Introduktion till streams

INTRODUKTION TILL STREAMS

Läsa bestämt antal tecken

Lägga tillbaka tecken

Kontrollera antalet inlästa tecken

Användardefinierad överlagring av >>

Medlemmar i klassen ios

Medlemmar i klassen ostream

Medlemmar i klassen istream

Övningsuppgifter

- Uppgift 1
- Uppgift 2
- Uppgift 3

Introduktion till streams

I C++ finns som standard ett klassbibliotek för in- och utmatning som heter **streams**. Detta består av ett stort antal klasser för in- och utmatning till och från bildskärm, filer coh minne.

I detta kapitel skall vi går igenom den grundläggande strukturen samt det viktigaste i avseende på formaterad in- och utmattning till bildskärm och från tangentbord. Klasser för io mot minne och filer behandlas **inte** i denna introduktion.

Klasserna som hör ihop med strömmar i C++ bildar ett omfattande klassbibliotek. Klasserna kan användas direkt som de är eller användas som basklasser till dina egna klasser. Med strömbibliotekets klasser kan du utföra formaterad I/O på både på för- och användardefinierade datatyper.

För att kunna använda strömmar för vanlig in och utmatning måste ditt program inkludera filen **iostream.h**. Andra headerfiler som du kanske måste inkludera är **strstream.h** för formatering av data till och från minnet och **fstream.h** för filhantering. Både **strstream.h** och **fstream.h** inkluderar **iostream.h**.

Vad är en ström?

En ström är en abstraktion som refererar till något dataflöde från en källa till en datamottagare. Vi använder även synonymerna erhålla, få och hämta när vi talar om inmatning av tecken från en källa.

Vid utmatning använder vi infoga, placera och lagra. Oavsett namnet så kan en strömklass användas till att forma data aven när det inte gäller input eller output. Du kommer att få se att formatering i minnet är möjligt för såväl tecken, arrayer och andra strukturer.

I/O ström biblioteket.

I/O ström biblioteket har två parallella klasser: streambuf och ios. Båda är klasser på lägsta nivå.

Streambuf

Klassen **streambuf** erbjuder metoder för in och utmatning på strömmar när det krävs lite eller ingen formatering. Streambuf används som basklass av andra klasser i strömbiblioteket och kan även användas som basklass i dina egna klasser och bibliotek.

De flesta av streambufs funktioner är implementerade som inline för högsta effektivitet. Klasserna **strstreambuf** och **filebuf** är subklasser till **streambuf**.

ios

Klassen **ios** (och samtliga klasser som härleds från denna klass) innehåller en pekare till ett objekt av typen streambuf. Ios är basklass till två klasser: **istream** för inmatning och **ostream** för utmatning. Klassen **iostream** är sedan härledd från **istream** och **ostream** genom multipelt arv.

```
class ios;
class istream : virtual public ios;
class ostream : virtual public ios;
class iostream :public istream, public ostream;
```

