

Del 7 – Effektivitet

Ämnesområden denna föreläsning:

- Kompilering av översättningsenheter (*translation units*)
 - Inkludering av headerfiler och framåtdeklaration
 - När behövs klassdefinitionen?

Slide 1

- `inline`
 - När bör man/bör man inte använda `inline`
 - Optimering med `inline`
- Temporära objekt
- Minneshantering med minnespool – specialiserad `new` och `delete`

Effektivitet

Innehållet i denna föreläsning är inspirerat av

- Mayhew och Bulka, “*Efficient C++: performance programming techniques*” [MB]

Slide 2

• Boken jämför hastigheten hos olika konstruktioner i C++.
samt

- Scott Meyers, “*Effective C++*” [SM]
- Boken nämner 50 specifika tips om hur man bör skriva C++-program.

Kompilering och länkning

Slide 3

- En **källkodsfil** (med suffix `.cpp`) som är körd genom preprocessorn kallas en **översättningsenhet** (*translation unit*).
- I C++ kompileras källkodsfilerna separat till **objektfiler** (med suffix `.o`).
- Objektfilerna **länkas** sedan till ett körbart program.
- Kompilatorn kan inte veta om anrop sker till funktion som inte är definierad.
- Länkaren har översikt över alla definierade funktioner och varnar om anrop sker till funktion som ej är definierad.

Beroenden

För att minska kompileringstiderna vill man ha så få beroenden som möjligt mellan filer:

Slide 4

- Undvik att inkludera onödiga filer i dina headerfiler.
- Undvik att använda `inline` på funktioner som använder andra objekt eftersom deras definition då måste vara synlig.
- Använd framåtdeklaration där detta är möjligt.

Framåtdeklaration

När kan man använda framåtdeklaration och när måste definitionen vara synlig?

- Definitionen måste vara **synlig** för alla **basklasser**.
 - Definitionen måste vara **synlig** då man **definierar objekt**, t.ex. som medlem.
- Slide 5**
- Definitionen måste vara **synlig** då man **använder ett objekt**, t.ex. tilldelar en av objektets medlemmar.
 - Definitionen **behöver ej vara synlig** då man deklarerar/definierar **pekare eller referens till objekt**, t.ex. som medlem eller returtyp från medlemsfunktion.
 - Definitionen **behöver ej vara synlig** då man använder objekt som **returtyp för funktion** – definitionen för objektet kan då inkluderas endast där funktionen används.

Exempel på framåtdeklaration och inklusion av headerfiler:

```
// i filen A.h
#include "D.h"      // inkludera D:s definition

class B;           // (framåt)deklaration av B
class C;           // (framåt)deklaration av C
class A            // definition av klassen A
{
    B foo();        // B behöver ej vara definierad!
    C *bar();       // C behöver ej vara definierad
    B &b;           // B behöver ej vara definierad
    D *pc;          // D behöver ej vara definierad
    D c;            // ok: D är definierad
    B b;            // fel: B måste vara definierad
};
```

Slide 6

Slide 7

```

// i filen A.cpp
#include "A.h"           // A måste vara definierad
#include "B.h"           // B måste definieras
                        // eftersom foo() returnerar B
extern C *get_Cp();     // extern funktion, dvs
                        // definierad i annan fil

B A::foo()
{
    return B();         // B måste vara definierad
}

C *A::bar()
{
    C *p = get_Cp();    // returnerar pekare till C
    return p;           // C behöver ej vara definierad
}                       // om vi inte använder C

```

inline

- **inline** är ett **förslag** till kompilatorn att ersätta funktionsanrop med den kod som anropas.
- **inline** är till stor hjälp när man vill skriva effektiva program.
- Utan **inline** kan man använda preprocessorn för att uppnå samma effekt. Man får då inte samma typkontroll.

