Dynamischer Speicher

- malloc wird in c++ eher nicht mehr verwendet.
- Speicher wird aus dem free store ausgefasst.
- Objekte werden mit new angelegt.
- new stellt Speicher bereit und ruft den passenden Constructor auf.

- new ist ein Operator.

```
btime *pt1=new btime;
btime *pt2=new btime(11,55);
cout<< "pbt1: ";
pt1->btimeShow();
cout<<endl;
cout<< "pbt2: ";
pt2->btimeShow();
cout<<endl;
delete p2; delete pt2;
```

Dynamischer Speicher

- Auch Variablen oder Arrays der Standarddatenypen können mit new angelegt werden.

```
int length;
char *Ptc;

Ptc = new char[length];
. . .
delete []Ptc;
```

Dynamischer Speicher

- Über new können auch Arrays von Objekten angelegt werden.
- Diese Arrays werden mit dem Defaultconstructor initialisiert.

```
btime* btx=new btime[2];
cout<< "btx[0]: ";
btx[0].btimeShow();
cout<<endl;
cout<< "btx[1]: ";
btx[1].btimeShow();
cout<<endl;
delete[]btx;
```

Dynamischer Speicher

- Das Pendant zu new ist der Operator delete.
- delete gibt den Speicher zurück und ruft ggf. den Destructor auf.
- Wird delete auf ein Array angewendet, so sind Klammern [ ] zu ergänzen.

- Delete auf den Nullpointer angewandt bleibt ohne Wirkung, es gibt keinen Fehler/Absturz.

```
delete[]btx;
delete pt1;
delete pt2;
```

Dynamischer Speicher

- Steht der mit new angeforderte Speicherplatz nicht zur Verfügung, so wird in Analogie zu malloc ein NULL-Pointer zurückgegeben.
- Mit der Funktion set_new_handler kann eine benutzerdefinierte Funktion eingestellt werden, die ausgeführt wird, wenn der angeforderte Speicher nicht verfügbar ist.
- Nach Ausführung der Behandlungsfunktion wird erneut versucht, den Speicher zu allokieren.
- set_new_handler(NULL) erzeugt eine bad-alloc-Exception, die das Programm beendet.

Dynamischer Speicher

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

void NotEnoughMem ()
{
  cerr<<"no space left..."<<endl;
  set_new_handler(NULL);
  exit(-1);
}

int main()
{
  int*p;
  set_new_handler(NotEnoughMem);
  p=new int[1000000000ul];
  return 0;
}
```

Klassen mit Pointermembern

- Enthält eine Klasse Pointer als Member, so sind Besonderheiten zu beachten.
- Als Beispiel gelte eine Klasse cstring, die eine Zeichenkette im Sinne von C verwalte.
- Durch Verwendung dieser Klasse soll die Zeichenkettenarbeit vereinfacht und sicherer werden.
- Die Klasse cstring enthalte als Member einen Pointer auf die Zeichenkette und die Länge ohne terminierende 0.

Klassen mit Pointermembern

```
class cstring
{
  public:
    cstring (){ pbuf  = NULL; len = 0;}
    cstring (const char* pbuf);
    cstring (const char Chr,const int len=1);

    int     getLen  ()const{return len;}
    char*   getPbuf ()const{return pbuf;}
    . . .

  private:
    char * pbuf;
    int    len;
};
```

Klassen mit Pointermembern

```
cstring :: cstring (const char * pbufP)
{
  if (pbufP)
  {
    len   = strlen(pbufP);
    pbuf = new char[len+1];
    strcpy(pbuf, pbufP);
   } else {pbuf=NULL; len=0;}
}

cstring :: cstring (const char Chr, const int
lenP)
{
  Len  = lenP;
  pbuf = new char[len+1];
  memset(pbuf,Chr,len);
  pbuf[len]='\0';
}
```

Klassen mit Pointermembern

```
cstring :: cstring (const cstring *Other)
{
  len   = Other->len;
  pbuf = new char[len+1];
  strcpy(pbuf, Other->pbuf);
}
```

- In den Constructoren wird der Speicher für die Zeichenkette einschl. der teminierenden 0 bereitgestellt und die Zeichenkette hineingeschrieben.

Klassen mit Pointermembern

- Wird nun ein Objekt vernichtet, so wird der Pointer pbuf auch vernichtet, der ausgefasste Speicher bleibt aber erhalten und kann nicht mehr freigegeben werden.
- Den Ausweg bildet ein Destructor, er kann dafür sorgen, dass Ressourcen, die über das Objekt in Anspruch genommen und verwaltet werden, korrekt zurückgebaut werden.

