EDAF30 – Programmering i C++

Avslutning. Sammanfattning och frågor

Sven Gestegård Robertz
Datavetenskap, LTH

2016

Innehåll

Pekare
Syntax

```
struct Foo {
  int x;
}
```

```
Foo* ptr;       // Deklaration av en pekare till Foo
Foo object;     // Deklaration av en foo-instans
ptr = &object;  // ptr tilldelas adressen till object
                // "ptr pekar på object"
```

Två sätt att komma åt medlemmen x i object, genom pekaren:

```
int x1 = (*ptr).x;

int x2 = ptr->x;
```

ptr *är en pekare,* *ptr *är objektet som* ptr *pekar på*

Pekare
Exempel

```
struct Foo {
    int x;
};

void assign17(Foo* f)
{
    f->x = 17;
}

void test()
{
    Foo test;
    test.x = 10;

    assign17( &test );
}
```


Most vexing parse
Exempel 1

```
struct Foo {
    int x;
};

int main()
{
#ifdef ERROR1
    Foo f(); // funktionsdeklaration
#else
    Foo f{}; // variabeldeklaration C++11
    // Foo f; //C++98 (men inte initierat)
#endif
    cout << f.x << endl; // Fel

    Foo g = Foo(); // OK     // C++11: auto g = Foo();
    cout << g.x << endl;
```

```
error: request for member 'x' in 'f', which is of
non-class type 'Foo()
```


Most vexing parse
Exempel 2

```
struct Foo {
    int x;
};

struct Bar {
    int x;
    Bar(Foo f) :x{f.x} {}
};

int main()
{
#ifdef ERROR2
    Bar b(Foo()); // funktionsdeklaration
#else
    Bar b{Foo()}; // variabeldeklaration (C++11)
    // Bar b((Foo())); // C++98 : extra parenteser --> uttryck
#endif
    cout << b.x << endl; // FEL!
```

```
error: request for member 'x' in 'b', which is of
non-class type 'Bar(Foo (*)())'
```

Most vexing parse
Exempel: faktisk funktion

```
struct Foo {
    Foo(int i=0) :x{i} {}
    int x;
};

struct Bar {
    int x;
    Bar(Foo f) :x{f.x} {}
};

Bar b(Foo()); // forward declaration

Foo make_foo()
{
   return Foo(17);
}

Bar b(Foo(*f)())
{
    return Bar(f());
}
```

```
void test()
{
    Bar tmp = b(make_foo);
    cout <<  tmp.x << endl;
}
```

17

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

~~Två~~ Tre sorters minnesallokering:

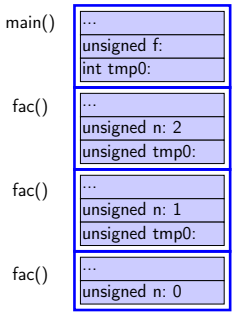
- på *stacken* - *automatiska* variabler. Förstörs när programmet lämnar det *block* där de deklarerats. *Initieras inte automatiskt*
- på *heapen* - *dynamiskt allokerade* variabler. Överlever tills de explicit avallokeras. *Initieras inte automatiskt*
- *static storage duration* - hela programmets exekvering.
  - lokala variabler (deklarerade **static** i en funktion) .
  - bara synliga inom filen (**static** eller **const** i file scope): *internal linkage*
  - globala variabler (deklarerade i file scope) t ex
    - `int x;`
    - `extern const int fortytwo = 42;`

    external linkage
  - *Initieras till "noll"* om ingen explicit initiering görs

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
ägarskap för resurser

För dynamiskt allokerade objekt är *ägarskap* viktigt

- Ett objekt eller en funktion kan *äga* ett objekt
- *Ägaren* är ansvarig för att avallokera objektet
- 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.*

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);
   char *res = new char[strlen(temp)+1];
   strcpy(res, temp);
   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);
   char *res = new char[strlen(temp)+1];
   strcpy(res, temp);
   return res; // Dynamiskt allokerat överlever
}
void exempel () {
   cout << "Ange ditt namn: ";
   char* namn = read_line();  // Ta över ägarskap av sträng
   cout << "Ange din bostadsort: ";
   char* ort = read_line();
   cout << "Goddag " << namn << " från " << ort << endl;

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


Minneshantering
Annat vanligt fel

```
struct Vektor{
    Vektor(size_t sz) :size{sz},p{new int[sz]} {}
    ~Vektor() {delete[] p;}

    size_t size;
    int* p;
};

void print(Vektor v)
{
    for(size_t i=0; i != v.size; ++i){
        cout << v.p[i] << " ";
    }
    cout << endl;
}

void fill(Vektor v, int val)
{
    for(size_t i=0; i != v.size; ++i){
        v.p[i] = val;
    }
}
```

```
void test1()
{
    Vektor v(10);

    fill(v,7);
    print(v);

    fill(v,17);
    print(v);
}
```

