

Undantagshantering, fortsättning

Fel kan detekteras och hanteras på olika sätt, t.ex. med

- Felhanteringskod mer eller mindre inflätad i den ordinarie koden
 - **if**-sats för att testa felvillkor och avgränsa tillhörande felhanteringskod.
 - hopp (**break**, **goto**) för att avbryta och eventuellt hoppa till speciella avsnitt med felhanteringskod
 - svårt att särskilja ordinarie kod och felhanteringskod
- Makrot `assert`
 - testar ett uttryck (**int**-kompatibelt) och avslutar programmet om detta beräknas till 0 (falskt)
 - används t.ex. för att verifiera förvillkor – ”preconditions” – till ett programavsnitt
 - bör främst användas som ett hjälpmedel under utveckling och testning av programvara
- Undantagshantering – ”exception handling”
 - idén är att ha ett systematiskt och enhetligt sätt att hantera fel och andra ”exceptionella händelser”
 - när ett undantag – ”exception” – kastas, avbryts det normala utförandet av ett kodavsnitt
 - undantag fångas in i undantagshanterare – ”catch” – innehåller kod för att hantera undantaget

Makrot assert

```
// Fil: test.cc
#include <cassert>
...
void f(char* s)
{
    assert(s != 0);
    ...
}
```

Om testuttrycket blir **false** avbryts programmet och ett felmeddelande skrivs ut:

```
% a.out
Assertion failed: file "test.cc", line 23
```

Framför allt ett *utvecklingshjälpmedel* – kan slås av då man kompilerar för produktionskörning:

```
g++ -DNDEBUG test.cc
```

Flaggan `-D` fungerar som om man i filen skrivit makrot

```
#define NDEBUG
```

Det finns även en motsvarighet för att testa statiska uttryck: `static_assert (statiskt-uttryck, textsträng) ;`

- om *statiskt-uttryck* blir **false** kommer *textsträng* att ingå i diagnosmeddelande erhålls (kompileringsfel)

Undantag

- Undantag definieras normalt som klasser
 - standardundantagen definieras av en klasshierarki med `exception` som gemensam basclass
- **try**-block
 - en speciell blocksats som används för att avgränsa kod där undantag kan kastas, antingen uttryckligen i blockets satser eller indirekt av funktioner som anropas i blocket.
- **throw**-uttryck
 - avbryter den normala programexekveringen genom att ”kasta” ett objekt

```
throw out_of_range{"otillåtet index"}; // out_of_range är ett standardundantag
```

- objektets typ avgör vad för slags undantag det gäller – i detta fall `out_of_range`.

- **catch**-hanterare

```
catch (const out_of_range& e)  
{  
    cout << e.what() << '\n';  
    throw;  
}
```

- placeras efter ett **try**-block – ett **try**-block kan ha flera hanterare
- fångar undantag som kastats i **try**-blocket, direkt eller indirekt
- vilken typ av undantag som fångas anges i *undantagsspecifikationen* – `const out_of_range&` i exemplet
- en hanterare kan innehålla ett enkelt **throw**-uttryck för att kasta samma undantag vidare

Hantering av standardundantag – ”ellips” – enkelt throw-uttryck

```

try
{
    do
    {
        // Antag att även andra undantag kan kastas i detta try-block ...
        ...
        cout << "Ange en radposition, avsluta med -1: ";
        cin >> pos;
        cout << line.at(pos) << endl;    // at() kan kasta out_of_range
        ...
    }
    while (pos > -1);
}
catch (const out_of_range& e) {           // fångar out_of_range
    cout << e.what() << endl;
}
catch (const exception& e) {           // fångar alla slags exception
    cout << e.what() << endl;
}
catch (...) {                           // ”ellips” – ”catch all”
    cout << "Ett oväntat fel har inträffat\n";
    throw;                               // enkelt throw-uttryck – endast tillåtet i catch
}

```

- Om ett undantag kastas i **try**-blocket kontrolleras, i tur och ordning, om någon av **catch**-hanterarna matchar undantaget.
- Om ingen hanterare passar kastas undantaget vidare till nästa block i den dynamiska anropskedjan

Standardundantag

exception

logic_error

domain_error
invalid_argument
length_error
out_of_range
future_error

runtime_error

range_error
overflow_error
underflow_error
system_error

ios_base::failure

bad_typeid

bad_cast

bad_weak_pointer

bad_exception

bad_function_call

bad_alloc

bad_array_new_length

Kan exempelvis kastas av/vid:

otillåtna funktionsvärden

bitset-konstruktor

objekts längd överskrids

at()

funktioner i trådbiblioteket

vissa beräkningar

bitset::to_long()

vissa beräkningar

systemnära funktioner

ios_base::clear()

typeid

dynamic_cast

konstruktörer `std::shared_ptr`

exception-specifikation

`std::function::operator()`

new

new[]