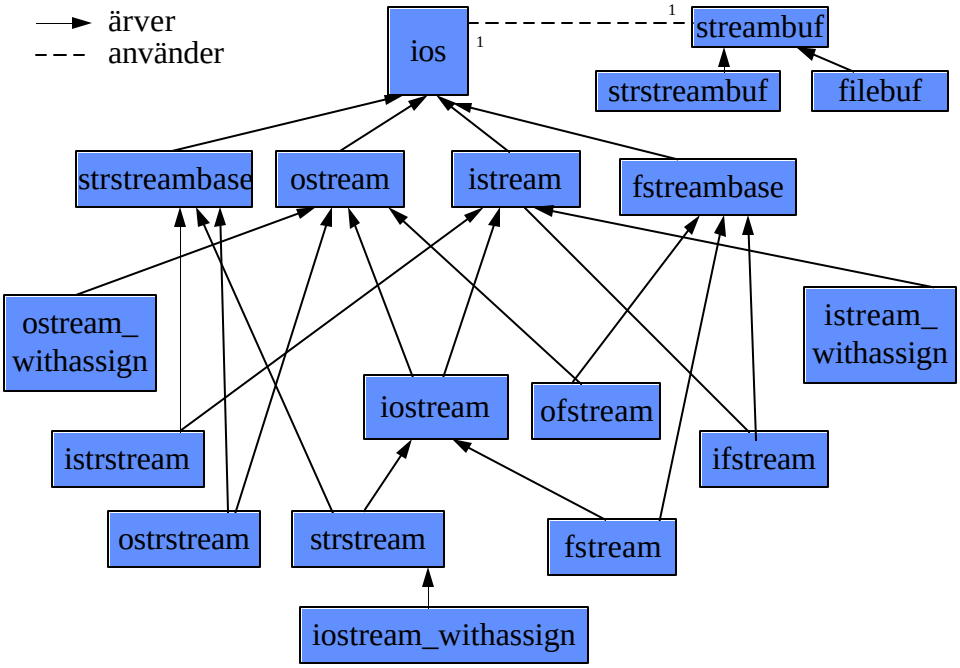
Dessutom finns det tre **withassign**-klasser som är härledda från **istream**, **ostream** respektive **iostream**.

```
class istream_withassign : public istream;
class ostream_withassign : public ostream;
class iostream_withassign : public iostream;
```

Dessa tillför en tilldelningsoperator för respektive klass.

Klasshierarkin

Man kan beskriva klasshierarkin på nedanstående sätt. Vi visar här hur de klasser som är deklarerade i **iostream.h, fstream.h** samt **strstrea.h** är relaterade till varandra:



Klassen ios innehåller variabler för att ta hand om kommunikationen med klassen streambuf samt felhantering.

Klassen istream stödjer både formaterad och oformaterad omvandling av teckenströmmar hämtade från streambufs.

Klassen ostream stödjer både formaterad och oformaterad omvandling av teckenströmmar som skall lagras i streambufs.

Iostream klassen kombinerar istream och ostream för tvåvägsoperationer där en och samma ström fungerar som källa och datamottagare.

Withassignklasserna tillför en tilldelningoperator till respektive basklass på nedanstående sätt:

```
class istream_withassign : public istream {
     istream_withassign();
     istream& operator=(istream&);
     istream& operator=(streambuf*);
};
```

På samma sätt överlagras tilldelningsoperatorn för ostream_withassign och iostream_withassign.

Standardströmmar

Ett C++ program börjar med tre fördefinierade öppna strömmar. Dessa ska vara deklarerade som objekt av typen withassign enligt följande:

```
extern istream_withassign cin;
extern ostream_withassign cout;
extern ostream_withassign cerr;
```

Deras konstruktorer anropas varje gång iostream.h inkluderas. Dock görs själva initieringen bara en gång.

Dessa fyra standardströmmar motsvarar de som normalt brukar användas av C-programmeraren enligt nedan:

cin motsvarar standard input dvs stdin (fil nr 0)
cout motsvarar standard output dvs stdout (fil nr 1)
cerr motsvarar standard error dvs stderr (fil nr 2)

Man brukar kunnaomdirigera dessa standardströmmar till andra filer eller teckenbuffertar efter det att programmet har startat.

Utmatning

<< operatorn

Data skickas till en ström med hjälp av den binära << operatorn. Som vänster operand används ett objekt av klassen ostream.eller av någon type som är härledd från ostream. Den högra operanden kan vara någon av standardtyp dvs.

```
char (signed och unsigned),
short (signed och unsigned),
int (signed och unsigned),
long (signed och unsigned),
char* (som sträng),
float, double,long double och void*.
```

Genom överlagring av bitskiftoperatorn << enligt nedan låter man kompilatorn välja vilken funktion som ska anropas.

```
// Utdrag från iostream.h

    ostream&  operator<< (signed char);
    ostream&  operator<< (unsigned char);
    ostream&  operator<< (short);
    ostream&  operator<< (unsigned short);
    ostream&  operator<< (int);
    ostream&  operator<< (unsigned int);
    ostream&  operator<< (long);
    ostream&  operator<< (unsigned long);
    ostream&  operator<< (float);
    ostream&  operator<< (double);
    ostream&  operator<< (long double);
    ostream&  operator<< (const   signed char *);
    ostream&  operator<< (const unsigned char *);
    ostream&  operator<< (void *);
```

Genom att skriva

```
cout << "Hello!\n";
```

skriver man strängen "Hello" till cout (standard output, vanligtvis din skärm) följt av en ny rad.