Vad skrivs ut?
0 0 7 7 7 7 7 7 7 7

Minneshantering
Annat vanligt fel

```
struct Vektor{
    Vektor(size_t sz) :size{sz},p{new int[sz]} {}
    ~Vektor() {delete[] p;}

    size_t size;
    int* p;
};

void print(Vektor v)
{
    for(size_t i=0; i != v.size; ++i){
        cout << v.p[i] << " ";
    }
    cout << endl;
}

void fill(Vektor v, int val)
{
    for(size_t i=0; i != v.size; ++i){
        v.p[i] = val;
    }
}
```

```
void test1()
{
    Vektor v(10);

    fill(v,7);
    print(v);

    fill(v,17);
    print(v);
}
```

0 0 7 7 7 7 7 7 7 7

*** Error: double free or corruption (fasttop): 0x0000000002138010 ***

Minneshantering
Annat vanligt fel: värdeanrop och brott mot *rule of three*

Problem:

```
void fill(Vektor v, int val)
{
    for(size_t i=0; i != v.size; ++i){
        v.p[i] = val;
    }
}

void print(Vektor v)
{
    for(size_t i=0; i != v.size; ++i){
        cout << v.p[i] << " ";
    }
    cout << endl;
}
```

- `fill` och `print` *värdeanropas*
- värdet av *pekaren* `Vektor::p` kopieras
- destruktorn körs *efter första anropet av* `fill` ⇒ **`delete`**`[] p`
- `v.p` (i `test1()`) blir *dangling pointer*

Lösning?

Minneshantering
Annat vanligt fel: värdeanrop och brott mot *rule of three*

Problem:

- `fill` och `print` *värdeanropas*
- värdet av *pekaren* `Vektor::p` kopieras
- destruktorn körs *efter första anropet av* `fill` ⇒ **`delete`**`[] p`
- `v.p` (i `test1()`) blir *dangling pointer*

Lösningar:

- ändra till referensanrop:**`void`** `fill(Vektor&,` **`int`**`)`, **`void`** `print(`**`const`** `Vektor&,` **`int`**`)`
  - Fungerar för både `print` och `fill`
- Rule of three: implementera copy-constructor och **`operator`**`=`.
  - Undviker dubbel delete och dangling pointer
  - *Både* värde- och referensanrop fungerar i `print`
  - `fill` måste ha en utparameter ⇒ referensanrop

Det säkra är att göra båda.

Råd

resurshantering

- resurshantering: RAII och *rule of three (five)*
- undvik "nakna" **`new`** och **`delete`**
- Använd konstruktorer till att säkerställa *invarianter*
  - kasta exception om det misslyckas

för polymorfa klasser

- Kopiering leder ofta till katastrof.
- =**`delete`**
  - Copy/Move-constructor
  - Copy/Move-assignment
- Om kopiering behövs, implementera en virtuell `clone()`

Råd

klasser

- skapa bara medlemsfunktioner för sådant som behöver tillgång till *representationen*
- som default, gör konstruktorer med en parameter **`explicit`**
- gör bara funktioner **`virtual`** om du vill ha polymorfism

polymorfa klasser

- åtkomst genom referens eller pekare
- En klass som har virtuella funktioner ska ha en virtuell destruktor
- använd **`dynamic_cast`** om du behöver navigera klasshierarkin
- använd `override` för tydlighet och för att få hjälp av kompilatorn att hitta fel

Råd

parametrar och returvärden, "reasonable defaults"

- *return by value* om inte *mycket dyrt* att kopiera
- använd referens-anrop om inte *mycket billigt* att kopiera
  - in-parametrar: **`const`** `T&`
  - in/ut- eller ut-parametrar: `T&`

Råd

säkrare kod

- initiera alla variabler
- använd exceptions istället för felkoder
- använd *named casts* (om du måste typomvandla)
- använd bara **`union`** som implementationsteknik inuti klasser
- undvik pekararitmetik, utom
  - för trivial array-traversering (t ex p++)
  - i väldigt specialiserad kod (t ex implementation av minneshanterare)

Råd

standardbiblioteket

- använd standard-biblioteket om möjligt
  - standard-containers
  - standard-algoritmer
- använd hellre `std::string` än C-strängar (**char**`[]`)
- använd hellre containers (t ex `std::vector<T>`) än arrayer (`T[]`)

Ofta både

- säkrare och
- effektivare

än egenskriven kod

Råd

standardbibliotekets container-klasser

- använd `std::vector` som default
- använd `std::forward_list` för sekvenser som ofta är tomma
- var uppmärksam på när iteratorer blir ogiltiga
- använd `at()` istället för `[]` för att få index-kontroll
- använd *range for* för enkla traverseringar
- initiering: använd `()` för storlekar och `{}` för listor av element

Skriv enkel kod som är korrekt och går att förstå

Lycka till på tentan

Frågor?