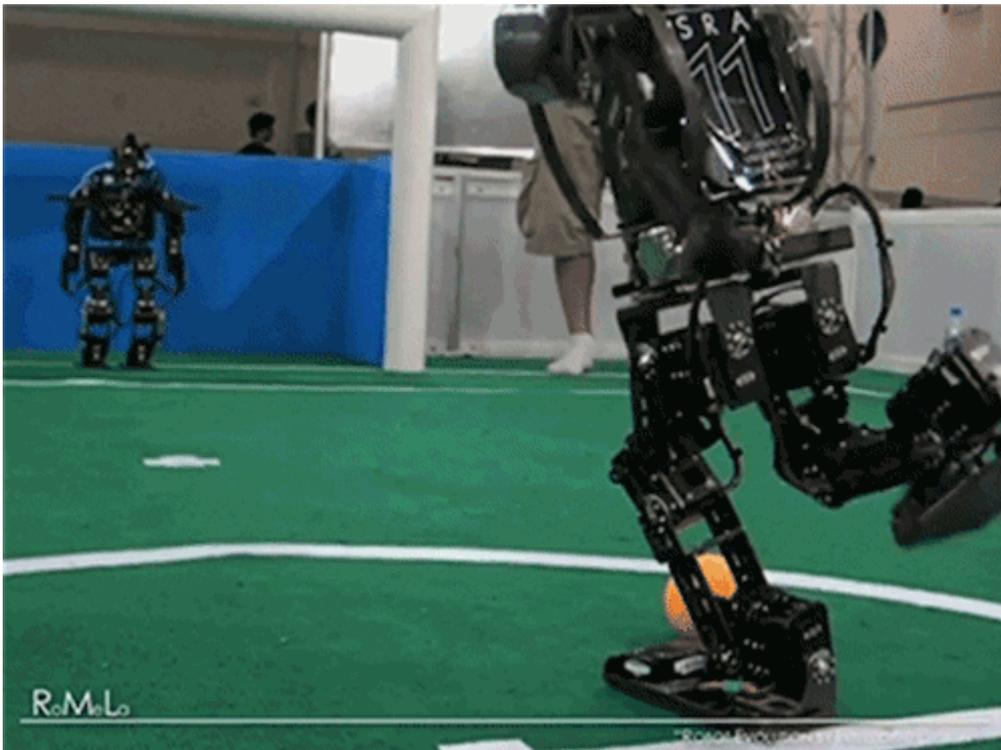


# Speicher und Pointer

Parameter	Kursinformationen
Veranstaltung:	Softwareprojekt Robotik
Semester	Wintersemester 2023/24
Hochschule:	Technische Universität Freiberg
Inhalte:	Grundlagen der Speicherverwaltung unter C++
Link auf GitHub:	<a href="https://github.com/TUBAF-lfl-LiaScript/VL_Softwareentwicklung/blob/master/02_SpeicherUndPointer.md">https://github.com/TUBAF-lfl-LiaScript/VL_Softwareentwicklung/blob/master/02_SpeicherUndPointer.md</a>
Autoren	Sebastian Zug & Georg Jäger