Detta innebär ett anrop av medlemsfunktionen

```
ostream&  operator<< (const   signed char *);
```

vilket också skulle kunna göras som

```
cout.operator<<( "Hello!\n");
```

<< operatorn är vänsterassociativt och returnerar en referens till den ostream för vilken den är anropad. Detta tillåter upprepad användning på nedanstående sätt:

```
void display(int i, double d)
{
    cout << "i=" << i << ", d=" << d << "\n";
}
.
display( 8, 2.34);
```

Detta kommer att resultera i att:

```
i = 8, d = 2.34
```

skrivs på standard output.

Raden

```
cout << "i=" << i << ", d=" << d << "\n";
```

motsvarar

```
(((((cout << "i=") << i) << ", d=") << d) << "\n");
```

där vi ska vara glada att vi har möjlighet att använda den första varianten.

<<-operatorn finns även överlagrad för pekare till void (void *);

```
int i = 1;
cout << &i; // visar adress i hexformat
```

Utmatning av enstaka objekt av typen char:

```
char ch = 'a';
cout << ch; // visa A
```

Metoderna put och write

För att mata ut ett enskilt tecken kan du använda medlemsfunktionen put deklarerad i ostream enligt följande:

```
ostream& ostream::put(char ch);
```

Med deklarationen

```
int ch='x';
```

blir nedanstående två raderna likvärdiga.

```
cout.put(ch);
cout << (char) ch;
```

Medlemsfunktionen write ger möjlighet till utmatning av större (binära) datamängder.

ostream& ostream::write(const signed char* ptr,int n);
ostream& ostream::write(const unsigned char* ptr,int n);

Write funktionen matar ut exakt n tecken (char) oberoende av eventuella '\0'-tecken.

T ex så skriver man med:

```
cout.write( (char *) &x, sizeof(x))
```

den binära formen av x till standard output.

Det finns en liten skillnad mellan << operatorn och funktionerna put och write. Den formaterande << operatorn kan orsaka att en låst ström frigörs (flushas) och utdata kan ha en bestämd fältlängd. Därför kan "cout << 'a';" och "cout.put ('a');" ge olika resultat.

Alla formatflaggor påverkar << men inga påverkar put och write.

Formatering av utdata

Formatflaggor

Formatering av både in och utdata bestäms av formatflaggor i klassen ios. Lägena bestäms av bitarna i en long int enl. följande:

```
// Utdrag från klassen ios
     public:
     enum {
          skipws     = 0x0001, // hoppa över space vid input
          left  = 0x0002,       // vänster just. utdata
          right= 0x0004,        // höger just utdata
          internal   = 0x0008,  // fyll ut efter tecken eller
                                // talsystemstecken
          dec   = 0x0010,       // decimal omvandling
          oct   = 0x0020,       // oktal omvandling
          hex   = 0x0040,       // hex. omvandling
          showbase  = 0x0080,   // visa basindikator för
                                // utdata
          uppercase = 0x0100,   // uppercase hex. data
          showpos    = 0x0200,  // visa + vid pos. integer
          scientific= 0x0400,   // använd 1.2345E2 notation
                                // utdata
          fixed= 0x1000,             // använd 123.45  notation
          unitbuf    = 0x2000,  // rensa alla strömmar efter input
          stdio= 0x4000,             // rensa stdout, stderr efter input
     };
```

Flaggorna ärvs av de härledda klasserna ostream och istream. Det finns funktioner för att sätta, testa och ta bort flaggorna antingen individuellt eller i grupp. En del flaggor tas bort automatiskt efter input eller output.

