EDAF30 – Programmering i C++

6. Resurshantering

Sven Gestegård Robertz
Datavetenskap, LTH

2017

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* Användardefinierade typer som uppför sig som inbyggda

Minnesallokering

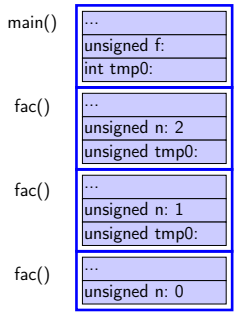
Två sorters minnesallokering:

- 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*"

- Dynamiskt allokerat minne
  - allokeras på *heapen*, med **new** (som i Java)
  - tillhör inget *scope*
  - finns kvar tills det avallokeras med **delete** (skillnad mot Java)

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

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 odefinierad 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

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
};
```

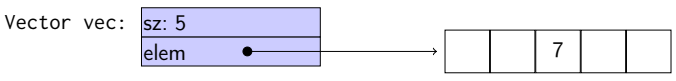
Manuell minneshantering

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

Klasser
Resurshantering, representation

```
struct Vector {
  Vector(int s) :sz{s},elem{new double(sz)} {}
  ~Vector() {delete[] elem;}
  double& operator[](int i) {return elem[i];}
  int sz;
  double* elem;
};

void test()
{
   Vector vec(5);
   vec[2] = 7;
}
```



- *Resource handle* – Vector äger sin **double[]**
- objektet: pekare + storlek, arrayen ligger på heapen

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
}
```

Destruktorn för p körs alltid

Smart pointer, exempel
Dynamiskt minne behövs sällan

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

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

  x.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`)

"Rule of three"
Canonical construction idiom

Om en klass äger någon resurs, ska den ha en egen

1. Destruktor
2. Copy constructor
3. Copy assignment operator

för att inte läcka minne. T ex. klassen `Vektor`

Regel: Om du definerar *någon*, ska du definiera *alla*.

Varnande exempel
Default-kopiering

För klasser som äger resurser fungerar inte default-kopiering.

- värdeanrop
- kopiering
- destruktorn körs vid **return**
- *dangling pointer*

Exempel: `Vector`

Move semantics

- Onödigt att kopiera om man vet att källan ska förstöras direkt
- Bättre att "stjäla" innehållet
- Gör *resource handles* ännu effektivare
- Vissa objekt får/kan inte kopieras
  - `std::move` gör om till en *rvalue-referens* (`T&&`)

Move semantics (C++11)
Exempel: Vektor

Copy-konstruktor

```
Vector::Vector(const Vector& v) :elem{new double[v.sz]}, sz{v.sz}
{
    for(int i=0; i < sz; ++i) {
        elem[i] = v[i];
    }
}
```

Move-konstruktor

```
Vector::Vector(Vector&& v) : elem{v.elem}, sz{v.sz}
{
    v.elem = nullptr;
    v.sz = 0;          // v har inga element
}
```

Move semantics (C++11)
Exempel: Vektor

Copy-assignment

```
Vector& Vector::operator=(const Vector& v) {
   if(this != &v) {
       double* tmp = new double[v.sz];
       for(int i = 0; i < v.sz; ++i) {
           tmp[i] = v[i];
       }
       delete[] elem;
       elem = tmp;
       sz = v.sz;
   }
   return *this;
}
```

Move-assignment

```
Vector& Vector::operator=(Vector&& v) {
    if(this != &v) {
        elem = v.elem;      // "flytta" arrayen från v
        v.elem = nullptr;   // markera att v är ett "tomt skrov"
        sz = v.sz;
        v.sz = 0;
    }
    return *this;
}
```