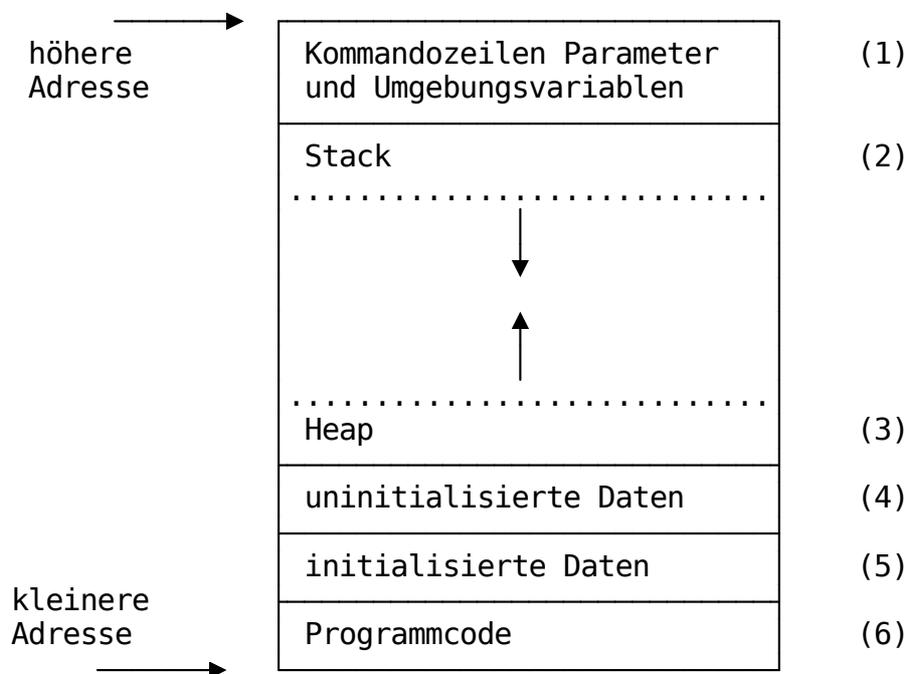
Zielstellung der heutigen Veranstaltung

- Unterscheidung von Stack und Heap
- Gegenüberstellung von Basis-Pointern und Smart-Pointern
- Herausforderungen beim Speichermanagement

**Organisatorisches** Zur Terminfindung für die erste Übung werden wir eine OPAL Nachricht versenden.

## Komponenten des Speichers

Wie wird der Speicher von einem C++ Programm eigentlich verwaltet? Wie wird diese Struktur ausgehend vom Start eines Programmes aufgebaut?



	Speicherbestandteil	engl.	Bedeutung
1	Parameter		
2	Stack		LIFO Datenstruktur
3	Heap		"Haldenspeicher"
4	Uninitialisierte Daten	.bss	
5	Initialisierte Daten	.data	Globale und statische Daten,
6	Programmcode	.text	teilbar, häufig als read only Speicher

## Untersuchung der ausführbaren Datei

Welche Anteile der Speicherstruktur sind zur Compilezeit analysierbar?

```
memory.cpp

#include <iostream>

int main(void)
{
    return EXIT_SUCCESS;
}
```

```
gcc memory.c -o memory
size memory
text      data      bss      dec      hex      filename
1918      640       8        2566     a06      memory
```

Der reine Quellcode wird im `.text` Segment abgelegt. Im `.data` und `.bss` Segment werden lediglich die globalen und statischen, lokalen Variablen abgelegt.

Achtung: Im Beispiel wurde keine Optimierung verwendet. Je nach Konfiguration ergeben sich hier unterschiedliche Resultate!

# Stack vs Heap

Zunächst mal ganz praktisch, was passiert auf dem Stack?

Ausgangspunkt unserer Untersuchung ist ein kleines Programm, das mit dem gnu Debugger `gdb` analysiert wurde:

```
g++ -m32 StackExample.cpp -o stackExample
gdb stackExample
(gdb) disas main
(gdb) disas calc
```

## StackExample.cpp

```
1  #include <iostream>
2
3  int calc(int factor1, int factor2){
4      return factor1 * factor2;
5  }
6
7  int main()
8  {
9      int num1 {0x11};
10     int num2 {0x22};
11     int result {0};
12     result = calc(num1, num2);
13     std::cout << result << std::endl;
14     return EXIT_SUCCESS;
15 }
```

