

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;
```

Dynamischer Speicher

- Auch Variablen oder Arrays der Standarddatentypen 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 allokkieren.
- `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 Pointermembere

- Enthält eine Klasse Pointer als Member, so sind Besonderheiten zu beachten.
- Als Beispiel gelte eine Klasse cstring, die eine Zeichenkette im Sinne von C verwaltet.
- 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 Pointermembemern

```
class cstring
{
    public:
        cstring () { PtStr = NULL; Len = 0; }
        cstring (const char* PtStr);
        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 Pointermembnern

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

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

Klassen mit Pointermembemern

```
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 terminierenden 0 bereitgestellt und die Zeichenkette hineingeschrieben.

Klassen mit Pointermembemern

- 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 Pointermembemern

- 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 Pointermembemern

- 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 Pointermembemern

- Ein ähnliches Problem ergibt sich bei der Zuweisung von Objekten mit Pointermembemern.
- 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 referenzieren, was zum Fehler beim Löschen des Speichers beim Abbau eines der Objekte führt.
- Hier schafft eine zu programmierende Operatorfunktion Abhilfe.

Klassen mit Pointermembemern

```
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 Pointermembemern

Merke

- Verwaltet eine Klasse Ressourcen mit Hilfe von Pointern, so sind drei spezielle Funktionen nötig:
 1. Copierconstructor
 2. Überladenener Zuweisungsoperator
 3. Destructor

Klassen mit Pointermembemern

```
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;