Standardundantagen

- klassen `exception` är basklass för nästan alla andra standardundantag
- en del direkta subklasser till `exception` används för att kasta undantag
 - `bad_exception`, t.ex.
- `logic_error` representerar sådant som beror på fel i programmets logik och i teorin förebyggbart
 - `domain_error`, `invalid_argument`, `length_error`, `out_of_range`
- `runtime_error` representerar sådant som beror på fel utanför programmets kontroll och inte enkelt kan förutses
 - `range_error`, `overflow_error`, `underflow_error`, `system_error`
- några undantagsklasser är inte härledda från `exception`
 - `nested_exception`, t.ex.
- `exception` tillhör inkluderingen `<exception>` övriga `<stdexcept>`

Basklassen *exception*

```
class exception
{
public:
    exception() noexcept;
    exception(const exception& e) noexcept;

    virtual ~exception();

    exception& operator=(const exception& e) noexcept;

    virtual const char* what() const noexcept;
};
```

- polymorf klass
 - inte abstrakt men bör endast användas som basklass för mer användbara undantagsklasser
 - defaultkonstruktör
 - kopieringskonstruktör
 - virtuell destruktör (kastar per definition inte undantag)
 - kopieringstilldelningsoperator
 - virtuell funktion `what()`
- **noexcept** specificerar att funktionen inte kastar undantag
 - viktigt att undantag inte i sin tur kan kasta nya undantag

En av de direkta subclasserna – logic_error

Standarden anger endast konstruktorerna men i praktiken definieras `logic_error` så här.

```
class logic_error : public std::exception
{
public:
    explicit logic_error(const std::string& what_arg) : msg_(what_arg) {}

    explicit logic_error(const char* what_arg) : msg_(what_arg) {}

    virtual const char* what() const noexcept { return msg_.data(); }

private:
    string msg_;
};
```

- defaultkonstruktor genereras *inte*, eftersom en annan konstruktor har deklarerats
- kopieringstilldelningsoperator genereras
- flyttkonstruktor och flyttilldelningsoperator genereras *inte*, eftersom kopieringskonstruktorn deklarerats
- destruktör kommer att genereras och vara **virtual**
- `what()` överskuggar `what()` deklarerad i `exception`

En av subclasserna till `logic_error` – `length_error`

```
class length_error : public std::logic_error
{
public:
    explicit length_error(const std::string& what_arg) : std::logic_error(what_arg) {}

    explicit length_error(const char* what_arg) : std::logic_error(what_arg) {}
};
```

- defaultkonstruktor genereras *inte*, eftersom en annan konstruktor har deklarerats
- kopieringstilldelningsoperator genereras
- flyttkonstruktor och flyttilldelningsoperator genereras *inte*, eftersom en kopieringskonstruktor deklarerats
- destruktör genereras
- `what()` ärvs som den är från `logic_error`

Egendefinierad undantagsklass

Härled från exempelvis `logic_error`

```
class some_error : public std::logic_error
{
public:
    explicit some_error(const std::string& what_arg) : std::logic_error(what_arg) {}

    explicit some_error(const char* what_arg) : std::logic_error(what_arg) {}
};
```

Dessutom följande genererade eller ärvda funktionalitet:

- kopieringskonstruktor
- kopieringstilldelningsoperator
- destruktör
- `what()`

Funktioner och undantag

En funktion som kastar ett undantag avbryts direkt och undantaget kastas till anroparen. `string`-klassens `at`-funktion kastar `out_of_range` om otillåten position anges:

```
char& string::at(size_type pos)
{
    if (pos > size())
    {
        throw out_of_range{"otillåtet index"};
    }

    return buffer[pos];
}
```

Undantagsspecifikationer

```
char& string::at(size_type pos) noexcept(false);
```

- *noexcept-specifikation*

```
noexcept (konstant-uttryck)
```

```
noexcept    enbart är detsamma som    noexcept ( true )
```

- *dynamisk exception-specifikation är deprecated* i C++11 (använd alltså inte)

```
char& string::at(size_type pos) throw(out_of_range);
```

Några rekommendationer

- använd undantag för att hantera fel och separera ordinarie kod och felhanteringskod
 - använd *inte* undantag om lokala konstruktioner och åtgärder är fullt tillräckliga för att hantera fel
- försök minimera antalet **try**-block
- funktioner ska vanligtvis inte skriva ut felmeddelanden eller avsluta program
 - kasta undantag och låt anroparen/användaren bestämma vad som ska göras
- se upp med minnesläckor för minne som tilldelats med **new** och andra resurser
 - undantagskastning kan innebära att **delete** hoppas över eller att en resurs inte frisläpps
 - *undantags säker programmering* syftar till att hantera sådana problem på bästa sätt
- låt `main()` fånga och rapportera alla slags fel – ”sist utpost”
- förutsätt inte att alla fel är härledda från `exception`
 - använd ellips i lämpliga situationer
- ofta fördelaktigt att härleda egna undantag från en standardundantagsklass
 - enkelt – endast två konstruktörer behöver deklareraras
 - egna undantag får samma egenskaper som standardundantagen
 - egna undantag kan hanteras som om de vore standardundantag
- om en funktion behöver specificera om den kan kasta undantag eller ej, använd *noexcept-specifikation*