578

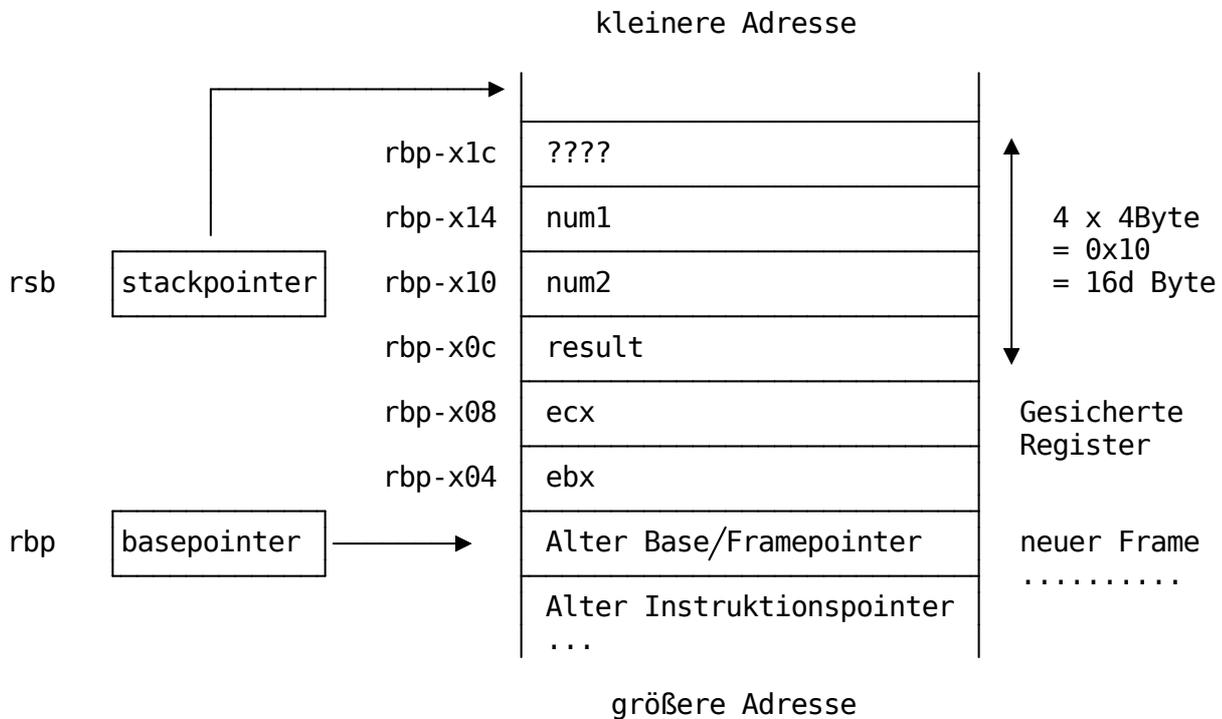
Was passiert beim starten des Programmes und beim Aufruf der Funktion `calc` "unter der Haube"? Schauen wir zunächst die Einrichtung des Stacks von Seiten der `main` funktion bis zur Zeile 12.

```
0x06ed <+10>:    push    %ebp
0x06ee <+11>:    mov     %esp,%ebp
0x06f0 <+13>:    push    %ebx
0x06f1 <+14>:    push    %ecx
0x06f2 <+15>:    sub     $0x10,%esp
... # Initialisierung der Variablen
0x0700 <+29>:    movl   $0x11,-0x14(%ebp)
0x0707 <+36>:    movl   $0x22,-0x10(%ebp)
0x070e <+43>:    movl   $0x0,-0xc(%ebp)
... # Aufruf des Unterprogramms
0x071b <+56>:    call   0x6cd <calc(int, int)>
0x0720 <+61>:    add    $0x8,%esp
```

```

0x0723 <+64>:   mov     %eax,-0xc(%ebp)
0x0726 <+67>:   sub     $0x8,%esp
... # Aufruf Betriebssystemschnittstell für Ausgabe
0x0733 <+80>:   call   0x570 <_ZNSolsEi@plt>
....

