EDAF30 – Programmering i C++

7. Felhantering. Funktionsmallar

Sven Gestegård Robertz
Datavetenskap, LTH

2017

Innehåll

Felhantering
Tre nivåer av felhantering:

1 Vidta lämplig åtgärd direkt för att möjliggöra fortsatt exekvering
2 Kategorisera och skicka vidare felet till någon annan programenhet, som förväntas hantera det
3 Identifiera felet, ge något felmeddelande samt låt därefter programmet krascha *("fail-fast", t ex* `assert`*)*

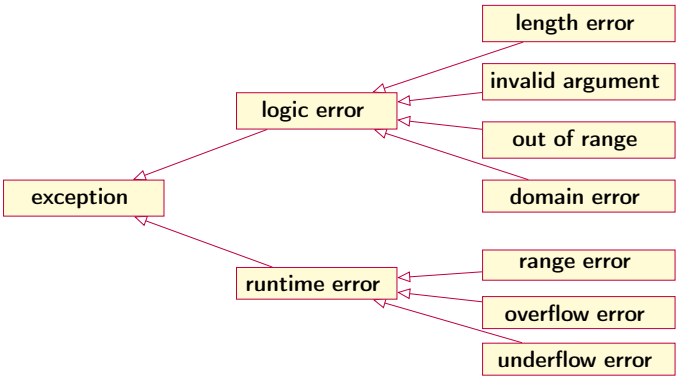
Nivå 2: exceptions (eller returvärde)

Exceptionella händelser (*exceptions*, "undantag")

- Felhantering kan göras med **`throw`** och **`catch`**, (t.ex. vid indexering utanför gränser). Likt Java.
- Vid **`throw`** poppas aktiveringsposter från stacken tills en funktion som innehåller ett matchande **`catch`** hittas.
- Om ett exception inte fångas kommer programmet att krascha. (Programmet avslutas genom att `terminate()` anropas.)
- Standardklasser för exceptions: **`#include`** `<stdexcept>`

Standardbibliotekets exception-klasser

Klassträd för klasserna i `<stdexcept>`



Felhantering
Att fånga exceptionella händelser

```
try {
        // Programkod där fel kan uppstå
}
catch (parameter av någon typ) {
        // Kod som tar hand om denna typ av fel
}
catch (parameter av någon typ) {
        // Kod som tar hand om denna typ av fel
}
catch (...) {
        // Kod som tar hand om resten (default)
}
```

Den första **`catch`**-sats som matchar typen väljs.
⇒ Fånga subklasser före superklassen.

Att fånga exceptionella händelser

Exempel:

```
try {
        cout << "Ange ett tal: ";
        int i;
        if (cin >> i){
            int r = f(i);
            cout << "Resultat: " << r << endl;
        }
}
catch(overflow_error) {
        cout << 'Resultat utanför giltigt område');
}
catch(exception& e) {
        cout << typeid(e).name() << ": " << e.what() << endl;
}
```


Att fånga exceptionella händelser

Exempel:

```
try {
        cout << "Ange ett tal: ";
        int i;
        if (cin >> i) {
            int r = f(i);
            cout << "Resultat: " << r << endl;
        }
}
catch(overflow_error&) {
        cout << 'Resultat utanför giltigt område');
}
catch(exception& e) {
        cout << typeid(e).name() << ": " << e.what() << endl;
}
```

Fördefinierad funktion i klassen exception

Att fånga exceptionella händelser
... och skicka vidare

```
try{
  do_something();
}
catch {length_error& le) {
  // hantera length error
}
catch {out_of_range&) {
  throw;   // skicka vidare
}
catch (...) {
  // default
}
```


Att generera exceptionella händelser

Syntax för throw

```
throw std::exceptionnamn("Extra info");
```

Exempel:

```
throw invalid_argument("till f2");
```


Att generera exceptionella händelser

Deklaration av egna undantag som subklasser

```
#include<stdexcept>

class communication_error : public runtime_error {
public:
        communication_error(const string& mess = "")
                : runtime_error(mess) {}
};
```

Användning av egendeklararade undantag

```
throw communication_error("Checksum error");
```


Att generera exceptionella händelser

Deklaration av helt egna undantag

```
struct MyOwnException{
  MyOwnException(const std::string& msg, int val)
          : m{msg},v{val} {}
    std::string m;
    int v;
};
```

Användning av egendeklararade undantag

```
void f() {
    throw MyOwnException("An error occurred", 17);
}

void test1() {
    try{
        f();
    } catch(MyOwnException &e) {
        cout << "Exception: " << e.m << " - " << e.v << endl;
    }
}
```

Att fånga exceptionella händelser
Resurshantering: destruktorer körs vid "*stack unwinding*"

```
struct Foo {
  int x;
  Foo(int ix) :x{ix} {
    cout << "Foo("<<x<<")\n";}
  ~Foo() {
    cout << "~Foo("<<x<<")\n";}
};

void test(int i)
{
  Foo f(i);
  if(i == 0) {
    throw std::out_of_range("noll?");
  } else {
    Foo g(100+i);
    test(i-1);
    cout << "after call to test("
         << i-1 << ")\n";
  }
}
```

```
int main() {
 Foo f(42);
 try{
   Foo g(17);
   test(2);
 } catch(std::exception& e) {
    cout<<e.what()<< endl; }
}
```

```
Foo(42)
Foo(17)
Foo(2)
Foo(102)
Foo(1)
Foo(101)
Foo(0)
~Foo(0)
~Foo(101)
~Foo(1)
~Foo(102)
~Foo(2)
~Foo(17)
noll?
~Foo(42)
```


Specifikation av exceptionella händelser i C++11

Nyckelordet `noexcept` anger om en funktion får lov att generera exceptions eller inte.
Att inte ange något är samma som `noexcept(false)`.

