Dynamisk minneshantering

- Dynamisk minneshantering
  - Dynamisk minneshantering
    - Kodsegment
    - Datasegment
    - Stacksegment
    - Heapsegment
  - new och delete
    - Allokering av enstaka objekt
    - Allokering av fält med objekt
    - Exempel på klass som alltid är beroende av att man använder []
    - Resultat
    - Placeringsoperator
  - Andra funktioner för dynamisk minneshantering
    - Malloc
    - Calloc
    - realloc
    - Free
    - Felhantering och malloc,calloc och realloc
  - Defaultversioner av new och delete
  - Egendefinierad överlagring av new och delete
    - Egen överlagring av globala newoperatorn
    - Egen överlagring av globala deleteoperatorn
    - Anrop av vår new och deleteoperator
    - Överlagring av globala placeringsoperatorn
    - Anrop av våra placeringsoperatorer
  - New och delete operatorn som klassmedlemmar
    - Newoperatorn som klassmedlem
    - Deleteoperatorn som klassmedlem
  - Exempel med egendefinierad minneshantering
    - Definition av statiska datamedlemmar
    - Definition av metoder
    - Definition av våra egendefinerade newoperatorer
    - Exempel: newplac1.cpp
    - Resultat
    - Exempel:newplac2.cpp
    - Resultat
    - Exempel:newplac3.cpp
    - Resultat
  - Exempel: Egendefinierad minnesanvändning för avlusning
    - En klass för minnesadministration: klassen Memory
    - Definition av static datamedlemmar i klassen Memory

Definition av static funktionsmedlemmar i klassen Memory
MemoryBase, en basklass för klasser som skall använda klassen Memory
Definition av static datamedlemmar i klassen MemoryBase
Definition av funktionsmedlemmar i klassen MemoryBase
Definiera subklasser till MemoryBase
Användning av Memory och MemoryBase
Resultat

Dynamisk minneshantering Dynamisk minneshanteringDynamisk minneshanteringDynamisk minneshantering

Ett program består av den **kod** som kompilatorn genererat, dvs instruktioner för den CPU som används. Dessutom reserveras utrymme för **data** dvs för den information som skall behandlas eller användas.

Vi kan skilja på de data för vilket det finns minne reserverat under hela programexekveringen, dvs **statiskt minne** , och de data som finns under vissa tidsperioder, **temporärt minne**. Data kan t.ex. vara konstanter, variabler eller egendefinierade typer.

När programmet exekveras tilldelas programmet en minnesarea. Hur stor denna är kan t.ex bero på i vilken miljö (operativsystem) vi exekverar, hur många andra program som körs samtidigt, total minnesstorlek mm. Det minne som vårt program har fått sig tilldelat kan oftast delas in i fyra delar:

Kodsegment

Den del där maskinkoden lagras. Från denna del hämtas de instruktioner (maskinkod) som styr vad programmet skall göra.

Datasegment

Den del som innehåller konstanta data samt plats för de globala variabler som vi deklarerat i vårt program. Denna del har normalt statisk storlek.

Stacksegment

Ett från början ledigt utrymme som används som tillfällig lagringsplats vid funktionsanrop (för överföring av parametrar, lokala variabler, överföring av returvärden, återhoppsadress mm).

Heapsegment

Ett från början ledigt utrymme som programmeraren kan använda dynamiskt dvs. för olika typer objekt vid olika tillfällen under en programexekvering. Genom att använda heapen kan vi hantera större datamängder än om vi enbart använder datasegmentet.

Hur ovanstående indelning stämmer överens med ditt system är miljöberoende. Detta gäller också gränser för hur stor varje del kan vara samt programmets totala minnesutrymme. Annat som kan skilja är namn på de olika delarna, åt vilket håll som stack och heap "växer". I UNIX-miljö brukar kodsegmentet t.ex. kallas för TEXT. I stora drag stämmer dock alltid ovanstående och du bör kunna identifiera motsvarande delar i ditt system.

new och delete

I C++ använder vi operatorn new för att begära minne från heapen. Det är sedan programmerarens som ansvarar för att minnet senare frigörs. Det gör vi genom att använda pekarvärdet vi tidigare fick från new som argument till delete.

Allokering av enstaka objekt

När vi vill reservera plats på heapen för ett enstaka objekt gör vi på nedanstående sätt:

```
// allokera minne
char *chptr = new char;
char *iptr   = new int;
Bankkund  *kundptr  = new Bankkund("Lisa Svensson");
Bankkonto *kontoptr = new Bankkonto(kundptr,"1234567-5");
....
// Använd objekt
...

// Frigör minne....
delete chptr;
delete iptr;
delete kundptr;
delete kontoptr;
...
```

Allokering av fält med objekt

När vi vill reservera plats på heapen för fält med objekt gör vi på nedanstående sätt. Lägg märke till att vi anger [] när vi använder delete på en pekare som vi har fått vid allokering av ett fält:

```
// Allokera minne
char *chptr = new char[100];
int *iptr   = new int[1000];
Bankkund  *kundptr  = new Bankkund[1000];
Bankkonto *kontoptr = new Bankkonto[1000];
....
// Använd objekt
........

// Frigör minne....
delete [] chptr;
delete [] iptr;
delete [] kundptr;
delete [] kontoptr;
...
```

Om vi utelämnar [] kan vi få problem med minneshanteringen. Genom att vi anger [] kommer nämligen kompilatorn att se till att destruktorn anropas för varje objekt i fältet. Utelämnar vi [] kommer bara det första objektets destruktor att anropas.

I praktiken brukar detta ha betydelse om klassen har en destruktor som utför någon väsentlig uppgift. Det är därför vanligt att programmerare utlämnar [] när de frigör ett fält av någon fördefinierad typ (char, int osv) eftersom dessa typer inte har någon destruktor..

En god regel är dock att alltid ange [] oberoende av om du frigör ett fält av egen- eller fördefinierad typ eftersom detta kan ha betydelse i framtiden eller om du flyttar programmet till en annan miljö.

