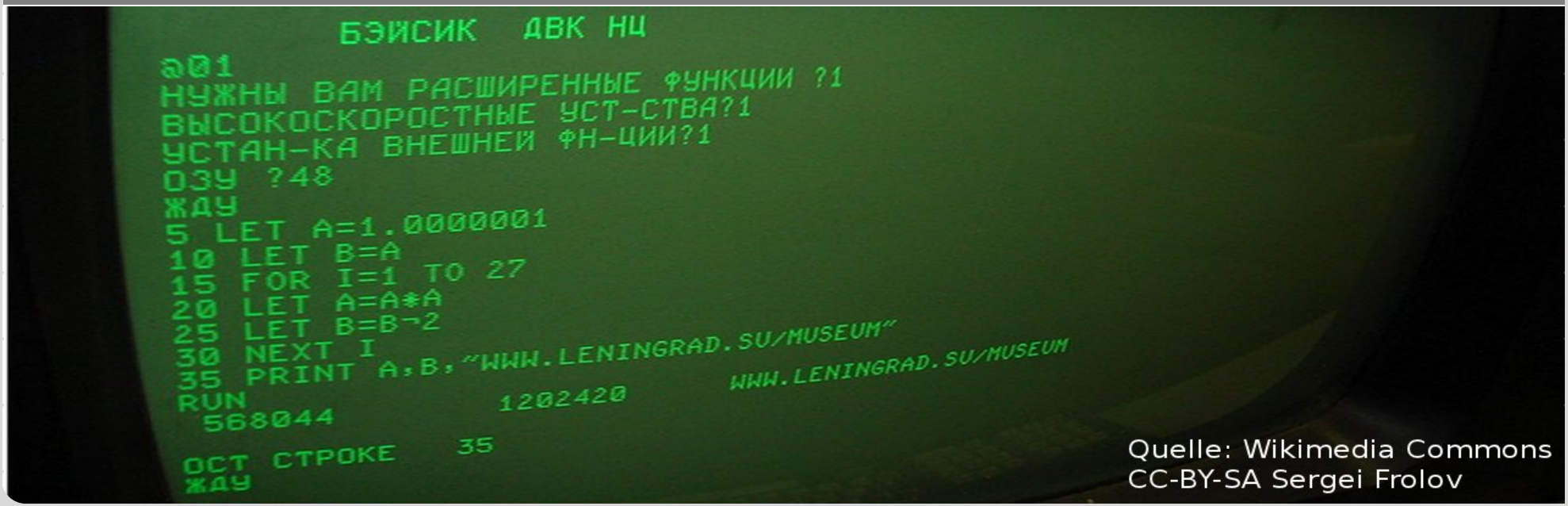
Karlsruher Institut für Technologie

C++ Workshop

2. Block, 04.05.2012

Robert Schneider, Sven Brauch | 4. Mai 2012



Quelle: Wikimedia Commons
CC-BY-SA Sergei Frolov

Gliederung

1. ≫Dinge≪, Referenzen und Pointer
   - ≫Dinge≪ und Speicher
   - Referenzen
   - Pointer
   - Arrays
2. Stack und Heap
   - Stack
   - Heap
3. Objektorientierte Programmierung

1. ≫Dinge≪, Referenzen und Pointer
   - ≫Dinge≪ und Speicher
   - Referenzen
   - Pointer
   - Arrays

≫Dinge≪

Das grundlegende Konzept in C++ nennt sich im Standard „object“. Hinsichtlich Java und Objektorientierung aber missverständlich! Wir nennen es daher ≫Ding≪.

≫Dinge≪

Das grundlegende Konzept in C++ nennt sich im Standard „object“. Hinsichtlich Java und Objektorientierung aber missverständlich! Wir nennen es daher ≫Ding≪.

Ein Beispiel

```
int foo;
```

Es gibt jetzt ein ≫Ding≪ mit dem Namen `foo` und dem Typen `int`.

Definition: ≫Ding≪

Standard, 1.8

Ein ≫Ding≪

1. ist ein Speicherbereich, aber *keine* Funktion (auch wenn diese Speicher belegt!).

Definition: ≫Ding≪

Standard, 1.8

Ein ≫Ding≪

1. ist ein Speicherbereich, aber *keine* Funktion (auch wenn diese Speicher belegt!).
2. hat eine Speicherdauer, einen Typen und *kann* einen *Namen* haben.

Definition: ≫Ding≪

Standard, 1.8

Ein ≫Ding≪

1. ist ein Speicherbereich, aber *keine* Funktion (auch wenn diese Speicher belegt!).
2. hat eine Speicherdauer, einen Typen und *kann* einen *Namen* haben.
3. hat eine Größe von einem oder mehr Bytes (abgesehen von bit-fields).
4. von ≫einfachem≪ (POD) Typ besetzt eine zusammenhängende Menge Bytes.
5. wird durch eine Definition, den `new`-Ausdruck oder vom Compiler erzeugt.

Speicher

Standard, 1.7

- Die fundamentale Speicher-Einheit im C++ Speichermodell ist das *Byte*.

Speicher

Standard, 1.7

- Die fundamentale Speicher-Einheit im C++ Speichermodell ist das *Byte*.
- Was ein Byte ist, ist recht abstrakt definiert.

Speicher

Standard, 1.7

- Die fundamentale Speicher-Einheit im C++ Speichermodell ist das *Byte*.
- Was ein Byte ist, ist recht abstrakt definiert.
- Der Speicher, welcher einem C++ Programm zur Verfügung steht, besteht aus einer oder mehreren Sequenzen von zusammenhängenden Bytes.

Speicher

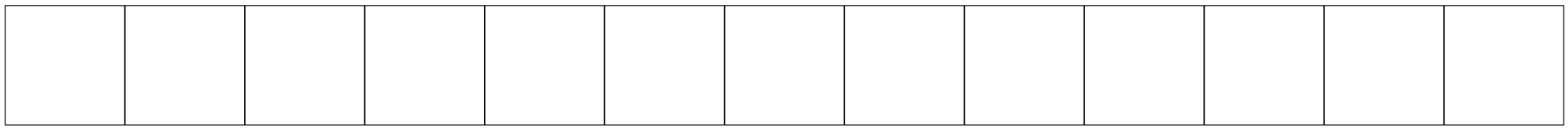
Standard, 1.7

- Die fundamentale Speicher-Einheit im C++ Speichermodell ist das *Byte*.
- Was ein Byte ist, ist recht abstrakt definiert.
- Der Speicher, welcher einem C++ Programm zur Verfügung steht, besteht aus einer oder mehreren Sequenzen von zusammenhängenden Bytes.
- Jedes Byte hat eine eindeutige Adresse.

