EDAF30 – Programmering i C++

7. Resurshantering, Felhantering

Sven Gestegård Robertz
Datavetenskap, LTH

2016

Innehåll

Resurshantering

En *resurs* är
- något som måste *allokeras*
- och senare *lämnas tillbaka*

Exempel:
- minne
- filer (*file handles*)
- sockets
- lås
- ...

Resource handles

Organisera resurshantering med klasser som *äger* resursen
- allokerar resurser i konstruktorn
- lämnar tillbaks resurser i destruktorn
- *RAII*

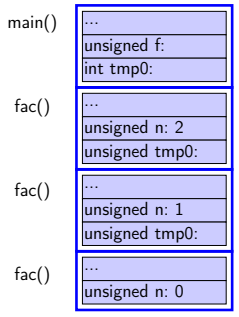
Minnesallokering

Två sorters minnesallokering:
- på *stacken* - *automatiska* variabler. Förstörs när programmet lämnar det *block* där de deklarerats.
- på *heapen* - *dynamiskt allokerade* variabler. Överlever tills de explicit avallokeras.

Minnesallokering
Exempel: allokering på *stacken*

```
unsigned fac(unsigned n)
{
  if(n == 0)
    return 1;
  else return n * fac(n-1);
}

int main()
{
  unsigned f = fac(2);
  cout << f;
  return 0;
}
```



Minnesallokering
Dynamiskt minne, allokering "på *heapen*", eller "i *free store*"

Utrymme för dynamiska variabler allokeras med `new`

```
double* pd = new double;       // allokera en double
*pd = 3.141592654;         // tilldela ett värde
float* px;
float* py;
px = new float[20];        // allokera array
py = new float[20] {1.1, 2.2, 3.3};   // allokera och initiera
```

Minne frigörs med `delete`

```
delete pd;
delete[] px; // [] krävs för C-array
delete[] py;
```


Minnesallokering
Varning! se upp med parenteser

Allokering av `char[80]`

```
char* c = new char[80];
```

Nästan samma...

```
char* c = new char(80);
```

Nästan samma...

```
char* c = new char{80};
```

De två senare allokerar *en byte*

och *initierar* den med värdet 80 ('P').

```
char* c = new char('P');
```


Minnesallokering
ägarskap för resurser

För dynamiskt allokerade objekt är *ägarskap* viktigt

- Ett objekt eller en funktion kan *äga* en resurs
- *Ägaren* är ansvarig för att avallokera resursen
- Om du har en pekare måste du veta *vem som äger objektet den pekar på*
- Ägarskap kan *överföras* vid funktionsanrop
  - men måste inte
  - var tydlig

Varje gång du skriver **new** är du ansvarig för att någon kommer att göra **delete** *när objektet inte ska användas mer*.

Klasser
RAII

- *RAII Resource Acquisition Is Initialization*
- Ett objekt initieras av en *konstruktor*
  - Allokerar de resurser som behövs (*"resource handle"*)
- När ett objekt tas bort körs dess *destruktor*
  - Frigör de resurser som ägs av objektet
  - Exempel: `Vector`: ta bort arrayen som `elem` pekar på

```
class Vector{
private:
  double elem*;  // en array som innehåller själva vektorn
  int sz;       // storleken
public:
  Vector(int s) :elem{new double[s]}, sz{s} {}  // konstruktor
  ~Vector() {delete[] elem;}   // destruktor, ta bort arrayen
};
```

Manuell minneshantering

- Objekt allokerade med **new** måste tas bort med **delete**
- Objekt allokerade med **new[]** måste tas bort med **delete[]**
- annars fås en *minnesläcka*

Minnesallokering

Typiskt misstag: Att glömma allokera minne

```
char namn[80];

*namn = 'Z'; // Ok, namn allokerad på stacken. Ger namn[0]='Z'



char *p;     // Oinitierad pekare
             // Ingen varning vid kompilering

*p = 'Z';    // Fel! 'Z' skrivs till  en slumpvis vald adress

cin.getline(p, 80); // Ger (nästan) garanterat exekveringsfel
                    // ("Segmentation fault") eller
                    // minneskorruption
```