Exempel på klass som alltid är beroende av att man använder []

Antag att vi har konstruerat en klass som allokerar minne i konstruktorn och firgör detta minne i destruktorn.

```
class Buffer {
public:
     Buffer(int size);
     Buffer(const Buffer&);
     ~Buffer();
     Buffer& operator=(const Buffer&);
     operator const char*() const;
     operator char*();
private:
     char *buffer;
     int size;
};
```

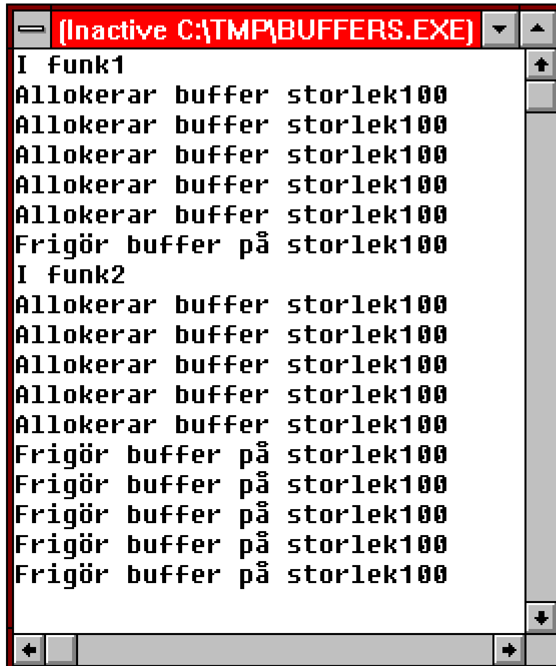
I konstruktorer och i tilldelningsoperatorn allokerar vi minne och i destruktor och i tilldelningsoperatorn frigör vi minne:

```
Buffer::Buffer(int s)
: buffer(new char[s]),size(s)
{
     memset(buffer,'\0',s);
     cout << "Allokerar buffer storlek"
          << size << endl;
}
Buffer::Buffer(const Buffer& bufref)
: buffer(new char[bufref.size]),size(bufref.size)
{
     memcpy(buffer,bufref.buffer,bufref.size);
     cout << "Allokerar buffer storlek"
          << size << endl;
}

Buffer::~Buffer()
{
     delete [] buffer;
     cout << "Frigör buffer på storlek"
          << size << endl;
}
```

```
Buffer& Buffer::operator=(const Buffer& bufref)
{
     if (this != &bufref)
        {
         delete [] buffer;
         buffer = new char[bufref.size];
         memcpy(buffer,bufref.buffer,bufref.size);
         cout << "Allokerar buffer storlek"
              << size << endl;
        }
     return *this;
}
Buffer:: operator const char *() const
{
     return buffer;
}

Buffer:: operator char *()
{
     return buffer;
}
void funk1()
{
     cout << "I funk1\n";
     Buffer *buffers = new Buffer[5];

     delete buffers;
}
void funk2()
{
     cout << "I funk2\n";
     Buffer *buffers = new Buffer[5];

     delete [] buffers;
}
main()
{
     funk1();
     funk2();
     return 0;
}
```

Resultat

Lägg märke till att vi har glömt ange **[]** i funk1. Detta medför att destruktorn endast anropas för det första objektet i fältet. I funk2 anger vi däremot **[]** vilket resulterar i att den funktionen städar heapen.



Placeringsoperator

Vi kan placera ett objekt på en godtycklig plats i minnet genom att använda en variant av new som går under beteckningen **placement** operator. Om vi gör det får vi inte använda delete på pekaren.

```
char buffert[sizeof(Anstelld)];

Anstelld *ptr = new (buffert) Anstelld;
...
ptr->input();
ptr->output();
```

Andra funktioner för dynamisk minneshantering Funktioner för dynamisk minneshanteringFunktioner för dynamisk minneshanteringFunktioner för dynamisk minneshantering

I C och C++ har vi alltid också tillgång till följande funktioner när vi vill använda oss av dynamiskt minne:

```
    * malloc
    * calloc
    * realloc
    * free
```

Det kan finnas fler funktioner som kan användas men ovanstående ingår enligt ANSI-C standarden och finns i praktiken även om du använder en C++-kompilator.

När vi använder C++ är ovanstående funktioner **inte** lämpliga för att direkt allokera plats för någon egendefinierad typ (class). Det är dock nödvändigt att du känner till dem för att förstå exempel som vi kommer att visa i det här kapitlet.

Vi kommer nämligen att visa hur man indirekt kan välja att låta **new-**operatorn använda ovanstående funktioner för minnesallokering. Om vi gör detta måste vi sedan låta **delete-**operatorn använda **free** för att frigöra minne.

Malloc.1 Malloc.1 Malloc.1 Malloc

Funktionen **malloc** är deklarerad på följande sätt:

```
void *malloc(size_t storlek);
```

Med funktionen malloc får vi tillgång till ett så stort minnessegment vi anger som argument (ett heltal). Som returvärde får vi ett pekarvärde (en adress) av typen **pekare till void**.

```
int  *intptr;
char *charptr;
charptr = (char *) malloc( 2000 * sizeof(char));
intptr = (int *) malloc( 2000 * sizeof(int));
```

Hur många byte som vi faktiskt får oss tilldelade blir då systemberoende men vi kan vara säkra på att vi angett rätt storlek för respektive typ.

Calloc.2 Calloc.2 Calloc.2 Calloc

calloc är en funktion som fungerar nästan på samma sätt som malloc. Den är deklarerad på nedanstående sätt. Lägg märke till att den tar två argument:

```
void *calloc( int  antal, size_t storlek);
```

calloc skiljer sig från **malloc** på följande sätt:

- calloc allokerar **antal** * **storlek** byte och hjälper således till med att multiplicera argumenten.

- calloc fyller det tilldelade minnessegmentet med nollor ('\0') vilket kan vara praktiskt i många sammanhang.