```

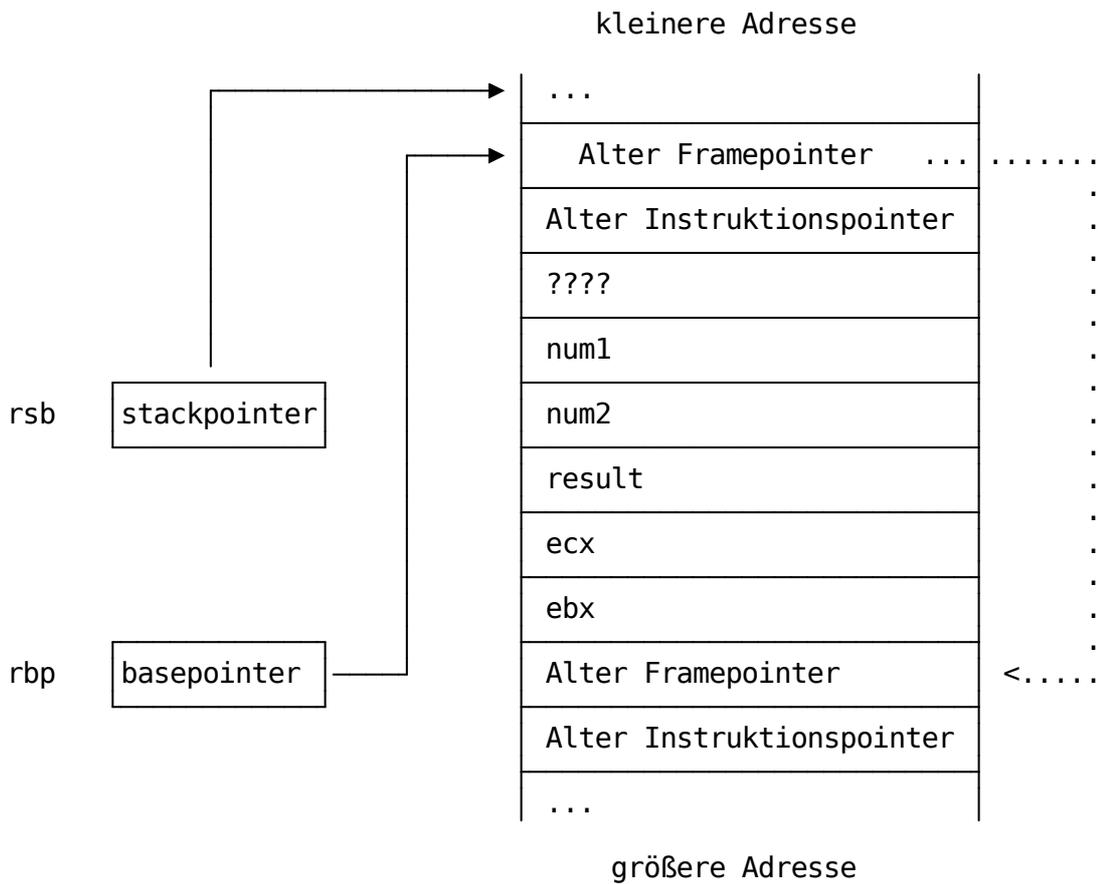


Nun rufen wir die Funktion `calc` auf und führen die Berechnung aus. Dafür wird ein neuer Stackframe angelegt. Wie entwickelt sich der Stack ausgehend von dem zugehörigen Assemblercode weiter?

```

0x06cd <+0>:   push   %ebp
0x06ce <+1>:   mov    %esp,%ebp
0x06da <+13>:  mov    0x8(%ebp),%eax
0x06dd <+16>:  imul  0xc(%ebp),%eax
0x06e1 <+20>:  pop    %ebp
0x06e2 <+21>:  ret

```



## Heap

Der Heap ist ein dedizierter Teil des RAM, in dem von der Applikation Speicher dynamisch belegt werden kann. Speichergrößen werden explizit angefordert und wieder frei gegeben.

Unter C(!) erfolgt dies mit den Funktionen der Standardbibliothek. C++ übernimmt diese Funktionalität.

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t n, size_t size);
void *realloc(void *ptr, size_t size)
free(void *ptr);
```

C++ erhöht den Komfort für den Entwickler und implementiert ein alternatives Konzept.

```
new «Datentyp» »(«Konstruktorargumente»)«
delete «Speicheradresse»
```

	<code>new</code>	<code>malloc</code>
Bedeutung	Schlüsselwort, Operator	Funktion der Standardbibliothek
Rückgabewert	Pointer vom Typ des Objektes	<code>void</code> Pointer
Fehlerfall	Ausnahme	NULL Pointer
Größe	wird vom Compiler bestimmt	muss manuell festgelegt werden
Überschreibbarkeit	ja	

### Beispiel

```
#include <iostream>

struct Bruch{
    int zaehler = 1;
    int nenner = 1;

    Bruch(int z, int n) : zaehler{z}, nenner{n} {};
    void print(){
        std::cout << zaehler << "/" << nenner << std::endl;
    }
};

int main(){
    Bruch einhalb {1,2};
    Bruch* einviertel = static_cast<Bruch*>(malloc(sizeof(Bruch)));
    einviertel->zaehler = 1;
    einviertel->nenner = 4;
    einviertel->print();
    Bruch* einachtel = new Bruch(1,8);
    einachtel->print();
    return EXIT_SUCCESS;
}
```

[Link auf pythontutor](#)

### Zusammenfassung

Parameter	Stack	Heap
Allokation und Deallokation	Automatisch durch den Compiler	Manuel durch den Programmierer
Kosten	gering	ggf. höher durch Fragmentierung
Flexibilität	feste Größe	Anpassungen möglich

Und noch mal am Beispiel

### StackvsHeap.cpp

```

1  #include <iostream>
2
3  class MyClass{
4      public:
5          MyClass()
6          {
7              std::cout << "Constructor executed" << std::endl;
8          }
9          ~MyClass()
10         {
11             std::cout << "Destructor executed" << std::endl;
12         }
13 };
14
15 int main()
16 {
17     { // Scope I
18         MyClass A;
19     }
20     { // Scope II
21         MyClass* B = new MyClass;
22     }
23     std::cout << "Memory Leak !" << std::endl;
24     return EXIT_SUCCESS;
25 }
```