Minnesallokering

Exempel: misslyckad `read_line`

```
char* read_line() {
   char temp[80];
   cin.getline(temp, 80);
   return temp;
}

void exempel () {
  cout << "Ange ditt namn: ";
  char* namn = read_line();

  cout << "Ange din bostadsort: ";
  char* ort = read_line();

  cout << "Goddag " << namn << " från " << ort << endl;
}
```

"Dangling pointer": pekare till objekt som inte längre finns

Minnesallokering

Delvis korrigerad version av `read_line`

```
char* read_line() {
   char temp[80];
   cin.getline(temp, 80);
   size_t len=strnlen(temp,80);
   char *res = new char[len+1];
   strncpy(res, temp, len+1);
   return res; // Dynamiskt allokerat överlever
}
void exempel () {
   cout << "Ange ditt namn: ";
   char* namn = read_line();
   cout << "Ange din bostadsort: ";
   char* ort = read_line();
   cout << "Goddag " << namn << " från " << ort << endl;
}

Fungerar, men minnesläcka !
```


Minnesallokering

Ytterligare korrigerad version av `read_line`

```
char* read_line() {
   char temp[80];
   cin.getline(temp, 80);
   size_t len=strnlen(temp,80);
   char *res = new char[len+1];
   strncpy(res, temp, len+1);
   return res;  Dynamiskt allokerat objekt överlever
}
void exempel () {
   cout << "Ange ditt namn: ";
   char* namn = read_line();  NB! Anropande funktion blir ägare
   cout << "Ange din bostadsort: ";
   char* ort = read_line();
   cout << "Goddag " << namn << " från " << ort << endl;

   delete[] namn;      Avallokera strängarna
   delete[] ort;
```


Använd `std::string`

Enklare och säkrare med `std::string`

```
#include <iostream>
#include <string>

using std::cin;
using std::cout;
using std::string;

string read_line()
{
  string res;
  getline(cin, res);
  return res;
}

void exempel()
{
    cout << "Namn:";
    string namn = read_line();
    cout << "Bostadsort:";
    string ort = read_line();

    cout << "Goddag, " << namn <<
    " fraan " << ort << endl;
}
```

- `std::string` är en *resource handle*
- *RAII*
- Dynamiskt minne behövs sällan

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

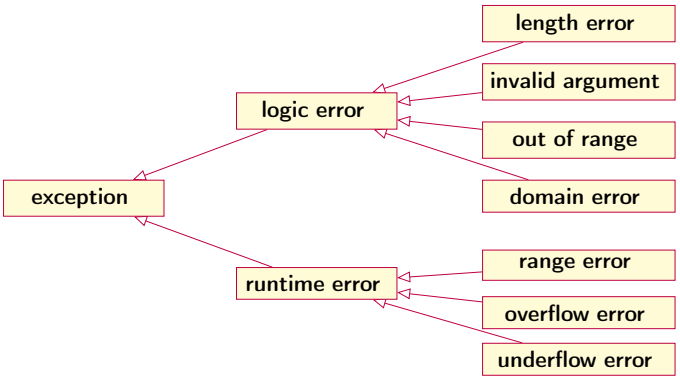
```
throw exceptionnamn("Extra info");
```

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 egendeklarerade 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`.

