EDAF30 – Programmering i C++

12. Exempel, kommentarer och råd

Sven Gestegård Robertz
Datavetenskap, LTH

2017

Innehåll

`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

- Histioriskt: 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

Whitespace i kod

- Whitespace ignoreras (oftast) av C++-kompilatorn
- men spelar stor roll för läsbarhet
- Var konsekvent, följ standard för indentering etc.
- Exempel:

```
void loop()
{
    int i = 5;

    do{
        cout << i << endl;
    }while( i --> 0);          d v s while( i-- > 0)
}
```

- Se upp med fel som t ex `if (a =! b)` istf `if (a != b)`
  (D v s tilldelningen `a = !b` *i stället för jämförelsen* `a != b`*.)*

Exempel
En klasshierarki

```
struct Foo{
    virtual void print() const {cout << "Foo" << endl;}
};

struct Bar :Foo{
     void print() const override {cout << "Bar" << endl;}
};

struct Qux :Bar{
     void print() const override {cout << "Qux" << endl;}
};
```

Polymorf klass
exempel, *object slicing*

Vad skrivs ut?

```
void print1(const Foo* f)
{
  f->print();
}
void print2(const Foo& f)
{
  f.print();
}
void print3(Foo f)
{
  f.print();
}
```

```
void test()
{
    Foo* a = new Bar;
    Bar& b = *new Qux;
    Bar  c = *new Qux;

    print1(a);
    print1(&b);
    print1(&c);
    std::cout << std::endl;
    print2(*a);
    print2(b);
    print2(c);
    std::cout << std::endl;
    print3(*a);
    print3(b);
    print3(c);
}
```


Arv och *scope*

- En subklass *scope* är kapslat (eng: *nested*) inuti superklassens
  - Namn i superklassen syns i subklasser
  - *om de inte döljs* av samma namn i subklassen
- Använd *scope-operatorn* `::` för att komma åt dolda namn
- Namnuppslagning sker vid kompilering
  - *Statisk typ* för en pekare eller referens styr vilka namn som syns (som i Java)
  - Virtuella funktioner måste ha samma parametertyper i subklasser

Ingen överlagring mellan nivåer i en klasshierarki

Exempel

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.

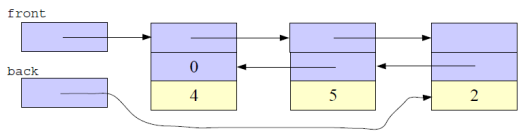
Exempel

```
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
```


Standardklassen `list`

- Liksom `vector` och `deque` utgör `list` en implementering av sekvenser
- Intern representation är en s.k. *länkad lista* (och inte en array som i fallen `vector` och `deque`)
- Varje element är ett eget objekt, med pekare till nästa (och föregående) element. List-objektet har pelare till första (och sista) elementet

```
list<int> l;
l.push_back(4); l.push_back(5); l.push_back(2);
```



Standardklassen `list`

Operationerna på en `list` är samma som på en `deque` förutom att
- vektorindexering (`[]` och `at()`) inte är tillåten
- det även finns följande operationer (*i stället för motsvarande algoritmer*):

l.reverse()	Vänder bak och fram på listan l
l.remove(e)	Tar bort alla e:n från listan l
l.unique()	Tar bort alla förekomster, utom den första, ur varje sammanhägande grupp av lika element i listan l
l.sort()	sorterar listan
l.merge(l2)	Sorterar in listan l2 i listan l. l2 blir tom (för sorterade listor)
l.splice(p,l2)	Skjuter in elementen i listan l2 i listan l, före platsen p (iterator). Listan l2 blir tom.

\+ varianter av vissa av dessa med fler parametrar

std::list

- I teorin effektiv insättning på godtycklig plats
  - $O(1)$: konstant-tid
  - ingen kopiering krävs
  - i praktiken ofta långsammare än `std::vector` på en modern dator
- använd `std::forward_list` för listor som oftast är tomma

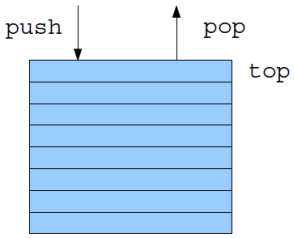
Köer och stackar

- Förenklade standardklasser, s.k. *adapterklasser*, implementerade med hjälp av någon av de andra standardklasserna: `stack`, `queue`
- Enklare gränssnitt med färre operationer
- Kan inte använda iteratorer
- underliggande container: `std::deque` som default

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

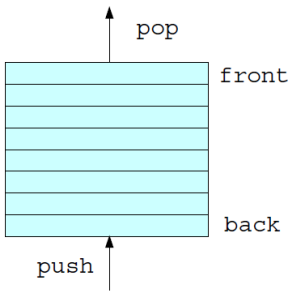
Köer och stackar

- Stack: LIFO-struktur (Last In First Out)
- Operationer: `push`, `pop`, `top`, `size` och `empty`



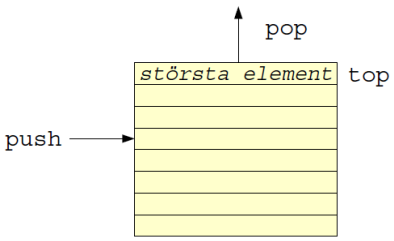
Köer och stackar

- Kö: FIFO-struktur (First In First Out)
- Operationer: `push`, `pop`, `front`, `back`, `size` och `empty`



Köer och stackar

- Prioritetskö: Som kö fast elementen har prioritet. Elementet med högst prioritet ligger först i kön.
- Operationer: `push`, `pop`, `top`, `size` och `empty`



Köer och stackar

Exempel:

- Använda stack för att behandla element "baklänges" t ex läsa in värden och skriva ut dem baklänges
- Använda prioritetskö för sortering

Köer och stackar

Exempel: Använda stack för baklängesutskrift

```
#include <stack>
#include <iostream>
using namespace std;

int main () {
    stack<char> s;
    char c;
    cout << "Skriv in text och avsluta med <CR>";
    while ((c = cin.get()) != '\n')
        s.push(c);
    while (!s.empty()) {
        cout << s.top();
        s.pop();
    }
}
```


Köer och stackar

Exempel: Lägga in heltal i kö och skriva ut dem

```
#include <queue>
#include <iostream>
using namespace std;

int main () {
    queue<int> q;
    int i;
    cout << "Skriv in tal och avsluta med Ctrl-D" << endl;
    while (cin >> i)
        q.push(i);
    while (!q.empty()) {
        cout << q.front() << ' ';
        q.pop();
    }
}
```

Köer och stackar

Exempel: Skriva ut tal i storleksordning

```
#include <queue>
#include <iostream>
using namespace std;
int main () {
    priority_queue<int> p;
    int i;
    cout << "Skriv in tal och avsluta med Ctrl-D" << endl;
    while (cin >> i)
        p.push(i);
    while (!p.empty()) {
        cout << p.top() << ' ';
        p.pop();
    }
}
```

NB! `ctrl-D` *är* `EOF` *på Linux/Mac. I DOS/windows används* `ctrl-Z`

Containerklasser – Klassificering

Sekvenser

- `vector<T>`
- `deque<T>`
- `list<T>`

adapterklasser (begränsade versioner av ovanstående)

- `queue<T, Sequence>`
- `priority_queue<T, Sequence>`
- `stack<T, Sequence>`

Läsanvisningar

Referenser till relaterade avsnitt i Lippman

Container adapters 9.6