Sätta flaggor

Man kan sätta och kontrollera formatflaggor med funktionen **setf**. Funktionen returnerar tidigare inställning.

```
long old;
old = cout.setf(ios::showbase|ios::hex|ios::uppercase);
cout << 255 << endl;
cout.setf(old);
```

ger t.ex.

```
0XFF
```

som resultat.

Funktionen **setf** finns också i en annan variant där man kan ange vilken typ av flaggor som skall påverkas:

setf(long ny, long vilka);

där det första argumentet är nya värden och det andra vilka som skall ändras.

```
// Konstanter som kan användas som andra argument
// till setf...
static  const long basefield;   // dec | oct | hex
static  const long adjustfield; // left | right | internal
static  const long floatfield;  // scientific | fixed
```

Om man t.ex. bara vill ändra flyttalsformat kan man skriva som följer:

cout.setf(ios::fixed,ios::floatfield);

Talsystem

Normalt matas heltal in i decimalform. Detta kan ändras genom att flaggorna ios::dec, ios::oct och ios::hex sätts.
Om alla flaggor är noll (defaultvärde) gäller decimalform.

Vidd

Normalt används minsta möjliga antal tecken för att beskriva den högra operanden. För att ändra och kontrollera detta kan du använda viddfunktionerna.

```
int ios::width(int w);// sätt bredd till w
                      // returnera tidigare bredd
int ios::width();  // returnera aktuell bredd
                      // ingen förändring
```

Det förutbestämda värdet på width är noll vilket ger utdata utan padding. En bredd som är annat än noll ger en output på åtminstone så många tecken, eller padding, så att bredden fylls ut. Ingen trunkering äger rum: om bredden är mindre än antalet tecken som behövs, ignoreras detta. (Liksom om width = 0.) Ett exempel

```
int i = 123;
int old_w = cout.width(6);
cout << i;      // output bbb123 b=blank
                // bredden sätts till 0
cout.width(old_w);// sätt tillbaka tidigare bredd
```

Lägg märke till att bredden sätts till noll efter varje formaterad utmatning. I följande exempel:

```
int i, j;
...
cout.width(4);
cout << i << " " << j;
```

används åtminstone använda 4 teckenpositioner för att presentera i medan j och blanktecknet visas utan utfyllnad.

Manipulatorer

Ett enklare sätt att förändra bredden (width) och andra formatvariabler är att använda en speciell funktion som kallas manipulator. Manipulatorer tar en strömreferens som ett argument och returnerar en referens till samma ström.

Manipulatorer kan alltså stoppas in i en kedja av infogare eller extraktorer. Detta görs för att förändra strömmars tillstånd som en bieffekt utan att egentligen utföra någon infogning. T ex är

```
cout << setw(4) << i << setw(6) << l;
```

likvärdigt med:

```
cout.width(4);
cout << i;
cout.width(6);
cout << j;
```

Setw är en parametiserad manipulator som finns deklarerad i **iomanip.h**. Andra parametiserade manipulatorer som **setbase**, **setfill**, **setprecision**, **setiosflags** och **resetiosflags** fungerar på samma sätt.

För att kunna använda dessa så måste ditt program inkludera iomanip.h. Du kan skriva dina egna manipulatorer utan parametrar:

```
ostream& beep( ostream& os)
{
    return os << "\a\a";
}
...
cout << "Signal kommer " << beep;
```

Manipulatorer med en parameter är mer komplicerade och kräver iomanip.h.

