}
```

```
12, 1.68156e-44, 5.92879e-323, ^L
745170248, 3.33096e-12, 1.90387e-306, Hej, du
```

union
kapsla en union i en klass för att minska risken för fel

```
struct Bar{
    enum {undef, i, f, d, c} kind;
    Foo u;
};
void print(Bar b) {
    switch(b.kind){
    case Bar::i:
        cout << b.u.i << endl;
        break;
    case Bar::f:
        cout << b.u.f << endl;
        break;
    case Bar::d:
        cout << b.u.d << endl;
        break;
    case Bar::c:
        cout << b.u.c << endl;
        break;
    default:
        cout << "???" << endl;
        break;
    }
}
```

```
void test_kind()
{
    Bar b{};

    b.kind = Bar::i;
    b.u.i = 17;

    print(b);

    Bar b2{};
    print(b2);
}
```

```
17
???
```


union
anonym union – slipp en nivå

Ett annat alternativ är följande:

```
struct FooS{
  enum {undef, k_i, k_f, k_d, k_c} kind;
  union{
    int i;
    float f;
    double d;
    char c[10];
  };
};

FooS test;

test.kind = FooS::k_c;
strcpy(test.c, "Testing");
if(test.kind == FooS::k_c)
    cout << test.c << endl;
```

```
Testing
```


union
klass med anonym union och access-funktioner

```
struct FooS{
  enum {undef, k_i, k_f, k_d, k_c} kind;
  union{
    int i;
    float f;
    double d;
    char c[10];
  };
  FooS()           :kind{undef} {}
  FooS(int ii)     :kind{k_i},i{ii} {}
  FooS(float fi)   :kind{k_f},f{fi} {}
  FooS(double di) :kind{k_d},d{di} {}
  FooS(const char* ci) :kind{k_c} {strncpy(c,ci,10);}
  int     get_i() {assert(kind==k_i); return i;}
  float   get_f() {assert(kind==k_f); return f;}
  double get_d() {assert(kind==k_d); return d;}
  char*   get_c() {assert(kind==k_c); return c;}
  FooS& operator=(int ii)    {kind=k_i; i = ii; return *this;}
  FooS& operator=(float fi)  {kind=k_f; f = fi; return *this;}
  FooS& operator=(double di) {kind=k_d; d = di; return *this;}
  FooS& operator=(const char* ci){kind=k_c; strncpy(c,ci,10);
                                  return *this;}
};
```


Bit-operatorer
Operationer på låg nivå: Bit-operatorer

Alla variabler antas vara av typen **unsigned short int** vilket innebär 16 bitars positiva heltal

```
a = 77;          // a = 0000 0000 0100 1101
b = 22;          // b = 0000 0000 0001 0110
c = ~a;          // negera varje bit (bitvis komplement)
d = a & b;       // en bit i d = 1 om motsvarande bit i a OCH b == 1
e = a | b;       // en bit i e = 1 om motsvarande bit i a ELLER b == 1
f = a ^ b;       // en bit i f = 1 om motsvarande bit i a XOR b == 1
g = a << 3;      // skifta bitarna 3 steg vänster
h = c >> 5;      // skifta bitarna 6 steg höger (se upp med signed)
i = a & 0x000f; // bitmask: plocka ut de fyra lägsta bitarna i a
j = a | 0xf000; // sätt de högsta fyra bitarna till 1
k = a ^ (1<<4); // negera femte biten
```

Vanliga operationer:

set	clear	toggle
`a = a | (1 << 4);` `a |= (1 << 4);`	`a = a & ~(1 << 4);` `a &= ~(1 << 4);`	`a = a ^ (1 << 4);` `a ^= (1 << 4);`

Bit-operatorer

Operationer på låg nivå: Bit-operatorer

Alla variabler antas vara av typen **unsigned short int** vilket innebär 16 bitars positiva heltal

```
a = 77;          // a = 0000 0000 0100 1101
b = 22;          // b = 0000 0000 0001 0110
c = ~a;          // c = 1111 1111 1011 0010
d = a & b;       // d = 0000 0000 0000 0100
e = a | b;       // e = 0000 0000 0101 1111
f = a ^ b;       // f = 0000 0000 0101 1011
g = a << 3;      // g = 0000 0010 0110 1000
h = c >> 5;      // h = 0000 0111 1111 1101
i = a & 0x000f;  // i = 0000 0000 0000 1101
j = a | 0xf000;  // j = 1111 0000 0100 1101
k = a ^ (1 << 4); // k = 0000 0000 0101 1101
```


Bit-fält
Kan användas för att spara minne

Ange explicit antalet bitar med `var : antalbitar`