Slide 8

- Följande funktioner kan inte göras **inline** med dagens kompilatorer
 - Funktioner som är rekursiva
 - Funktioner med statiska variabler (gäller vissa kompilatorer)
 - Komplicerade funktioner med många **if** eller loopar.
- Funktionspekare till en funktion som deklarerats **inline** skapar en funktion i minnet som man kan peka till.

`inline` eller inte?

När bör man undvika att använda `inline`?

Slide 9

- Använd `inline` där du lokaliserat en flaskhals.
- Överdrivet användande ger större exekverbara program pga expansion av koden (*code bloat*).
- `inline` kan öka kompileringstiderna eftersom programkoden ligger i headerfilerna.
 - Ändringar ger omkompilering pga beroenden.
 - Använda objekt måste finnas definierade.
- `inline` kan ge långsammare kod pga sidfel (*page faults*).

Exempel på snabbare program med `inline`: Optimering mellan funktionsanrop är svårt, med `inline` blir det lättare.

```
char foo(int i)
{
    return i % 7;
}
```

Slide 10

```
// utan inline och därför utan optimering
int i = 9;
char c = "abcdefg"[foo(i)];

// med inline och optimering
int i = 9;
char c = 'c';
```

`inline` och 90-10-regeln

Slide 11

- I de flesta program spenderas 90% av körtiden i 10% av koden.
- De funktioner som utför arbetet vill man snabba upp.
- Felhanteringskod används sällan och bör inte göras `inline`.

Betrakta följande exempel:

Slide 12

```
void foo() {
    if(error) {
        /* 500 rader felhantering */
    }
    /* 10 rader kod */
}

for(int i = 0; i < 100000; i++)
    foo();    // foo är tidskritisk

/* foo används på flera andra ställen */
```

Vi konstaterar följande:

- Funktionen `foo` är tidskritisk.
- Gör vi `foo` inline får vi en stor ökning i kodens storlek pga felhanteringsrutinen.
- Vi väljer att dela upp `foo`:
 - Den del av koden som ska exekvera snabbt gör vi `inline`.
 - Felhanteringen låter vi ske genom funktionsanrop.

Slide 13

Vårt nya, effektiva program blir:

```
inline                                // vi gör foo inline
void foo() {
    if(error) {
        foo_handle_error(); // ej inlineanrop
    }
    /* 10 rader kod */
}
void foo_handle_error() { // felhanteringen ej inline
    /* 500 rader felhantering */
}

for(int i = 0; i < 100000; i++)
    foo(); // foo expanderar till < 15 rader kod
```

Slide 14

Temporära objekt och kopiering

För att slippa onödiga instruktioner vill man undvika att skapa temporära objekt:

Slide 15

- Använd anrop genom referens istället för värdesanrop där så är möjligt.
- Underlätta för kompilatorn att genomföra returvärdesoptimering (*return value optimization*).
- Omformulera uttryck för att undvika temporära objekt.

Exempel på hur man kan undvika temporära objekt:

```
std::string s;  
std::string r("foo");
```

Slide 16

```
s = r + "bar";    // temp objekt som innehåller "foobar"  
  
s += r;          // bättre: inget temp objekt  
s += "bar";     // samma resultat
```

Annat, vanligt förekommande, exempel på onödiga instruktioner:

Slide 17

```

class Huge {
    Huge() : e(0) { heavy(); }
    Huge(int i) : e(i) { heavy(); }
    Huge &operator=(const Huge &h)
    { e = h.e; heavy(); return *this; }
    heavy() { for(int i = 0; i < 10000000; i++); }
    int e;
};

class A {
    Huge e;
    A() { e = 7; } // kör heavy tre gånger
    A(int) : e(7) {} // kör heavy en gång
};

```

Specialiserad minneshantering

Om man ska allokera många objekt av samma typ kan man minska körtiderna genom att specialisera minnesallokeringen:

- Slide 18
- De inbyggda `new` och `delete` är byggda för att klara alla krav på minnesallokering.
 - Eftersom de är generella kan de vara för långsamma för vissa krav.
 - Man kan då skapa specialbyggda `new` och `delete` som sköter allokeringen.
 - Vi använder den inbyggda `new` för att allokera stora mängder minne; denna mängd kallar vi vår **minnespool**.
 - Minnet delas sedan ut i lagom stora portioner i vår egen `new`.