I deklarationen av funktionen

```
struct Foo {
    void f();
    void g() noexcept;
};
```

och i definitionen av funktionen

```
#include<stdexcept>
void Foo::f() {
    throw std::runtime_error("f failed");
}
void Foo::g() noexcept{
    throw std::runtime_error("g lied and failed");
}
```


Specifikation av exceptionella händelser
Excempel på användning

```
#include<typeinfo> // for typeid

void test_noexcept()
{
    Foo f;

    try {
        f.f();
    }catch(std::exception &e) {
        cout << typeid(e).name() << ": " << e.what() << endl;
    }
    try {
        f.g();
    }catch(std::exception &e) {
        cout << typeid(e).name() << ": " << e.what() << endl;
    }
    cout << "done\n";
}
```

```
St13runtime_error: f failed
terminate called after throwing an instance of 'std::runtime_error'
  what():  g lied and  failed
```


Specifikation av exceptionella händelser
äldre C++, använd inte

I äldre C++ fanns "händelselistor" för en funktion: typerna för de händelser som kan genereras av funktionen specificeras med nyckelordet **`throw`**.

Exempel på händelselista:

```
int f(int) throw(typ1, typ2, typ3) {
        //...
        throw typ1("Fel av typ 1 har inträffat");
        throw typ2("Fel av typ 2 har inträffat");
        throw typ3("Fel av typ 3 har inträffat");
        //...
}
```

Ingen lista ⇒ Alla typer av händelser kan genereras
Tom lista (`throw()`) ⇒ Inga händelser kan genereras

Tumregler för *exceptions*

- Tänk på felhantering tidigt i designen
- Använd specifika exception-typer, inte inbyggda typer. (använd inte `throw 17;`, `throw false;` , etc.)
- "Throw by value, catch by reference"
- Om en funktion inte ska kasta exceptions, deklarera `noexcept`.
- Specificera *invarianter* för dina typer
  - Konstruktorn säkerställer invarianten, eller kastar ett exception.
  - Medlemsfunktioner kan lita på invarianten.
  - Medlemsfunktioner måste upprätthålla invarianten.
  - Exempel: `Vektor`
    - storleken `ant` är ett positivt heltal
    - arrayen `p` pekar på är av storlek `ant`
    - Om allokeringen av arrayen misslyckas kastas `std::bad_alloc`
- Om nånting kan kontrolleras vid kompilering, använd `static_assert`.

Static assert

```
constexpr int some_param = 10;

int foo(int x)
{
    static_assert(some_param > 100, "some_param too small");
    return 2*x;
}

int main()
{
    int x = foo(5);

    std::cout << "x is " << x << std::endl;
    return 0;
}

// error: static assertion failed: some_param too small
```

Generisk programmering
Templates (mallar)

- Använder *typparametrar* för att skriva mer generella klasser och funktioner
- Slipper manuellt skriva en ny klass/funtion för varje datatyp som ska hanteras
- statisk polymorfism
- En mall *instansieras* av kompilatorn för typer den används för
  - varje instans är en separat klass/funktion
  - vid kompilering: ingen kostnad vid exekvering

Funktionsmallar

Exempel: Vanliga minimifunktionen

```
template<class T>
const T& minimum(const T& a, const T& b) {
    if (a < b)
        return a;
    else
        return b;
}
```

Kan instansieras för alla typer som har operatorn <

Funktionsmallar

Instansiering av funktionsmallar sker i vissa fall automatiskt t.ex. vid utskrift:

```
double x = 7.0, y = 5.0;
long m = 5, n = 7;
//...
cout << minimum(x, y) << endl;
cout << minimum(m, n) << endl;

// Blandat funkar inte (ingen casting görs!)
cout << minimum(m, y) << endl;

// För att forcera casting ger man typ-parametern explicit
cout << minimum<double>(m, y) << endl;
```

En explicit instans av en funktionsmall är en vanlig funktion
⇒ implicit typkonvertering av argument

Funktionsmallar
Överlagring med vanlig funktion

```
struct Name{
    string s;
    //...
};
```

Överlagring för `Name&`

```
const Name& minimum(const Name& a, const Name& b)
{
    if(a.s < b.s)
        return a;
    else
        return b;
}
```


