

Minneshantering

- Lokala objekt **allokeras på stacken** och har kort livslängd
- Objekt med längre livslängd måste allokeras **dynamiskt** på heapen (free store)
- Dynamisk allokering görs med **new** och **delete**
- Statiska och globala objekt finns under programmets hela körtid

Minneshantering

- Minne (objekt) som allokerats med `new` ska deallokeras med `delete`
- Minne (vektorer av objekt) som allokerats med `new[]` ska deallokeras med `delete[]`

```
void foo()
{
    A a;                // a allokerad på stacken
    A *ap = new A;     // dynamiskt allokerad
    A *aa = new A[5];  // vektor med 5 A-objekt
    delete ap;         // frigör allokerat minne
    delete aa;         // fel: odefinierat beteende!
    delete [] aa;      // ok: destruktör för 5 element
} /* vid funktionens slut frigörs a automatiskt */
```

Minneshantering

- C++ har ingen automatisk minneshantering (*garbage collection*)
- Alla dynamiska objekt som inte lämnas tillbaka läcker minne
- Speciellt viktigt är det att hålla ordning på minnet i objekt som skapas och destrueras ofta som t.ex. strängar.

Undvik pekare

- Undvik pekare!
 - Det är lätt att referera otillåtet minne (t.ex. genom att avreferera `NULL`)
 - Det är lätt att referera ogiltigt minne (redan frigjort med `delete`)
- Skapa istället objekt på stacken
- Använd referenser! Dessa är garanterade att alltid referera ett giltigt objekt

Referenser

Exempel på hur referenser och pekare används som alias för ett objekt

```
class A { public: int i; };

A a;           // objekt
A &arf;        // fel: arf måste initieras
A &ar = a;     // ok: ar refererar a

int i;
i = a.i;       // vi kommer åt a.i genom a
i = ar.i;      // ar används istället för a

void foo(const A &); // deklaration
foo(a);           // skicka a som referens
```

Preprocessor

- Körs innan programmet ses av kompilatorn
- Används för att inkludera/exkludera källkod
- Kan användas för att definiera konstanter och makron
- C++ har många tekniker för att undvika preprocessor, såsom `inline`, `template` och `const`

Preprocessor

```
#include <iostream>    // inkluderar filen iostream
#include "myfile.h"    // letar i lokal katalog först

#ifdef MSDOS           // inkludera endast om MSDOS-miljö
#include <conio.h>
#endif
```

Klasser

Medlemmar

Åtkomst

Statiska variabler

Klasser

Klasser används för att

- strukturera koden
- kapsla in data
- knyta metoder till data
- skapa återanvändbara komponenter

Medlemmar och åtkomst

- Alla kommer åt `public`
- Endast subklasser kommer åt `protected`
- Endast klassen själv kommer åt `private`

Medlemmar och åtkomst

```
class Account
{
public:                                // tillgängliga för alla
    void deposit(double d)    { balance += d; }
    void withdraw(double d)  { balance -= d; }

protected:
    double balance;          // endast tillgänglig i klassen
};
// glöm inte semikolon!

int main()
{
    Account savings;

    savings.deposit(10);
    savings.withdraw(3.50);

    return 0;
}
```

Medlemmar och åtkomst

Medlemsfunktioner som definieras i klassdeklarationen blir inline

```
/* myfile.h -- headerfil med deklarationer */
class A
{
    int i;                // i är private
public:                  // foo är inline
    int foo(char) { return 7; }
    int bar();           // bar, baz är inline om
    char baz();          // definitionen anger det
protected:
    double d;
};
inline int A::bar() { return 2; } // inline i .h-fil

/* myfile.cpp -- källkodsfil med definitioner */
char A::baz() { return 'x'; } // ej inline i .cpp-fil
```

Statiska medlemmar

- Statiska medlemmar är gemensamma för alla instanser av samma klass
- De kan t.ex. användas till att räkna instanser av klassen

Statiska medlemmar

```
class A { // i headerfil myfile.h
    static const int size = 4711; // statisk medlem
    static int cnt; // statisk medlem
    int array[size]; // vanlig medlem
public:
    static int count() {return cnt;} // statisk funktion
};

// i implementationsfil myfile.cpp
const int A::size; // statiska medl. måste definieras
int A::cnt; // ... och defaultvärde är 0
...
int i = A::count();
```