Speicher

Standard, 1.7

- Die fundamentale Speicher-Einheit im C++ Speichermodell ist das *Byte*.
- Was ein Byte ist, ist recht abstrakt definiert.
- Der Speicher, welcher einem C++ Programm zur Verfügung steht, besteht aus einer oder mehreren Sequenzen von zusammenhängenden Bytes.
- Jedes Byte hat eine eindeutige Adresse.



≫Dinge≪ und Speicher

Ein ≫Ding≪ ist ein Speicherbereich. Wie hängt dieser Speicherbereich mit dem Speicher zusammen, der meinem Programm zur Verfügung steht?

≫Dinge≪ und Speicher

Ein ≫Ding≪ ist ein Speicherbereich. Wie hängt dieser Speicherbereich mit dem Speicher zusammen, der meinem Programm zur Verfügung steht?

Beispiel: Ein paar ≫Dinge≪

Am einfachsten legt man ≫Dinge≪ mittels einer Definitionen an.

Ausnahme: `static`

```
int foo;
MyType myObj;
int* pi;
```

≫Dinge≪ und Speicher

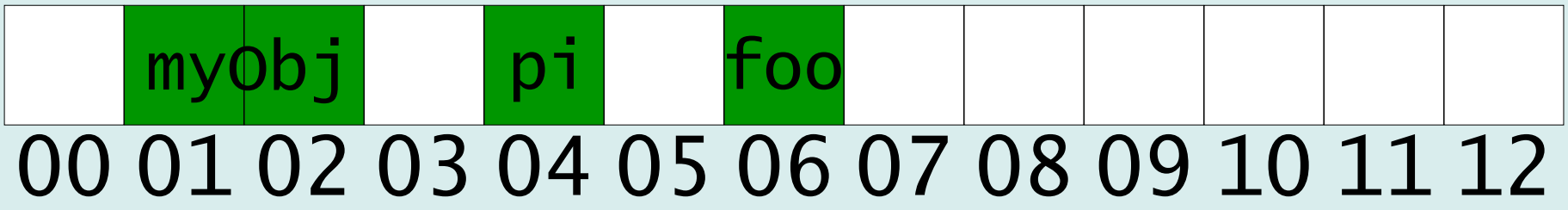
Ein ≫Ding≪ ist ein Speicherbereich. Wie hängt dieser Speicherbereich mit dem Speicher zusammen, der meinem Programm zur Verfügung steht?

Beispiel: Ein paar ≫Dinge≪

Am einfachsten legt man ≫Dinge≪ mittels einer Definitionen an.

Ausnahme: `static`

```
int foo;
MyType myObj;
int* pi;
```



≫Dinge≪ und Referenzen

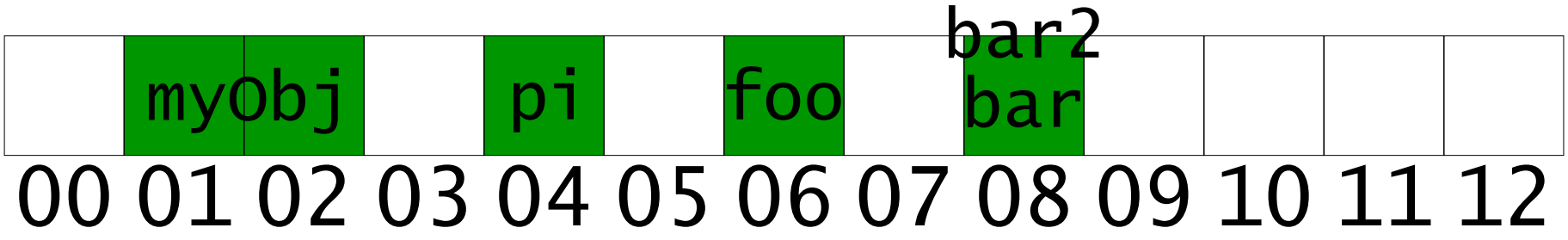
Eine Referenz ist ein zusätzlicher Namen für *dasselbe* ≫Ding≪.
Die Speicherdauer wird dadurch *nicht* beeinflusst.

```
bar  = 4;
bar2 = 4;
```

≫Dinge≪ und Referenzen

Eine Referenz ist ein zusätzlicher Namen für *dasselbe* ≫Ding≪.
Die Speicherdauer wird dadurch *nicht* beeinflusst.

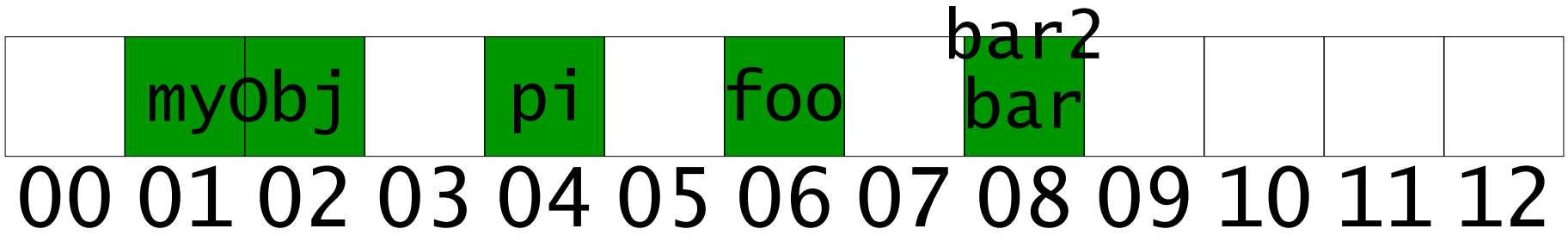
```
bar  = 4;
bar2 = 4;
```



≫Dinge≪ und Referenzen

Eine Referenz ist ein zusätzlicher Namen für *dasselbe* ≫Ding≪.
Die Speicherdauer wird dadurch *nicht* beeinflusst.

```
bar  = 4;
bar2 = 4;
```




```
int  bar;
int& bar2 = bar;
```

Referenz: Referenzen

`TYPE&addName = origName;`

- Führt `addName` als zusätzlichen Namen für das ≫Ding≪ mit dem Namen `origName` ein.
- `addName` heißt auch eine Referenz auf `origName`.

Achtung: Eine Referenz muss immer gleich ≫initialisiert≪ werden, d.h. mittels = einem ≫Ding≪ zugeordnet. Bei Klassen-Membern in der Initialisations-Liste des Konstruktors.