Precis som **malloc** returnerar **calloc** ett värde av typen **pekare till void**. Det är lämpligt att explicit typomvandla detta värde.

```
int  *intptr;
char *charptr;

charptr = (char *) calloc( 2000 , sizeof(char));
intptr = (int *)  calloc( 2000  , sizeof(int));
```

realloc.3 realloc.3 realloc.3 realloc

Med **realloc** kan du öka eller minska det minnessegment du förut fått dig tilldelat med calloc, malloc eller tidigare anrop av **realloc**. Som argument skall du använda en pekare till segmentet som skall ändra storlek samt den nya storleken:

```
   void *realloc( void * ptr , size_t nystorlek);
```

Vid användning av realloc garanteras att den del som finns kvar är oförändrad efter anropet.

```
int *ptr;

/* Allokera 2000 int...*/
ptr = (int * ) malloc( 2000 * sizeof(int));
    .
    .
/* Öka storleken ...*/
ptr = (int *) realloc( ptr , 4000 * sizeof(int));
    .
    .
/* Minska storleken...*/
ptr = (int *) realloc( ptr , 1000 * sizeof(int));
    .
    .
/* Frigör..*/
free(ptr);
```

Observera att argumentet till **realloc** måste vara ett värde som vi fått i retur från malloc, calloc eller realloc.

Free.4 Free.4 Free.4 Free

När vi hanterar dynamiskt minne måste vi komma ihåg att frigöra minne som vi inte längre behöver. Det finns oftast en gräns för hur mycket minne som finns tillhands. Det är när vi också frigör minne som vi kan börja tala om dynamik. Ett minnessegment som vi inte längre behöver kan vi frigöra med funktionen **free**. Den har nedanstående prototyp:

```
void free( void *ptr);
```

Argumentet till **free** måste vara ett pekarvärde som vi fått i retur från malloc, calloc eller realloc.

```
int  *intptr;
char *charptr;

intptr = (int * ) malloc( 2000 * sizeof(int));
    .
    .
    .
    .
free( intptr );


charptr = (char * ) malloc( 2000 * sizeof(char));
    .
    .
    .
    .
free( charptr );
```

I ovanstående exempel används **antagligen** samma minnessegment till två olika objekt vid olika tillfällen. Var minnet finns och hur detta hanteras vet vi antagligen inte och behöver oftast inte bry oss om så länge det räcker.

Felhantering.5 Felhantering.5 Felhantering.5 Felhantering och malloc,calloc och realloc

Du **skall** alltid kontrollera returvärdet från calloc, malloc och realloc. Det är inte säkert att det finns så mycket minne som vi begär. Om det inte finns så mycket minne som du begär så **allokeras inget minne** och funktionen returnerar värdet NULL (0). I dina program ska du därför använda någon av nedanstående metoder (stilar). Vad ett NULL-returvärde ska få för konsekvenser är sammanhangsberoende. Här väljer vi att avbryta programmet med ett returvärde som är skiljt från noll (t.ex. 1):

```
char *charptr;

intptr = (int * ) malloc( 2000 * sizeof(int));
if ( intptr == NULL)  {
   fprintf( stderr , " Slut på minne !\n");
   exit(1);
}
    .
    .
```

Vi kan också väva samman kontroll och tilldelning:

```
    .
    .
if ((charptr=(char*) malloc( 2000*sizeof(char))) == NULL)
{
         fprintf( stderr , " Slut på minne !\n");
         exit(1);
}
    .
    .
```

Defaultversioner av new och delete

Nedan ser vi ett exempel på hur new och delete-operatorn skulle kunna vara definierade. Exakt hur dessa funktioner ser ut är miljö- och leverantörsberoende. Lägg märker till placementsoperatorns enkelhet.

```
extern void (*_new_handler)();

// Default global newoperator...
void* operator new ( size_t storlek)
{
     if (storlek==0)
          storlek = 1;

     void *ptr = NULL;

     while (1){
     {
          ptr = malloc(storlek);
          if (_new_handler ==NULL || ptr!=NULL)
               break;
          (*_new_handler)();
     }
     return ptr;
}
// Default placementoperator
void* operator new(size_t,void* addr)
{
     return addr;
}
// Default deleteoperator
void operator delete(void* p)
{
     if (p!=NULL)
          free(p);
}
```

Egendefinierad överlagring av new och delete

Egen överlagring av globala newoperatorn

Du kan överlagra new och deleteoperatorn för allokering av minne för enstaka objekt. I en framtida C++-standard kommer man även att kunna överlagra både new och delete för fält. För närvande (dec-93) är det dock få C++-kompilatorer som stödjer överlagring av new och delete för fält.

Kompilatorn kommer att generera anrop av vår version av new med storleken på önskat minnessegment som parameter. Här följer ett exempel som skulle kunna användas för att alltid få minnessegmentet nollställt.

```
//Överlagring av new och delete för enstaka
//objekt
void *operator new(size_t size)
{
     if (size==0)
          size=1;
     return calloc(size,1);
}
```

Egen överlagring av globala deleteoperatorn

För säkerhets skull bör vi se till att minnet frigörs på ett sätt som matchar den metod som vi använde när vi allokerade minnet. Om vi alltid allokerar minne med **calloc** måste vi se till att det frigörs med **free.** Den globala deleteoperatorn kan bara överlagras på nedanstående sätt (i avseende på prototyp):

```
void operator delete(void *ptr)
{
     free(ptr);
}
```

Anrop av vår new och deleteoperator

Om vi sedan anropar new kommer våra egna funktioner att anropas. Kompilatorn kommer att se till att vår funktion anropas med det antal byte som skall reserveras. Vi behöver inte göra några ändringar för att det skall fungera.

```
// allokera minne,Vår newfunktion kommer att anropas
char *chptr = new char;
int *iptr   = new int;
Bankkund  *kundptr  = new Bankkund("Lisa Svensson");
Bankkonto *kontoptr = new Bankkonto(kundptr,"1234567-5");
....
// Använd objekt
...

// Frigör minne. Vår deletefunktion kommer att anropas
delete chptr;
delete iptr;
delete kundptr;
delete kontoptr;
...
```

