C++

En snabböversikt

Tommy Olsson, Institutionen för datavetenskap, © 2001

Detta är en kort överblick av C++. Naturligtvis finns det en hel del detaljer, både intressanta och viktiga, som har utelämnats.

Lexikala element

Följande tecken är tillåtna för att skriva C++-program. I stränglitteraler och teckenlitteraler är alla grafiska tecken tillåtna och dessutom en del specialsymboler.

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
+ - * / = ( ) { } [ ] < > ' " ! # ~ % ^ & _ : ; , . ? \ |
mellanrum, tabulertecken och andra ”vita tecken”
```

Tecken sätts samman till symboler (tokens), t.ex. namn på variabler, funktioner, klasser, reserverade ord, operatorsymboler, avgränsare, etc. Sådana symboler utgör de minsta lexikala elementen i ett program.

Kommentarer

Det finns två slags kommentarer. En *radslutskommentar* inleds med dubbla snedstreck, //, och avslutas av radslut.

```
// File: document.cc

static int page = 0;  // aktuellt sidnummer
```

Den andra typen av kommentar, *”C-kommentar”*, inleds med /* och avslutas av */. Den kan sträcka sig över flera rader.

```
/* Bortkommenterat avsnitt */
```

Kommentarer kan ej nästlas.

Identifierare

En identifierare består av ett eller flera tecken. Första tecknet måste vara bokstav eller understrykningstecken. Efterföljande tecken kan vara bokstäver, siffror, eller understrykningstecken. Identifierade som inleds med dubbla understrykningstecken eller ett understrykningstecken följt av en stor bokstav är reserverade av systemet. Identifierare bör ej vara längre än 31 tecken.

Nyckelord

Nyckelord har en speciell mening i språket och kan ej användas för att deklarera namn. Följande 63 identifierare är reserverade.

asm	**else**	**new**	**this**
auto	**enum**	**operator**	**throw**
bool	**explicit**	**private**	**true**
break	**export**	**protected**	**try**
case	**extern**	**public**	**typedef**
catch	**false**	**register**	**typeid**
char	**float**	**reinterpret_cast**	**typename**
class	**for**	**return**	**union**
const	**friend**	**short**	**unsigned**
const_cast	**goto**	**signed**	**using**
continue	**if**	**sizeof**	**virtual**
default	**inline**	**static**	**void**
delete	**int**	**static_cast**	**volatile**
do	**long**	**struct**	**wchar_t**
double	**mutable**	**switch**	**while**
dynamic_cast	**namespace**	**template**	

Datatyper

Grundläggande datatyper

De grundläggande typerna är inbyggda i språket. De utgörs av två huvudsakliga slag av typer, *integral types* och *floating point types*. Dessutom finns **void**, som kan användas för att ange att inget värde returneras av en funktion, eller för att ange en generell pekartyp, **void***.

bool		
char	**signed char**	**unsigned char**
wchar_t		
short	**int**	**long**
unsigned short	**unsigned**	**unsigned long**
float	**double**	**long double**

Typningen då det gäller de grundläggande typer är svag. Booletypen **bool** och teckentyperna **char** och **wchar_t** är heltalskompatibla, liksom delvis uppräkningstyper **enum**. De grundläggande typerna ingår i ett regelverk för automatiska typomvandlingar.

Egendefinierade datatyper

Utifrån de grundläggande datatyperna kan nya datatyper definieras. Hit hör uppräkning (**enum**), fält ([ ]), post (**struct**), klass (**class**), union (**union**), referens (&), funktion (()), pekare (*). Med hjälp av pekare kan länkade datastrukturer utformas.

```
enum Button { OFF, ON };    // uppräkningstyp

float  vector[10];    // fältobjekt

struct Coordinate {    // posttyp
   double  x;
   double  y;
};

struct Bitfield {    // bitfält
   unsigned int   a : 6;
   int : 5;                    // unused
   unsigned int  b : 4;
   bool             c : 1;
};

class Stack {    // klass (typ Stack)
public:
   Stack() : top(0) {} ;
   void   push(int x);
   int  pop();
private:
   unsigned  _top;
   int  _stack[100];
};

union Int_Or_Float {    // unionstyp
      int       i;
      float     f;
};

int       i;
int&      i_ref = i;    // referens (måste initieras)

void swap(int& a, int& b);    // funktion void (*)(int&, int&)

int*      i_prt = &i;    // pekare till int

typedef struct Listnode* ListnodePtr;    // länkad lista
typedef ListnodePtr List;

struct Listnode {
      int             element;
      ListnodePtr  next;
};