Funktionsmallar
Funktionsmall för minsta elementet i en array

```
template<typename T>
T& min_element(T a[], size_t n)
{
    cout << "[min_element<"<<typeid(T).name()<<"[]>]\n";
    size_t idx = 0;

    for(size_t i = 1; i < n; ++i) {
        if(a[i] < a[idx])
            idx = i;
    }
    return a[idx];
}
```

Användning (*kompilatorn härleder typparametern T*)

```
int a[] {3,5,7,6,8,5,2,4};

int ma = min_element(a, sizeof(a)/sizeof(int));
```


Funktionsmallar kan överlagras

Generalisering: överlagra `min_element` för fler datastrukturer

Funktionsmallar
Funktionsmall för minsta elementet i iterator-par

```
template<typename FwdIterator>
ForwardIterator min_element(FwdIterator start, FwdIterator end)
{
    if(start==end)
        return end;
    ForwardIterator res=start;

    auto it = start;
    while(++it != end){
        if(*it < *res)
            res = it;
    }
    return res;
}
```

- Standardalgoritmerna är templates
- Detta är en version av `std::min_element`

Användning

```
int a[] {3,5,7,6,8,5,2,4};
int ma2 = *min_element(begin(a), end(a));
int ma3 = *min_element(a+2,a+4);

vector<int> v{3,5,7,6,8,5,2,4};
int mv = *min_element(v.begin(), v.end());
```


Funktionsmallar
Funktionsmall för minsta elementet i en `vector<T>`

Parametriserad på *element-typen* för `vector`

```
template<typename T>
T& min_element(vector<T>& c)
{
    if(v.begin()==v.end())
        throw std::range_error("empty vector");
    return *min_element(v.begin(), v.end());
}
```

Användning

```
vector<int> v{3,5,7,6,8,5,2,4};
int mv2 = min_element(v);
```


Funktionsmallar
Funktionsmall för minsta elementet i en container

Variant som kan användas för en godtycklig klass som

- har `begin()` och `end()`
- har en **typedef** som definierar namnet `value_type`

(detta uppfylls av alla standard-containers)

```
template<class Container>
typename Container::value_type& min_element(Container& c)
{
    if(c.begin()==c.end())
        throw std::range_error("empty container");
    return *min_element(c.begin(), c.end());
}
```

Kan även deklareras med två typ-parametrar:

```
template<typename Container,
         typename T = typename Container::value_type>
T& min_element(Container& c)
```


Funktionsmallar
Funktionsmall för minsta elementet i en array

Trick: värdeparameter för arrayens storlek, och inferens

```
template<typename T, size_t N>
T& min_element(T (&a)[N])
{
    size_t idx = 0;

    for(size_t i = 1; i < N; ++i) {
        if(a[i] < a[idx])
            idx = i;
    }
    return a[idx];
}
```

Användning

```
int a[] {3,5,7,6,8,5,2,4};

int ma4 = min_element(a);
```

Här vet kompilatorn storleken på `a[]` och fyller i mall-parametern `N`

Funktionsmallar
minsta elementet i *nånting man kan iterera över*

Användning

```
int a[] {3,5,7,6,8,5,2,4};
int& ma = min_element(a);

vector<int> v{3,5,7,6,8,5,2,4};
int& mv = min_element(v);

deque<int> d{v.begin(), v.end()};
int& md = min_element(d);

string s{"kontrabasfiol"};
char& ms = min_element(s);
```


Funktionsmallar
`std::min_element` för typer som inte har `<`

Överlagring med en template-parameter: `LESS` (en egenskapsklass)

```
template<class IT, class LESS>
IT min_element(IT first, IT last, LESS cmp)
{
    IT m = first;
    for (IT i = ++first; i != last; ++i)
    if (cmp(*i, *m))
        m = i;
    return m;
}
```

`LESS` måste ha **operator**() och typerna måste stämma, t ex:

```
class Str_Less_Than {
public:
    bool operator () (const char *s1, const char *s2)
    {
        return strcmp(s1, s2) < 0;
    }
};
```

Funktionsmallar
`std::min_element` för typer som inte har <

Exempel på användning med stränglista:

```
list<const char *> tl = { "hej", "du", "glade" };
Str_Less_Than lt; // funktionsobjekt

cout << *min_element(tl.begin(), tl.end(), lt);
```

`Str_Less_Than`-objektet kan skapas direkt i parameterlistan:

```
cout << *min_element(tl.begin(), tl.end(), Str_Less_Than());
```

(C++11) lambda: anonymt funktions-objekt

```
auto cf= [](const char* s, const char* t){return strcmp(s,t)<0;};

cout << *min_element(tl.begin(), tl.end(), cf);
```


Nästa föreläsning

Typomvandlingar. Klasser: Operatorer och arv

Referenser till relaterade avsnitt i Lippman

Typomvandingar 4.11

Operatorer och typomvandling kapitel 14

Dynamisk poymorfism och arv kapitel 15 – 15.2.2

Pekaren `this` s. 257–260

Läsanvisningar

Referenser till relaterade avsnitt i Lippman

Exceptions 5.6, 18.1.1

Funktionsmallar 16.1.1