Överlagring av globala placeringsoperatorn

Vi kan överlagra **placeringsversionen** av new. Denna skall definieras som en funktion som tar en storlek samt ytterligare en parameter av godtycklig typ. Här har vi överlagrat new för pekare av typen int*, double* och char*. Vi väljer här att alltid hantera **dynamisk minne** genom att returnera en pekare till **statiskt minne**.

```
void* operator new (size_t s, int*)
{
   static int faelt[100];
   if (s<sizeof(faelt))
       return faelt;
   else
       return NULL;
}
void* operator new (size_t s, char *)
{
   static char faelt[100];
   if (s<sizeof(faelt))
       return faelt;
   else
       return NULL;
}
void* operator new (size_t s, double *)
{
   static double faelt[100];
   if (s<sizeof(faelt))
       return faelt;
   else
       return NULL;
}
```

Anrop av våra placeringsoperatorer

Om vi anropar new med någon typ som matchar våra operatorer kommer dessa att anropas.

Det är viktigt att vi **inte anropar delete** eftersom detta antagligen skulle innebära att **free** får i uppgift att frigöra statiskt minne. Detta skulle kunna få fatala följder för vår program.

```
double *dptr;
dptr = new (dptr) double[5];
...
int  *ipttr;
iptr = new (iptr) int[5];
....
char *cptr;
cptr = new (cptr) char[5];
....
```

New och delete operatorn som klassmedlemmar

Newoperatorn som klassmedlem

Vi kan överlagra **new** som klassmedlem i vår egendefinerade klass. Vi kan överlagra både som en funktion som har samma prototyp som globala new och som placeringsoperator med godtycklig typ som andra parameter.

New-operatorn blir alltid en **static-**medlem. Det betyder att det inte finns någon **this**-pekare när vi befinner oss i funktionen. Och vi kommer därför inte heller åt några medlemmar i objektet som vi skall allokera minne för (det existerar inte ännu).

```
class Special {
public:
     Special();
     void* operator new(size_t s);      // Standardnew
     void  operator new(size_t s,char*); // Place.
private:
     ....
};
```

Deleteoperatorn som klassmedlem

Vi kan överlagra **delete** som klassmedlem i vår egendefinerade klass. Vi kan överlagra på två sätt. Antingen som funktion som tar en voidpekare **eller också** som en funktion som tar en voidpekare och ett heltal. Heltalet anger hur mycket minne som skall frigöras. Vi kan **inte** överlagra på bägge sätten samtidigt.

Deleteoperatorn blir alltid en **static-**medlem. Det betyder att det inte finns någon **this**-pekare när vi befinner oss i funktionen. Och vi kommer därför inte heller åt några medlemmar i objektet som vi skall allokera minne för (det existerar inte längre).

```
class Special {
public:
     Special();
     void* operator new(size_t s);      // Standardnew
     void  operator new(size_t s,char*); // Place.
     void  operator delete(void* p,size_t s);
//   void  operator delete(void* p)
private:
     ....
};
```

Exempel med egendefinierad minneshantering

Antag att vi vill låta en klass administrera användning av new på egen hand. Vi låter klassen ha en egen buffer i statiskt minne med plats för ANTAL objekt. När placeringsoperatorn används utnyttjas i första hand denna buffer.Ett objekt kan befinna sig på heap, statiskt eller på någon oidentifierad plats (stack,heap eller statiskt datasegment).

```
class Anstelld {
public:
     enum AnstelldMemory { STATIC ,HEAP,
                                     PLACEUNKNOWN};
     enum {ANTAL=500};
     Anstelld();
     void input();
     void output();
     void * operator new (size_t size);
     void * operator new(size_t size,
                               AnstelldMemory a );
     void  operator delete(void *ptr);
     int isstatic();
     int onheap();
     static int isstatic(Anstelld* ptr);
     static int onheap(Anstelld* ptr);
     static void freeallstatic();
     static int  morestatic();
private:
     char namn[30];
     char adresss[30];
     AnstelldMemory  whereis;
     static int nr;
     static Anstelld * const array;
     static const Anstelld * ptrs[ANTAL];
     static int heapflag;
};
```

AnstelldMemory är en enumtyp som används för att definera flaggor som kan användas för att placera objekt på rätt ställe.

ANTAL definierar hur många objekt som buffert skall rymma;

I klassens konstruktor **Anstelld()** kommer vi att initiera datamedlemmen **whereis** som är av typen **AnstelldMemory** och därför kommer att kunna anta något värdena **`STATIC ,HEAP`** `och` **`PLACEUNKNOWN.`**

Metoderna **input** och **output** är funktioner som har med själva datatypen att göra. I det

exemplet simulerar dessa in- och utmatning.

Operatorn **new** är överlagrad på vanligt sätt samt som en funktion som också tar ett argument av typen **AnstelldMemory** dvs en konstant av vår enumtyp.

Med metoderna **onheap** och **isstatic** kan vi fråga ett objekt om det ligger på heap eller är statiskt lagrat.

Med metoden **freeallstatic** frigör vi på ett snabbt sätt vår egen buffert.

Med **morestatic** kan vi fråga om det finns utrymme för fler objekt i vår buffert.

Attributen **namn** och **adress** är kopplat till vad det är för datatyp som vi har tänkt att hantera.

array är en pekare som vi kopplar till bufferten.

ptrs är ett fält med pekare som vi använder för att komma ihåg vilket minne som är blockerat.

heapflag används för att objekten i konstruktorn skall kunna veta att de t.ex. ligger på heapen.

nr kommer ihåg hur stor del av bufferten som är använd.