```

Constructor executed
Destructor executed
Constructor executed
Memory Leak !
```

# Stack Overflow

Wenn ein Programm mehr Speicherplatz als die Stapelgröße belegt, tritt ein Stapelüberlauf auf und es kann zu einem Programmabsturz kommen. Es gibt zwei Fälle, in denen ein Stapelüberlauf auftreten kann:

1. Wenn wir eine große Anzahl lokaler Variablen deklarieren oder ein Array oder eine Matrix oder ein höherdimensionales Array mit großer Größe deklarieren, kann dies zu einem Überlauf des Stapels führen.
2. Wenn sich eine Funktion hinreichend oft rekursiv selbst aufruft, kann der Stapel keine große Anzahl von lokalen Variablen speichern, die von jedem Funktionsaufruf verwendet werden, und führt zu einem Überlauf des Stapels.

Für Ubuntu 22.04 ergibt sich mit dem Befehl `ulimit` folgende Ausgabe:

```
ulimit -a
-t: cpu time (seconds)          unlimited
-f: file size (blocks)         unlimited
-d: data seg size (kbytes)     unlimited
-s: stack size (kbytes)       8192
-c: core file size (blocks)    0
-m: resident set size (kbytes) unlimited
-u: processes                  127043
-n: file descriptors           1024
-l: locked-in-memory size (kbytes) 4074208
-v: address space (kbytes)     unlimited
-x: file locks                 unlimited
-i: pending signals           127043
-q: bytes in POSIX msg queues  819200
-e: max nice                   0
-r: max rt priority            0
-N 15:                          unlimited
```

Die Stackgröße ist für diesen Rechner standardmäßig auf 8192kB beschränkt. Wenn wir also einen Stack-Overflow generieren wollen können wir dies realisieren, in dem wir Datenstrukturen generieren, die größer als dieser Wert sind.

$$8192kB = 8192.000B = 1024.000 \cdot 8Byte$$

## Stack Overflow.cpp

```
#include <iostream>

int recursion(int a)
{
    ..... return recursion(a+1);
}

int main()
{
    ..... std::cout << "Stack Overflow about to occur!\n";
    ..... double array[1024*1024];
    ..... // recursion(0);
    ..... return EXIT_FAILURE;
}
```

### Mögliche Lösungsansätze

#### Vermeidung

- Individuelle Analyse der statischen /dynamischen Codeelemente mit Hilfe des Compilers und Berücksichtigung der Aufrufhierarchie.
- Testen mit vorinitialisiertem Speicher zur Bestimmung der maximalen Stack-Größe.

#### Identifikation

- Verwendung von Stackpointer-Evaluationsregister (ARMv8-M Architekturen). Wenn der SP der CPU den in diesem Register festgelegten Wert (nennen wir es das SP\_Limit-Register) unterschreitet (oder je nach Stack-Wachstum darüber liegt), wird eine Ausnahme generiert.
- Die Memory Protection Unit (MPU) erlaubt die Integration von Sicherheitskorridoren zur Überwachung der Stackgröße.
- Integration sogenannter "Canaries" als Muster im Speicher, die Hinweise auf unberechtigte Schreiboperationen geben.

## (Raw-)Pointer / Referenzen

C und C++ unterstützen das Konzept des **Zeigers**, die sich von den meisten anderen Programmiersprachen unterscheiden. Andere Sprachen wie C#, C++(!), Java, Python etc. implementieren **Referenzen**.

Oberflächlich betrachtet sind sich Referenzen und Zeiger sehr ähnlich, in beiden Fällen greifen wir indirekt auf einen Speicher zu:

- Ein Zeiger ist eine Variable, die die Speicheradresse einer anderen Variablen enthält. Ein Zeiger muss mit dem Operator \* dereferenziert werden, um auf den Speicherort zuzugreifen. Pointer können auch ins "nichts" zeigen. C++11 definiert dafür den `nullptr`.
- Eine Referenz ist ein Alias, das heißt ein anderer Name für eine bereits vorhandene Variable. Eine Referenz wird wie ein Zeiger implementiert, indem die Adresse eines Objekts gespeichert wird. Entsprechend kann eine Referenz auch nur für ein bestehendes Objekt angelegt werden. In der Nutzungsphase unterscheidet sich der Zugriff auf eine Referenz nicht von einem Direktzugriff

## ReferencesVsPointer.cpp