Manipulatorer deklarerade i iostream.h:

dec	sätt decimal flagga
oct	sätt oktal flagga
hex	sätt hexadecimal flagga
ws	läs förbi alla blanktecken
endl	sätt i en nyrad och töm strömmen
ends	sätt i avslutande null i sträng
flush	töm en ostream

Manipulatorer deklarerade i iomanip.h:

```
setbase(int)  sätt omräkningsbas till n (0,8,10 eller 16)
resetiosflags(long)       nollställ omvandlingsflaggor
setiosflags(long)     sätt omvandlingsflaggor
setfill(int n)        sätt fyllnadstecken
setprecision(int)     sätt antal decimaler
setw(int)             sätt fältbredd
```

De icke parametiserade manipulatorerna dec, hex och oct (deklarerade i iostream.h) tar inga argument utan förändrar omräkningsbasen (och lämnar den förändrad).

```
int i = 36;
cout   << dec << i << " "
       << hex << i << " "
       << oct << i << endl;
// visar 36 24 44
```

Manipulatorn endl infogar ett tecken för ny rad och tömmer strömmen (flush). Du kan tömma en ostream när som helst med

```
ostream << flush;
```

Utfyllnad och padding

Utfyllnadstecknet och riktningen på paddingen beror på var man sätter utfyllnadstecknet samt på höger, vänster eller inre flaggor. Som utfyllnadstecken används normalt blanktecken. Du kan ändra detta genom att använda funktionen fill:

```
int i = 123;
cout.fill('*');
cout.width(6);
cout << i; // visar ***123
```

Normalt används högerjustering (paddingen på vänster sida). Du kan ändra detta (och andra formatflaggor) med funktionerna setf och unsetf.

```
int i = 56;
...
cout.width(6);
cout.fill('#');
cout.setf(ios::left,ios::adjustfield);
cout << i; // visar 56####
```

Det andra argumentet, ios::adjustfield, talar om för setf vilka bitarna som skall påverkas. Alternativt kan du använda manipulatorerna setfill, setioflags och resetioflags att förändra utfyllnadstecknet och paddingläget.

Användardefinierad överlagring av << operatorn

Du kan skriva infogare för utmatning av din data genom att överlagra << operatorn. Antag att du har en typ enligt följande:

```
struct Information {
   char *namn;
   double vaerde;
   char *enhet;
};
```

Du kan överlagra << enligt följande:

```
ostream& operator << (ostream& s, Information& m)
{
   s << m.namn << " " << m.vaerde << " " << enhet;
}
```

Uttrycken

```
   Information x;

   // Initiering....
   x.namn   = "Kapacitet";
   x.vaerde = 1.15;
   x.enhet  " "liter";

   // Utmatning....
   cout << x;
```

skulle ge utdata enligt

```
   "Kapacitet 1.15 liter"
```

Inmatning

>> operatorn

Att hämta data från en ström görs på ett sätt som i stort liknar utmatning men i stället används en överlagrad >> operator, normalt använd som bitoperator för högerskift. Operatorn >> ger ett kompakt, tydligt och lättanvänt alternativ till den traditionella scanf-funktionen.

Den vänstra operanden är ett objekt av typen istream. Liksom vid output, kan den högra operanden vara av vilken typ som helst för vilken >> blivit överlagrad.

Alla fördefinierade typer som visats tidigare för utdata har en överlagrad >> operator. Du kan också överlagra >> för strömindata till dina egna datatyper. Om operatorn >> överlagras för en viss "typ" kallas den "typ"- extraktor, t.ex:

```
typ  x;
cin >> x;
```

Detta hämtar ett värde från cin (standard input, vanligtvis ditt tangentbord) och tilldelar detta till x. Omvandling och formatering är beroende av typen på x, hur extraktorn är definierad och hur formatflaggorna är satta.

Normalt kastas inledande blanktecken bort (enligt isspace -makrot i ctype.h) och läser sedan in data som motsvarar typen på den högra operanden.

Förfarandet med att hoppa över blanktecken kontrolleras av flaggan ios::skipws.

Skipws flaggan är normalt satt till att ge överhopp vid blanktecken. Om man tar bort flaggan (t ex med setf) så slår man av överhoppandet. Lägg även märke till de speciella manipulatorerna för datamottagare, t.ex. ws som låter dig hoppa över blanktecken.