Definition av statiska datamedlemmar

Vi måste definiera våra staticmedlemmar utanför klassen. Vi definierar även **buffert1** som är den buffert som vi använder i första hand när vi skall placera objekten med **placeringsoperatorn**.

```
static
char buffert1[Anstelld::ANTAL*sizeof(Anstelld)];
Anstelld* const
Anstelld::array = (Anstelld*) buffert1;
const Anstelld * Anstelld::ptrs[Anstelld::ANTAL];
int Anstelld::heapflag = 0;
int  Anstelld::nr = 0;
```

Definition av metoder

I konstruktor bestämmer vi objektets placering.

```
Anstelld::Anstelld()
{
     switch (heapflag)
         {
     case HEAP  :  whereis = HEAP;
                   break;
     case STATIC:  whereis = STATIC;
                   break;
     default    :
                   whereis = PLACEUNKNOWN;
                   break;
     }
}
```

Metoderna **input** och **output** kommer att tills vidare vara tomma.

```
void Anstelld::input()
{
}
void Anstelld::output()
{
}
```

Metoderna **isstatic** och **onheap** returnerar sant eller falskt.

```
int Anstelld::isstatic()
{
     return whereis == STATIC;
}
int Anstelld::onheap()
{
     return whereis == HEAP;
}
int Anstelld::isstatic(Anstelld* p)
{
     return p->whereis == STATIC;
}

int Anstelld::onheap(Anstelld* p)
{
     return p->whereis == HEAP;
```

```
}
```

Definition av våra egendefinerade newoperatorer

I den "vanliga" newoperatorn sätter vi en flagga och anropar sedan **globala new.**

```
void * Anstelld::operator new (size_t size)
{
     heapflag = HEAP;
     return ::operator new(size);
}
```

I vår egendefinerade placeringsoperator undersöker vi om det finns plats i vår buffert. Annars placerar vi objektet på heapen.

```
void * Anstelld::operator new(size_t size,
                           AnstelldMemory where )
{
     void *returnptr = NULL;

     switch (where)
     {
     case STATIC:    if (nr<ANTAL) // Snabballokering
                     {
                          heapflag = STATIC;
                          ptrs[nr++] = &array[nr];
                          returnptr = &array[nr];
                     }
                     else
                     {    // Hitta en plats...
                          for(  int i=0;
                               ptrs[i]!=NULL &&
                     i<ANTAL;
                               i++)
                          {
                          }
                          if (i!=ANTAL) // Finns plats
                          {
                             heapflag = STATIC;
                             ptrs[i]  = &array[i];
                             returnptr =   &array[i];
                          }
                          else
                          {  // Placera på heap
                             heapflag = HEAP;
                             returnptr =  ::operator
                                         new(size);
                          }
                     }
                     break;
     case HEAP :     returnptr = ::operator new(size);
                     heapflag = HEAP;
                     break;
     default:
                     returnptr = NULL;
                     break;
     }
     return returnptr;
}
```

I vår egendefinierade **deleteoperator** undersöker vi om

```
minnet som skall frigöras finns i vår egen buffert.
Annars antar vi att det ligger på heapen.


void  Anstelld::operator delete(void *ptr)
{
     for(int i=0;ptrs[i]!= ptr && i<ANTAL;i++)
     {
     }
     if (i!=ANTAL)
      {
          ptrs[i] = NULL;
                 nr--;
      }
     else
      {
          ::operator delete (ptr);
      }
}
```

I **freeallstatic** NULL-ställer vi snabbt våra pekare och i **morestatic** kontrollerar vi om det finns någon ledig pekare och returnerar 0 eller 1;

```
void Anstelld::freeallstatic()
{
     for(int i=0;i<ANTAL;i++)
          ptrs[i] = NULL;
        nr = 0;
}
int Anstelld::morestatic()
{
     for(int i=0;i<ANTAL;i++)
         {
          if (ptrs[i]==NULL)
               return 1;
     }
     return 0;
}
```

Exempel: newplac1.cpp

Nu kan vi pröva vår klass. Vi börjar med ett exempel där vi allokerar plats för objekt på olika sätt. Vi frågar sedan objekten var de befinner sig. Lägg märke till att om hela vår buffert används kommer nästa objekt att placeras på heap.

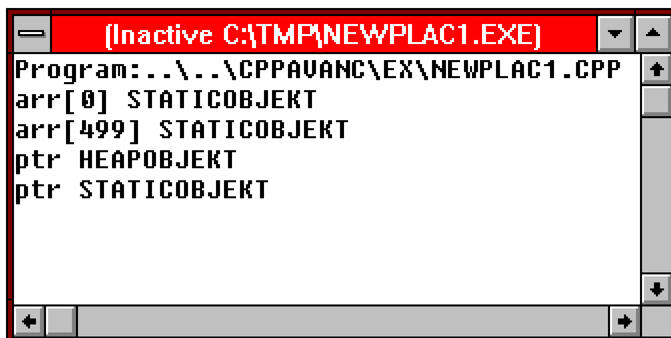
```
main()
{
      cout << "Program:" << __FILE__ << endl;
      Anstelld *arr[Anstelld::ANTAL];

      for(int i=0;i<Anstelld::ANTAL;i++)
            arr[i] = new (Anstelld::STATIC) Anstelld;

      if (arr[0]->onheap())
            cout << "arr[0] HEAPOBJEKT\n";
      else
            if (arr[0]->isstatic())
                  cout << "arr[0] STATICOBJEKT\n";
            else
                  cout << "ptr UNKNOWN\n";

      if (arr[Anstelld::ANTAL-1]->onheap())
            cout  << "arr[" << (Anstelld::ANTAL-1)
                  << "] HEAPOBJEKT\n";
      else
            if (arr[Anstelld::ANTAL-1]->isstatic())
                  cout  << "arr[" << (Anstelld::ANTAL-1)
                        << "] STATICOBJEKT\n";
            else
                  cout << "ptr UNKNOWN\n";

      Anstelld *ptr = new (Anstelld::STATIC) Anstelld;
      if (ptr->onheap())
            cout << "ptr HEAPOBJEKT\n";
      else
            if (ptr->isstatic())
                  cout << "ptr STATICOBJEKT\n";
            else
                  cout << "ptr UNKNOWN\n";
      delete ptr;

      for(i=0;i<Anstelld::ANTAL;i++)
            arr[i]->input();

      for(i=0;i<Anstelld::ANTAL;i++)
            arr[i]->output();

      for(i=0;i<Anstelld::ANTAL;i++)
            delete arr[i];
```

```
        ptr = new (Anstelld::STATIC) Anstelld;
        if (ptr->onheap())
              cout << "ptr HEAPOBJEKT\n";
        else
              if (ptr->isstatic())
                    cout << "ptr STATICOBJEKT\n";
              else
                    cout << "ptr UNKNOWN\n";
        delete  ptr;

        return 0;
}
```