List head = new ListNode;
```

Uppräknings-, post- , klass- och unionsnamn är *typnamn*.

Lagringsklasser

Lagringsklasser har att göra med optimering, minneshantering, räckvidd och läsbarhet.

auto anger att ett objekt är lokalt i ett block och att dess livstid är lika med blockets aktiveringtid. Detta är standardlagringsklass för objekt som deklareras i block (anges aldrig).

register anger att ett objekts värde, om möjligt, ska lagras i ett register.

extern anger att ett namn har global räckvidd.

static anger för ett objekt som deklareras i block att dess livstid är lika med programmets och att dess värde därmed bibehålls mellan successiva blockaktiveringar.
anger för en deklaration på filnivå att dess räckvidd endast är resten av aktuell filen.

typedef skapar synonym för datatyp.

Typspecificerare

const anger icke-modifierbarhet.

volatile anger att någon process utanför programmets kontroll kan ändra värdet på ett objekt, och att därför optimering avseende användningen av objektet ej får göras.

Typomvandling

C++ har både explicit typomvandling, s k *cast*, och implicit typomvandling (*coersion*). Implicit typomvandling kan inträffa i uttryck, vid parameteröverföring till funktioner och vid retur av funktionsvärden. Implicit typomvandling försvagar datatypningen och kan medföra svårupptäcka fel.

Automatisk typomvandling

Heltals- och flytpunktstyper kan blandas fritt i uttryck. Om möjligt typomvandlas värden så att ingen information förloras, men dessvärre tillåts värdeförstörande automatiskt omvandlingar. Värdebevarande omvandlingar kallas *promotions*. Man skiljer mellan *integral promotions* och *floating-point promotions*. Ej värdebevarande omvandlingar kallas *conversions*.

Integral promotions:

- **char**, **signed char**, **unsigned char**, **short int** eller **unsigned short int** omvandlas till **int**, om **int** kan representera alla värde för typen ifråga, i annat fall till **unsigned int**.
- **wchar_t** eller **enum** omvandlas till den första av typerna **int**, **unsigned int**, **long** eller **unsigned long**, som kan representera alla värden i den bakomliggande typen.
- *bitfält* omvandlas till **int**, om **int** kan representera alla värden för bitfältet, annars till **unsigned int** om denna kan representera alla värden. I annat fall finns ingen tilllåten omvandling.
- **bool** omvandlas till **int**; **false** blir 0 och **true** blir 1.

Floating-point promotions innebär omvandling från **float** till **double**.

Implicit omvandling av pekare kan ske från en viss pekartyp till **void***. Omvandling från **void*** till en viss pekartyp kan endast ske genom explicit omvandling.

En pekare till en klass kan omvandlas till en pekare till en publikt härledd basklass.

Conversions mellan de grundläggande datatyperna kan göras på alldeles för många olika sätt.

Uttrycklig typomvandling

Det finns fyra slags uttrycklig typomvandling. De har formen *form<resultattyp>(värde)*. För typomvandlingar som är väldefinierade, portabla och inverterbara används **static_cast**.

```
double d = static_cast<double>(i) ;          // i antas vara av typ int
```

För typomvandlingar som är representations- eller systemberoende använder man **reinterpret_cast**. Denna form av omvandlingar ska i princip undvikas.

```
int  i = reinterpret_cast<int>(ptr);          // ptr är en pekare
```

I klasshierarkier kan det i vissa fall vara användbart att typomvandla en pekare, eller referens, för en viss klass till en pekare, respektive referens, för en underordnad klass, s k *down-casting*. Detta görs med **dynamic_cast**. Derived antas vara en klass härledd från klassen Base.

```
Base*     base_ptr;
Derived*  dptr = dynamic_cast<Derived*>(base_ptr);
```

För konstanta objekt kan det i vissa fall var praktiskt att ta bort konstantheten (*casting away constness*), vilket görs med **const_cast**.

```
void f(const Object* const_objptr)
{
    Object*  objptr = const_cast<Object*>(const_objptr); ...
}
```

Det finns även två äldre former av uttrycklig typomvandling. Från C har formen (*typ*)*värde* ärvts, och sedan lade man i C++ till den vanligare funktionsformen *typ*(*värde*).

En konstruktor till en klass kan fungera som typomvandling, om den tar ett argument av annan typ än klassen ifråga. Man kan också deklarera medlemsfunktioner i en klass som omvandlar till annan typ än klassen.

```
class A {
public:
    A(int);                          // från int till A
    operator int() const;            // från A till int
};
```

Konstanter

Namngivna konstanter deklareras med nyckelordet **const**.

```
const double PI = 3.1415;
```

Litteraler

Litteraler är uttryckliga konstanta värden. Finns för varje *grundtyp*, vilket omfattar heltals-, flyttals-, och teckenlitteraler.

```
4711        3.1415        1.3e-2        'A'
```

Heltalslitteraler med suffix u eller U tolkas som **unsigned**, suffix l eller L anger **long**. Flyttalslitteraler med suffix f eller F anger **float**.

```
127U        3.14159L        3.14f
```

En *heltalslitteral* som inleds med 0 tolkas som oktal, en litteral som inleds med 0x tolkas som hexadecimal, för övrigt som decimal.

```
15          017          0xF
```

Stränglitteraler är teckenföljder omgivna med citattecken Ett citattecken som ska ingå i en stränglitteral föregås av ett bakstreck. Typen för en stränglitteral är **static char**[]. Tomma strängen skrivs "" och innehåller ett tomtecken, '\0'.

```
"Bjarne Stroustrup"        ""        "citattecken skrivs \"."
```

Flera stränglitteraler som skrivs i följd (även på successiva rader) sätts underförstått samman till en enda lång sträng. En enda stränglitteral kan skrivas över flera rader, om ett bakstreck skrivs sist på de rader där strängen ej är avslutad.

Teckenlitteraler anges inom apostrofer, t ex 'A'. Vissa ej skrivbara tecken och specialtecken anges med en s k kodskiftssekvens (*escape sequence*). Prefix L till en teckenlitteral anger typen wchar_t (*wide character*, 16-bitstecken).

'\"'	enkel apostrof	'\\'	bakstreck
'\0'	tomtecken	'\a'	alert
'\b'	backsteg	'\f'	sidmatning
'\n'	ny rad	'\r'	vagnretur
'\t'	tab (horisontell)	'\v'	vertikal tab
'\101'	oktalt 'A'		
'\x041'	hexadeciamlt 'A'		
L'ång'	**wchar_t** litteral		

Det finns en generell *tompekare* 0. I standardbibliotekets inkluderingsfiler deklareras NULL för att ange tompekare.

Värdena av *booletypen* **bool** skrivs false respektive true.

Uppräkningslitteraler (**enum**) definieras genom uppräkning.