"Rule of ~~three~~ five"
Canonical construction idiom, fr o m C++11

Om en klass äger någon resurs, ska den ha en egen

1. Destruktor
2. Copy constructor
3. Copy assignment operator
4. *Move* constructor
5. *Move* assignment operator

Typer
Två sorters konstanter

- En variabel deklarerad **const** får inte ändras (`final` i Java)
  - Betyder ungefär "Jag lovar att inte ändra denna variabel."
  - Kontrolleras av kompilatorn
  - Används ofta för att specificera gränssnitt
    - En funktion som inte ändrar värdet på argument
    - En medlemsfunktion ("metod") som inte ändrar objektets tillstånd.
  - Viktigt vid (medlems-)funktionsöverlagring
- En variabel deklarerad **constexpr** måste ha ett värde som kan beräknas vid kompilering.
  - Används för att specificera konstanter
  - Infördes i C++-11

Medlemsfunktioner kan vara **const**

- Betyder att de inte ändrar objektets tillstånd
- Viktigt vid överlagring
  - `T func(int)` och `T func(int) const` är olika funktioner

exempel:

```
class Container{
public:
  double& operator[]();
  double  operator[]() const;
};
```


Funktioner kan vara **constexpr**

- Betyder att de kan beräknas vid kompilering om argumenten är **constexpr**

exempel:

```
constexpr int square(int x)
{
    return x*x;
}

void test_constexpr_fn()
{
    char matrix[square(4)];

    cout << "sizeof(matrix) = " << sizeof(matrix) << endl;
}
```

Utan **constexpr** får man felet

```
error: variable length arrays are a C99 feature
```


const och pekare

const modifierar allt som står till vänster om (undantag: om **const** står först modifieras det som kommer omedelbart efter)

Exempel

```
      int* ptr;
const int* ptrToConst;  //NB! (const int) *
int const* ptrToConst,  // ekvivalent, tydligare?

int* const  constPtr;   // pekaren är konstant

const int* const constPtrToConst; // Både pekare och värde
int const* const constPtrToConst; // ekvivalent, tydligare?
```

Var noggrann när du läser:

```
char *strcpy(char *dest, const char *src);
```

`(const char)*`, inte `const (char*)`

const och pekare
Exempel

```
void Exempel( int* ptr,
              int const * ptrToConst,
              int* const constPtr,
              int const * const constPtrToConst )
{
    *ptr = 0;                   // OK: ändrar innehållet
    ptr  = nullptr;             // OK: ändrar pekaren

    *ptrToConst = 0;            // Fel! kan inte ändra innehållet
    ptrToConst  = nullptr;      // OK: ändrar pekaren

    *constPtr = 0;              // OK: ändrar innehållet
    constPtr  = nullptr;        // Fel! kan inte ändra pekaren

    *constPtrToConst = 0;       // Fel! kan inte ändra innehållet
    constPtrToConst  = nullptr; // Fel! kan inte ändra pekaren
}
```


Pekare

Pekare på konstanter och konstant pekare

```
int k;            // ändringsbar int
int const c = 100;// konstant int
int const * pc;   // pekare till konstant int
int *pi;          // pekare till ändringsbar int

pc = &c;   // OK
pc = &k;   // OK, men k kan inte ändras via *pc
pi = &c;   // FEL! pi får ej peka på en konstant
*pc = 0;   // FEL! pc pekar på en konstant

int* const cp = &k; // Konstant pekare
cp = nullptr;       // FEL! Pekaren ej flyttbar
*cp = 123;          // OK! Ändrar k till 123
```


char[], char* och const char*
const är viktigt för C-strängar

En *string literal* (t ex `"I am a string literal"`) är **const**.

- Kan lagras i icke-skrivbart minne
- **char*** `str1 = "Hello";` — *deprecated* i C++ – ger varning
- **const char*** `str2 = "Hello";` — OK, strängen är **const**
- **char** `str3[] = "Hello";` — `str3` går att ändra

Nästa föreläsning

Felhantering. Funktionsmallar

Referenser till relaterade avsnitt i Lippman

Exceptions 5.6, 18.1.1

Funktionsmallar 16.1.1

Läsanvisningar

Referenser till relaterade avsnitt i Lippman

Dynamiskt minne och smarta pekare 12.1

Dynamiskt allokerade arrayer 12.2.1

Klasser, resurshantering 13.1, 13.2

const och constexpr 2.4