```
1 #include <stdio.h>
2
3 void function(int *aux){
4     *aux++;
5     printf("Call by 'Reference': %i\n", *aux);
6 }
7
8 void function(int aux){
9     aux++;
10    printf("Call by Value: %i\n", aux);
11 }
12
13 int main() {
14     int i = 3;
15     int j = 5;
16
17     // Initialisierung
18     // -----
19     // Pointer auf eine Variable
20     int *ptr = &i;
21     int *empty_ptr;
22
23     // Referenz auf eine Variable
24     int &ref = i;
25     int &ref_b = i;
26     //int &empty_ref; // Error!
27
28     // Reassignment
29     ptr = &j;
30     //ref = i; // Reassignment nicht möglich
31
32     // Indirection - Funktioniert nicht mit Referenzen
33     int **ptr_ptr; //it is valid.
34     ptr = &i;
35     ptr_ptr = &ptr;
36     printf("0x%p points at %i\n", (void *)ptr, i);
37     printf("0x%p points at 0x%p\n\n", (void *)ptr_ptr, *ptr_ptr);
38     .....
39     // Funktionsaufruf
40     function(ptr);
41     printf("Resultierendes j: %i\n", j);
42     function(ref);
43     printf("Resultierendes i: %i\n", i);
44     return 0;
45 }
```

```
0x0x7ffc0f6b7100 points at 3
0x0x7ffc0f6b7108 points at 0x0x7ffc0f6b7100
```

```
Call by 'Reference': 5
Resultierendes j: 5
Call by Value: 4
Resultierendes i: 3
```

Führen Sie das Beispiel im [PythonTutor](#) aus und beobachten Sie den Unterschied zwischen Referenz und Pointer.

*Achtung:* Im obenstehenden Code ist ein Pointer Anfängerfehler verborgen. Dessen Wirkung sollte mit der Ausgabe in Zeile 41 deutlich werden - Finden Sie ihn?

Die Funktionsweise im Vergleich soll am Beispiel einer Parameterübergabe verdeutlicht werden:

### Constructor.cpp

```
1  #include <iostream>
2
3  void DoSomeCalculationsByValue(int number){
4      number = number * 2;
5  }
6
7  void DoSomeCalculationsByPointer(int* number){
8      *number = *number * 2;
9  }
10
11 void DoSomeCalculationsByReference(int& number){
12     number = number * 2;
13 }
14
15 int main()
16 {
17     int number {1};
18     DoSomeCalculationsByValue(number);
19     std::cout << number << std::endl;
20     int* ptr = &number;
21     DoSomeCalculationsByPointer(ptr);
22     std::cout << *ptr << std::endl;
23     int& ref = number;
24     DoSomeCalculationsByReference(ref);
25     std::cout << ref << std::endl;
26     return EXIT_SUCCESS;
27 }
```

1  
2  
4

## Nachteile bei der Verwendung von Raw Pointern

"Raw pointers have been a pain in the backside for most students learning C++ in the last decades. They had a multi-purpose role as raw memory iterators, nullable, changeable nullable references and devices to manage memory that no-one really owns. This lead to a host of bugs and vulnerabilities and headaches and decades of human life spent debugging, and the loss of joy in programming completely." [Link](#)

"In modernem C++ werden Rohzeiger nur in kleinen Codeblöcken mit begrenztem Gültigkeitsbereich, in Schleifen oder Hilfsfunktionen verwendet, in denen Leistung ausschlaggebend ist und keine Verwirrung über den Besitzer entstehen kann." [Microsoft](#)

## Smart Pointer

**Problemstellung:** Wir wollen den Komfort der Nutzung des Stacks auf dynamische Konzepte übertragen, die sich auf dem Heap befinden und mit Pointer adressiert werden.

Das folgende Beispiel greift einen Nachteil der Raw Pointer nochmals auf, um das Konzept höherabstrakter Zeigerformate herzuleiten. Wie können wir sicherstellen, dass im folgenden das Objekt auf dem Heap zerstört wird, wenn wir den Scope verlassen?

## SelfDesignedUniquePointer.cpp

```
1 #include <iostream>
2
3 class MyClass{
4     public:
5         MyClass()
6         {
7             std::cout << "Constructor executed" << std::endl;
8         }
9         ~MyClass()
10        {
11            std::cout << "Destructor executed" << std::endl;
12        }
13 };
14
15 int main()
16 {
17     {
18         MyClass* A = new MyClass();
19         // ... Some fancy things happen here ...
20         delete A;
21     }
22     return EXIT_SUCCESS;
23 }
```

```
Constructor executed
Destructor executed
```

Wir kombinieren eine stackbasierten Pointerklasse, die die den Zugriff auf das eigentliche Datenobjekt kapselt. Mit dem verlassen des Scopes wird auch der allozierte Speicher freigegeben.

Visualisierung der Lösung in [PythonTutor](#)

Was sind die Schwächen des Entwurfes?