```
enum RGB { red, green, blue };
```

Deklarationer och räckviddsregler

En *deklaration*, allmänt, förser en identifierare med *typ*, *lagringsklass* och *räckvidd*. En deklaration kan också vara en *definition*. För ett *objekt* är definitionen den deklaration som ger upphov till att objektet skapas och initieras. För en *funktion* är definitionen den fullständiga deklarationen av funktionskroppen.

```
double  PI = 3.1415;                        // objektdefinition

double circumference(double);               // funktionsdeklaration

double circumference(double radius)         // funktionsdefinition
{
    return 2 * PI * radius;
}
```

En *klassdefinition* inför en typ med klassens namn och med de operationer som deklareras i form av publika medlemsfunktioner för klassen, uttryckligen eller underförstått. En *klassdeklaration* anger endast att en identifierade är namnet på en klass, som definieras på annan plats.

```
class B;                // deklaration; B är en klass

class A {
    B*   b;             // pekare (och referenser) till B får nu deklareras
};

class B {               // definition av B
    A*   a;
};
```

Räckvidd

Deklarationer på *filnivå* har i princip räckvidd över hela programmet. Denna kan begränsas till enbart deklarationsfilen genom **static**-deklaration.

```
 int          global g = 1;
static int    local = 0;            // endast synlig i aktuell fil
```

Deklarationer som är *nästlade* i block har räckvidd inom deklarationsblocket. För statiskt deklarerade objekt gäller, att de skapas och initieras då deklarationsblocket aktiveras, destrueras och försvinner då deklarationsblocket avslutas.

```
void swap(int& a, int& b)
{
    int  temp = a;              // temp skapas och initieras
    a = b;
    b = temp;
}                               // temp upphör att existera
```

Livstiden för sådana objekt kan utvidgas till hela programkörningen genom en **static**-deklaration.

```
void print_heading()
{
    static int     page_number = 0;         // initiering första gången
    page_number++;
    cout << "Sida " << page_number << endl << 'f';
}
```

Ett dolt namn deklarerat på filnivå kan kommas åt med räckviddsoperatorn ::*namn*.

```
int i;

void f()
{
    int   i = ::i;          // deklarerat i initieras med värdet på globalt i
}
```

Räckviddsoperatorn används annars mest för att referera till namn i standardomgivningen (som finns i namnrymnden std), t.ex.

```
void print(std::ostream& os, int value);
```

eller för att deklarera och referera till klassmedlemmar.

```
class A {
public:
    void f();;              // medlemsfunktionen f deklareras
    static void g();        // klassfunktionen g deklareras
};

void A::f()                 // medlemsfunktionen f definieras
{ ...
}

class B {
public:
    void g() { A::g(); }    // anrop av en statisk medlemsfunktions
};
```

Deklarationer kan förekomma praktiskt taget var som helst i ett block. I villkorsstrukturer (speciellt **switch**) kan deklarationer och därmed initiering komma att hoppas förbi.

For-satsen har speciell regel för deklaration av en styrvariabel, vilken anses lokal i **for**-satsen.

```
for (int i = 0; i < 10; i++)            // i deklareras här
{
    cout << vector[i] << endl;
}                                       // i upphör att existera
```

Namnrymder

Namnrymder används för att dela upp ett programs totala namnrymd i olika, vanligtvis namngivna delrymder, främst för att kunna hantera namnkollisioner.

```
// File: stack.hh
namespace Stack {                       // specifikation
    void    push(int x);
    int     pop();
}
```

Namnrymder är *öppna*, dvs de kan successivt utvidgas.

```
// File: stack.cc
namespace Stack {                       // implementation
    const int   max_size = 100;
    int         stack[max_size];
    int         stack_top = 0;
}

void Stack::push(int x) { ... }
int Stack::pop() { ... }
```

```
// File: main.cc
#include "stack.hh"                    // hämta specifikationen
int main()
{
    Stack::push(4711);                 // använd stacken
    ...
}
```

En stack implementeras dock naturligtvis bäst som en *klassmall*. Detta medger att klassmallen kan instansieras för olika elementtyper och att flera stackobjekt kan skapas.

Typedef

En **typedef**-deklaration används för att skapa en *synonym* för den typ som den definierar.

```
typedef char* c_string;
```

Namnet c_string är typmässigt av typ **char*** (ingen unik typ införs) och **typedef** används vanligtvis för ökad läsbarhet.

Länkningsregler

C++-program byggs genom filinkludering, separatkompilering och länkning. Länkning av separatkompilerade moduler kräver lösning av externa referenser (objektreferenser, funktionsanrop).

Huvudregeln är att externa, icke-statiska objekt endast får definieras på ett ställe. Användning av nyckelordet **extern** tillsammans med en initierare i en objektdeklaration medför att deklarationen också är en definition. Användning av **extern** utan en initierare medför en deklaration som ej är en definition. Om **extern** utelämnas är deklarationen också en definition, vare sig en initierare anges eller ej.

```
// File: file1.cc
int  i;                    // definition av i

// File: file2.cc
extern int    i;           // deklaration av i

// File file3.cc
extern int    n = 1;       // definition av n

// File: file4.cc
int           i;           // otillåten omdefinition av namnet i
extern double n;           // typfel, n definierad som int
extern char   c;           // definition av c saknas
```

Konstantdefinitioner (**const**) och **inline**-definitioner på filnivå är lokala i filen. En klonstantdefinition kan uttryckligen deklareras **extern**.

Typedef-deklarationer och uppräkningslitteraler är lokala i sina deklarationsfiler. En uppräkning som definierats nästlad i en klass är lokal i klassen, och extern åtkomst kräver användning av räckviddsoperatorn.

Deklarationer placeras vanligtvis i inkluderingsfiler (*header files*) som inkluderas i de filer deklarationerna används. En inkluderingsfil ska förses med en *gard* för att undvika problem med multipel inkludering.