Referenzen als Parameter

Nutzt man eine Referenz als Parameter, so kann man innerhalb der Funktion den Wert des übergebenen ≫Dings≪ ändern:

```
void compute(double& p0, double& p1)
{
    p0 = 42.0;
    p1 = 21.0;
}
```

Referenzen als Parameter

Nutzt man eine Referenz als Parameter, so kann man innerhalb der Funktion den Wert des übergebenen ≫Dings≪ ändern:

```
void compute(double& p0, double& p1)
{
    p0 = 42.0;
    p1 = 21.0;
}
```

```
    double alice = 0;
    double bob   = 1;

    compute(alice, bob);
```

Adressen und Pointer

- Jedes Byte hat eine eindeutige Adresse.

Adressen und Pointer

- Jedes Byte hat eine eindeutige Adresse.
- Man darf keine Annahmen darüber treffen, wie diese Adresse tatsächlich aussieht oder wie groß sie ist!

Adressen und Pointer

- Jedes Byte hat eine eindeutige Adresse.
- Man darf keine Annahmen darüber treffen, wie diese Adresse tatsächlich aussieht oder wie groß sie ist!
- Was man tun darf, sind Operationen mit *Pointern*.

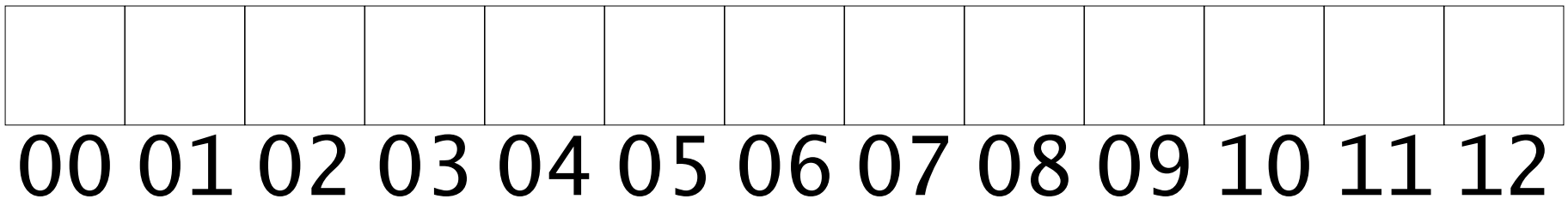
Adressen und Pointer

- Jedes Byte hat eine eindeutige Adresse.
- Man darf keine Annahmen darüber treffen, wie diese Adresse tatsächlich aussieht oder wie groß sie ist!
- Was man tun darf, sind Operationen mit *Pointern*.
- Ein Pointer ist ein ≫Ding≪, dessen Inhalt eine Adresse ist.

Adressen und Pointer

- Jedes Byte hat eine eindeutige Adresse.
- Man darf keine Annahmen darüber treffen, wie diese Adresse tatsächlich aussieht oder wie groß sie ist!
- Was man tun darf, sind Operationen mit *Pointern*.
- Ein Pointer ist ein ≫Ding≪, dessen Inhalt eine Adresse ist.

Wir wollen jedoch zwecks Anschauung eine Adresse mit der Nummer *der* Bytes eines ≫Dings≪ identifizieren.



≫Dinge≪ und Pointer

Sei `NAME` der Name eines ≫Dings≪. Mit `&NAME` erhalte ich dann die Adresse des ≫Dings≪, und zwar in der Form eines Pointers.

≫Dinge≪ und Pointer

Sei `NAME` der Name eines ≫Dings≪. Mit `&NAME` erhalte ich dann die Adresse des ≫Dings≪, und zwar in der Form eines Pointers.

Als ≫Ding≪ hat der Pointer einen Typ:

```
bool b;
bool* pb = &b;
```

≫Dinge≪ und Pointer

Sei `NAME` der Name eines ≫Dings≪. Mit `&NAME` erhalte ich dann die Adresse des ≫Dings≪, und zwar in der Form eines Pointers.

Als ≫Ding≪ hat der Pointer einen Typ:

```
bool b;
bool* pb = &b;
```

Das ≫Ding≪ `NAME` habe den Typen `TYPE`. Dann hat ein Pointer auf das ≫Ding≪ den Typ `TYPE*`.

≫Dinge≪, Pointer und Adressen

Ein Beispiel:

```
int foo;
MyType myObj;
int* pi;
```

≫Dinge≪, Pointer und Adressen

Ein Beispiel:

```
int foo;
MyType myObj;
int* pi;
pi = &foo;
```

≫Dinge≪, Pointer und Adressen

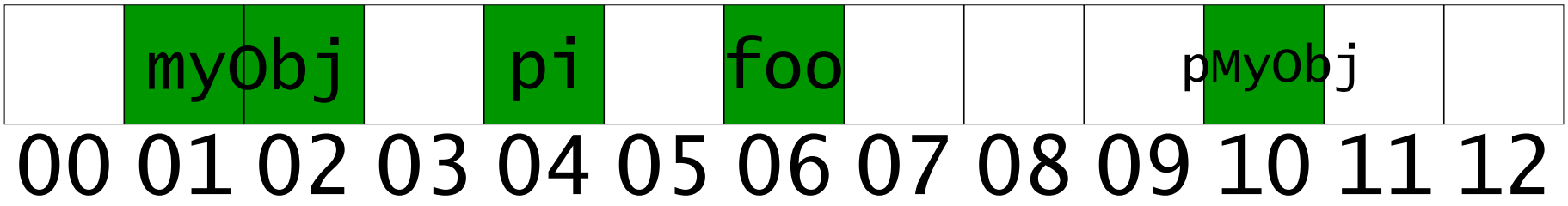
Ein Beispiel:

```
int foo;
MyType myObj;
int* pi;
pi = &foo;
MyType* pMyObj = &myObj;
```

≫Dinge≪, Pointer und Adressen

Ein Beispiel:

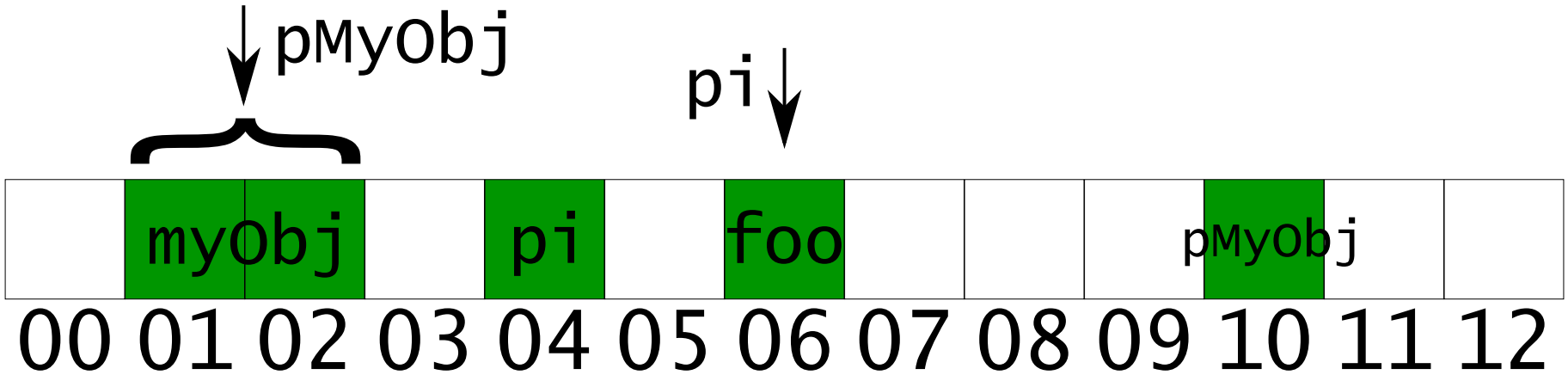
```
int foo;
MyType myObj;
int* pi;
pi = &foo;
MyType* pMyObj = &myObj;
```



≫Dinge≪, Pointer und Adressen

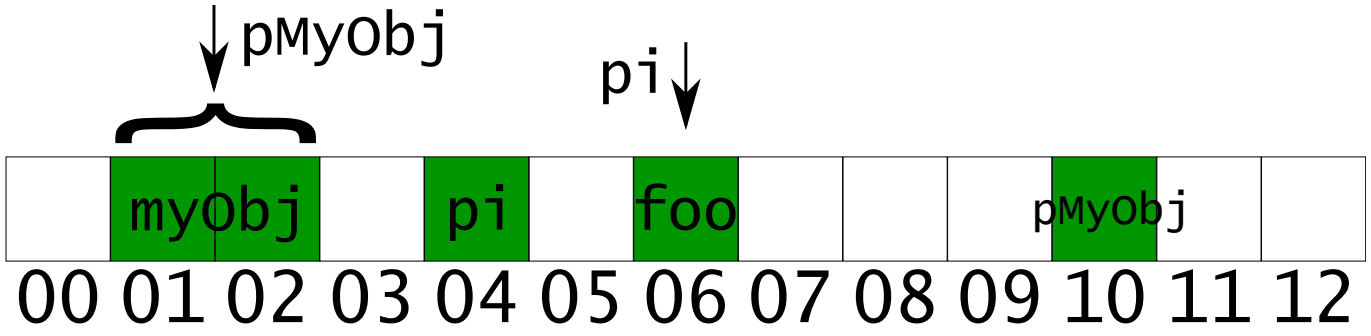
Ein Beispiel:

```
int foo;
MyType myObj;
int* pi;
pi = &foo;
MyType* pMyObj = &myObj;
```



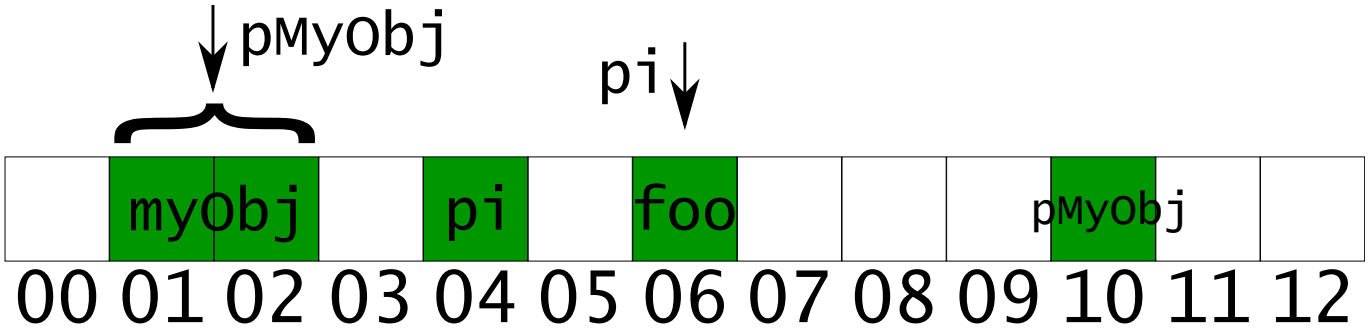
Dereferenzieren

Wie greife ich auf das ≫Ding≪ zu, auf welches ein Pointer verweist?



Dereferenzieren

Wie greife ich auf das ≫Ding≪ zu, auf welches ein Pointer verweist?



Mittels `*pointerName`. Man kann diesen Ausdruck deuten als:
das ≫Ding≪ an der Adresse, die in `pointerName` *gespeichert ist.*

Beispiel

```
int content = *pi;
*pi = 42;
```

Pointer vs. Referenzen

Pointer	Referenz
ist ein ≫Ding≪	ist nur ein zusätzlicher Name
enthält eine Adresse	enthält selbst nichts
Inhalt kann sich ändern	bezieht sich immer auf dasselbe ≫Ding≪
Inhalt muss nicht sinnvoll sein	bezieht sich auf ein existierendes ≫Ding≪

Pointer vs. Referenzen

Pointer	Referenz
ist ein ≫Ding≪	ist nur ein zusätzlicher Name
enthält eine Adresse	enthält selbst nichts
Inhalt kann sich ändern	bezieht sich immer auf dasselbe ≫Ding≪
Inhalt muss nicht sinnvoll sein	bezieht sich auf ein existierendes ≫Ding≪

```
MyType myObj;
MyType* pMyObj = &myObj;

MyType& rMyObj = myObj;
MyType myOtherObj;
pMyObj = &myOtherObj;
```

Definition eines Arrays

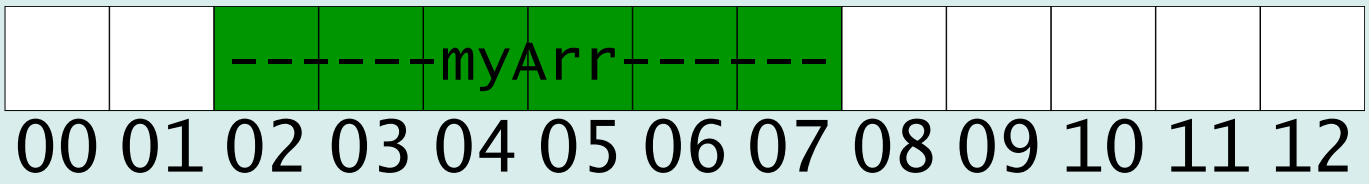
Mit `TYPE name[STATIC_NUMBER];` erzeuge ich `STATIC_NUMBER` ≫Dinge≪ hintereinander und zusammenhängend.

