

Del 5 – Undantag

Ämnesområden denna föreläsning:

- Undantag (*exceptions*), returvärden
- `throw`, `try` och `catch`
- `new`, `bad_alloc`, `nothrow`

Slide 1

- `throw()` i funktionsdeklaration
- Deklarationer för virtuella funktioner
- Polymorfi hos undantagsklasser
- `try` som funktionskropp (*function try*)
- Undantag i konstruktorns initieringslista
- `set_new_handler`, `set_unexpected`

Varför inte returvärden?

Det är inte alltid tillräckligt med returvärden då man ska returnera ett felvärde från en funktion.

Slide 2

- Vissa returtyper, såsom `int`, saknar speciellt värde som kan symbolisera fel.
- Dock har några returtyper speciella värden, t.ex. `double` har NaN (*not a number*), pekare har NULL.
- Man kan tvingas returnera och hantera felvärden i många steg i anropshierarkin, vilket leder till mycket kod att underhålla.

throw, try och catch

Istället för att hantera fel med returvärden kan man använda undantag (*exceptions*).

```
Slide 3      try
             {
               before();           // before() körs
               throw "error!";     // kasta sträng
               after();            // after() körs ej
             }
             catch(const char *s) // ta emot undantag
             {
               cout << s << endl;
             }
```

Om man inte vet vilken typ av undantag som kastas kan man ta emot alla:

```
Slide 4      try {
               before();           // before() körs
               throw 7;           // kasta heltal
               after();           // after() körs ej
             }
             catch(const char *s) // ta emot sträng
             {
               cout << s << endl;
             }
             catch(...)           // ta emot allt annat
             {
               cout << "unknown exception" << endl;
             }
```

Kasta vidare – `throw`;

Ibland vill man kasta vidare (*rethrow*) det undantag man fått:

```

try { foo(); }           // foo kastar heltal
catch(int i) { cout << i << endl; }
void foo()
{
  try {
    throw 7;           // kasta heltal
  }
  catch(...) {         // ta emot allt
    cout << "unknown exception" << endl;
    throw;           // kasta vidare
  }                  // obs, inget space
}                    // mellan throw och ;

```

Slide 5

Undantag i standardbiblioteket

Standarden definierar en undantagsklass `exception` som är bas till de undantag standardbiblioteket kastar:

```

class std::exception {
public:
  exception() throw();
  exception(const exception &) throw();
  exception &operator=(const exception &) throw();
  virtual ~exception() throw();
  virtual const char* what() const throw();
private:
  ...
};

```

Slide 6

Använd `#include <exception>` för att få tillgång till `exception`.

Exempel på användning av exception:

Slide 7

```

#include <stdexcept>           // out_of_range m.m.
...                           // #inkluderar <exception>
try
{
    throw out_of_range("array index out of bounds");
    throw invalid_argument("array size must be positive");
}
catch(exception &e)
{
    cout << e.what() << endl;    // skriver ut sträng
}

```

new kastar bad_alloc

När new misslyckas med att allokeras minne är default att kasta ett undantag bad_alloc som är nedärvd från exception:

Slide 8

```

#include <new>                  // innehåller bad_alloc
...
try
{
    A *a = new A[-1];           // otillräckligt minne
    A *b = new A[1000000000L]; // otillräckligt minne
    A *c = new (nothrow) A[-1]; // returnerar 0 om fel
}
catch(bad_alloc &e)
{
    cout << e.what() << endl; // ge felmeddelande
}

```

Funktionsdeklarationer med `throw()`

- Man kan specificera de undantag som en funktion kan kasta genom att ange dem med `throw()` i funktionens deklaration.
- Man är då garanterad att inga andra undantag kastas.
- Kollen sker **under körning** och kan kasta `std::unexpected`.

Slide 9

```
int foo() throw() {return 0;}

class A
{
public:
    int bar() throw();
    void baz() throw(bad_alloc);
};
```

Exempel på hur undantag bör hanteras för att följa specifikationen:

Slide 10

```
int foo() throw(bad_alloc)    // specificera undantag
{
    try {
        ...
    }
    catch(bad_alloc &) {      // ta emot minnesfel
        throw;               // kasta vidare
    }
    catch(...) {
        cout << "unknown exception" << endl;
    }
    // kastar inget undantag
}
```

Deklarationer för virtuella funktioner

Virtuella, nedärvda funktioner är begränsade till att kasta undantagen i basklassens funktion.

Slide 11

```
class A {
    virtual void f() throw(X, Y);
    virtual void g() throw(X);
    virtual void h() throw();
};

class B : public A {
    virtual void f() throw();           // ok: mer restriktiv
    virtual void g() throw(X);         // ok: lika restriktiv
    virtual void h() throw(X, Y);      // fel: mer tillåtande
};
```

Objekt eller referens till `catch`?

För att använda polymorfi måste man ha en pekare eller referens.

Om man anger ett objekt i `catch` kan man (som vanligt) inte använda polymorfi.

Slide 12

```
try { throw bad_alloc(); }
catch(exception e) {
    cout << e.what() << endl;    // skriver 'exception'
}

try { throw bad_alloc(); }
catch(exception &e) {
    cout << e.what() << endl;    // skriver 'bad_alloc'
}
```

try runt funktionskropp

Man kan låta en hel funktionskropp inneslutas i try:

Slide 13

```

void foo()                                // foo är en funktion
try
{
    for(int j = 0; ; j++)                // funktionskropp
        int *i = new int[100000];       // oändlig loop
                                        // som läcker minne
}
catch(exception &e) {                    // tar emot undantag
    cout << e.what() << endl;           // kastade i kroppen
}
catch(...) {
    cout << "unknown exception" << endl;
}

```

Konstruktorns initialiseringslista

Man kan med try runt konstruktorns funktionskropp hantera undantag som kastats i initieringslistan:

Slide 14

```

int foo() {throw 7; return 5;}
...
A::A() try : i(foo())                    // foo kastar int
{
    /* konstruktorns kropp */
}
catch(int) {
    cout << "int exception" << endl;
}
catch(...) {
    cout << "unknown exception" << endl;
}

```

Hanterare av undantag

Vissa vanligt förekommande undantag kan hanteras med speciella hanteringsrutiner:

```
void my_new_handler() { cout << "out of mem"; exit(1); }  
void my_unexpected_handler() {}
```

Slide 15

```
void (*old)(); // standardhanterare sparas  
old = set_new_handler(my_new_handler);  
new int[-1];  
set_new_handler(old); // sätt tillbaka gammal  
  
old = set_unexpected(my_unexpected_handler);  
...  
set_unexpected(old); // sätt tillbaka gammal
```