```
// File: stack.h
#ifndef STACK_H
#define STACK_H
class Stack
{ ...
};
#endif

// File: stack.cc
#include "stack.h"
...
```

Uttryck

Operatoruppsättningen är omfattande och möjligheterna att skriva komplicerade uttryck är stora, vilket man ska vara mycket restriktiv med.

Genom att avsluta ett uttryck med ett semikolon, kan man använda ett uttryck som en sats, en *uttryckssats*. Detta är meningsfullt för t.ex. operatorer som har sidoeffekter, t ex tilldelningsoperatorerna och stegningsoperatorerna.

En tabellöversikt över operatorerna finns på sista sidan.

Satser

Tomma satsen

En tom sats skrivs som ett semikolon.

Uttrycksatser

Ett uttryck som avslutas med ett semikolon utgör en sats.

```
x = y + z;
i++;
```

Blocksats

De sammansatta satsernas syntax kräver att alla sekvenser som består av fler än en sats skrivs som en sammansatt sats.

```
while (p != NULL)
{
    q = p;
    p = p->next;
}
```

En god regel (som av utrymmesskäl dock ej alltid tillämpas här) är att alltid använda den sammansatta satsen, även om endast en sats förekommer i en sekvens.

Selektionssatser

Det finns två former av selektionssatser, **if** och **switch**.

If-satsen är en villkorsstyrd selektionssats, som kan skrivas med eller utan **else**-gren.

```
if (queue_front == max_size) {
    front = 0;
}

if (a > b) {
    max = a;
}
else {
    max = b;
}
```

Då **If**-satser nästlas och den logiska strukturen är ett mångförgrenat val, är följande skrivsätt att rekommendera.

```
if (x < t->value) {
    find(x, t->left);
}
else if (x > t->value) {
    find(x, t->right);
}
else {
    return t;
}
```

Switch-satsen är ett mångförgrenat val som styrs av ett heltalsuttryck.

```
bool accept(char* question)
{
    char    answer = '\0';

    while (true) {    // bryts genom return-satser
        cout << question;
        cin >> answer;
        switch (answer) {
            case 'y':
            case 'Y'
                return true;
            case 'n':
            case 'N':
                return false;
            default:
                cerr << "Otillåtet val!");
                break;
        }
    }
}
```

Det är viktigt att avsluta de **case**-alternativ för vilka **switch**-satsen ska avbrytas med **break** eller **return**, annars sker s k *fall-through* till efterföljande **case**, eller till **default**.

Repetitionssatser

Det finns tre satser för repetition, **while**, **do**-**while** och **for**. **While**-satsen är en förtestad villkorsstyrd slinga.

```
while (p != NULL && p->value != x)
{
    p = p->next;
}
```

Do-**while**-satsen är en eftertestad villkorsstyrd slinga.

```
do
{
    cout << "ge ett tal 1 - 10: ";
    cin >> x;
}
while (x < 1 || x > 10);
```

For-satsen är egentligen en förtestad villkorsstyrd slinga, med en initieringssats, ett villkorsuttryck, samt ett uttryck som vanligtvis används för att stega slingvariabler. Den bör dock endast användas för räknarstyrda slingor.

```
for (int i = 0; i < n; i++)
{
    cout << vector[i] << endl;
}
```

Break och continue

Break-satsen kan användas, dels för att avbryta en **switch**-sats efter att det, eller de, **case**-alternativ som ska utföras har utförts (se **switch**-satsen för exempel).

Break-satsen kan också användas för att avbryta en slinga, dvs för att hoppa ur en **while**-, **do**-**while**- eller **for**-sats.

```
while (true) {
    ...
    if (...) break;          // while-satsen hoppas ur
    ...
}
```

Continue-satsen används i repetitionssatser för att avbryta pågående varv i en slinga och direkt påbörja nästa varv.

```
while (villkor) {
    ...
    if (...) continue;       // återstående satser t överhoppas
    ...                      // hopp till styrvillkoret ovan
}
```

Continue-satsen bör undvikas och istället låter man **if**-satsen omfatta de satser som återstår i slingan.

Goto

Goto-satsen är ett ovillkorligt hopp till ett namngivet läge i programmet. Ett programläge anges med en identifierare följd av ett kolon.

```
if (disaster) goto hell;
...
hell:
    grill();
```

Det finns regler som begränsar möjligheterna att hoppa hur som helst i koden. **Goto**-satsen bör normalt helt undvikas.

Return

Return-satsen används för återhopp från funktioner. För en funktion som ska returnera ett värde anges ett uttryck som ger returvärdet.

```
return i - 1;
```

I en **void**-deklarerad funktions skrivs enbart **return**. Det kan finnas flera **return**-satser i en funktion.

Deklarationssats

Deklaration av objekt är en sats på formen

typ *objektnamn*;

Funktioner

Det finns endast en form av underprogram i C++, *funktion*. En funktion som ”returnerar” **void**, inget värde, motsvarar den form av underprogram som brukar kallas *procedur*.

Prototyper

En prototyp deklarerar en funktion till namn, returtyp och parametrar. Definitionen av funktionen finns på annan plats.

```
double   sqrt(double x);
void     print(const char*);
```

Typerna hos parametrarna kallas funktionens *signatur*, eller *parameterprofil*, och i denna kan även returtypen ingå.

Parameteröverföring

Det finns två former av parameteröverföring, kopiering (*call-by-copy*) och referens (*call-by-reference*).

Kopieringsanrop innebär att en parameter, vid funktionsanropet, initieras med det motsvarande argumentets värde. För övrigt finns ingen koppling till argumentet och eventuell manipulering av parametervärdet i funktionen påverkar ej argumentets värde.