```
cstring :: ~cstring ()
{
  delete []pbuf;
}
```

Destructor, er wird bei Abbau des Objektes automatisch aufgerufen

Klassen mit Pointermembern

- Ein weiteres Problem ergibt sich wenn ein Objekt mit einem Objekt der eigenen Klasse initialisiert werden soll.
- Dieser Fall kommt vor, wenn ein Objekt als Parameter an eine Funktion übergeben oder als Returnwert zurückgegeben wird, wenn ein Objekt 1:1 kopiert wird.
- In diesem Fall würde der Pointer unserer Stringklasse auch kopiert, beim Abbau der Kopie würde der Destructor aufgerufen und der Speicher freigegeben.

Klassen mit Pointermembern

- In der Folge enthielte das originale Objekt einen Pointer auf Speicher, der nicht mehr valide ist – es käme zu fehlerhaftem Programmverhalten.
- Ausweg ist ein spezieller Constructor, der Copyconstructor. Er übernimmt als Parameter eine **Referenz auf das zu kopierende Objekt** und legt ein komplett neues Objekt an, bei dem neuer Speicher (hier für die Zeichenkette) ausgefasst wird.

Klassen mit Pointermembern

- Ein ähnliches Problem ergibt sich bei der Zuweisung von Objekten mit Pointermembern.
- Bei der Zuweisung von Objekten wird Member für Member kopiert. Das betrifft auch Member, die Pointer sind.
- Somit haben wir wieder zwei Objekte, die ein und den selben Speicher verwenden, was zum Fehler beim Löschen des Speichers beim Abbau eines der Objekte führt.
- Hier schafft eine zu programmierende Operatorfunktion Abhilfe.

Klassen mit Pointermembern

```
class cstring
{
  public:
    cstring (){ pbuf  = 0; len = 0;}
    cstring (const char* pbuf);
    cstring (const char Chr,const int len=1);
    cstring (const cstring *Other);
    cstring (const cstring &Other);
   ~cstring (); // Destructor
    cstring& operator= (cstring &other);

    int     getLen  ()const{return len;}
    char*   getPbuf ()const{return pbuf;}

  private:
    char * pbuf;
    int    len;
};
```

Klassen mit Pointermembern

Merke

- Verwaltet eine Klasse Ressourcen mit Hilfe von Pointern, so sind drei spezielle Funktionen nötig:

1. Copyconstructor
2. Überladenener Zuweisungsoperator
3. Destructor

Klassen mit Pointermembern

```
cstring :: cstring (const cstring &Other)
{
  len   = Other.len;
  pbuf = new char[len+1];
  strcpy(pbuf, Other.pbuf);
}
cstring& cstring :: operator= (cstring & other)
{
    char*tmp;
    tmp=new char(other.len+1);
    strcpy(tmp,other.pbuf);
    delete []pbuf;
    len=other.len;
    pbuf=tmp;
    return *this;
}
cstring :: ~cstring ()
{  delete []pbuf; }
```

Soll auch funktionieren für:
a=a;
Oder
a=b=c;

Der MoveConstructor

- Erzeugt eine Methode ein neues Objekt, das als Returnwert zurückgegeben wird, so wird dieses als lokale Variable erzeugt dann auf den Stack als temporäres Objekt kopiert. Das originale Objekt wird dann beim Verlassen vernichtet. Sodann wird das temporäre Objekt auf das Ziel kopiert und danach das temporäre Objekt vernichtet.
- Bei der Parameterübergabe eines Objektes geschieht ähnliches.

Der MoveConstructor

- Ein Moveconstructor überträgt die Member des eigenen Objektes auf ein neues Objekt. Die Member des eigenen Objektes werden gelöscht.
- Als Parameter erhält der MoveConstructor eine RValueReferenz (&&).
- Es handelt sich um eine Referenz, die nicht als LValue auftreten kann.
- Der Aufruf erfolgt explizit über move.
- Compilieren ggf mit: -std=c++11 oder -std=c++0x

Klassen mit Pointermembern

```
// Move Constructor (ab C++11)
CString :: CString (CString && Other)
{
  pbuf=Other.pbuf;
  Len  =Other.Len;
  Other.pbuf=nullptr;
  Other.Len  =0;
  cout<< "Move Called"<<endl;
}

CString CString :: concat (const CString & Other)
{
  char * PtTmp;
  Len = Len + Other.Len + 1;
  PtTmp = new char[Len];
  strcpy (PtTmp,pbuf);
  strcat (PtTmp,Other.pbuf);
  return move(PtTmp);
}
```

Hier wird zunächst der Constructor aufgerufen, der aus const char* ein Objekt baut. Der move-Constructor legt dieses dann auf den Stack zur weiteren Verarbeitung.

Der MoveConstructor

- Die Funktion concat verkettet den String (this) mit dem als Parameter übergegeben String.
- Es entsteht dabei ein neues Stringobjekt, dass als Kopie über den Stack zurückgegeben wird.
- Über den Move-Constructor wird das mehrfache Ausfassen und gleich wieder Freigeben von Speicher vermieden.
- Der Aufruf könnte wie folgt lauten:

```
CString sx=move(s1.concat(" & ").concat(s2));
```