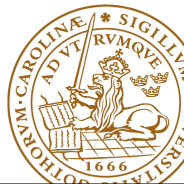
EDAF30 – Programmering i C++

12. Mer om polymorfism och containers

Sven Gestegård Robertz

Datavetenskap, LTH

2016



Innehåll

1. Klasser och arv
   - dynamisk polyformism
   - statisk polyformism
2. Mer om standard-containers
   - Listor
   - Adapterklasser: Köer och stackar

Polymorfa typer
Gränssnitt definieras av en (abstrakt) basklass

Abstrakta typer isolerar användaren från implementationsdetaljer och *skiljer gränssnittet från representationen*:

- Representationen av objekt (*inkl. storleken!*) är okänd
- Kan bara refereras via pekare eller referenser
- Måste (typiskt) allokeras på heapen
  - objekt av en abstrakt typ kan inte skapas
  - *factory-metod* måste returnera pekare eller referens
  - objekt på stacken blir inte polymorfa
  - . . . undantag: referensanrop

Exempel

```
class Animal{
public:
  void speak() const { cout << get_sound() << endl;}
  virtual string get_sound() const =0;
  virtual ~Animal() =default;
};

class Dog :public Animal{
public:
  string get_sound() const override {return "Woof!";}
};
class Cat :public Animal{
public:
  string get_sound() const override {return "Meow!";}
};
class Bird :public Animal{
public:
  string get_sound() const override {return "Tweet!";}
};
class Cow :public Animal{
public:
  string get_sound() const override {return "Moo!";}
```


Exempel

```
int main()
{
  Dog d;
  Cat c;
  Bird b;
  Cow w;

  d.speak();      Woof!
  c.speak();      Meow!
  b.speak();      Tweet!
  w.speak();      Moo!
}
```


Exempel
Container med polymorfa objekt

```
int main()
{
  Dog d;
  Cat c;
  Bird b;
  Cow w;

  vector<Animal> zoo{d,c,b,w};

  for(auto x : zoo){
    x.speak();
  };

}
```

```
error: cannot allocate an object of abstract type 'Animal'
```

Exempel
Använd pekare

```
int main()
{
  Dog d;
  Cat c;
  Bird b;
  Cow w;

  vector<Animal*> zoo{&d,&c,&b,&w};

  for(auto x : zoo){
    x->speak();        Woof!
  };                   Meow!
                       Tweet!
}                      Moo!
```


Exempel
Referensanrop

```
void test_polymorph(const Animal& a)
{
    a.speak();
}

int main()
{
    Dog d;
    Cat c;
    Bird b;
    Cow w;

    test_polymorph(d);          Woof!
    test_polymorph(c);          Meow!
    test_polymorph(b);          Tweet!
    test_polymorph(w);          Moo!
}
```


Exempel
Factory-metod

```
#include <random>
#include <stdexcept>

Animal* make_animal()
{
    static std::default_random_engine gen;
    static std::uniform_int_distribution<> dis(1, 4);

    switch(dis(gen)){
        case 1:
            return new Dog();
        case 2:
            return new Cat();
        case 3:
            return new Bird();
        case 4:
            return new Cow();
    };
    throw std::runtime_error("We should not come here");
}
```


Exempel
Factory-metod

```
void test_factory()
{
    cout << "test_factory:\n";
    for(int i=0; i != 10; ++i) {
        auto a = make_animal();
        a->speak();
        delete a;
    }
}
```


Exempel
Factory-metod med `std::unique_ptr`

```
#include <memory>
#include <random>
#include <stdexcept>

std::unique_ptr<Animal> make_unique_animal()
{
    static std::default_random_engine gen;
    static std::uniform_int_distribution<> dis(1, 4);

    switch(dis(gen)){
        case 1:
            return std::unique_ptr<Animal>{new Dog()};
        case 2:
            return std::unique_ptr<Animal>{new Cat()};
        case 3:
            return std::unique_ptr<Animal>{new Bird()};
        case 4:
            return std::unique_ptr<Animal>{new Cow()};
    };
    throw std::runtime_error("We should not come here");
}
```


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;}
};
```

Polyform klass
exempel

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);
}
```


Statiskt polyform klass
Curiously recurring template pattern(CRTP)

```
template <class D>
class Animal_static{
public:
  void speak() const { cout << get_sound() << endl;}
  string get_sound() const {
    return static_cast<const D*>(this)->get_sound();
  }
};

class Dog_static :public Animal_static<Dog_static>{
public:
  string get_sound() const{return "Woof!";}
};
class Cat_static :public Animal_static<Cat_static>{
public:
  string get_sound() const{return "Meow!";}
};
class Bird_static :public Animal_static<Bird_static>{
public:
  string get_sound() const{return "Tweet!";}
};
```


Statisk polyform klass
Användning

```
void test_crtp()
{
  cout << "test_crtp:\n";
  Dog_static d;
  Cat_static c;
  Bird_static b;
  Cow_static w;

  d.speak();       Woof!
  c.speak();       Meow!
  b.speak();       Tweet!
  w.speak();       Moo!

  vector<Animal_static*> zoo{&d, &c, &b, &w};
  for(auto a : zoo){
    a->speak();
  }
}
```

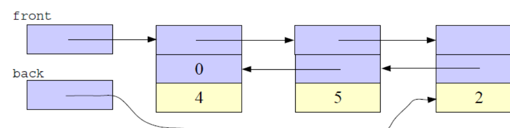
Error: template argument 1 is invalid

`Animal_static` *är inte en klass utan en klassmall.*

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)

```
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. Utöver det finns följande operationer

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.merge(l2)	Sorterar in listan `l2` i listan `l`. `l2` blir tom
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

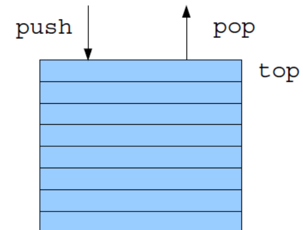
- I teorin effektiv insättning på godtycklig plats
  - $O(1)$: konstant-tid
  - ingen kopiering krävs
  - i praktiken ofta långsammare än `std::vector` på en modern dator
- använd `std::forward_list` för listor som oftast är tomma

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

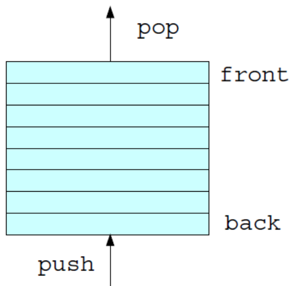
Köer och stackar

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



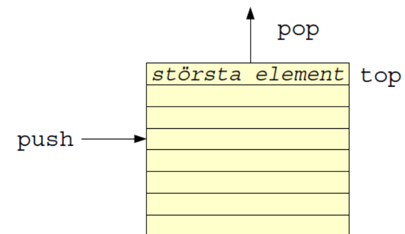
Köer och stackar

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


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

Virtuella funktioner 15.3

Containers och arv 15.8

Container adapters 9.6