Konstruktion och destruktion

Konstruktorer

Implicita konstruktorer

Initieringslistan

Destruktorer

Konstruktörer

- När ett objekt instansieras anropas dess **konstruktor**
- Konstruktorn saknar returtyp
- En **implicit konstruktor** tillhandahålls av kompilatorn "vid behov" och kör konstruktorn på alla objekt i klassen
- Konstruktorn utan argument kallas **defaultkonstruktor** och anropas vid konstruktion utan argument

Konstruktörer

- Konstruktorn har samma namn som klassen
- Konstruktorns jobb är att initiera objektet

```
class A
{
public:
    A()                // konstruktor
    {
        i = 0;        // konstruktorns kropp
    }

protected:
    int i;
};
```

Konstruktörer och initieringslistan

- Vissa typer kan inte tilldelas, bara initieras
- Tilldelning är onödigt arbete om objektet istället kan initieras till rätt värde
- Använd initieringslistan för initiering av allt data i klassen
- Initiera pekare till `NULL` och alloker eventuellt dynamiskt minne i kroppen

Konstruktorer och initieringslistan

```
class A
{
public:
    A(int &v)                // konstruktor
        : p(0), ref(v), str("hello") // initieringslista
    {}                       // tom kropp

protected:
    int          *p;
    int          &ref;
    std::string  str;
};
```

Konstruktörer

- Konstruktörer kan överlagras
- Använd `explicit` då konstruktorn har ett argument av enkel typ för att undvika ofrivillig konvertering

Konstruktörer

Exempel på överlagrad konstruktor och `explicit`

```
// date.h -- headerfil
class Date
{
public:
    Date(const Date &);           // kopieringskonstruktor
    Date(int y, int m, int d);   // ÅÅMMDD
    explicit Date(int c);        // dagar från 1900-01-01
    ...
    int year, month, date;
};

// date.cpp -- källkodsfil
Date::Date(const Date &d) { /* kopiering */ }
Date::Date(int y, int m, int d)
    : year(y), month(m), date(d) // initiering av medl.
{ }
```

Konstruktörer

Lite översikt:

- Konstruktör som är `protected` ger klass som bara kan skapas av subklasser
- Konstruktör som är `private` ger möjlighet att förhindra användandet av t.ex. kopieringskonstruktorn

Konstruktörer

- Kopieringskonstruktorn och tilldelningsoperatoren är privata
- De saknar definition, så om de används (kan endast ske i OS pga `private`) så får man länkarfel

```
class OS                                // Operating System
{
public:
    OS() {}                             // konstruktor
    ...
private:
    OS(const OS &);                      // ingen definition
    const OS &operator=(const OS &);
};
```

Destruktorn

- När ett objekt förstörs (t.ex. med `delete` eller då det hamnar utom räckvidd) anropas dess destruktör
- Destruktorn i en basklass bör vara deklarerad `virtual`
- Destruktorn har ingen returtyp och tar inga argument
- Syntaxen för destruktorn `~A()` kommer från operatoren för bitvis komplement (`~`), dvs icke

Destruktorn

```
// ligger i string.h -- headerfil
class String {
public:
    String() : str(0) {}           // defaultkonstruktör
    String(const char *s);       // kopierar s
    ~String();                   // destruktör
protected:
    char *str;
};

// ligger i string.cpp -- källkodsfil
String::String(const char *s) : str(0) {
    str = new char[strlen(s) + 1]; // glöm inte '\0'
    strcpy(str, s);
}
String::~~String() { delete [] str; } // obs! delete[]
```

`this`-pekaren

- I en medlemsfunktion kan man komma åt objektet genom att använda den fördefinierade pekaren `this`
- `this` går inte att tilldela

this-pekaren

```
class MyInt
{
    int i;
public:
    MyInt &increase(int);
};

inline MyInt &MyInt::increase(int j)
{
    i += j;
    return *this; // ger referens till objektet självt
}
```

Arv

Arv

Aggregation

Åtkomst

Multipelt arv

Virtuella basklasser

Arv och aggregation

Återanvändning kan ske genom:

- Aggregat (innehållande), "har en"
- Arv, "är en"

```
class Animal
{
    // Animal innehåller (har en) sträng
    std::string species;
};

// Bear ärver (är en) Animal
class Bear : public Animal
{
    ...
};
```

Arv och konstruktörer

- I en arvskedja måste basklasserna initieras före objektets kropp körs
- Annars skulle man kunna använda oinitierade referenser och konstanter som ligger i basklassen
- Basklassens konstruktor körs alltid först i initieringslistan
- Anges inget anrop till basklass anropas dess defaultkonstruktor

