EDAF30 – Programmering i C++

"De sista pusselbitarna"

Sven Gestegård Robertz
Datavetenskap, LTH

2015

Innehåll

Namnrymder `namespace`

- Avgränsa synligheten för namn
  - tydligare vilka funktioner/klasser/konstanter som hör ihop
  - minskar risken för namnkrockar
- Åtkomst av namn i namnrymder
  - fullständigt kvalificerat namn vid användning: `namnrymd::namn`
  - selektivt med **`using`** `namnrymd::namn`
  - import av alla namn med **`using namespace`** `namnrymd`
    - undvik generellt, eller använd avgränsat
    - använd aldrig i header-filer
- Namnrymder kan *utökas*
  - Utom (med några undantag) `std` (⇒ undefined behaviour)

Namnrymder `namespace` Exempel

deklarationer (.h)

```
namespace foo {
  void test();
}

namespace bar {
  void test();
}
```

definitioner (.cpp)

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

namespace foo {
  void test()
  {
    cout << "foo::test()\n";
  }
}

void bar::test()
{
  cout << "bar::test()\n";
}
```

```
int main()
{
    foo::test();
    bar::test();
    using namespace foo;
    test();
}
```

```
foo::test()
bar::test()
foo::test()
```


Namnrymder `namespace`

- Den namnlösa namnrymden
  - endast synlig i filen där den deklarerats
  - används för att dölja saker (jfr **`static`** i C)

```
namespace foo {
  void test()
  {
    cout << "foo::test()\n";
  }
}
namespace {
  void test()
  {
    cout << "::test()\n";
  }
}
```

```
int main()
{
  test();
  foo::test();
  ::test();
}
```

```
::test()
foo::test()
::test()
```

- Alternativa namn för namnrymder:

```
namespace test=my_namespace_for_testing_stuff;
```


`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
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 åtta lägsta bitarna i a
j = a | 0xf000; // sätt de högsta fyra bitarna till 1
k = a ^ (1<<4); // negera femte biten
```

Vanliga operationer:

set

```
a = a | (1 << 4);
a |= (1 << 4);
```

clear

```
a = a & ~(1 << 4);
a &= ~(1 << 4);
```

toggle

```
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

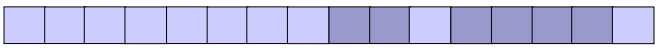
```
Bil b;
strncpy(b.reg_nr, "ABC123", 6);
b.arsmod = 97;
b.skatt_betald = true;
b.besiktigad = true;
b.avstalld = false;
cout << "Årsmodell: " << b.arsmod << endl;
if (b.skatt_bet && b.besiktigad)
    cout << "Bilen är OK";
```


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
```


istream_iterator<T>

istream_iterator<T> : konstruktorer

```
istream_iterator();  // ger en  end-of-stream istream iterator
istream_iterator (istream_type& s);
```

```
#include <iterator>

stringstream ss{"1 2 12 123 1234\n17\n\r42"};

istream_iterator<int> iit{ss};
istream_iterator<int> iit_end;

while(iit != iit_end) {
    cout << *iit++ << endl;
}
```

```
1
2
12
123
1234
17
42
```


istream_iterator<T>

Användning för att initiera vector<**int**>:

```
stringstream ss{"1 2 12 123 1234\n17\n\r42"};

istream_iterator<double> iit{ss};
istream_iterator<double> iit_end;

vector<int> v{iit, iit_end};

for(auto a : v) {
    cout << a << " ";
}
cout << endl;
```

```
1 2 12 123 1234 17 42
```


istream_iterator
Felhantering

```
stringstream ss2{"1 17 kalle 2 nisse 3 pelle\n"};
istream_iterator<int> iit2{ss2};
while(!ss2.eof()) {
  while(iit2 != iit_end) { cout << *iit2++ << endl; }
  if(ss2.fail()){
    ss2.clear();
    string s;
    ss2 >> s;
    cout << "ss2: not an int: " << s << endl;
    iit2 = istream_iterator<int>(ss2); // create new iterator
  }
}
cout << boolalpha << "ss2.eof(): " << ss2.eof() << endl;
```

```
1
17
ss2: not an int: kalle
2
ss2: not an int: nisse
3
ss2: not an int: pelle
ss2.eof(): true
```