Vi har följande program som vi vill optimera:

```
/* Fraction definierar inte new och delete */
class Fraction {...};

Slide 19 for(int i = 0; i < 5000000; i++)
{
    Fraction *f = new Fraction; // ineffektivt att
    foo(f);                      // allokera 5M objekt
    delete f;                    // ett i taget
}
```

Vi skapar en klass Pool som hanterar minnespoolen:

```
// i filen pool.h
template<class T>
class Pool
{
public:
    Pool();
    ~Pool();
    void *alloc();           // hämta en minnescell
    void free(void *);      // lämna tillbaka minnescell

protected:
    void expand();          // utöka minnet
    enum { EXPAND = 65536 }; // allokerad storlek i bytes
    ...
}
```

Slide 21

```

        std::vector<T *> blocks;    // block med minne

        struct List {              // länkad lista av celler
            List *next;            // pekare till nästa cell
        } freelist;                // lediga celler
};

#include "pool.cpp"                // templateklassens implem.

```

Vi definierar funktionerna i pool.cpp:

Slide 22

```

template<class T>
Pool<T>::Pool() : blocks(100)
{
    assert(sizeof(T) >= sizeof(Pool<T> *));
    freelist.next = 0;
    expand();
}

template<class T>
Pool<T>::~~Pool()
{
    std::vector<T *>::iterator it;
    for(it = blocks.begin(); it != blocks.end(); it++)
        delete [] *it;
}

```

```

template<class T>
inline
void *Pool<T>::alloc()
{
    if(!freelist.next)
        expand();

```

Slide 23

```

    List *tmp = freelist.next;
    freelist.next = tmp->next;
    return reinterpret_cast<void *>(tmp);
}

```

```

template<class T>
inline
void Pool<T>::free(void *p)
{

```

```

    List *tmp = reinterpret_cast<List *>(p);
    tmp->next = freelist.next;
    freelist.next = tmp;
}

```

```

template<class T>
void Pool<T>::expand()
{

```

Slide 24

```

    int size = EXPAND / sizeof(T);
    T *mem = new T[size];
    blocks.push_back(mem);

    List *oldp = reinterpret_cast<List *>(mem);
    oldp->next = freelist.next;
    for(int i = 0; i < size - 1; i++) {
        List *newp = reinterpret_cast<List *>(mem + i + 1);

```

Slide 25

```
        newp->next = oldp;
        oldp = newp;
    }
    freelist.next = oldp;
}
```

Vår testklass Fraction ser ut som:

Slide 26

```
class Fraction
{
public:
    Fraction(int d = 0, int n = 1) : n_(n), d_(d) {}

    void *operator new(size_t size)
    { return pool->alloc(); }
    void operator delete(void *p, size_t)
    { return pool->free(p); }

protected:
    int n_;    // nominator
    int d_;    // denominator
    ...
}
```

```

public:
    static void init() { pool = new Pool<Fraction>; }
    static void exit() { delete pool; }

```

Slide 27

```

protected:
    static Pool<Fraction> *pool; // deklaration
};

Pool<Fraction> *Fraction::pool; // definition

```

Den optimerade huvudfilen blir:

```

Fraction::init();
clock_t start = clock();

for(int i = 0; i < 5000000; i++)
{
    Fraction *f = new Fraction;
    foo(f);
    delete f;
}

cout << "test took(s): "
     << ((double)clock()-start) / CLOCKS_PER_SEC
     << endl;
Fraction::exit();

```

Slide 28

Mätningarna ger:

Utan minnespool - test took(s): 5.10

Slide 29 Med minnespool - test took(s): 0.39

Hastighetsökning = $5.10/0.39 = 13$ ggr