Definition eines Arrays

Mit `TYPE name[STATIC_NUMBER];` erzeuge ich `STATIC_NUMBER` ≫Dinge≪ hintereinander und zusammenhängend.

Beispiel

```
double myArr[3];
```



Zugriff auf die Elemente

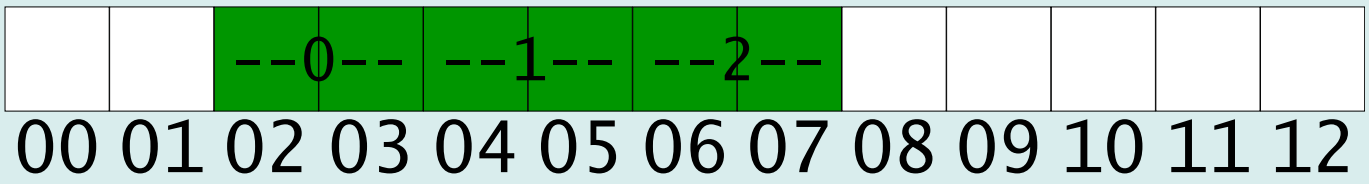
Mit `name[DYNAMIC_NUMBER]` greife ich auf ein Element zu, d.h. auf das `DYNAMIC_NUMBER`-te ≫Ding≪ im Array.

Zugriff auf die Elemente

Mit `name[DYNAMIC_NUMBER]` greife ich auf ein Element zu, d.h. auf das `DYNAMIC_NUMBER`-te ≫Ding≪ im Array.

Beispiel

```
double myArr[3];
myArr[1] = 13.37;
```



Arrays und Pointer

- Ein ≫Ding≪ erzeugt durch `TYPE name[N];` hat den Typen *Array von N TYPEs*.

Arrays und Pointer

- Ein ≫Ding≪ erzeugt durch `TYPE name[N];` hat den Typen *Array von N TYPEs*.
- Ein *Array von N TYPEs* wird automatisch (wo erforderlich) zu einem `TYPE*` konvertiert.

Arrays und Pointer

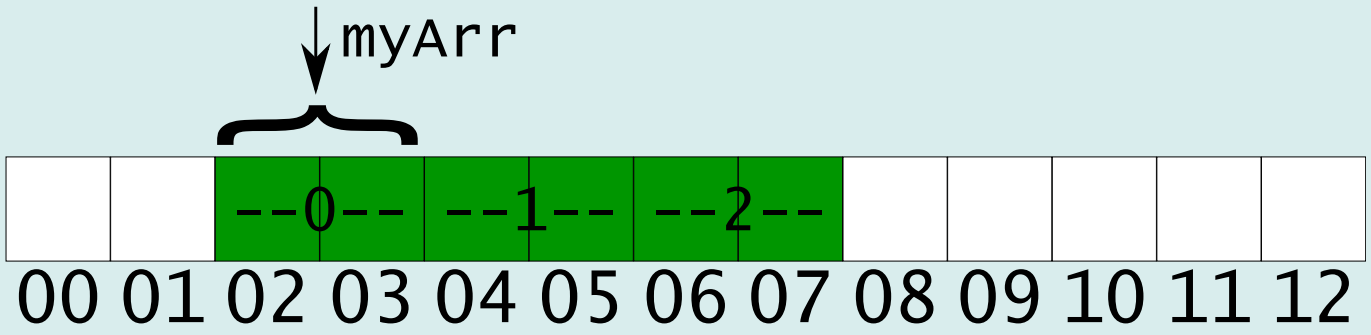
- Ein ≫Ding≪ erzeugt durch `TYPE name[N];` hat den Typen *Array von N TYPEs*.
- Ein *Array von N TYPEs* wird automatisch (wo erforderlich) zu einem `TYPE*` konvertiert.
- Der so entstehende Pointer zeigt auf `name[0]`.

Arrays und Pointer

- Ein ≫Ding≪ erzeugt durch `TYPE name[N];` hat den Typen *Array von N TYPEs*.
- Ein *Array von N TYPEs* wird automatisch (wo erforderlich) zu einem `TYPE*` konvertiert.
- Der so entstehende Pointer zeigt auf `name[0]`.

Beispiel

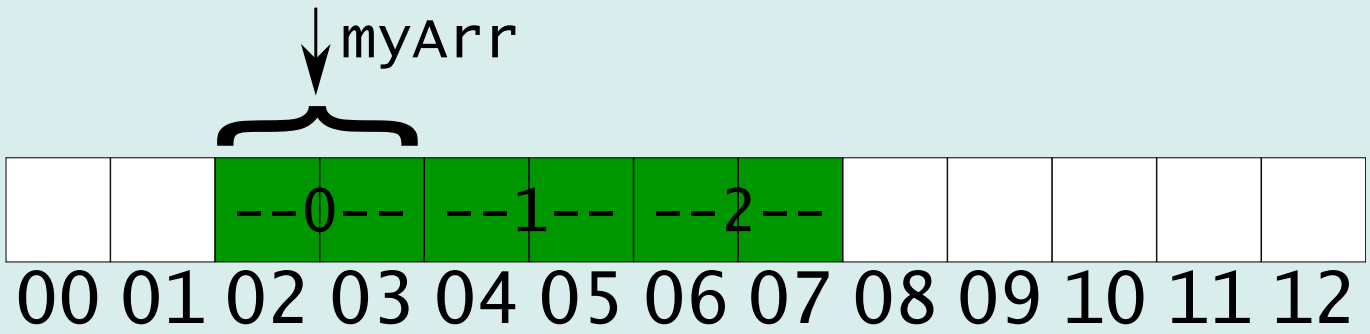
```
double  myArr[3];
double* pFirstElem  =    myArr;
double* pFirstElem2 = &( myArr[0] );
```