- vid fel sätts `fail`-biten för strömmen
- iteratorn sätts till `end`
- om man ändrar strömmen måste man skapa en ny iterator

Iteratorers giltighet

Generellt, om man ändrar strukturen en iterator refererar in i *blir iteratorn ogiltig*. Exempel:

- insättning
  - sekvenser
    - `vector`, deque* : alla iteratorer blir ogitiga
    - `list` : iteratorer påverkas inte
  - associativa containers (`set`, `map`)
    - iteratorer påverkas inte
- borttagning
  - sekvenser
    - `vector` : iteratorer *efter* de borttagna elementen blir ogitiga
    - deque : alla iteratorer blir ogitiga (i princip*)
    - `list` : iteratorer till de borttagna elementen blir ogiltiga
  - associativa containers (`set`, `map`)
    - iteratorer påverkas inte
- storleksförändring (`resize`): som insättning/borttagning

ostream_iterator och algoritmen copy

ostream_iterator

```
ostream_iterator (ostream_type& s);
ostream_iterator (ostream_type& s, const char_type* delimiter);
```

```
    stringstream ss{"1 2 12 123 1234\n17\n\r42"};

    istream_iterator<double> iit{ss};
    istream_iterator<double> iit_end;

    cout << fixed << setprecision(2);
    ostream_iterator<double> oit{cout, " <-> "};

    std::copy(iit, iit_end, oit);
```

```
1.00 <-> 2.00 <-> 12.00 <-> 123.00 <-> 1234.00 <-> 17.00 <-> 42.00 <->
```

Tumregler, "defaults"

- Iteration, *range for*
- *return value optimization*
- värde- eller referensanrop?
- referens- eller pekarparameter? (utan överföring av ägarskap)
- default-konstruktor och initiering
- resurshantering: RAII och *rule of three (five)*
- var försiktig med typomvandling. Använd *named casts*

använd *range for*

```
for(auto e : collection) {
   // ...
}
```

Använd *range for* om du ska iterera över *hela* intervallet:

- tydligare och säkrare
- ingen risk att råka tilldela iteratorn
- ingen risk att råka tilldela loop-variabeln
- ingen pekararitmetik

Fungerar på varje typ `T` som har

- medlemsfunktioner `begin` och `end`, eller
- fria funktioner `begin(T)` och `end(T)`

return value optimization (RVO)

Kompilatorn får lov att optimera bort kopiering av objekt vid **`return`** från funktioner

- *return by value* ofta effektivt, även för större objekt
- RVO tillåtet *även om copy-konstruktorn eller destruktorn har sidoeffekter*
- undvik sådana sidoeffekter för att göra koden portabel

Tumregler för funktionsparametrar

- Returnera värde oftare
- Överanvänd inte värdeanrop

"reasonable defaults"

	cheap to copy	moderately cheap to copy	expensive to copy
Out	`X f()`		`f(X&)`
In/Out	`f(X&)`		
In	`f(X)`	`f(const X&)`	

För resultat (returvärde), om kostnaden för kopiering är

- liten, eller måttlig ($< 1\ k$, sammanhängande): returnera värde (moderna kompilatorer gör RVO: return value optimization)
- stor : använd referensanrop som utparameter
  - eller kanske allokera på heapen och returnera pekare

parametrar: referens eller pekare?