Genom att vi överlagrar **new** och **delete** kan vi kontrollera att vår buffert används. Om den inte räcker till så använder vi heapen.

Resultat

Objekten vet om de ligger på heapen eller är lagrade i statiskt minne.



Exempel:newplac2.cpp

I nästa exempel simulerar vi användning av vår egen buffert samt användning av heap och gör en utvärdering av hur det påverkar prestanda. Först simulerar vi **intensiv slumpvis** användning av new, delete och placeringsoperatorn.

```
// Funktion som genomgående allokerar objekt med
placementoperatorn
void funkstatic()
{
   Anstelld *arr[Anstelld::ANTAL];

   for(int i=0;i<Anstelld::ANTAL;i++)
       arr[i] = new (Anstelld::STATIC) Anstelld;
   for(i=0;i<Anstelld::ANTAL;i++)
       arr[i]->input();
   for(i=0;i<Anstelld::ANTAL;i++)
       arr[i]->output();
   for(i=0;i<Anstelld::ANTAL/2;i++)
   {
       int slump = rand()%Anstelld::ANTAL;
       if (arr[slump]!=NULL)
       {
          delete arr[slump];
          arr[slump] = NULL;
       }
   }
   for(i=0;i<Anstelld::ANTAL;i++)
   {
       if (arr[i]==NULL)
       {
          arr[i] = new (Anstelld::STATIC)
                                       Anstelld;
          arr[i]->input();
       }
   }
   for(i=0;i<Anstelld::ANTAL;i++)
          arr[i]->output();
   for(i=0;i<Anstelld::ANTAL;i++)
       delete arr[i];
}
```

Sedan gör vi en matchande funktion som i stället konsekvent använder heap.

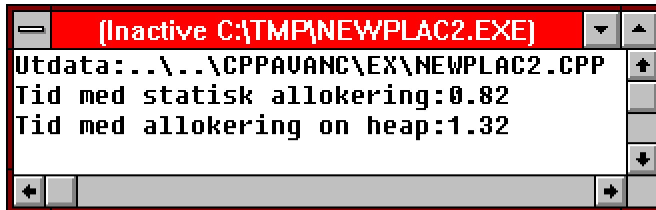
```
// Funktion som genomgående allokerar objekt på heapen
void funkheap()
{
   Anstelld *arr[Anstelld::ANTAL];

   for(int i=0;i<Anstelld::ANTAL;i++)
       arr[i] = new Anstelld;
   for(i=0;i<Anstelld::ANTAL;i++)
       arr[i]->input();
   for(i=0;i<Anstelld::ANTAL;i++)
       arr[i]->output();
   for(i=0;i<Anstelld::ANTAL/2;i++)
   {
       int slump = rand()%Anstelld::ANTAL;
       if (arr[slump]!=NULL)
       {
           delete arr[slump];
           arr[slump] = NULL;
       }
   }
   for(i=0;i<Anstelld::ANTAL;i++)
   {
       if (arr[i]==NULL)
       {
           arr[i] = new Anstelld;
           arr[i]->input();
       }
   }

   for(i=0;i<Anstelld::ANTAL;i++)
           arr[i]->output();

   for(i=0;i<Anstelld::ANTAL;i++)
           delete arr[i];
}
```

Till sist anropar vi våra två funktioner och gör en utvärdering genom att ta tid på respektive funktionsanrop.

```
#include <stdlib.h>
#include <sys/timeb.h>
main()
{
   cout << "Utdata:" __FILE__ "\n";
   srand(0);
   timeb start,stopp;

   ftime(&start);
   funkstatic();
   funkstatic();
   funkstatic();
   funkstatic();
   funkstatic();
   funkstatic();
   ftime(&stopp);
   cout << "Tid med statisk allokering:"
        << ((stopp.time+stopp.millitm/1000.0) -
           (start.time+start.millitm/1000.0))
        << endl;

   srand(0);
   ftime(&start);
   funkheap();
   funkheap();
   funkheap();
   funkheap();
   funkheap();
   funkheap();
   ftime(&stopp);
   cout << "Tid med allokering on heap:"
        << ((stopp.time+stopp.millitm/1000.0) -
           (start.time+start.millitm/1000.0))
        << endl;

   return 0;
}
```

Resultat

Vi kan konstatera att i den miljö som programmet kördes denna gång var det mycket effektivare att administrera heapanvändning på egen hand.