Elemente und Pointer

Beispiel

```
double  myArr[3];
double* pFirstElem  =    myArr;
```

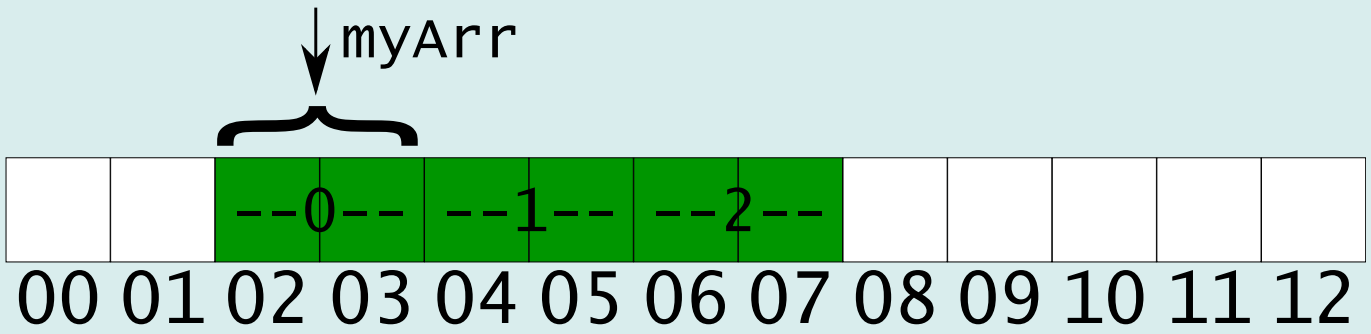


Elemente und Pointer

Beispiel

```
double  myArr[3];
double* pFirstElem  =    myArr;

double* pElem       = &( myArr[2] );
double* pElem2      = &( myArr[0] ) + 2;
```



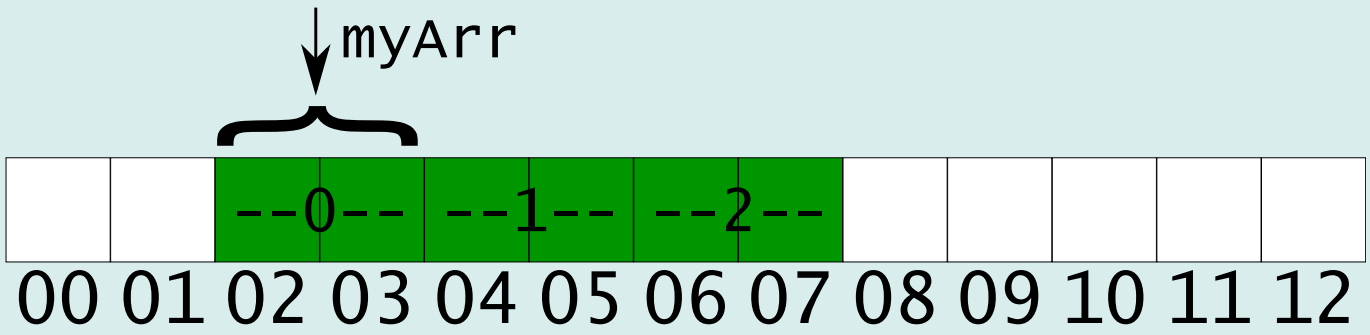
Elemente und Pointer

Beispiel

```
double  myArr[3];
double* pFirstElem  =    myArr;

double* pElem       = &( myArr[2] );
double* pElem2      = &( myArr[0] ) + 2;

double* pElem3      =    pFirstElem[2];
double* pElem4      =    pFirstElem + 2;
```



2. Stack und Heap
   - Stack
   - Heap

Speicherdauer von ≫Dingen≪

Block

- Ein *Block* beginnt mit { und endet mit }.

Speicherdauer von ≫Dingen≪

Block

- Ein *Block* beginnt mit { und endet mit }.

Definiere ich ein ≫Ding≪, so wird es bis zum Ende des Blocks, in welchem es erzeugt wurde, gespeichert.

Diese Art der vorgegebenen Speicherdauer nennt sich *automatic storage duration*.

Speicherdauer von ≫Dingen≪

Block

- Ein *Block* beginnt mit { und endet mit }.
- Ein Block kann mehrere *statements* enthalten – z.B. Definitionen, Zuweisungen, Funktionsaufrufe usw.
- Anderer Name: *compound statement*
- Die geschweiften Klammern bezeichnen *keine* Blöcke bei: Klassen, `namespace`, `enum`

Definiere ich ein ≫Ding≪, so wird es bis zum Ende des Blocks, in welchem es erzeugt wurde, gespeichert.

Diese Art der vorgegebenen Speicherdauer nennt sich *automatic storage duration*.

Referenz: Definition von ≫Dingen≪, Auf-den-Stack-legen

`|TYPEname;`— bzw. `TYPE name(PARAMETER);`

- Stellt sicher, dass es Speicher für das ≫Ding≪ vom Typ `TYPE` gibt.
- Führt den Namen `name` für das ≫Ding≪ ein.
- Ruft den Konstruktor des ≫Dings≪ auf (und übergibt die PARAMETER).

`TYPE name[STATIC_NUMBER];` – die Variante mit Parametern *existiert nicht!*

- Stellt sicher, dass es Speicher für `STATIC_NUMBER` ≫Dinge≪ gibt.
- Führt den Namen `name` ein, dieser ist (fast) identisch zur Adresse des nullten Elements.
- Ruft nacheinander für jedes Element den *default*-Konstruktor auf, beginnend beim nullten Element.

Reihenfolge der Definitionen

- Wenn ich (mehrere) ≫Dinge≪ innerhalb eines Blocks definiere, werden diese in der Reihenfolge der Definition auf einen Stapel abgelegt – den *Stack*.
- Das ist nicht wörtlich zu nehmen! Der Stack ist nur gedanklich!

Reihenfolge der Definitionen

- Wenn ich (mehrere) ≫Dinge≪ innerhalb eines Blocks definiere, werden diese in der Reihenfolge der Definition auf einen Stapel abgelegt – den *Stack*.
- Das ist nicht wörtlich zu nehmen! Der Stack ist nur gedanklich!
- Beim Ende des Blocks werden die ≫Dinge≪ wieder vom Stack genommen, und zwar entgegengesetzt der Reihenfolge, in welcher sie definiert wurden (Standard, 6.6 2).

Reihenfolge der Definitionen

- Wenn ich (mehrere) ≫Dinge≪ innerhalb eines Blocks definiere, werden diese in der Reihenfolge der Definition auf einen Stapel abgelegt – den *Stack*.
- Das ist nicht wörtlich zu nehmen! Der Stack ist nur gedanklich!
- Beim Ende des Blocks werden die ≫Dinge≪ wieder vom Stack genommen, und zwar entgegengesetzt der Reihenfolge, in welcher sie definiert wurden (Standard, 6.6 2).

Man kann dieses Verhalten nicht beeinflussen!

Referenz: Vom-Stack-nehmen

Für ein ≫Ding≪, welches als `TYPE name` definiert wurde:

- Ruft den Destruktor des ≫Dings≪ auf.
- Gibt den reservierten Speicher wieder frei.

Für ein ≫Ding≪, welches als `TYPE name[STATIC_NUMBER]` definiert wurde:

- Ruft nacheinander für jedes Element den Destruktor auf, beginnend beim nullten Element.
- Gibt den reservierten Speicher wieder frei.

Am Anfang war die Leere...

