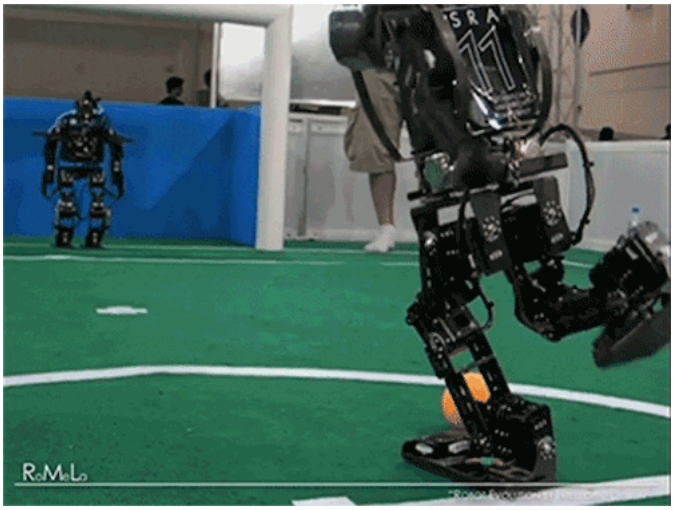
Speicher und Pointer

Parameter	Kursinformationen
Veranstaltung:	`Softwareprojekt Robotik`
Semester	`Wintersemester 2022/23`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Grundlagen der Speicherverwaltung unter C++`
Link auf GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/02_SpeicherUndPointer.md
Autoren	Sebastian Zug & Georg Jäger



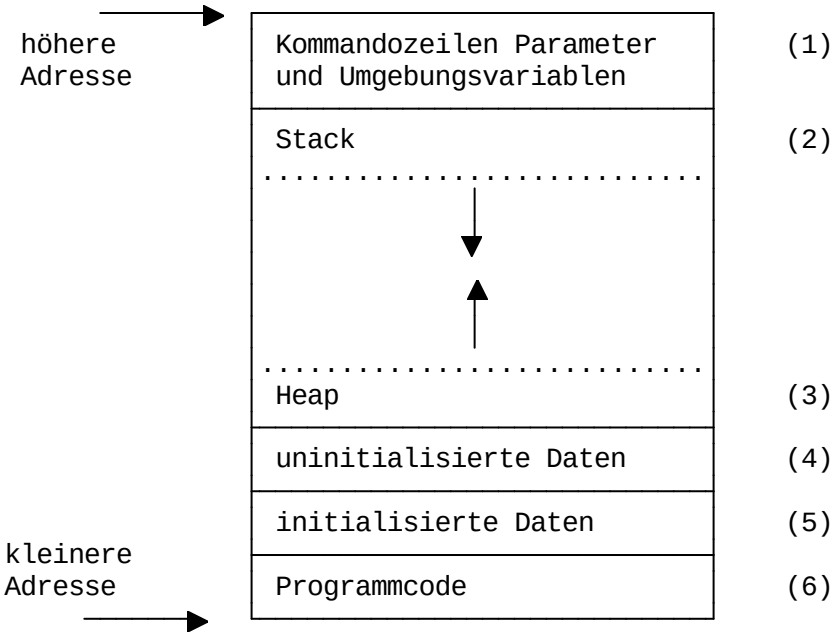
Zielstellung der heutigen Veranstaltung

- 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!

Code	text	data	bss	dec	Bemerkung

`#include <iostream>`
 	
`int main(void)`
`{`
`  return EXIT_SUCCESS;`
`}`
	1918	640	8/4	2566	bss hängt von der Compiler-Konfiguation ab. Mit `-m32` entsteht eine lediglich 4Byte große, nicht initialisierte Variable

`#include <iostream>`
 	
`int global;`
 	
`int main(void)`
`{`
`  static int local;`
`  return EXIT_SUCCESS;`
`}`
	1918	640	16	2566	Rückgabewert, global, local = 3 x 4 Byte

`#include <iostream>`
 	
`int global = 5;`
 	
`int main(void)`
`{`
`  static int local = 3;`
`  return EXIT_SUCCESS;`
`}`
	1918	648	8	2574	Initialisierte globale/statische Variablen

`#include <iostream>`
 	
`int main(void)`
`{`
`  int local = 3;`
`  return EXIT_SUCCESS;`
`}`
	1918	640	8	2566	Keine globalen/statischen Variablen