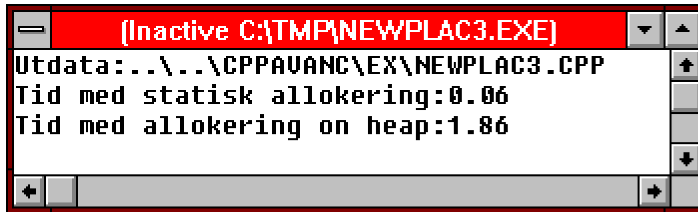
Exempel:newplac3.cpp

Vi skriver sedan om funktionerna och allokerar och frigör objekten i sekvens dvs en annan typ av användning.

```
void funkstatic()
{
    Anstelld *arr[Anstelld::ANTAL];
    for(int i=0;i<Anstelld::ANTAL;i++)
         arr[i] = new (Anstelld::STATIC) Anstelld;
    for(i=0;i<5;i++)
        {
         Anstelld::freeallstatic();
         for(i=0;i<Anstelld::ANTAL;i++)
              arr[i] = new (Anstelld::STATIC) Anstelld;
        }
    Anstelld::freeallstatic();
}
void funkheap()
{
    Anstelld *arr[Anstelld::ANTAL];

    for(int i=0;i<Anstelld::ANTAL;i++)
         arr[i] = new  Anstelld;
    for(i=0;i<5;i++)
        {
         for(i=0;i<Anstelld::ANTAL;i++)
              delete arr[i];
         for(i=0;i<Anstelld::ANTAL;i++)
              arr[i] = new Anstelld;
    }
    for(i=0;i<Anstelld::ANTAL;i++)
         delete arr[i];
}
```

Resultat

Vi kan konstatera att om vi allokerar minne på detta sätt så vinner vi ändå mer på att implementera egna newfunktioner.



Exempel: Egendefinierad minnesanvändning för avlusning

En klass för minnesadministration: klassen Memory

I nästa exempel kommer vi att visa hur man kan kontrollera dynamisk minnesanvändning genom att överlagra **globala new** och **delete**-operatorn. Vi inför klassen **Memory** som kommer att användas för minneskontroll.

```
// memory.h
#ifndef MEMORY_H
#define MEMORY_H
#define MSIZE 1000
class Memory {
public:
    static int registrera(void *ptr, size_t size);
    static int registrera_namn(void *ptr,const char *name);
    static int avregistrera(void *ptr);
    static void dump(const char *where="");
private:
    static struct MemLogg  {
         const char *name;
         void *ptr;
         size_t size;
    } marray[MSIZE];
    static long minne;
};
#endif
```

Klassen **Memory** håller en tabell med **MSIZE** stycken objekt av typen **MemLogg** . Dessa innehållet en pekare till ett minnessegment som allokerats på heapen (**ptr**) samt storleken på respektive segment (**size**). Dessutom finns möjlighet att koppla segmentet till en förklarande text (**name**).

Med klassattributet **minne** kommer vi att hålla reda på hur mycket minne vår applikation har allokerat på heapen.

Med metoden **registrera** registreras ett minnessegment med en pekare och storlek i tabellen.

Med metoden **registrera_namn** kopplar vi segmentet till en text.

Med metoden **dump** skrivs tabellen ut på standard error.

Definition av static datamedlemmar i klassen Memory

```
long Memory::minne = 0;
MemLogg Memory::marray[MSIZE];
```

Definition av static funktionsmedlemmar i klassen Memory

När vi registrerar minne söker vi upp en ledig plats i tabellen.

```
int Memory::registrera(void *ptr, size_t size)
{
       minne+=size;
       for(int i=0;i<MSIZE;i++)
       {
            if (marray[i].ptr==NULL)
            {
                  marray[i].ptr  = ptr;
                  marray[i].size = size;
                  break;
            }
       }
       if (i==MSIZE)
            return 0;
       else
            return 1;
}
```

När vi registrerar namn söker vi upp pekaren i vår tabell. Hittar vi den så sätter vi **name**-pekaren till det föreslagna namnet.

```
int Memory::registrera_namn(void *ptr,const char *name)
{
       for(int i=0;i<MSIZE;i++)
       {
            if (marray[i].ptr==ptr)
                 {
                  marray[i].name  = name;
                  break;
            }
       }
       if (i==MSIZE)
            return 0;
       else
            return 1;
}
```

När vi avregistrerar ett segment söker vi upp pekaren i tabellen och sätter den till NULL. Dessutom minskar vi uppgiften om hur mycket minne som allokerats.

```
int Memory::avregistrera(void *ptr)
{
       for(int i=0;i<MSIZE;i++)
       {
            if (marray[i].ptr==ptr)
            {
                  marray[i].name  = NULL;
                  marray[i].ptr  = NULL;
                  minne-= marray[i].size;
                  marray[i].size = 0;
                  break;
            }
       }
       if (i==MSIZE)
            return 0;
       else
            return 1;
}
```

I metoden **dump** listar vi hela tabellen. Vi presenterar också en godtycklig text så att det kan framgå vi vilket tillfälle som dumpen är gjord.

```
void Memory::dump(const char *where)
{
       cerr << "<<<<" << where
            << ":Allokerat Minne:" << minne
            << ">>>>" << endl;
       for(int i=0;i<MSIZE;i++)
       {
            if (marray[i].ptr!=NULL)
            {
                  cerr << "På adress:" << marray[i].ptr <<" "
                       << "Size      :" << marray[i].size;
                  if (marray[i].name!=NULL)
                  {
                        cerr << "   " << "Objekt:"
                             << marray[i].name;
                  }
                  cerr << endl;
            }
       }
}
```

Vi har också överlagrat globala new och deleteoperatorn. Genom att vi har gjort detta kommer all användning av new och delete att loggas. I newoperatorn registrerar vi minne och i delete avregisteras det.

```
void * operator new ( size_t storlek)
{
      if (storlek==0)
            storlek = 1;
      void *ptr = calloc(storlek,1);
      if (ptr==NULL)
      {
            if (_new_handler!=NULL)
            {
                  (*_new_handler)();
                  ptr = calloc(storlek,1);
            }
      }
      if (ptr!=NULL)
      {
            Memory::registrera(ptr,storlek);
      }

      return ptr;
}
void operator delete ( void *ptr)
{
      Memory::avregistrera(ptr);
      free(ptr);
```

```
}
```