```
int max(int a, int b)
{
    return a > b ? a : b;
}
```

Referensanrop innebär att en parameter, vid funktionsanropet initieras med en referens till det motsvarande argumentet, och att därför varje manipulation av parametern också direkt påverkar argumentets värde.

```
void swap(int& a, int& b)
{
    int  old_a = a;
    a = b;
    b = old_a;
}
```

inline-funktioner

inline-deklaration är ett direktiv till kompilatorn att funktionsanrop helst ska ersättas med, i princip, funktionskroppen och därigenom inget egentligt funktionsanrop kommer att genereras.

```
inline int max(int a, int b)
{
    return a > b ? a : b;
}
```

Standardargument

Funktionsparametrar kan deklareras att ha standardvärden, som används om inget motsvarande argument ges i ett anrop.

```
double* make_vector(unsigned size = 100, initvalue = 0.0)
{
    double*     v = new double[size];
    for (int i = 0; i < size; i++) v[i] = initvalue;
    return v;
}

double*  d1 = make_vector();
double*  d2 = make_vector(1.0);
double*  d3 = make_vector(200, 0.0);
```

Standardargument endast kan utnyttjas sammanhängande från höger till vänster, vilket ej tillåter godtyckliga kombinationer av uttryckliga argument och standardargument.

Överlagring

Överlagring (*overloading*) innebär att samma namn kan ges till olika funktioner, eller operatorer. Överlagrade funktioners, eller operatorers, signatur måste skilja för att kompilatorn utifrån de argument som anges ska kunna avgöra vilken som ska användas i ett anrop.

```
void  print(int);
void  print(char*);

print(5);
print("överlagring");
```

Medlemsfunktioner deklarerade i olika klasser och funktioner deklarerade i olika namnrymder kan ha samma namn och identiska signaturer, utan att detta medför problem eller olösbara problem.

Exempel på operatoröverlagring visas nedan i klassen IntVector. Samtliga fördefinierade operatorsymboler utom ::, **?:**, . och .*, får överlagras. Den fördefinierade prioriteten och associativiteten gäller.

Klasser

En klass är en sammansatt datatyp som kan ha både medlemsfunktioner och datamedlemmar av varierande typ. Olika åtkomst kan specificeras för olika medlemmar och med detta kan man dölja medlemmar som ej ska vara allmänt tillgängliga. Genom härledning kan man låta en klass ärva från en eller flera andra klasser och på så sätt återanvända kod.

Konstruktorer och destruktorer

En *konstruktor* är en speciell medlemsfunktion som automatisk utförs då ett klassobjekt skapas, och används normalt för att initiera objektet. En klass kan ha flera konstruktorer och vilken som används beror på vilka argument man ger.

En *destruktor* är en speciell medlemsfunktion som automatiskt utförs då ett klassobjekt upphör att existera, och används främst för klasser som använder dynamiskt minne för att återlämna detta. En klass kan endast ha en destruktor och den kan inte ha några parametrar.

```
class IntVector {
    friend operator<<(const IntVector&);
public:
    IntVector(unsigned size = 100);          // standardkonstruktor
    IntVector(const IntVector&);             // kopieringskonstruktor
    ~IntVector() { delete [] _vector; }      // destruktor

    IntVector& operator=(const IntVector&);          // tilldelning

    int&     operator[](unsigned i);                 // indexerings-
    int      operator[](unsigned i) const;           // operatorer

    unsigned size() const { return _size; }
    ...
private:
    int*         _vector;      // int-fält, skapas av konstruktor
    unsigned     _size;        // initieras av konstruktor
};

IntVector    v1(50);      // standardkonstruktor, size=50
Intvector    v2 = v1;     // kopieringskonstruktor
IntVector    v3;          // standardkonstruktor, size=100
v3 = v1;                  // tilldelningsoperator
```

Kopieringskonstruktor används också automatiskt vid parameteröverföring (*call-by-copy*) och vid retur av funktionsargument.

Explicit konstruktor

En konstruktor med *en* parameter definierar även en implicit typomvandling, vilket i vissa fall är önskvärt, i andra inte. Standardkonstruktorn i klassen IntVector ovan definierar en typomvandling från **unsigned int** till IntVector, vilket man nog inte vill ha som en implicit omvandling. För att förbjuda implicit typomvandling görs **explicit**-deklareration:

```
explicit IntVector(unsigned size = 100);
```

Medlemsfunktioner

En medlemsfunktion är en funktion som deklareras inuti en klass. Den tillhör klassen och har fri tillgång till **public**-, **protected**- och **private**-delarna. Om funktionen *definieras* i klassen, som size() i klassen IntVector ovan, behandlas den som om den vore **inline**-deklarerad.

Vänfunktioner

En vän (**friend**) till en klass kan antingen vara en vanlig funktion, en angiven medlemsfunktion i en annan klass eller samtliga medlemsfunktioner i en annan klass, i vilket fall man anger klassen som vän. Att vara en vän till en klass innebär att man har åtkomst även till skyddade och privata medlemmar. **friend**-deklarationerna kan skrivas var som helst i den klass som ska ha vänner.

```
class A {
    friend bool operator==(const A&, const A&);
    friend B::f();
    friend class C:
public:
    ...
};
```

this-pekaren

I en icke-statisk medlemsfunktions kropp är nyckelordet **this** ett uttryck vars värde är adressen till det objekt för vilket medlemsfunktionen har anropats. Typen för **this** i en medlemsfunktion för klassen T är T*, **const** T* om funktionen är **const**-deklarerad.

Följande medlemsfunktion i klassen DList (dubbellänkad lista) anropas för ett nytt listobjekt för att få det inlänkat till höger i listan om ett befintligt objekt utpekat av d. Här finns ingen annan åtkomst till det nya objektet än **this**.

