

Del 3 – Klassanvändning, operatorer och pekare

Ämnesområden denna föreläsning:

- Synlighet
 - Överlagring av operatorer
- Slide 1**
- Vänner till klasser och funktioner
 - Virtuella funktioner och polymorfi
 - Abstrakta basklasser och strikt virtuella funktioner
 - Pekare till funktioner och medlemsfunktioner
 - Typkonvertering

Synlighet

Ett problem med moduler i C är att alla variabel- och funktionsnamn ligger globalt synliga. C++ botar detta genom att införa det mycket användbara nyckelordet **namespace**.

- För att kapsla in klasser, funktioner och data finns **namnrymder** genom **namespace**
- Slide 2**
- Med synlighetsoperatorn (*scope operator*) **::** kan man komma åt en namnrymd och medlemmar i klasser
 - Med **using** exponerar man innehåll i en namnrymd
 - Med **odöpta namnrymder** (*unnamed namespaces*) kan man deklarerar data som är synliga endast lokalt (påminner mycket om **static**) för att undvika namnkonflikter mellan namnrymder.

```

namespace Spc      // håll namnen på namnrymder korta
{
    class A;      // framåt(forward)deklaration
    int foo();
    int j;
    enum {arms = 2, legs = 1, hair = 517};
}
class Spc::A      // A är inte synlig utan Spc::
{
public :
    int foo();
    int j;
};
inline int Spc::A::foo() { return j; } // använder Spc::A::j
inline int Spc::foo() { return j; }   // ok: Spc::j synlig

int main()
{
    if(Spc::j > Spc::legs) // Använd Spc::j, Spc::legs explicit
    { /*...*/ }

    using Spc::j;          // j synlig i hela main()
    j = Spc::foo();       // Spc::foo() ej synlig, använd Spc::
    Spc::A a;
    j = a.foo();          // ok: Spc::A::foo() synlig genom a

    using Spc::A;         // hela klassen A synlig
    using namespace Spc; // allt data i Spc synligt

    return 0;
}

```

Slide 3

Slide 4

Överlagring av operatorer

Man kan omdefiniera operatorerna i C++, dock endast för egna typer såsom klasser, enums etc.

Slide 5

```
class MyInt {
public:
    MyInt &operator=(const MyInt &);        // tilldeln.
    MyInt &operator+=(const MyInt &);      // tilldeln.
    const MyInt &operator+(int);           // addition
    const MyInt &operator+(const MyInt &); // addition
};
...
MyInt i, j;
i = j + 7;           // använd oper+() och oper=()
j += 3;             // använd oper+=()
i.operator+=(j);    // operatorer kan användas explicit
```

Slide 6

```
class B;
class A {
    int operator()(int, double);        // överlagring av anrop
    int operator[](int) const;          // indexering
    B *operator[](const char *) const; // (i t.ex. map)
    A &operator++(int);                  // postfix inkrement
    A &operator--();                     // prefix dekrement
    const A operator*(const A &) const; // multiplikation
    A &operator*() const;                // avreferering

    friend const A operator*(int, const A &);
    friend ostream &operator<<(ostream &, const A &);
};

const A operator*(int, const A &);      // multiplikation
ostream &operator<<(ostream &, const A &); // utmatning
```

Vänner

Slide 7

- Ibland måste en klass ha tillgång till data i en annan
- Man vill dock inte använda åtkomsttypen `public` eftersom man får oönskad insyn
- Man specificerar sina vänner (klass eller funktion) med `friend`
- De får då åtkomstnivå `public` på data/funktioner

Slide 8

```
class MyInt {
    int i;                // i är private
public:
    MyInt(int j) : i(j) {}
    friend const MyInt operator*(int, const MyInt &);
    friend ostream &operator<<(ostream &, const MyInt &);
};

inline const MyInt operator*(int i, const MyInt &j)
{ return MyInt(i * j.i); }           // kommer åt MyInt::i
inline ostream &operator<<(ostream &s, const MyInt &j)
{ s << "{MyInt = " << j.i << "}"; } // kommer åt MyInt::i
```

Virtuella funktioner

- Normalt sett: kompilatorn vet vilken funktion som skall köras vid funktionsanrop, **statisk bindning**
- Virtuella funktioner: vilken funktion som körs bestäms av vilket objekt funktionen anropats genom, **dynamisk bindning**. Detta kallas **polymorfi**.

