

Synlighet

Namespace

Scope-operatorn

Klasser

Vänner

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`
- I **namnrymder** kan man kapsla in klasser, funktioner och data
- Med synlighetsoperatoren (*scope operator*) `::` kan man komma åt en namnrymd och medlemmar i klasser

Synlighet

- Med `using` exponerar man innehåll i en namnrymd
- Med **odöpta namnrymder** (*unnamed namespaces*) kan man deklarera data som endast är synliga i filen där de är definierade
- Obsolet metod: I C (och därför även C++) skapar man filstatiska funktioner genom nyckelordet `static`
- Genom att ha fillokala data minskar risken för namnkonflikter

Synlighet

```
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
```

Synlighet

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

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

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

    return 0;
}
```

Några överkursexempel på namnrymder och using

```
namespace XYZ_version1
{
    int    i;
    void   foo();    // deklaration
}
namespace XYZ_version2
{
    int    i;
    int    j;
    void   foo();    // deklaration
}
namespace XYZ = XYZ_version2; // alias för namnrymd

void XYZ::foo() {} // definition av XYZ_ver2::foo
int i;             // global variabel i
int j;             // global variabel j

// exempel på usingdeklaration (enstaka variabel)
{
    using XYZ::i; // lokal i, gömmer global i
    i = 3;        // ok: lokal i
}

// exempel på usingdirektiv (hel namnrymd)
{
    using namespace XYZ; // introducerar i, j och foo
    j = 4;               // fel! ::j eller XYZ::j ?
}
```

Synlighet

Man kan byta synlighet på medlemmar med `using`

```
struct A {  
    protected:  
        int i;  
};  
  
struct B : public A {  
    using A::i;  
};  
  
A a;  
B b;  
a.i = 1;           // fel: i är protected  
b.i = 2;           // ok
```

Synlighet och klasser

- För att komma åt data i klasser använder man synlighetsoperatoren `::`
- Ett bra exempel på hur synlighetsoperatoren används är STLs iteratorer
- Iteratorerna är inkapslade i behållarklasserna och döpta till samma sak: `T::iterator` resp. `T::const_iterator`

Synlighet och klasser

Nedanstående exempel visar hur man kan använda klassens synlighetoperator för att abstrahera sin kod

```
// här kan vi byta till lämpligare behållare vid behov  
// t.ex. std::vector eller std::deque  
typedef std::list<std::string> T;  
T t;  
...  
  
// oavsett klass, hämta dess const_iterator  
T::const_iterator i = t.begin();  
for(; i != t.end(); ++i)  
    std::cout << *i << std::endl;
```

const

Deklarationer med `const` ger skrivskydd.

```
class A
{
    const int ci;           // ci är skrivskyddad (ssk)
    const int *p_ci;       // pekare till ssk int
    int *const cp_i;       // ssk pekare till int
    const int *const cp_ci; // ssk pekare till ssk int

    int get() const;       // objektet är ssk i get
    int i;
};
// källkodsfil
int A::get() const { return i; } // ok: ändrar inte
int A::get() const { return i = 2; } // fel: ändrar obj
```

const och mutable

Funktioner deklarerade med `const` gör objektet read-only. Med `mutable` inför man ett undantag.

```
class A {
    int get() const;    // objektet är read-only i get
    int i;
    mutable int access_count;
};

inline
int A::get() const
{
    access_count++;    // ok: får ändras trots const
    return i;
}
```

Vänner

- 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 och funktioner
- `friend`-förhållandet ärvs inte
- `friend` bör **undvikas** för att inte äventyra inkapslingen

Vänner

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

inline const MyInt operator*(int i, const MyInt &j)
{ return MyInt(i * j.i); } // kommer åt MyInt::i

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

Överlagring av operatörer

Konstanta referenser

Returtyper

Argumenttyper

Överlagring av operatörer

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

Överlagring av operatörer

Exempel på operatoröverlagring

```
class MyInt {
public:
    const MyInt &operator=(const MyInt &); // tilldeln.
    const MyInt &operator+=(const MyInt &); // tilldeln.
    const MyInt operator+(int) const;      // addition
    const MyInt operator+(const MyInt &) const;
    ...
};

MyInt i, j;
i = j + 7;           // använd oper+( ) och oper=( )
j += i;             // använd oper+=( )
i.operator+=(j);    // operatörer kan användas explicit
```

Överlagring av operatörer

Det är viktigt att tänka på returtyperna för operatorerna

- Ta addition som exempel: inget av de ingående objekten innehåller resultatet och därför måste en temporär skapas
- Tar man däremot prefix ++-operatören så håller objektet själv rätt värde och kan returneras som referens
- Om returnerade referenser ska vara `const` är olika från fall till fall och beroende på klassens användningsområden (ta exempel heltalsklass):
 - helst `const` i fallet `(a = b) = c`
 - inte nödvändigtvis `const` om `(rect = r).normalize()`
- Är man osäker härmar man beteendet hos `int`

Överlagring av operatörer

Ytterligare exempel på operatoröverlagring

```
class B;
class A {
    int operator()(int, double);           // anropsoper
    B &operator[](int) const;              // indexing
    B &operator[](const char *) const;    // (i t.ex. map)
    const A operator++(int);               // postfix inkr.
    const A &operator--();                 // prefix dekr.
    const A operator*(const A &) const;   // multiplikation
    A &operator*();                        // avreferering
};

const A operator*(int, const A &);       // extern mult
std::ostream &operator<<(std::ostream &, const A &);
```

Polymorfi

Dynamisk bindning

Virtuella funktioner

Virtuella destruktorer

Strikt virtuella
funktioner

Abstrakta basklasser

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** – beteendet bestäms vid körningstillfället
- Polymorfi är endast applicerbart på **pekare och referenser** eftersom typen på ett objekt är känt vid kompileringstillfället.

Virtuella funktioner

```
struct A {
    void print()           { std::cout << "A"; }
    virtual void vprint() { std::cout << "A"; }
};
struct B : public A {
    void print()           { std::cout << "B"; }
    virtual void vprint() { std::cout << "B"; }
};

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

ap->vprint();         // polymorfi: skriver ut "B"
ar.vprint();         // polymorfi: skriver ut "B"
ap->print();          // ej polymorfi: skriver ut "A"
ar.print();          // ej polymorfi: skriver ut "A"
b.vprint();          // objekt: skriver ut "B"

B *bp = &a;          // fel: A är inte nedärvd från 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(); };

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
```

Abstrakta basklasser och strikt virtuella funktioner

- En **strikt virtuell** (*pure virtual*) funktion tvingar subklasser att ge en definition
- Basklassen kan då inte instantieras och kallas därför **abstrakt basklass**
- En abstrakt basklass definierar ett **interface** som alla nedärvda klasser måste följa

Abstrakta basklasser och strikt virtuella funktioner

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

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