Liksom << så är >> vänsterassociativ och returnerar sin vänstra operand. Den vänstra operanden är en referens till det objekt av typen istream för vilket det är anropat. Detta tillåter flera inputoperatorer att kombineras i ett uttryck. Betrakta följande exempel där vi i samma sats tilldelar värden till två variabler:

```
int i;
double d;
cin >> i >> d;
```

Först hoppas eventuella blanktecken över. Därefter tolkas indata t.o.m. nästa blanktecken som ett heltal som tilldelas variablen i. Nästa serie med icke blanktecken tolkas som en double och värdet tilldelas variablen d. Siffror som läses från standard input blir omvandlade till intern binär form och sparade i variabeln i respektive d.

Extraktorer för inbyggda typer

Det finns 3 kategorier av typer; heltalstyper, flyttalstyper samt strängar (dvs teckenfält avslutade med '\0'). Var och ett beskrivs nedan.

För heltals och flyttalstyper gäller att om det första tecknet som inte är blankt inte är en siffra eller . (punkt för decimalpunktsomräkning), sätts strömmen i ett feltillstånd och ingen ytterligare indata kan tas emot innan felet är avhjälpt.

Extraktorerför heltalstyper

För typerna short, int och long (signed och unsigned) så är det förutbestämda utförandet av >> att hoppa över blanktecken och att sedan omvandla ett heltalsvärde och att läsa tecken tills ett tecken kommer som inte passar in i typen. Formatet på heltalsvärden är det samma som heltalskonstanter i C++.

```
int i;
long l;
cin >> i;
cin >> l;
```

Extraktorer för flyttal

För typerna float och double blir effekten av >> operatorn överhopp av blanktecken och omvandling av flyttalsvärden ända tills ett tecken kommer upp som inte passar in. Formatet på flyttalsvärden är det samma som flyttalskonstanter men med utelämnat flyttalssuffix.

```
float f;
cin >> f;
```

Extraktorer för char

För typen char (signed eller unsigned) hoppar >> operatorn att över inledande blanktecken och tilldelar nästa tecken som inte är blankt till den högra operanden.

```
char ch;
cin >> ch;  // ch tilldelas nästa char som inte
          // är blank eller tab
```

Extraktorer för teckenfält

För inläsning till ett teckenfält gäller att tilldelning sker fram tom. första blanktecknet (jmfr med scanf("%s",str)).

```
char enamn[100];
```

```
char fnamn[100];
cout << "Ange för- och efternamn:";
cin  >> fnamn >> enamn;
```

Funktionen get

Funktionen get är finns i ett antal överlagrade varianter.
Om du t.ex. vill läsa nästa tecken, oavsett om det är blanktecken eller inte så kan du använda get(char&).

```
char ch;
cin.get(ch);  // ch tilldelas nästa char i
                    // istream även om det är blank
```

Medlemsfunktionen get för indata motsvarar put-funktionen för utdata.

En annan variant av **get** ger möjlighet till kontroll över antalet inlästa tecken, var de skall placeras samt avslutande tecken.

istream& istream::get(char *buf, int max, int term='\n');

Funktionen läser in tecken från indata strömmen till fältet som buf pekar på tills antalet tecken är max - 1 eller tills den stöter på det tecken som anges som tredje argument (defaultinitierat till '\n'), beroende på vilket som kommer första. Ett avslutande null läggs in automatiskt.

Fältet buf måste vara åtminstone max tecken. Observera att tecknet term (t.ex. '\n') ligger kvar i strömmen varför detta måste läsas för sig. Här är ett exempel på ett program som beräknar antalet rader i en fil eller som kommer från standard input:

Läsa textrad med getline

Vi kan också välja att läsa in textrader med funktionen **getline**. Denna fungerar precis som **get** men med den skillnaden att vi får hjälp med att plocka bort det avslutande tecknet (oftast '\n').

Läsa bestämt antal tecken

Du kan läsa in högst n tecken genom att första ange hur många tecken som skall läsas in:

```
char array[SIZE];
cin.width(sizeof(array));
cin >> array; //undviker overflow
```

Lägga tillbaka tecken

Medlemsfunktionen:

istream& istream::putback(char c);

Lägger tillbaka tecknet c i istream. Om detta inte är möjligt sätts stream till "fail".