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
```

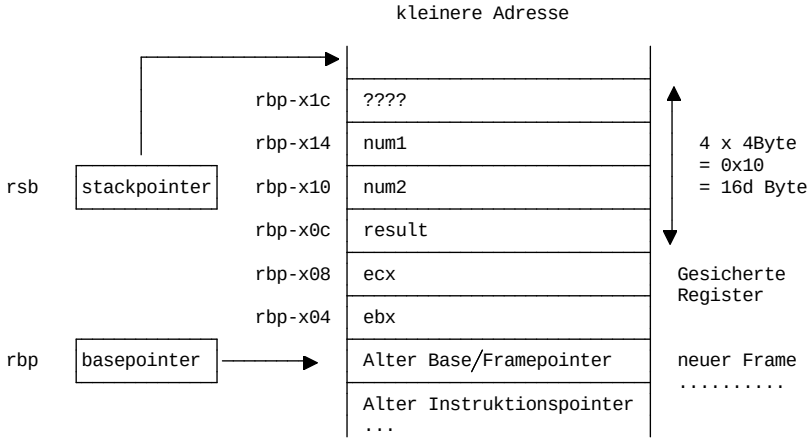
StackExample.cpp

```
 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  }
```

```
Failed to execute 'send' on 'WebSocket': Still in CONNECTING state.
```

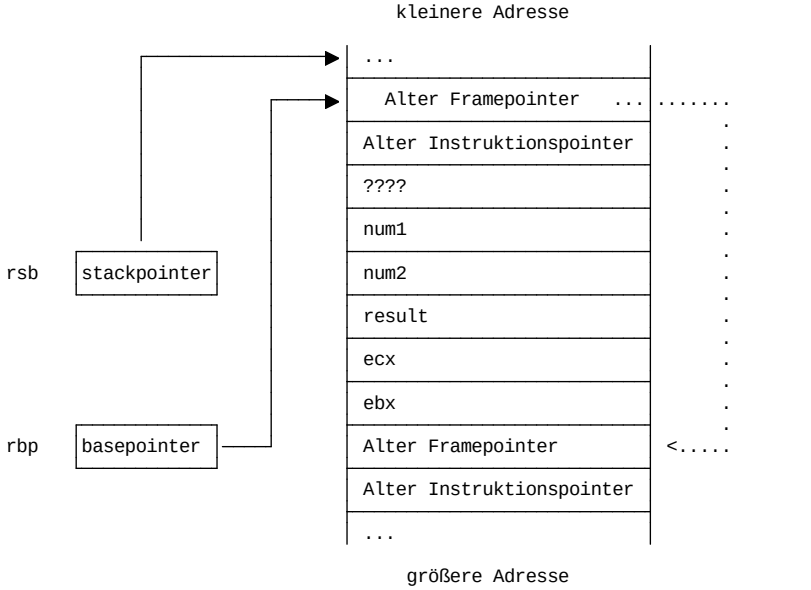
Was passiert beim starten des Programmes und beim Aufruf der Funktion `calc` "unter der Haube"? Schauen wir zunächst die Einrichtung des Stacks von Seiten der `main` funktion bis zur Zeile 12.

```
0x06ed <+10>:    push   %ebp
```



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»
```

	`new`	`malloc`
Bedeutung	Schlüsselwort, Operator	Funktion der Standardbibliothek
Rückgabewert	Pointer vom Typ des Objektes	`void` Pointer
Fehlerfall	Ausnahme	NULL Pointer
Größe	wird vom Compiler bestimmt	muss manuell festgelegt werden
Überschreibarkeit	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

```cpp
#include <iostream>

class MyClass{
  public:
    MyClass()
    {
      std::cout << "Constructor executed" << std::endl;
    }
    ~MyClass()
    {
      std::cout << "Deconstructor executed" << std::endl;
    }
};

int main()
{
  {  // Scope I
    MyClass A;
  }
  {  // Scope II
    MyClass* B = new MyClass;
  }
  std::cout << "Memory Leak !" << std::endl;
  return EXIT_SUCCESS;
}
```

```
Failed to execute 'send' on 'WebSocket': Still in CONNECTING state.
```

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

```cpp
#include <iostream>

int main(void)
{
    double array[1024*1024];
    std::cout << array[1000] << std::endl;
    std::cout << sizeof(double) << std::endl;
    return EXIT_SUCCESS;
}
```

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 de 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

```cpp
#include <stdio.h>

void function(int *aux){
  *aux++;
  printf("Call by 'Reference': %i\n", *aux);
}

void function(int aux){
  aux++;
  printf("Call by Value: %i\n", aux);
}

int main() {
  int i = 3;
  int j = 5;

  // Initialisierung
  // --------------------------------------
  // Pointer auf eine Variable
  int *ptr = &i;
  int *empty_prt;

  // Referenz auf eine Variable
  int &ref = i;
  int &ref_b = i;
  //int &empty_ref; // Error!

  // Reassignment
  ptr = &j;
  //ref = i; // Reassigment nicht möglich

  // Indirection - Funktioniert nicht mit Referenzen
  int **ptr_ptr;  //it is valid.
  ptr = &i;
  ptr_ptr = &ptr;
  printf("0x%p points at %i\n", (void *)ptr, i);
  printf("0x%p points at 0x%p\n\n", (void *)ptr_ptr, *ptr_ptr);

  // Funktionsaufruf
  function(ptr);
  printf("Resultierendes i: %i\n", j);
  function(ref);
  printf("Resultierendes i: %i\n", i);
  return 0;
}
```

```
Failed to execute 'send' on 'WebSocket': Still in CONNECTING state.
```