Slide 9

```
class A {
public:
    virtual void print() { cout << "A" << endl; }
};
class B : public A {
public:
    void print() { cout << "B" << endl; }
};
```

Slide 10

```
A a;
B b;
A *ap = &b;      // ok: B är subclass till A
A &ar = a;      // ok: ar refererar a

ap->print();     // skriver ut "B"
ar.print();     // skriver ut "A"
B *bp = &a;     // fel: A är inte subclass till B
```

Virtuella destruktorer

För att alla objekt i en arvskedja skall få sina destruktorer anropade krävs att basklassen har en **virtuell destruktör**.

```
class Av {public: virtual ~Av() {}};
class Bv : public Av {public: ~Bv() {}};
class A {public: ~A() {}};
class B : public A {public: ~B() {}};
```

Slide 11

```
B *bp = new B;
A *ap = bp;
delete ap;          // fel?: B's destruktör anropas inte

Bv *bvp = new Bv;
Av *avp = bvp;
delete avp;        // ok: B's destruktör anropas
```

Abstrakt basklass

En **strikt virtuell** (*pure virtual*) funktion tvingar subklasser att ge en definition. Basklassen kan då inte instantieras och kallas därför **abstrakt basklass**.

```
class A
{
public:
    virtual void print() = 0; // strikt pga '=0'
};
class B : public A
{
public:
    void print();           // definition måste ges om
};                          // B skall instantieras
```

Slide 12

Slide 13

```

void B::print()
{
    cout << "B" << endl;
}
...
A a;          // fel: A är en abstrakt basklass
B b;          // ok: B har alla funktioner definierade
b.print();    // kör B::print();

```

Pekare till funktioner

Antag att vi har en lista och vill göra något med varje element. En **funktionspekare** ger oss en möjlighet att slippa upprepa kod. Vi skickar en funktionspekare till en iteratorfunktion.

Slide 14

```

typedef void (*fp)(int &);          // funktionspekare
int iterate(int a[], int size, fp action) {
    for(int j = 0; j < size; j++) // j giltig i loopen
        action(a[j]);
}
void mod3(int &i) { i %= 3; }        // rest vid division
void print(int &i) { cout << i << " "; }
...
int a[7] = {1, 2, 3, 4, 5, 6, 7};
iterate(a, 7, mod3);
iterate(a, 7, print);    // ger utmatning 1 2 3 0 1 2 3

```

Slide 15

```

class A {
public:
    void foo(int) {}
    static void bar(int) {}
};
typedef void(*fp)(int);
fp p1 = &A::foo;    // fel: foo är void(A::*)(int)
fp p2 = &A::bar;    // ok: bar är void(*)(int)

typedef void(A::*fpm)(int);
fpm p3 = &A::foo;   // ok: foo är void(A::*)(int)
A a;
(a.*p3)(4711);     // kör foo från a
A *pa = &a;
(pa->*p3)(4711);   // kör foo från a via pa

```

Typkonvertering

Man använder **typkonvertering** för att göra om en pekare till en klass till en pekare till en annan.

- Slide 16
- Inbyggda datatyper konverterar till varandra
 - Subklasser konverteras till basklassen
 - Pekare utan kvalifierare (**const**) konverterar automatiskt till void-pekare
 - **static_cast** är konvertering under kompilering
 - **dynamic_cast** är konvertering som endast kan lösas under körning
 - **reinterpret_cast** är villkorslös konvertering
 - **const_cast** tar bort kvalifieraren **const** (detta går inte att göra med någon av ovanstående)

```
class A { virtual int f() {} };  
class B : public A { int f() {} };
```

```
const char inst[] = "C++";  
void *p;  
p = inst; // fel: kastar bort const  
p = (void *)inst; // ok, men riktigt fult
```

Slide 17

```
char *s = const_cast<char *>(inst); // ok  
p = s; // ok: samma kvalifierare  
s = reinterpret_cast<char *>(p); // ok  
s = reinterpret_cast<char *>(inst); // fel: const_cast  
// måste användas
```

```
B b;  
A *ap = &b;  
B *bp = dynamic_cast<B *>(ap); // bp != 0 eftersom  
// ap pekar på b  
int i = static_cast<int>(3.14); // undviker varning
```

Slide 18