```
class MyPointer{
    private:
        MyClass* m_Ptr;
    public:
        MyPointer(MyClass* ptr) : m_Ptr (ptr) {}
        ~MyPointer(){
            delete m_Ptr;
        }
};

//myPointer C(new MyClass()); // oder
MyPointer C = new MyClass();
```

Die Wirkungsweise eines intelligenten C++-Zeigers ähnelt dem Vorgehen bei der Objekterstellung in Sprachen wie C#: Sie erstellen das Objekt und überlassen es dann dem System, das Objekt zur richtigen Zeit zu löschen. Der Unterschied besteht darin, dass im Hintergrund keine separate Speicherbereinigung ausgeführt wird – der Arbeitsspeicher wird durch die C++-Standardregeln für den Gültigkeitsbereich verwaltet, sodass die Laufzeitumgebung schneller und effizienter ist.

C++11 implementiert verschiedene Pointer-Klassen für unterschiedliche Zwecke. Diese werden im folgenden vorgestellt:

- `std::unique_ptr` – smart pointer der den Zugriff auf eine dynamisch allokierte Ressource verwaltet.
- `std::shared_ptr` – smart pointer der den Zugriff auf eine dynamisch allokierte Ressource verwaltet, wobei mehrere Instanzen für ein und die selbe Ressource bestehen können.
- `std::weak_ptr` – analog zum `std::shared_ptr` aber ohne Überwachung der entsprechenden Pointerinstanzen.

## Unique Pointers

Die `unique` Pointer stellen sicher, dass das eigentliche Objekt nur durch einen einzelnen Pointer adressiert wird. Wird der entsprechende Scope verlassen, wird das Konstrukt automatisch gelöscht.

### UniquePointer.cpp

```
1 #include <iostream>
2 #include <memory>    //<-- Notwendiger Header
3
4 class MyClass{
5     public:
6     MyClass(){
7         std::cout << "Constructor executed" << std::endl;
8     }
9     ~MyClass(){
10        std::cout << "Destructor executed" << std::endl;
11    }
12    void print(){
13        std::cout << "That's all!" << std::endl;
14    }
15 };
16
17 int main()
18 {
19     {
20         std::unique_ptr<MyClass> A (new MyClass());
21         A->print();
22     }
23     return EXIT_SUCCESS;
24 }
```

```
Constructor executed  
That's all!  
Destructor executed
```

Merke: Zu jedem Zeitpunkt verweist nur eine Instanz des `unique_ptr` auf eine Ressource. Die Idee lässt keine Kopien zu.

Wie wird das softwaretechnisch abgefangen? Die entsprechenden Methoden wurden mit "delete" markiert und generieren entsprechend eine Fehlermeldung.

```
unique_ptr(const _Myt&) = delete;  
_Myt& operator=(const _Myt&) = delete;
```

Dies ist dann insbesondere bei der Übergabe von `unique_ptr` an Funktionen von Bedeutung.

## UniquePointerHandling.cpp

```
1 #include <iostream>
2 #include <memory>
3
4 void callByValue(std::unique_ptr<std::string> input){
5     std::cout << *input << std::endl;
6 }
7
8 void callByReference(const std::unique_ptr<std::string> & input){
9     std::cout << *input << std::endl;
10 }
11
12 void callByRawPointer(std::string* input){
13     std::cout << *input << std::endl;
14 }
15
16 int main()
17 {
18     // Variante 1
19     //std::unique_ptr<std::string> A (new std::string("Hello World"))
20     // Variante 2
21     std::unique_ptr<std::string> A = std::make_unique<std::string>("Hello
22     World");
23     // Ohne Veränderung der Ownership
24     callByValue(A); // Compilerfehler!!!
25     //callByValue(std::move(A)); // Mit Übergabe der Ownership
26     callByReference(A);
27     callByRawPointer(A.get());
28     //std::cout << A << std::endl;
29     return EXIT_SUCCESS;
30 }
```