Arv och konstruktörer

- Antag arvshierarki $A \leftarrow B \leftarrow C$ och skapandet av ett objekt av typen C
- Exekveringsordningen blir:
 - Initieringslistan för C anropar initieringslistan för B som anropar initieringslistan för A
 - Initieringslistan för A utförs
 - Kroppen för A utförs
 - Initieringslistan för B utförs
 - Kroppen för B utförs
 - Initieringslistan för C utförs
 - Kroppen för C utförs

Arv och konstruktorer

```
class Vehicle
{
public:
    Vehicle(int w) : weight(w) {}
    int weight;
};
class Car : public Vehicle
{
public:
    Car(int p, int w) : Vehicle(w), persons(p) {}
    int persons;
};
...
Car volvo(5, 1700);
int i = volvo.weight;
```

Arv och åtkomst

Olika typer av arv: `public`,
`protected` och `private`

funktion \ arv	<code>public</code>	<code>protected</code>	<code>private</code>
<code>public</code>	<code>public</code>	<code>protected</code>	<code>private</code>
<code>protected</code>	<code>protected</code>	<code>protected</code>	<code>private</code>
<code>private</code>			

Åtkomst vid arv

- Klasser som är till hjälp för implementationen bör ärvas `private`
- Man vill inte ha implementationsdetaljer i sitt gränssnitt

Multipelt arv

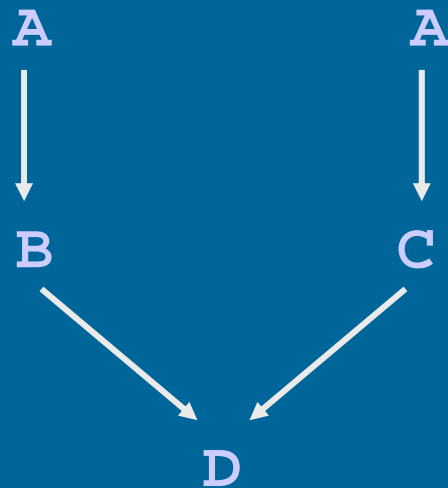
C++ stöder multipelt arv av godtyckligt antal klasser.

```
class Boat
{
public:
    Boat(bool s) : has_sail(s) {}
    bool has_sail;
};
class Amphibian : public Car, public Boat
{
public:
    Amphibian(int pers, int wt, bool sail = false)
        : Car(pers, wt), Boat(sail) {}
};
```

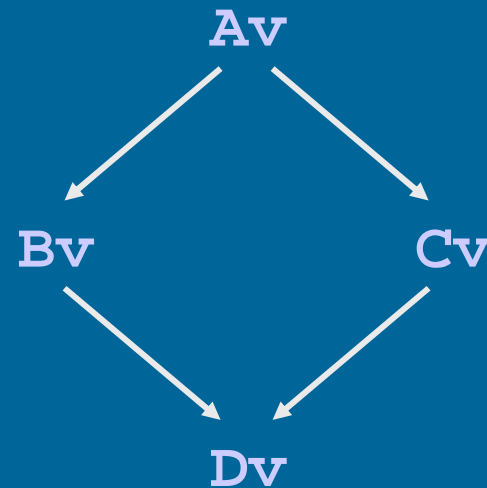
Virtuell basklass

För att använda gemensamt data i en delad basklass använder man *virtuella* basklasser genom nyckelordet `virtual`.

Vanligt arv



Virtuellt arv



Virtuell basklass

Ett exempel på hur data kan delas mellan basklasserna:

```
class A { public: int i; };
class B : public A {};
class C : public A {};
class D : public B, public C {};

class Bv : virtual public A {};
class Cv : virtual public A {};
class Dv : public Bv, public Cv {};

Dv dv;
dv.A::i = 7;      // alla ändrar samma data
dv.Bv::i = 8;
dv.Cv::i = 9;

D d;
d.A::i = 7;      // fel: tvetydigt
d.B::i = 8;      // ok: ändrar D->B->A1
d.C::i = 9;      // oj, ändrar D->C->A2
```

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 this

- `this`-pekaren går inte att tilldela, dvs pekaren går inte att flytta eftersom den är skrivskyddad
- Pekaren är "deklarerad"
`T *const this;`
då funktionen inte är `const`
- Pekaren är "deklarerad"
`const T *const this;`
då funktionen är `const`

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;
}
```