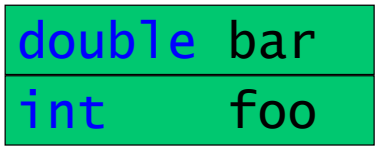
```
int main()
```

Definition von ≫Dingen≪ (1)

```
int    foo
```

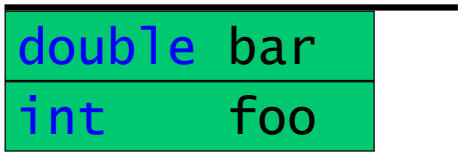
```
int main()
{
    int foo;
```

Definition von ≫Dingen≪ (2)



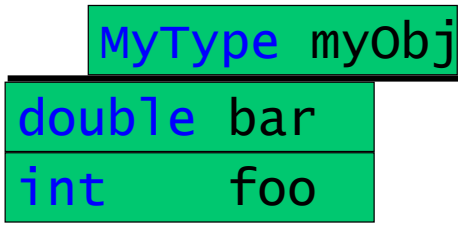

```
int main()
{
    int foo;
    double bar;
```

Definition von ≫Dingen≪ (3)



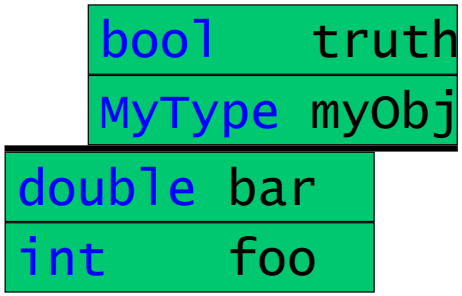

```
int main()
{
    int foo;
    double bar;
    {
```

Definition von ≫Dingen≪ (4)



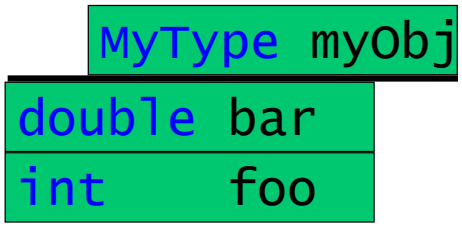

```
int main()
{
    int foo;
    double bar;
    {
        MyType myObj;
```

Definition von ≫Dingen≪ (5)



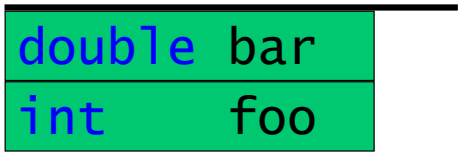

```
int main()
{
    int foo;
    double bar;
    {
        MyType myObj;
        bool truth;
```

Definition von ≫Dingen≪ (6)



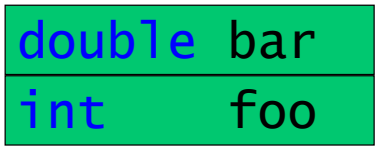

```
int main()
{
    int foo;
    double bar;
    {
        MyType myObj;
        bool truth;
    }
```

Definition von ≫Dingen≪ (7)




```
int main()
{
    int foo;
    double bar;
    {
        MyType myObj;
        bool truth;
    }
```

Definition von ≫Dingen≪ (8)




```
int main()
{
    int foo;
    double bar;
    {
        MyType myObj;
        bool truth;
    }
```

Dynamic storage duration

- ≫Dinge≪ „auf dem Stack“ werden automatisch verwaltet

Dynamic storage duration

- ≫Dinge≪ „auf dem Stack“ werden automatisch verwaltet
- Wie verwalte ich selbst ≫Dinge≪?

Dynamic storage duration

- ≫Dinge≪ „auf dem Stack“ werden automatisch verwaltet
- Wie verwalte ich selbst ≫Dinge≪?
- Wie nutze ich sehr große ≫Dinge≪?

Dynamic storage duration

- ≫Dinge≪ „auf dem Stack“ werden automatisch verwaltet
- Wie verwalte ich selbst ≫Dinge≪?
- Wie nutze ich sehr große ≫Dinge≪?

Die Lösung: „dynamische“ Speicherdauer bzw. der *Heap*

Allokation und Deallokation

- Mit `new TYPE` lege ich ein ≫Ding≪ auf dem Heap an.

Allokation und Deallokation

- Mit `new TYPE` lege ich ein ≫Ding≪ auf dem Heap an.
- Der `new`-Ausdruck gibt einen Pointer zurück (`TYPE*`).

Allokation und Deallokation

- Mit `new TYPE` lege ich ein ≫Ding≪ auf dem Heap an.
- Der `new`-Ausdruck gibt einen Pointer zurück (`TYPE*`).
- Mit `delete POINTER;` lösche ich das ≫Ding≪ vom Heap, auf welches der `POINTER` verweist.

Allokation und Deallokation

- Mit `new TYPE` lege ich ein ≫Ding≪ auf dem Heap an.
- Der `new`-Ausdruck gibt einen Pointer zurück (`TYPE*`).
- Mit `delete POINTER;` lösche ich das ≫Ding≪ vom Heap, auf welches der `POINTER` verweist.

ACHTUNG

- Ohne einen Pointer lässt sich ein ≫Ding≪ nicht vom Heap löschen!

Allokation und Deallokation

- Mit `new TYPE` lege ich ein ≫Ding≪ auf dem Heap an.
- Der `new`-Ausdruck gibt einen Pointer zurück (`TYPE*`).
- Mit `delete POINTER;` lösche ich das ≫Ding≪ vom Heap, auf welches der `POINTER` verweist.

ACHTUNG

- Ohne einen Pointer lässt sich ein ≫Ding≪ nicht vom Heap löschen!
- *Daran denken*, den Speicher wieder freizugeben, wenn er nicht mehr gebraucht wird!

Allokation und Deallokation

- Mit `new TYPE` lege ich ein ≫Ding≪ auf dem Heap an.
- Der `new`-Ausdruck gibt einen Pointer zurück (`TYPE*`).
- Mit `delete POINTER;` lösche ich das ≫Ding≪ vom Heap, auf welches der `POINTER` verweist.

ACHTUNG

- Ohne einen Pointer lässt sich ein ≫Ding≪ nicht vom Heap löschen!
- *Daran denken*, den Speicher wieder freizugeben, wenn er nicht mehr gebraucht wird!
- *Niemals* ein ≫Ding≪ mehrmals freigeben!

Allokation und Deallokation

- Mit `new TYPE` lege ich ein ≫Ding≪ auf dem Heap an.
- Der `new`-Ausdruck gibt einen Pointer zurück (`TYPE*`).
- Mit `delete POINTER;` lösche ich das ≫Ding≪ vom Heap, auf welches der `POINTER` verweist.