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

```cpp
#include <iostream>

void DoSomeCalculationsByValue(int number){
  number = number * 2;
}

void DoSomeCalculationsByPointer(int* number){
  *number = *number * 2;
}

void DoSomeCalculationsByReference(int& number){
  number = number * 2;
}

int main()
{
  int number {1};
  DoSomeCalculationsByValue(number);
  std::cout << number << std::endl;
  int* ptr = &number;
  DoSomeCalculationsByPointer(ptr);
  std::cout << *ptr << std::endl;
  int& ref = number;
  DoSomeCalculationsByReference(ref);
  std::cout << ref << std::endl;
  return EXIT_SUCCESS;
}
```

```
Failed to execute 'send' on 'WebSocket': Still in CONNECTING state.
```

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

```cpp
#include <iostream>

class MyClass{
  public:
    MyClass()
    {
      std::cout << "Constructor executed" << std::endl;
    }
    ~MyClass()
    {
      std::cout << "Deconstructor executed" << std::endl;
    }
};

int main()
{
  {
    MyClass* A = new MyClass();
    // ... Some fancy things happen here ...
    delete A;
  }
  return EXIT_SUCCESS;
}
```

```
Failed to execute 'send' on 'WebSocket': Still in CONNECTING state.
```

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?

```cpp
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

```cpp
#include <iostream>
#include <memory>   //<-- Notwendiger Header

class MyClass{
  public:
    MyClass(){
      std::cout << "Constructor executed" << std::endl;
    }
    ~MyClass(){
      std::cout << "Deconstructor executed" << std::endl;
    }
    void print(){
      std::cout << "That's all!" << std::endl;
    }
};

int main()
{
  {
    std::unique_ptr<MyClass> A (new MyClass());
    A->print();
  }
  return EXIT_SUCCESS;
}
```

```
Failed to execute 'send' on 'WebSocket': Still in CONNECTING state.
```

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.

```cpp
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

```cpp
#include <iostream>
#include <memory>

void callByValue(std::unique_ptr<std::string> input){
    std::cout << *input << std::endl;
}

void callByReference(const std::unique_ptr<std::string> & input){
    std::cout << *input << std::endl;
}

void callByRawPointer(std::string* input){
    std::cout << *input << std::endl;
}

int main()
{
  // Variante 1
  //std::unique_ptr<std::string> A (new std::string("Hello World"))
  // Variante 2
  std::unique_ptr<std::string> A = std::make_unique<std::string>("Hello
    World");
  // Ohne Veränderung der Ownership
  callByValue(A);                    // Compilerfehler!!!
  //callByValue(std::move(A));       // Mit Übergabe der Ownership
  callByReference(A);
  callByRawPointer(A.get());
  //std::cout << A << std::endl;
  return EXIT_SUCCESS;
}
```

```
Failed to execute 'send' on 'WebSocket': Still in CONNECTING state.
```

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 mittles 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

```cpp
#include <iostream>
#include <memory>

class MyClass{
  public:
    MyClass(){
      std::cout << "Constructor executed" << std::endl;
    }
    ~MyClass(){
      std::cout << "Deconstructor executed" << std::endl;
    }
    void print(){
      std::cout << "That's all!" << std::endl;
    }
};

int main()
{
  {
    std::shared_ptr<MyClass> A = std::make_shared<MyClass>();
    A->print();
  }
  std::cout << "Scope left" << std::endl;
  return EXIT_SUCCESS;
}
```

```
Failed to execute 'send' on 'WebSocket': Still in CONNECTING state.
```

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

```cpp
#include <iostream>
#include <memory>

std::weak_ptr<int> gw;

void f()
{
    if (auto spt = gw.lock()) { // Has to be copied into a shared_ptr
      before usage
    std::cout << *spt << "\n";
    }
    else {
        std::cout << "gw is expired\n";
    }
}

int main()
{
    {
      auto sp = std::make_shared<int>(42);
      gw = sp;
      f();
    }
    f();
}
```

```
Failed to execute 'send' on 'WebSocket': Still in CONNECTING state.
```

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/. Der Kurs fasst die Konzepte in 6 Präsentationen zusammen.