```

main.c: In function 'int main()':
main.c:23:14: error: use of deleted function 'std::unique_ptr<_Tp,
_Dp>::unique_ptr(const std::unique_ptr<_Tp, _Dp>&) [with _Tp =
std::__cxx11::basic_string<char>; _Dp =
std::default_delete<std::__cxx11::basic_string<char> >]
   23 |     callByValue(A);           // Compilerfehler!!!
      |     ~~~~~^~
In file included from /usr/include/c++/11/memory:76,
               from main.c:2:
/usr/include/c++/11/bits/unique_ptr.h:468:7: note: declared here
   468 |     unique_ptr(const unique_ptr&) = delete;
      |     ^~~~~~
main.c:4:47: note:   initializing argument 1 of 'void
callByValue(std::unique_ptr<std::__cxx11::basic_string<char> >)'
    4 | void callByValue(std::unique_ptr<std::string> input){
      |                               ~~~~~^~~~~

```

Diese konzeptionelle Einschränkung bringt aber einen entscheidenden Vorteil mit sich. Der `unique` Pointer wird allein durch eine Adresse repräsentiert. Es ist kein weiterer Overhead für die Verwaltung des Konstrukts notwendig! Vergleichen Sie dazu die Darstellung unter [cppreference](#)

## Shared Pointers

`std::shared_ptr` sind intelligente Zeiger, die ein Objekt über einen Zeiger "verwalten". Mehrere `shared_ptr` Instanzen können das selbe Objekt besitzen. Das Objekt wird zerstört, wenn:

- der letzte `shared_ptr`, der das Objekt besitzt, zerstört wird oder
- dem letzten `shared_ptr`, der das Objekt besitzt, ein neues Objekt mittels `operator=` oder `reset()` zugewiesen wird.

Das Objekt wird entweder mittels einer `delete`-expression oder einem benutzerdefiniertem deleter zerstört, der dem `shared_ptr` während der Erzeugung übergeben wird.

## SharedPointer.cpp

```
1 #include <iostream>
2 #include <memory>
3
4 class MyClass{
5     public:
6     MyClass(){
7         std::cout << "Constructor executed" << std::endl;
8     }
9     ~MyClass(){
10        std::cout << "Destructor executed" << std::endl;
11    }
12    void print(){
13        std::cout << "That's all!" << std::endl;
14    }
15 };
16
17 int main()
18 {
19     {
20         std::shared_ptr<MyClass> A = std::make_shared<MyClass>();
21         A->print();
22     }
23     std::cout << "Scope left" << std::endl;
24     return EXIT_SUCCESS;
25 }
```

```
Constructor executed
That's all!
Destructor executed
Scope left
```

Gegenwärtig sind Arrays noch etwas knifflig in der Representation und Handhabung. C++17 schafft hier erstmals die Möglichkeit einer adquaten Handhabung (vgl. StackOverflow Diskussionen).

Vergleiche [cppreference](#) für die Nutzung der API.

## Weak Pointers

Ein `weak_ptr` ist im Grunde ein shared Pointer, der die Referenzanzahl nicht erhöht. Es ist definiert als ein intelligenter Zeiger, der eine nicht-besitzende Referenz oder eine schwache Referenz auf ein Objekt enthält, das von einem anderen shared Pointer verwaltet wird.

## WeakPointer.cpp

```
1 #include <iostream>
2 #include <memory>
3
4 std::weak_ptr<int> gw;
5
6 void f()
7 {
8     if (auto spt = gw.lock()) { // Has to be copied into a shared_ptr
9         // before usage
10        std::cout << *spt << "\n";
11    }
12    else {
13        std::cout << "gw is expired\n";
14    }
15 }
16
17 int main()
18 {
19     {
20         auto sp = std::make_shared<int>(42);
21         gw = sp;
22         f();
23     }
24     f();
25 }
```

42

gw is expired

## Regeln für den Einsatz von Pointern

Eine schöne Übersicht zu den Fehlern im Zusammenhang mit Smart Pointer ist unter <https://www.acodersjourney.com> zu finden:

- *Using a shared pointer where an unique pointer suffices*
- *Not making resources/objects shared by shared pointer threadsafe*
- ...
- *Not using `make_shared` to initialize a shared pointer*
- ...

## Aufgabe der Woche

1. Machen Sie sich mit den wichtigsten gdb Mechanismen vertraut. Explorieren Sie den Stack eines Beispielprogrammes und versuchen Sie die Abläufe nachzuvollziehen.
2. Vergleichen Sie die Resultate der Codegrößen für verschiedene Optimierungsstufen. Gelingt es dem Compiler zum Beispiel eine einfache Aufgabe, die Sie geschickt strukturiert haben als solche zu identifizieren?

Eine Hilfe dabei kann die Webseite <https://godbolt.org/> Zudem existiert eine ganze Reihe von youtube Filmen, die Beispiele diskutieren: [Link](#)

3. Experimentieren Sie mit SmartPointern anhand von [https://thispointer.com/learning-shared\\_ptr-part-1-usage-details/](https://thispointer.com/learning-shared_ptr-part-1-usage-details/). Der Kurs fasst die Konzepte in 6 Präsentationen zusammen.