ACHTUNG

- Ohne einen Pointer lässt sich ein ≫Ding≪ nicht vom Heap löschen!
- *Daran denken*, den Speicher wieder freizugeben, wenn er nicht mehr gebraucht wird!
- *Niemals* ein ≫Ding≪ mehrmals freigeben!
- *Niemals* einen Pointer auf ein freigegebenes ≫Ding≪ derefenzieren!

Allokation und Deallokation

- Mit `new TYPE` lege ich ein ≫Ding≪ auf dem Heap an.
- Der `new`-Ausdruck gibt einen Pointer zurück (`TYPE*`).
- Mit `delete POINTER;` lösche ich das ≫Ding≪ vom Heap, auf welches der `POINTER` verweist.

ACHTUNG

- Ohne einen Pointer lässt sich ein ≫Ding≪ nicht vom Heap löschen!
- *Daran denken*, den Speicher wieder freizugeben, wenn er nicht mehr gebraucht wird!
- *Niemals* ein ≫Ding≪ mehrmals freigeben!
- *Niemals* einen Pointer auf ein freigegebenes ≫Ding≪ derefenzieren!
- Gute Angewohnheit: Nach `delete p;` sofort `p = 0;`, sorgt für (fast) sicheren Programmabsturz im Falle von `*p`

Arrays auf dem Heap

Arrays lassen sich auch „auf dem Heap anlegen“:

- Mit `new TYPE[DYNAMIC_NUMBER]` lege `DYNAMIC_NUMBER` ≫Ding≪ auf dem Heap an.

Arrays auf dem Heap

Arrays lassen sich auch „auf dem Heap anlegen“:

- Mit `new TYPE[DYNAMIC_NUMBER]` lege `DYNAMIC_NUMBER` ≫Ding≪ auf dem Heap an.
- Der `new`-Ausdruck gibt einen Pointer auf das nullte Element zurück (`TYPE*`).

Arrays auf dem Heap

Arrays lassen sich auch „auf dem Heap anlegen“:

- Mit `new TYPE[DYNAMIC_NUMBER]` lege `DYNAMIC_NUMBER` ≫Ding≪ auf dem Heap an.
- Der `new`-Ausdruck gibt einen Pointer auf das nullte Element zurück (`TYPE*`).
- Mit `delete[] POINTER;` lösche ich den Array vom Heap, auf welchen der `POINTER` verweist.

Referenz: Allokation auf dem Heap

`|newTYPE;—` bzw. `new TYPE(PARAMETER);`

- Alloziert das ≫Ding≪ auf dem Heap (den Speicher!).
- Ruft den Konstruktor von TYPE auf (und übergibt die PARAMETER).
- Gibt die Adresse des Speichers zurück, als `TYPE*`.

`new TYPE[DYNAMIC_NUMBER];` – die Variante mit Parametern *existiert nicht!*

- Alloziert `DYNAMIC_NUMBER` ≫Dinge≪ auf dem Heap (den Speicher!).
- Ruft für jedes Element den *default*-Konstruktor von TYPE auf.
- Gibt die Adresse des Speichers des nullten Elements zurück, als `TYPE*`.

Referenz: Deallokation vom Heap

`|deleteADDRESS;—` – ADDRESS muss eine Rückgabe von `new` sein, sei hier vom Typ TYPE*

- Ruft den Destruktor vom Objekt auf (`ADDRESS->~TYPE();`)
- Deallociert den Speicher vom Heap.

`delete[] ADDRESS;` – ADDRESS muss eine Rückgabe von `new [...]` sein

- Ruft nacheinander die Destruktoren der Elementes auf, beginnend mit dem letzten.
- Deallociert den Speicher vom Heap.

Stack vs. Heap

Stack

- Schnelle Allokation & Deallokation (z.B. ein einzige Operation für beliebig viel ≫Dinge≪!)
- Direkter, schneller Zugriff
- Optimierungen gut und einfach(er) für den Compiler
- Erzeugung & Aufräumen ist automatisch
- Begrenzter Speicher (z.B. 1 MB)
- Größe des ≫Dings≪ und Anzahl muss dem Compiler bekannt sein

Heap

- Für große ≫Dinge≪
- Eigene Verwaltung der Speicherdauer
- Speicherverwaltung während das Programm läuft

3 Objektorientierte Programmierung

Structures

structs

Ein `struct` ist eine Zusammenfassung mehrerer Objekte zu einem größeren. Zum Beispiel könnte man ein `struct` „Quader“ erstellen, welches drei Fließkommazahlen beinhaltet.

Instanzen

Eine solche `struct` ist lediglich eine abstrakte Beschreibung des Objekts; man arbeitet schließlich mit sogenannten *Instanzen* des Objekts. Beispiel Quader: Die Structure an sich beschreibt das abstrakte Objekt, die Instanz einen konkreten Quader („der Quader auf meinem Tisch“).

Beispiel für eine Structure

```
struct Box {
    double length;
    double width;
    double height;
};

int main() {
    Box myBox;
    myBox.length = 3;
    myBox.width = 5;
    myBox.height = 2.7;
}
```

Klassen

Klassen

Eine `class` ist eine `struct`, die zusätzlich zu Daten noch Funktionen enthält, die auf diesen Daten operieren.

Konstruktor und Destruktor

Eine Klasse hat zwei besondere Funktionen, den Konstruktor und den Destruktor; der Konstruktor wird aufgerufen, wenn eine neue Instanz der Klasse erstellt wird, und der Destruktor, wenn die Instanz wieder gelöscht wird. Der Konstruktor heißt `klassenname`, der Destruktor `~klassenname`.

Beispiel für eine Klasse

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

class Box {
public:
    void Box(double l, double w, double h) {
        length = l;
        width = w;
        height = h;
    };
    void ~Box() {
        cout << "box is being destroyed";
    };
    double getVolume() {
        return length * width * height;
    };
private:
    double length;
    double width;
    double height;
};

int main() {
    Box myBox(3, 5, 2.7);
    cout << myBox.getVolume();
}
```

Etwas anderes Beispiel für eine Klasse

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

class Box {
public:
    Box(double l, double w, double h);
    ~Box();
    double getVolume();
private:
    double length;
    double width;
    double height;
};

double Box::getVolume() {
    return length * width * height;
}

Box::Box(double l, double w, double h) : width(w), length(l), height(h) {
    cout << "Box is being created";
}

Box::~Box()
{
    cout << "Box is being destroyed";
}

int main() {
    Box* myBox = new Box(3, 5, 2.7);
    cout << myBox->getVolume();
    cout << myBox->length; // Fehler!
    delete myBox;
}
```