```
struct Bil {// post i ett bilregister
    char reg_nr[6];
    unsigned int arsmod : 7;
    unsigned int skatt_betald : 1;
    unsigned int besiktigad : 1;
    unsigned int avstalld : 1;
};
```

Bit-fält
Exempel

Åtkomst av bit-fält

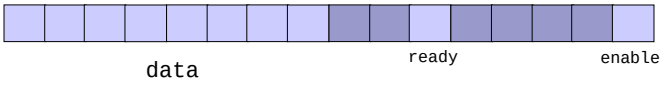
```
Bil b;
strncpy(b.reg_nr, "ABC123", 6);
b.arsmod = 97;
b.skatt_betald = true;
b.besiktigad = true;
b.avstalld = false;
cout << "Årsmodell: " << b.arsmod << endl;
if (b.skatt_bet && b.besiktigad)
    cout << "Bilen är OK";
```


Bit-fält

Register med 16 bitar:

```
struct Register {
    unsigned int enable :1;
    unsigned int        :4;
    unsigned int ready  :1;
    unsigned int        :2;
    unsigned int data   :8;
};
```

```
struct Register_alt{
    bool enable       :1;
    bool              :4;
    bool ready        :1;
    bool              :2;
    unsigned int data :8;
};
```



Bit-fält
Varningar

Bit-fält kan vara användbara i speciella fall, men de är *inte portabla*

- hur de läggs ut i minnet är *implementation defined*
- kompilatorn kan lägga till "utfyllnad" (*padding*)
- man kan inte ta *adressen till* (&) en bitfält-medlem
- ange alltid **signed** eller **unsigned**
  - **int**-fält av storlek 1 bör vara **unsigned**
- åtkomst kan bli långsammare än en "vanlig" struct
- heltalsvariabler och bitoperationer ofta ett alternativ

std::bitset (<bitset>)

- effektiv klass för att lagra ett antal bitar
  - kompakt
  - snabb
- har praktiska funktioner
  - test, **operator**[]
  - set, reset, flip
  - any, all, none, count
  - omvandling till/från string, och I/O
- jfr std::vector<**bool**>
  - std::bitset har fix storlek
  - en std::vector kan växa
  - men uppför sig inte riktigt som std::vector<T>

bitset, exempel:
Exempel: lagra 50 flaggor i 8 bytes

```
void test_bitop(){
  bool status;
  cout << std::boolalpha;

  unsigned long quizA = 0;

  quizA |= 1UL << 27 ;
  status = quizA & (1UL << 27);
  cout << "student 27: " ;
  cout << status << endl;

  quizA &= ~(1UL << 27);
  status = quizA & (1UL << 27);
  cout << "student 27: ";
  cout << status << endl;
}
```

```
student 27: true
student 27: false
```

```
void test_bitset(){
  bool status;
  cout << std::boolalpha;

  std::bitset<50> quizB;

  quizB.set(27);
  status = quizB[27];
  cout << "student 27: ";
  cout << status << endl;

  quizB.reset(27);
  status = quizB[27];
  cout << "student 27: ";
  cout << status << endl;
}
```

```
student 27: true
student 27: false
```


Komma-operatorn
(Orientering och varning)

Komma-operatorn expression1, expression2

- Utvärderar först expression1, därefter expression2
- uttrycket får värdet av expression2
- Exempel

```
string s;
while(cin >> s, s.length() > 5)
{
  //do something
}
```

Använd inte komma-operatorn!

`volatile` – "flyktiga" variabler

- Betyder (ungefär) att variabeln måste läsas/skrivas till minnet
- Maskinberoende
- Används i program som interagerar direkt med hårdvaran
  - T ex en variabel som uppdateras av hårdvaran själv eller en avbrottsrutin
- syntaxen fungerar på samma sätt som **`const`**

Trigraphs

- Historiskt: svensk 7-bitars teckenkod (variant på ASCII)

```
int main()                          int main()
{                                   å
  int x[10];          skrevs          int xÅ10Ä;
  ...                                 ...
}                                   ä
```

- i C infördes *trigraphs*, så att man även kunde skriva

```
int main()
??<
  int x??( 10 ??);
  ...
??>
```

- Om man i C++ skriver `cout << "what??!";` så skrivs `what|` ut.
- Försvinner troligen fr o m C++17

Läsanvisningar

Referenser till relaterade avsnitt i Lippman

Inbyggda typer 2.1
Bitoperatorer 4.8
Komma-operatorn 4.10
C-strängar 3.5.4
Union 19.6
Bitfält 19.8.1
volatile 18.8.2