```
void DList::insert_right(DList* d)
{
    if (d->right != NULL) d->right->left = this;
    this->right = d->right;
    d->right = this;
    this->left = d;
}
```

Härledning och arv

Genom härledning (*derivation*) kan man låta en klass (*subklass*) ärva (*inherit*) från en annan klass (*basklass*). Normalt innebär härledning att subklassen ska ha alla publika metoder som basklassen, men att man eventuellt modifierar vissa av dessa (*”overriding”*), lägger till fler samt kan lägga till nya datamedlemmar.

C++ har flera former av arv, **public**, **protected**, **private**, samt enkelt, multipelt och upprepat. De tre första anger i princip hur **public**-medlemmarna i basklassen ska ärvas, till subklassens **public**-, **protected**- eller **private**-del.

Enkelt arv innebär att subklassen har *en* direkt basklass. Detta är i grunden det naturliga sättet att ärva.

```
class Derived : public Base { ... };
```

Multipelt arv innebär att basklassen har två eller flera direkta basklasser. Problem kan uppstå om det i basklasserna finns medlemmar med samma namn.

```
class Derived : public Base1, private Base2 { ... };
```

Repeterat arv uppstår vid multipelt arv då de direkta basklasserna i sin tur är härledda och det åtminstone för vissa av dessa finns en gemensam basklass.

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

Problem till följd av detta är att medlemmarna i klassen A ärvs både via B och C och därmed i två instanser till D.

Det finns olika sätt att hantera de problem som nämnts ovan, bl a med operatorn :: för att uttryckligen ange klasstillhörighet och *virtuellt arv* för att undvika repeterat arv.

Konstruktorers exekveringsordning

Exekveringsordning för initierande konstruktorer och medlemmars konstruktorer:

1. Basklasserna för en klass initieras i deklarationsordning.
2. Medlemmarna i en klass initieras i deklarationsordning.

Observera att deklarationsordningen går före t ex den ordning i vilken man skriver initierare i klassens konstruktor.

Virtuell medlemsfunktion

En virtuell medlemsfunktion (deklareras som **virtual**) kan under exekvering väljas bland funktionerna i en klasshierarki. En förutsättning för detta är att objekt refereras via pekare eller referenser.

```
class A {
public:
    virtual void f() const;
};

class B : public A {
public:
    virtual void f() const;     // ”overriding” A::f()
};

A*  x = new A;
x->f();                 // A::f anropas
x = new B;
x->f();                 // B::f anropas; utan virtual A::f
```

Att pekaren x ibland refererar ett objekt av typ A och ibland ett objekt av typ B kallas *polymorfism* (”många former”).

Rent virtuell medlemsfunktion och abstrakt klass

En rent virtuell (*pure virtual*) funktion har ingen definition, utan istället avslutas prototypen med ”= 0”. En klass som har minst en rent virtuell medlemsfunktion är en *abstrakt klass*.

```
class Abstract {
    ...
    virtual void f() = 0;
    ...
};
```

För abstrakta klasser kan inga objekt skapas. Sådana klasser används vanligt i typhierarkier för att deklarera ett gemensam gränsyta för klasshierarkins subklasser. Rent virtuella medlemsfunktioner måste definieras i konkreta klasser.

En abstrakta klass kan också åstadkommas genom att man deklarerar klassens konstruktorer så att inga initieringskonstruktorer finns i klassens **public**-del.

Typidentifiering

Ett **typeid**-uttryck kan användas för att dynamiskt avgöra typen på objekt. Värdet på ett **typeid**-uttryck är en referens till en fördefinierad klass type_info. Denna klass har jämförelseoperationerna == och !=, samt medlemsfunktionen name, med vilken man kan ta reda på namnet på objektet typ.

```
class A { ... };
A*  a = new A;;

if ( typeid(*a) == typeid(A) ) ...          // uttrycket ger värdet sant

cout << typeid(*a).name() << endl;      // ger utskriften A
```

Ett annat sätt att undersöka typen för ett objekt är att använda **dynamic_cast**. Om man försöker typomvandla en pekare och denna inte tillhör den typ man försöker omvandla till eller någon subtyp till denna, erhåller man värdet 0. Är det en referens man försöker typomvandla och omvandlingen ej är tillåten genereras undantaget bad_cast.

Mallar

Mallar (*templates*) tillåter definition av funktioner och klasser parametriserade på, främst, avseende datatyper men även på värden.

```
template<class T> T max(const T& a, const T& b)
{
    return a > b ? a : b; ;
}

int       i = 1, j = 2, k;
double  a = 1.0, b = 2.0, c;

k = max(i, j);          // instans för int skapas
c = max(a, b);          // instans för double skapas
```

Följande klassmall visar hur en stacktyp kan parametriseras, både med avseende på elementtyp och stackstorlek.

```
template<class T, unsigned size = 100>
class Stack {
public:
    Stack();
    void    push(const T&);
    T       pop();
private:
    int     top;
    T       element[size];
};

Stack<int, 50>      stack_1;    // int-stack, storlek 50
Stack<double>       stack_2;    // double-stack, storlek 100

typedef Stack<int, 10> IntStack10;
IntStack10          stack_3;
```

Klassmallar kan även användas i härledning.

```
template<class T>
class Base { ... };

template<class T>
class Derived : public Base<T> { ... };
```

Undantag

Undantag (*exceptions*) används för att hantera ”exceptionella händelser” under programkörning. Några undantag kan genereras av exekveringssystemet. Undantagen i sig är objekt och kan vara av varierande typ.

throw-uttryck används för att generera undantag.

```
throw "Panik!";         // genererar undantag av typen const char*

throw;      // kan användas i en hanterare för att regenerera
            // det infångade undantaget
```

För att fånga in undantag inför man **try**-block kring kod där undantag uttryckligen kan genereras eller kan erhållas från anropade funktioner, och tillhörande hanterare, **catch**, där undantagen kan fångas in, baserat på deras typ, för hantering.