MemoryBase, en basklass för klasser som skall använda klassen Memory

Vi kommer nu att visa hur man kan designa klasser som skall använda klassen **Memory**. Vi samlar grundläggande egenskaper i klassen **MemoryBase**. Denna klass kan sedan användas som basklass av klasser som skall ha automatisk registrering.

```
#ifndef MEMBASE_H
#define MEMBASE_H
#include "memory.h"
class MemoryBase {
public:
    MemoryBase();
    MemoryBase(const MemoryBase& m);
    virtual ~MemoryBase();
    void* operator new (size_t storlek);
    void* operator new (size_t storlek, char *buffer);
    void operator delete (void* ptr);
    void dump();
    int isheap() { return dynamic;}
    static int antal()  { return ant;}
    virtual const char *ClassName()
    { return "class MemoryBase";}
    static void setclassname(const char* name)
    { classname=name;}
protected:
    static const char *classname;
private:
    int dynamic;
    static int mflag;
    static int ant;
};
#endif
```

Definition av static datamedlemmar i klassen MemoryBase

mflag flaggar för om objektet ligger på heapen. **ant** håller reda på hur många **MemoryBase**-objekt som det finns. **classname** används för att subklasserna skall kunna berätta för basklassens **new-**operator vilket namn som gäller för subklassen. Om denna pekare är satt till något annat än en ""-sträng kommer klassnamnet att registreras.

```
int MemoryBase::mflag = 0;
int MemoryBase::ant = 0;
const char *MemoryBase::classname = "";
```

Definition av funktionsmedlemmar i klassen MemoryBase

```
MemoryBase::MemoryBase()
: dynamic(mflag?1:0)
{
   if (mflag)
   {
       if (classname[0]=='\0')
               classname = "class MemoryBase";
       Memory::registrera_namn(this, classname);
       classname = "";
       mflag=0;
   }
   ant++;
}
MemoryBase::MemoryBase(const MemoryBase& m)
: dynamic(mflag?1:0)
{
   ant++;
}
MemoryBase::~MemoryBase()
{
   ant--;
   if (dynamic)
       mflag =1;
}
void* MemoryBase::operator new (size_t storlek)
{
   if (storlek==0)
       storlek = 1;
   mflag = 1;
   void *ptr = ::operator new(storlek);
   return ptr;
}
void* MemoryBase::operator new (size_t storlek, char
*buffer)
{
   return buffer;
}
void MemoryBase::operator delete (void* ptr)
{
   if (mflag)
   {
       ::operator delete(ptr);
       mflag = 0;
   }
}
```

I **dump** berättar ett objekt om det är ett heapobjekt, på vilken adress det ligger samt vad det är av för typ:

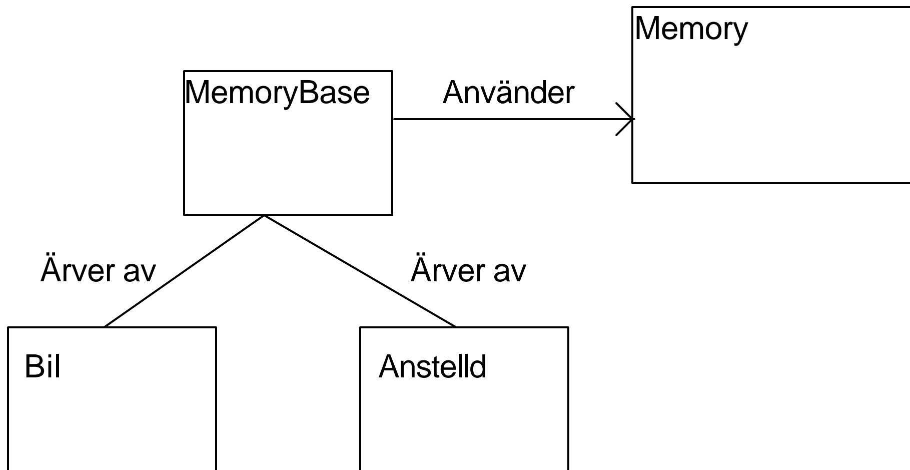
```
void MemoryBase::dump()
{
   if (dynamic)
       cerr << "Heapobjekt    ";
   else
       cerr << "Ej Heapobjekt ";
   cerr << ClassName() << ":" <<  this << endl;
}
```

Definiera subklasser till MemoryBase

Vi kan sedan definiera subklasser till **MemoryBase**. Dessa måste registrera sig själva enligt ett visst protokoll:

1. De måste definiera metoden **ClassName**
2. De måste överlagra **new**, registera sitt klassnamn i den funktionen samt anropa **MemoryBase** variant av new.

```
class Anstelld : public MemoryBase {
public:
   Anstelld(char *n="") { strcpy(namn,n);}
   virtual const char *ClassName()
   { return "class Anstelld";}
   void * operator new (size_t s)
   {
       MemoryBase::setclassname("class Anstelld");
       return MemoryBase::operator new(s);
   }
private:
   char namn[100];
};
class Bil : public MemoryBase {
public:
   Bil(char *n="",long m):mil(m)
   { strcpy(namn,n);}
   virtual const char *ClassName()
   { return "class Anstelld";}
   void * operator new (size_t s)
   {
       MemoryBase::setclassname("class Bil");
       return MemoryBase::operator new(s);
   }
private:
   char namn[80];
   long mil;
};
```



Användning av Memory och MemoryBase

Nu kan vi pröva våra klasser. Vi dumpar vår minnestabell när vi går in in main, mitt i main och innan vi går ur main.

```
main()
{
     set_new_handler(minnesfel);

     Memory::dump("Main 1");

     double *p = new double[1000];
     Memory::registrera_namn(p," double[1000]");

     Anstelld lokalt("LOKALT");
     Anstelld *aptr  = new Anstelld("HEAP");
     Anstelld *aptr2 = new Anstelld[10];

     Bil *bptr = new Bil("Volvo",7000);

     Memory::dump("Main 2");

     delete p;
     delete aptr;
     delete [] aptr2;
     delete bptr;

     Memory::dump("Main 3");

    return 0;
}
```

Resultat

Det visar sig att vi har bra kontroll på användning av dynamiskt minne. Åtminstone det som vi själva använder.

Fråga: **Hur kommer det sig att det redan finns sex segment som är registerade när vi kommer in i main?**