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
```

Dynamiskt minne, exempel
Felhantering

```
void f(int i, int j)
{
    X* p=new X;             // allocate new X
    //...
    if(i<99) throw E{};     // may throw an exception
    if(j<77) return;        // may return "early"
    //
    p->do_something();      // may throw
    //
    delete p;
}
```

Läcker minne om inte **delete** p anropas

Minnesallokering
C++: Smarta pekare

I standardbiblioteket <memory> finns två "smarta" pekare (C++11):

- std::unique_ptr<T> – *ensamt ägarskap*
- std::shared_ptr<T> – *delat ägarskap*

som är *"resource handles"*:

- deras destruktor avallokerar objektet de pekar på.
- Andra exempel på *"resource handles"*:
  - std::vector<T>
  - std::string

shared_ptr innehåller en *referensräknare*: när *sista* shared_ptr till ett objekt förstörs avallokeras objektet. Jfr. *garbage collection* i Java.

Smart pointer, exempel

```
void f(int i, int j)
{
    unique_ptr<X> p{new X};// allocate new X and give to unique_ptr
    //...
    if(i<99) throw E{};     // may throw an exception
    if(j<77) return;        // may return "early"
    //
    p->do_something();      // may throw
    //
    delete p;
}
```

Destruktorn för p körs alltid

Smart pointer, exempel
Dynamiskt minne behövs sällan

```
void f(int i, int j)
{
  X x1{};
  X x2{};

  if(i<99) throw E{};       // may throw an exception
  if(j<77) return;          // may return "early"

  x1.do_something();        // may throw
  x2.do_something();        // may throw
}
```

Använd lokala variabler om möjligt

read_line med unique_ptr

```
unique_ptr<char[]> read_line()
{
  char temp[80];
  cin.getline(temp, 80);
  int size = strlen(temp)+1;
  char* res = new char[size];
  strncpy(res, temp, size);
  return unique_ptr<char[]>{res};
}

void exempel()
{
  cout << "Ange ditt namn: ";
  unique_ptr<char[]> namn = read_line();
  cout << "Ange din bostadsort: ";
  unique_ptr<char[]> ort = read_line();
  cout << "Goddag " << namn.get() << " fraan " << ort.get() << endl;
}
```

- För att få en **char*** används unique_ptr<**char**[]>::get().

read_line med unique_ptr
helt utan explicita **new** och **delete** (c++14)

```
unique_ptr<char[]> read_line()
{
    char temp[80];
    cin.getline(temp, 80);
    int size = strlen(temp)+1;
    auto res = std::make_unique<char[]> (size);
    strncpy(res.get(), temp, size);
    return res;
}
```

Smarta pekare
Vector från tidigare

```
class Vector{
public:
    Vector(int s) :elem{new double[s]}, sz{s} {}
    double& operator[](int i) {return elem[i];}
    int size() {return sz;}
private:
    std::unique_ptr<double[]> elem;
    int sz;
};
```


Minnesallokering
C++: Smarta pekare

Tumregler för parametrar till funktioner:

om ägarskap inte ska överföras

- Använd "råa" pekare
- Använd `std::unique_ptr<T> const &`

om ägarskap ska överföras

.

- Använd *värdeanropad* `std::unique_pointer<T>`
  (då måste man använda `std::move()`)

- Detta är bara en orientering om att smarta pekare finns.
- "Råa" pekare är så vanliga att ni måste lära er behärska dem.

C++: Smarta pekare
Grov sammanfattning

"Råa" ("nakna") pekare:

- Programmeraren ansvarar för allt
- Risk för minnesläckor
- Risk för *dangling pointers*

Smarta pekare:

- Ingen (mindre) risk för minnesläckor
- (liten) Risk för *dangling pointers* om de används fel
  (t ex mer än en `unique_ptr`)

Läsanvisningar

Referenser till relaterade avsnitt i Lippman

Arrayer 3.5, 12.2.1, (12.2.2)
Exceptions 5.6, 18.1.1
Dynamiskt minne och smarta pekare 12.1
`unique_ptr` 12.1.5 (s. 470–471)

Nästa föreläsning
Klasser: resurshantering och polymorfism

Referenser till relaterade avsnitt i Lippman

Klasser, resurshantering 13.1, 13.2
Move semantics 13.6
Operatorer och typomvandling kapitel 14
Dynamisk poymorfism och arv kapitel 15 – 15.4
Arv och resurshantering 15.7
Pekaren `this` s. 257–260