EDAF30 – Programmering i C++

Felhantering. Strömmar och filer.

Sven Gestegård Robertz
Datavetenskap, LTH

2015

Innehåll

1. Felhantering med exceptions
   - Att generera exceptionella händelser
   - Att fånga exceptionella händelser
   - Specifikation av exceptionella händelser
2. Strömmar och filer

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>`

Att generera exceptionella händelser

Syntax för throw

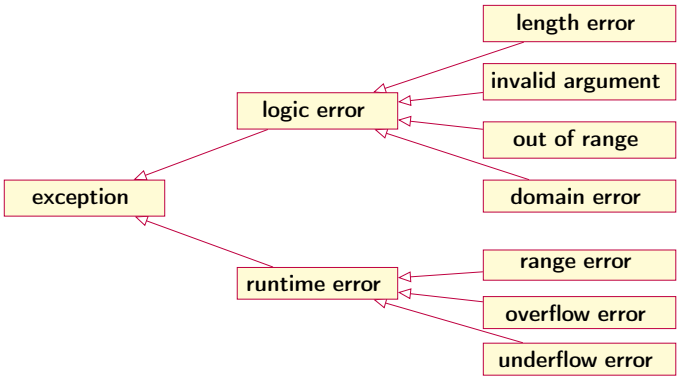
```
throw exceptionnamn("Extra info");
```

Exempel:

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


Att generera exceptionella händelser

Klassträd för klasserna i `<stdexcept>`



Att generera exceptionella händelser

Deklaration av egna undantag som subklasser

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

Användning av egendeklararade undantag

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


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:

```
int i;
try {
        cout << "Nästa tal? ";
        if (!(cin >> i))
                break;
        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:

```
int i;
try {
        cout << "Nästa tal? ";
        if (!(cin >> i))
                break;
        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 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`.

Strömmar och filer

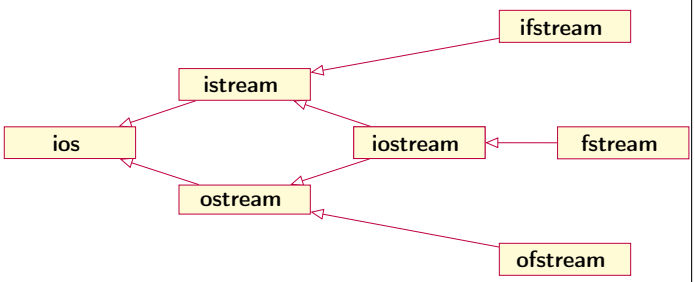
Innehåll

- Klassen `ios`
- Läsning av strömmar
- Utskrift av strömmar
- Koppling av filer till strömmar
- Direktaccess

Klassen `ios`

- Ström (stream) = Följd av tecken (bytes)
- Grundläggande klass för strömmar: `ios`

Klassträd för strömmar



Klassen `ios`

- Fördefinierade strömmar deklarerade i `<iostream>`: `cin`, `cout`, `cerr`, och `clog`
- Klasser deklarerade i inkluderingsfilen: `<fstream>`: `ifstream`, `ofstream`, `fstream`

Läsning av strömmar

Strömmar från klassen `istream` eller från subklass till denna

- Oformaterad inmatning: Läser data från strömmen utan att konvertera till annat format
  – Görs via medlemsfunktioner
- Formaterad inmatning: Data från strömmen görs om till annat format (enligt typen på variablen som ska tilldelas)
  – Görs med operatorn `>>`

Formaterad inmatning

Formaterad inmatning med `>>`

```
// Kodavsnitt
string avr_ort;
int dag, manad;
cout << "Ange avreseort, dag och månad: ";
cin >> avr_ort >> dag >> manad;
cout << "Avresa från " << avr_ort
         << " den " << dag << '/' << manad << endl;

// In- och utmatning
Ange avereseort, dag och månad: Lund 24 10
Avresa från Lund den 24/10
```


Formaterad inmatning

Manipulatorer vid inmatning:

setw(n)	Max-antalet tecken i inläsningssträngen
ws	Hoppa fram till nästa icke-vita tecken
skipws	Hoppa över inl. vita tecken vid anv av `>>`
noskipws	Hoppa ej över inl. vita tecken vid anv av `>>`
dec	Tolka följande heltal som decimalt
oct	Tolka följande heltal som oktalt
hex	Tolka följande heltal som hexadecimalt
boolalpha	Indata för `bool` på formen `false` / `true`
noboolalpha	Indata för `bool` på formen `0` / `1`