```
try {
    ...
    if (disaster) throw "Panik!";       // normal körning avbryts.
    ...
}
catch (const char* error) {     // undantag av typ const char* fångas.
    ...                         // någon lämplig form av hantering.
}
// programmet fortsätter här, såvida inga ohanterade undantag.
```

Preprocessor

Som ett första steg i en kompliering utförs alltid en bearbetning av källkoden av en preprocessor. Den utför speciella preprocessordirektiv, som vanligtvis finns i varje källprogram. Ut från preprocessorn kommer ”ren” C++-kod, som därefter kompileras. En vanlig användning av preprocessorn är *filinkludering*, där inkluderingsdirektivet (**#include**), av preprocessorn, ersätts med källkoden på den angivna filen.

```
#include <ifstream>     // inkludering av standardbibliotek
#include "list.h"       // inkludering av annan fil

std::fstream  input("INPUT.TXT");
```

För att undvika multipel inkludering av deklarationsfiler använder man en s k *gard*.

```
// File: list.h
#ifndef LIST_H
#define LIST_H
...     // här finns deklarationerna på filen
#endif
```

Det finns ytterligare direktiv till preprocessorn. Man bör undvika att använda preprocessorn så långt möjligt.

In- och utmatning

In- och utmatning erhålls genom flera standardbibliotek, dels C++:s strömbibliotek, dels det från C ”ärvda” in- och utmatningsbiblioteket *stdio*. Strömbiblioteket är i allmänhet att föredra, då det är både enklare att använda och typsäkrare.

Standardsströmmar

Det finns fyra standardströmmar.

cin — standard inström, typ istream. Buffrad.

cout — standard utmatningsström, typ ostream. Buffrad.

cerr — standard felutmatningsström, typ ostream. Obuffrad.

clog — standard loggutmatningsström, typ ostream. Buffrad variant av cerr.

Dessa är fördefinierade och normalt associerade med tangentbordet (cin) resp. skärmen. Man får tillgång till dem genom att inkludera filen iostream.h.

Grundläggande skrivning och läsning av strömmar görs med operatorn << resp. >>. Dessa finns överlagrade för de flesta grundtyper.

```
#include <iostream>
using namespace std;

int main()
{
    int     i;
    double  d;

    cout << "Skriv ett heltal och ett reellt tal: "
    cin >> i >> d;
    cout << "Du skrev talen " << i << " och " << d << endl;
}
```

Formatterad utmatning

Insättningsoperatorn << har vissa standardformat definierade för olika typers utmatning. För att styra utmatningsformatet finns s k manipulatorer tillgängliga. Vissa manipulatorer är parameterlösa och tillgängliggörs genom iostream.h, medan man måste inkludera även iomanip för att kunna använda parametriserade manipulatorer.

```
int main()
{
    int  i = 10;
    cout << oct << setw(4) << i << endl;
    cout << dec << setw(4) << i << endl;
    cout ' << hex << setw(4) << i << endl;
}
```

Vissa manipulatorers effekt gäller till dess vidare, medan vissas effekt endas har effekt på närmast efterföljande utskrift.

Manipulator	*Funktion*
endl	ny rad + flush (se nedan)
ends	skriver ut ett tomtecken i en sträng
flush	skriver ut eventuellt buffrat innehåll i utmatningsbufferten
dec	decimal utskrift av tal

oct	oktal utskrift av tal
hex	hexadecimal utskrift av tal
ws	läser förbi ”vita tecken” i indata
skipws/noskipws	slår av/på förbiläsning av vita tecken.
setw(**int**)	anger utskriftsfältstorlek (antal tecken)
setfill(**int**)	sätter utfyllnadstecken
setbase(**int**)	sätter basformat
setprecision(**int**)	sätter önskad flyttalsprecision
setiosflags(**long**)	sätter formatbitar
resetiosflags(**long**)	återställer formatbitar

Filströmmar

Genom att inkludera filen fstream.h får man tillgång till två klasser, ifstream för *infilströmmar* och ofstream för *utfilströmmar*, med tillhörande operationer för att öppna och stänga filer, läsa och skriva, samt testa filers tillstånd.

```
int main()
{
    char        inputname[20];
    ifstream    infile;
    ofstream    outfile("output.txt");    // öppnar namngiven fil
    char        c;

    cout << "Ange utdatafilens namn: " ;
    cin.getline(inputname);
    infile.open(inputname);

    if ( ! infile.good() ) return 1;