Kontrollera antalet inlästa tecken

Med funktioen **gcount** kan man kontrollera antalet inlästa tecken från en istream.

```
char buff[100];
cin    >> buff;
cout   << "Antal nlästa tecken är "
       << cin.gcount();
```

Användardefinierad överlagring av >>

Du kan göra en extraktor för dina egna typer på samma sätt som du kan göra infogare:

```
struct Information {
   char *namn;
   double vaerde;
   char *enhet;
};


istream& operator >> (istream& s, Information& m);
{
   s >> m.namn >> m.vaerde >> m.enhet;
   return s;
}
```

För att läsa indata som "kapacitet 1.25 liter" kan du använda en rad som:

```
   Information m;
   cin >> m;
```

vilket innebär att du gör anropet

```
   operator << ( cin, m);
```

Medlemmar i klassen ios

```
int bad();
static long bitalloc();
void clear(int = 0);
int eof();
int fail();
char fill();
char fill(char);
long flags();
long flags(long);
int good();
void init(streambuf *);   /* PROTECTED */
int precision(int);
int precision();
streambuf* rdbuf();
int rdstate();
long setf(long _setbits, long _field);
long setf(long);
void setstate(int);   /* PROTECTED */
static void sync_with_stdio();
ostream* tie();
ostream* tie(ostream*);
long unsetf(long);
int width();
int width(int);
static int xalloc();
```

Medlemmar i klassen ostream

```
ostream& flush();
int opfx();
void osfx();
ostream& put(char);
ostream& put(signed char);
ostream& put(unsigned char);
ostream& seekp(streamoff offset);
ostream& seekp(streamoff offset, seek_dir);
streampos tellp();
ostream& write(const char*, int n);
ostream& write(const signed char*, int n);
ostream& write(const unsigned char*, int n);
```

Medlemmar i klassen istream

```
void eatwhite(); /* PROTECTED */
int gcount();
int get();
istream& get(char *buf, int len, char delim = '\n');
istream& get(unsigned char *buf, int len, char delim = '\n');
istream& get(unsigned char *buf, int len, char delim = '\n');
istream& get(char &ch);
istream& get(signed char &ch);
istream& get(unsigned char &ch);
istream& get(streambuf&, char = '\n');
istream& (char *buf, int len, char);
istream& getline(char *buf, int len, char);
istream& getline(signed char *buf, int len, char delim = '\n');
istream& getline(unsigned char *buf, int len, char delim = '\n');
istream& ignore(int n = 1, int delim = EOF);
istream& ipfx(int n = 0);
int peek();
istream& putback(char);
istream& read(char*, int);
istream& read(signed char*, int);
istream& read(unsigned char*, int);
istream& seekg(streampos pos);
istream& seekg(streamoff offset, seek_dir dir);
streampos tellg()
```

Övningsuppgifter

Uppgift 1

Använd manipulatorerna **dec**, **oct** och **hex** för att skriva ut en tabell för talen from. 0 tom. 255. Skriv i högerställda kolumner med 8 teckens bredd. Stanna utskriften vid var tjugonde rad så att användaren hinner läsa tabellen och sedan trycka på <ENTER> för att fortsätta.

Uppgift 2

Gör ett program som läser tecken för tecken från standard input (cin) tills eof, och som skriver ut varje tecken på standard output (cout) efter att ha konverterat alla små bokstäver (även å,ä och ö) till stor bokstav.

Uppgift 3

Gör ett program som läser textrader från standard input (cin) tills eof, sorterar dessa och som sedan skriver ut dessa sorterade i bokstavsordning på standard output (använd t.ex. ett fält med pekare till char och funktionen qsort).

Antag att vi vill göra ett program som beräknar antalet rader, ord och tecken som matas in från standard input

Normalt skall antalet rader, ord och tecken skrivas ut men genom att ange en flagga -l,-w eller -c kan användaren styra vad som skall skrivas ut.