Formaterad inmatning

Formaterad inmatning med `>>` och manipulatorer

```
// Kodavsnitt
#include <iomanip>

int i, j, k;
cout "Ange tre heltal: "
cin >> oct >> i >> hex >> j >> k;
cout << i << " " << j << " " << k << endl;

// In- och utmatning
Ange tre heltal:
12 34 56
10 52 56
```


Oformaterad inmatning

Oformaterad inmatning med medlemsfunktioner

`gcount()`	Anger antalet tecken vid senaste inläsn.
`int get()`	Läser in och returnerar nästa tecken
`get(char& c)`	Läser in tecken till `c`
`getline(s, n, t)`	Läser `n` tecken till `s` med `t` som radseparator
`get(s, n, t)`	Som getline men sep. `t` läses ej
`read(char* s, n)`	Läser in `n` st tecken till `s`
`ignore(n, t)`	Hoppar över max `n` st tecken eller till första `t`
`peek()`	Returnerar nästa tecken (som förblir oläst)
`putback(c)`	Lägger tillbaka `c` i strömmen
`unget()`	Lägger tillbaka senast lästa tecken

Klassen ios
Statusflaggor

Flaggor definierade i ios, vilka beskriver en ströms tillstånd:

failbit	Senaste operationen misslyckades
eofbit	Ett filslut påträffades vid senaste operationen
badbit	Ett allvarligare fel av intern art har inträffat

Klassen ios
Statusflaggor

Medlemsfunktioner i klassen ios:

`void clear();`	Slår av alla tillståndsflaggorna
`bool good();`	Ger `true` om alla flaggor `false`
`bool fail();`	Ger `true` om `failbit` el. `badbit` satt
`bool eof();`	Ger `true` om `eofbit` är satt
`bool bad();`	Ger `true` om `badbit` är satt
`bool operator!();`	Ger resultatet `fail()`

cast av en stream s till **bool** ger `!s.fail()`

Klassen ios
Inläsning tecken för tecken

OBS! `istream::eof()` sätts när man *har försökt läsa* `EOF`.

Exempel: räkna tecken (fel)

```
int n=0;
while(!cin.eof()) {
    char c;
    cin.get(c);
    ++n;
}
cout << "read " << n << " chars\n";
```

Användaren skriver `1234` och trycker `<ENTER>`:

```
read 6 chars
```

OBS!

- När `eof` sätts tilldelar inte `get(char& c)` ut-parametern `c`
- men `int get()` returnerar `EOF` (`== -1`)

Utskrift till strömmar

Formaterad utmatning med `<<`

```
cout << uppercase << "hej svejs" << endl;
```

ger utskriften

```
hej svejs
```

```
cout << scientific << 123456789.0 << endl;
cout << uppercase << scientific << 123456789.0 << endl;
```

ger utskriften

```
1.234568e+08
1.234568E+08
```


Utskrift till strömmar

Manipulatorer vid utmatning:

uppercase	Ger stort `E` vid flyttalsutskrift
fixed	Utskrift med fixnotation
scientific	Utskrift med flytnotation
hex	Utskrift som hexadecimalt tal
oct	Utskrift som oktalt tal
dec	Utskrift som decimalt tal
setw(n)	Sätter minimalt antal positioner
setfill(c)	Anger tecken för utfyllnad (padding)
setprecision(n)	Sätter antalet decimaler (om `fixed`) eller antalet sign. siffror (om `scientific`)
setbase(n)	`setbase(16)`, `setbase(8)` osv
boolalpha	`bool` på formen `false` / `true`
flush	Tömmer utskriftsbufferten
endl	Lägger in radslut i strömmen

Utskrift till strömmar

Oformaterad utmatning med medlemsfunktioner

`put(char c)`	Skriv ut tecknet `c` till strömmen
`write(const char* s, streamsize n)`	Skriv ut `n` tecken från `s`
`flush()`	Töm utskriftsbufferten

Koppling av filer till strömmar

Öppnande av fil för läsning

```
ifstream infil("infilen.txt");

// Alt.
ifstream infil;
infil.open("infilen.txt");
```