- nödvändig/obligatorisk parameter: skicka referens
- valfri parameter: skicka pekare (kan vara `nullptr`)

```
void f(widget& w)
{
    use(w); //required parameter
}

void g(widget* w)
{
    if(w) use(w); //optional parameter
}
```


Default-konstruktor och initiering

- (automatiskt genererad) default-konstruktor (=**`default`**) initierar inte alltid medlemmar
  - *globala variabler* initieras till 0 (motsv)
  - *lokala variabler* initieras inte

```
struct A { int x; };

int a; // a initieras till 0
A b;   // b.x initieras till  0

int main() {
    int c;          // c initieras inte
    int d = int(); // d initieras till 0

    A e;            // e.x initieras inte
    A f = A();      // f.x initieras till 0
    A g{};          // g.x initieras till 0
}
```

- *använd alltid initierarlista*
- *implementera alltid default-konstruktor (eller =**`delete`**)*

RAII: Resource aquisition is initialization

- Allokera resurser för ett objekt i konstruktorn
  - OBS! allokering kan kasta exception
- Släpp resurserna i destruktorn
- Enklare resurshantering, inga nakna **new** och **delete**
- Exception-säkerhet: destruktorer körs när block lämnas
- *Resource-handle*
  - Objektet självt är litet
  - Pekare till större data på heapen
  - Exempel, vår `Vektor`-klass: pekare + storlek
  - Drar nytta av move-semantics
- `unique_ptr` är en *handle* till ett specifikt objekt. Använd *om du behöver pekar-semantik*, t ex för polymorfa typer.
- Föredra specifika *resource handles* framför smarta pekare.

"Rule of three"
Canonical construction idiom

Om en klass implementerar någon av dessa:

1. Destruktor
2. Copy constructor
3. Copy assignment operator

ska den (i princip alltid) implemetera alla tre.

Om en av de automatiskt genererade inte passar, gör troligen inte de andra det heller.

"Rule of ~~three~~ five"
Canonical construction idiom, fr o m C++11

Om en klass implementerar någon av dessa:

1. Destruktor
2. Copy constructor
3. Copy assignment operator
4. *Move* constructor
5. *Move* assignment operator

ska den (i princip alltid) implemetera alla ~~tre~~ fem.

Smarta pekare: `unique_ptr`
Exempel

```
struct Foo {
    int i;
    Foo(int ii=0) :i{ii} { std::cout << "Foo(" << i <<")\n"; }
    ~Foo() { std::cout << "~Foo("<<i<<")\n"; }
};
void test_move_unique_ptr()
{
   std::unique_ptr<Foo> p1(new Foo(1));
   {
      std::unique_ptr<Foo> p2(new Foo(2));
      std::unique_ptr<Foo> p3(new Foo(3));
      // p1 = p2; // fel! kan ej kopiera unique_ptr
      std::cout << "Assigning pointer\n";
      p1 = std::move(p2);
      std::cout << "Leaving inner block...\n";
   }
   std::cout << "Leaving program...\n";
}
```

```
Foo(1)
Foo(2)
Foo(3)
Assigning pointer
~Foo(1)
Leaving inner block..
~Foo(3)
Leaving program...
~Foo(2)
```

`Foo(2)` överlever inre blocket eftersom `p1` *övertar ägarskapet.*

Typomvandlingar (*casting*)
Namngivna typomvandlingar

- **`static_cast`**`<new_type> (expr)`
  - omvandlar mellan kompatibla typer (*kollar inte talområden*)
- **`reinterpret_cast`**`<new_type> (expr)`
  - inget skyddsnät, samma som C-stil: `(new_type) expr`
- **`const_cast`**`<new_type> (expr)` - lägger till eller tar bort **const**
- **`dynamic_cast`**`<new_type> (expr)` - används för pekare till klasser. Gör typkontroll vid *run-time*, som i Java.

Exempel

```
char c;              // 1 byte
int *p = (int*) &c;  // pekar på int: 4 bytes

*p = 5; // fel vid exekvering, stack-korruption:
        // skriver över 3 bytes efter &c

int *q = static_cast<int*> (&c); // kompileringsfel
```


Tumregler, "defaults"

- använd *range for* om du ska iterera över *hela* intervallet
- *return value optimization*
- värde- eller referensanrop?
- referens- eller pekarparameter? (utan överföring av ägarskap)
- default-konstruktor och initiering
- resurshantering: RAII och *rule of three (five)*
- var försiktig med typomvandling. Använd *named casts*