    while ( infile.get(c) ) {
        outfile.put(c);
    }

    infile.close();
    outfilr.close();
}
```

Standardbibliotek

Det finns ett omfattande standardbibliotek för C++. I detta finner man, utöver ovan nämnda bibliotek för in- och utmatning, stort att beskriva här. Nedanstående listning ger en uppfattning om standardbibliotekets omfattning och innehåll.

Namn som har ett c som prefix, som t ex <ctime>, motsvarar samma namn utan c i standardbiblioteket för C. Detta stöds ännu ej av alla C++-system och inkludering sker i så fall enligt tidigare konvention, exempelvis ”#**include** <ctype.h>”.

Containers (behållare)

<string>	strängar
<vector>	endimensionellt fält.
<list>	dubbellänkad lista.
<deque>	dubbeländkö.
<queue>	kö (omfattar även priority_queue).
<stack>	stack.
<map>	associativt fält (omfattar även multimap).
<set>	mängd (omfattar även multiset).
<bitset>	boolemängd.

General utilities

<utility>	operatorer och par.
<functional>	funktionsobjekt.
<memory>	minnestilldelare för behållare (containers).
<ctime>	datum och tid (C).

Iterators

<iterator>	iteratorer och iteratorstöd.

Algorithms

<algorithm>	allmäna algoritmer.
<cstdlib>	bsearch(), qsort() (C).

Diagnostics

<stdexcept>	standardundantag.
<cassert>	assert-makro (C).
<cerrno>	felhantering (C).

Strings

<string>	strängar.
<cctype>	teckenklassificering (C).
<cwtype>	*wide-character*-klassificering (C).
<cstring>	strängfunktioner (**char***; C).
<cwchar>	funktioner för *wide-character*-strängar (C).
<cstdlib>	C-funktioner (C).

Input/Output

<iosfwd>	framåtdeklaration av I/U-faciliteter.
<iostream>	standard iostream-objekt och operationer.
<ios>	iostream-basklasser.
<streambuf>	strömbuffertar.
<istream>	inströmmar.
<ostream>	utströmmar.
<iomanip>	manipulatorer.
<sstream>	strängströmmar.
<cstdlib>	teckentypsfunktioner (C).
<fstream>	filströmmar.
<cstdio>	formaterad in- och utmatning (C).
<cwchar>	in- och utmatning av *wide-character* (C).

Localization

<locale>	språk- och kulturskillnader (datumformat, valutasymbol, strängkollationering).
<clocale>	språk- och kulturskillnader (C).

Language support

<limits>	numeriska gränser.
<climits>	numeriska gränser för skalärer (C).
<cfloat>	numeriska gränser för flytpunktsvärden (C).
<new>	dynamisk minneshantering.
<typeinfo>	stöd för typidentifiering under körning (RTTI)
<exception>	stöd för undantagshantering.
<cstddef>	språkstöd (C).
<cstdarg>	funktionsargumentlistor av variabel längd (C).
<csetjmp>	stackhantering.
<cstdlib>	programavslutning (C).
<ctime>	systemklockan (C).

<csignal> signalhantering (C).

Numerics

<complex> komplexa tal och operationer.

<valarray> numeriska vektorer och operationer.

<numerics> generaliserade numeriska operationer.

<cmath> matematiska standardfunktioner.

<cstdlib> slumptal (C).

Kompilatorer och standardförslaget

Alla kompilatorer stödjer ännu hela C++ enligt den nya standarden från 1998 (ISO/IEC 14882). Bland det som eventuellt inte stöds finner man t ex:

- namnrymder; **namespace**, **using**.
- undantag (*exceptions*), **try-throw-catch**.
- mallar; **template**.
- möjligheten att i koden för en mall deklarera att ett mall-argument,som T i **<class** T>, är ett typnamn; **typename**.
- det nya standardbiblioteket.
- typinformation under körning (*run-time type information*, RTTI); **typeid**, **dynamic_cast**.
- de nya typomvandlingarna; **static_cast**, **dynamic_cast**, **const_cast**, **reinterpret_cast**.
- deklaration av konstruktorer så att de endast får användas uttryckligen; **explicit**.
- möjligheten att deklarera komponenter så att de får ändras även om objektet de tillhör är ett konstant objekt; **mutable**.
- strömbiblioteket i alla detaljer.

Suns kompilator för C++, CC, stödjer i mycket högr grad standarden.

Operatorer, deras prioritet och associativitet

Operatoruppsättningen är omfattande och prioritetsnivåerna många. Varje tabellrad nedan anger en prioritetsnivå.

Operator	**Associativitet**[a]	**Operatortyp**
::*namn* *klassnamn*::*medlem* *namnrymd*::*medlem*	höger->vänster vänster->höger	global räckvidd, enställig räckvidd, tvåställig
() [ ] . -> ++ -- **typeid**(…) **dynamic_cast**<*typ*>(…) **static_cast**<*typ*>(…) **reinterpret_cast**<*typ*>(…) **const_cast**<*typ*>(…)	vänster->höger	funktionsanrop, fältindexering, val av postkomponent, postfix upp- och nedstegning, typidentifiering, nya typomvandlingarna.
+ - ++ -- ! & * ~ **sizeof** *uttryck* **sizeof**(*typ*) **new** **delete** **delete[]** (*typ*)	höger->vänster	enställiga: plus, negation, prefix stegning, logisk negation, adressberäkning, avreferering av pekare, ett-komplement, storlek för värde reså. typ, dynamisk minnes-tilldelning och återlämning, dynamisk minnesåterlämning fält, typomvandling (föråldrad).
objekt.**pekare-till-medlem* *pekare*->*.pekare-till-medlem*	vänster->höger	val av medlem
* / %	vänster->höger	tvåställiga multiplikativa: multiplikation, division (heltalsdivision för heltal), rest vid heltalsdivision
+ -	vänster->höger	tvåställiga additiva: addition, subtraktion
<< >>	vänster->höger	bitskift (kan vara maskinberoende): vänster, höger
< <= > >=	vänster->höger	relationer
== !=	vänster->höger	likhet, skilt från
&	vänster->höger	bitvis OCH
^	vänster->höger	bitvis XOR (exklusivt eller)
|	vänster->höger	bitvis ELLER
&&	vänster->höger	logiskt OCH (kortslutande)
||	vänster->höger	logiskt ELLER (kortslutande)
= += -= *= /= %= &= |= ^= <<= >>=	höger->vänster	tilldelning (x *op*= y motsvarar x = x *op* y)
?:	höger->vänster	villkorsoperatorn (uttrycket x?y:z får värdet y om x är sant, annars z)
throw	höger->vänster	generera undantag (exception)
,	vänster->höger	sekvensiell beräkning (högeroperandens värde blir kommauttryckets värde)

a. Associativiteten bestämmer beräkningsriktningen för sammansatta uttryck, där de ingående operatorer har *lika* prioritet (och alltså inte prioritet kan styra beräkningsordningen).