Öppnande av fil för skrivning

```
ofstream utfil("utfilen.txt");

// Alt.
ofstream utfil;
utfil.open("utfilen.txt");
```


Koppling av filer till strömmar

Stängning av fil

```
infil.close();

utfil.close();
```

Ofta behövs inte `close` användas eftersom destruktorerna för `ifstream` och `ofstream` automatiskt anropas då den associerade strömmen destrueras

Koppling av filer till strömmar

Kopiering av fil

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

main (int argc, char* argv[]) {
        if (argc != 3) {
                cout << "Syntax: " << argv[0]
                        << " from_file to_file" << endl;
        }
        char c;
        ifstream f1(argv[1], ios::binary);//Binärfil
        ofstream f2(argv[2], ios::binary);

        while (f1.get(c)) // Snabbare sätt (via filbuffert):
                f2.put(c);    // f2 << f1.rdbuf();
}
```


Koppling av filer till strömmar

Filflaggor i klassen `ios`

in	Filen skall existera och vara läsbar.
out	Om filen existerar skall den skrivas över.
	Om filen inte finns skall en ny skrivbar fil skapas.
app	Om filen existerar skall skrivning läggas till i slutet.
	Om filen inte finns skall en ny skrivbar fil skapas.
trunc	Om filen redan finns skall den skrivas över
ate	Efter öppning flyttas filpekaren till slutet av filen
binary	Filen skall hanteras som en binärfil

Kombination av flaggor med `operator|` (bitvis eller)

```
ofstream filen("fil.dat",ios::trunc|ios::binary);
```


Direktaccess

Direkt indexering av filposition där index är av typen `streampos` (heltalstyp)

is.tellg()	Ger aktuell position i inströmmen `is`
os.tellp()	Ger aktuell position i utströmmen `os`
is.seekg(pos)	Sätter aktuell position i `is` resp. `os`
os.seekp(pos)	till pos (av typen streampos)
is.seekg(off, dir)	Sätter positionen till `off+dir` där
os.seekp(off, dir)	`off` är en (ev neg.) offset och dir är `ios::beg` (start), `ios::end` (slut) eller `ios::cur` (akt. pos.)

<stringstream> : strängar som strömmar

```
stringstream ss;

ss << "Hello, string!\n";
cout << ss.str();

ss.str("Brave new string");

while(ss) {
    std::string s;
    ss >> s;
    cout << s << endl;
}
```

```
Hello, string!
Brave
new
string
```

Hämta/ändra innehållet i strängen med

- `string stringstream::str() const;`
- `void stringstream::str (const string& s);`

Tips: Använd `stringstream` för att enkelt experimentera med strömmar, t ex direktaccess.

<stringstream> : strängar som strömmar
Exempel: seek

```
void print_pos(stringstream& ss) {
    cout<<"tellg: "<<ss.tellg()<<", tellp: "<<ss.tellp()<< endl;
}

stringstream ss{"1234567890abcdef"};
ss.seekg(5);
cout<<"peek: "<<char(ss.peek())<<endl;   peek: 6
ss.seekg(-1,ios::end);
cout<<"peek: "<<char(ss.peek())<< endl;  peek: f
ss.seekg(ios::beg);
cout<<"peek: "<<char(ss.peek())<< endl;  peek: 1
ss.seekg(2, ios::beg);
cout<<"peek: "<<char(ss.peek())<< endl;  peek: 3
ss.seekg(2, ios::cur);
cout<<"peek: "<<char(ss.peek())<< endl;  peek: 5
print_pos(ss);                           tellg: 4, tellp: 0
ss << "XYZ";   // Skriver över
print_pos(ss);                           tellg: 4, tellp: 3
cout << "str(): " << ss.str() << endl;   str(): XYZ4567890abcdef
ss.seekp(0, ios::end);
ss << "ABC";   // Lägger till
print_pos(ss);                           tellg: 4, tellp: 19
cout << "str(): " << ss.str() << endl;   str(): XYZ4567890abcdefABC
```


Vi har talat om

- Felhantering med **throw** och **catch**
- iostream

Nästa föreläsning:

Vi kommer att introducera standardbibliotekets

- datastrukturer och
- algoritmer