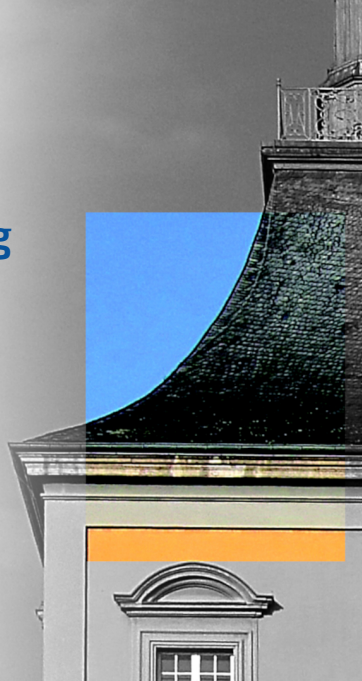
UNIVERSITÄT BONN

Algorithmen und Programmierung

Imperative Programmierung mit C++

Dr. Felix Jonathan Boes
boes@cs.uni-bonn.de
Institut für Informatik



KURZE WIEDERHOLUNG

Die Programmiersprache C++

Wiederkehrenden Programmcode zusammenfassen in Funktionen

UNIVERSITÄT BONN

Funktionen definieren

Wiederkehrende Codeabschnitte werden in Funktionen zusammengefasst. Funktionen werden in C++ wie folgt definiert.

```
RÜCKGABETYP FUNKTIONSNAME (TYP_1 NAME_1, ... TYP_N NAME_N) {
  ...
  return ...;
}
```

Die Parameter sind Variablen die innerhalb der Funktion zur Verfügung stehen. Der Funktionskörper muss in jedem Fall ein **`return`**-Statement erreichen welches eine Expression vom Typ RÜCKGABETYP produziert.

UNIVERSITÄT BONN

Funktionen aufrufen

Beim Nutzen einer Funktion müssen alle geforderten Parameter übergeben werden. Die Parametervariablen sind nur im Funktionskörper bekannt. Pro Parameter wird eine Expression ausgewertet. Der Wert jeder Expression wird dann in der zugehörigen Parametervariablen gespeichert.

Sobald im Funktionskörper ein **`return`**-Statement erreicht, wird die Berechnung der Funktion beendet. Die Funktion selbst ist eine Expression und wertet zu der Expression des **`return`**-Statements aus.

Die Programmiersprache C++

Einschub: Text und Textausgabe

Offene Frage

Wie werden in C++ Texte ein- und ausgegeben?

Welche Textoperationen stellt und C++ zur Verfügung?

UNIVERSITÄT BONN

Einleitung

Da wir Klassen und den Präprozessor noch nicht eingeführt haben, können wir noch nicht im Detail verstehen wie Text verarbeitet werden.

Hier beschreiben wir nur wie man einfach mit Texten arbeitet.

Um mit Texten arbeiten zu können, müssen wir zum Beginn der Programmdatei die folgende Zeile einfügen.

```
#include <string>  // Muss eingebunden werden um mit Texten zu arbeiten
```

UNIVERSITÄT BONN

Die Klasse String

Einfache Texte werden als Instanzen der Klasse `std::string` repräsentiert.

```
#include <string> // Muss eingebunden werden um mit Strings zu arbeiten
...
std::string text = "Hallo Welt"; // Legt einen String an.
```

Strings können verbunden werden. Dabei wird ein neuer String erzeugt.

```
std::string hallo = "Hallo ";
std::string welt  = "Welt";
std::string hallo_welt = hallo + welt;
```

Mit `std::to_string(...)` wird aus einem Objekt ein neuer String erzeugt.

```
std::string text = "Ich bin " + std::to_string(42) + " Jahre alt.";
```

UNIVERSITÄT BONN

Texte ausgeben und einlesen

Um Texte auf der Konsole auszugeben oder von der Konsole einzulesen, müssen wir zum Beginn der Programmdatei die folgende Zeile einfügen.

```
#include <iostream>
```

Diese Zeile sorgt dafür, dass die beiden Objekte `std::cout` und `std::cin` zur Verfügung stehen. Um Texte auszugeben bzw. einzulesen wird auf dem Objekt `std::cout` bzw. `std::cin` operiert.

UNIVERSITÄT BONN

Texte ausgeben

Um Texte auf der Konsole auszugeben wird auf dem Objekt **std::cout** operiert. Dabei wird dem Objekt **std::cout** ein **std::string** mithilfe des Operators << übergeben.

```
#include <iostream>
...
std::cout << "Hallo Welt"; // Druckt Hallo Welt ohne neue Zeile.
```

Der Operator << wird von links nach rechts ausgewertet und erzeugt bei jeder Auswertung das Objekt **std::cout**. Deshalb können auch mehrere Strings durch wiederkehrende << ausgegeben werden.

Die neue Zeile wird durch das Objekt **std::endl** repräsentiert.

```
std::cout << "Ich mag die Zahl " << std::to_string(42) << std::endl;
```

UNIVERSITÄT BONN

Texte einlesen I

Um Texte von der Konsole einzulesen wird auf dem Objekt **std::cin** operiert. Dabei empfängt **std::cin** eine Textzeile von der Texteingabe. Dieser wird an einen **std::string** mithilfe des Operators >> übergeben.

```
#include <iostream>
...
std::string eingegebener_text;
std::cout << "Bitte Text eingeben:" << std::endl;
std::cin  >> eingegebener_text;

std::cout << "Der eingegebene Text lautet '" << eingegebener_text << "'" << std::endl;
```

UNIVERSITÄT BONN

Texte einlesen II

Mit `std::cin` können neben Texten auch andere Datentypen eingelesen werden. Bei korrekter Texteingabe konvertiert die Operation `std::cin >> ...` den eingelesenen Text in den auf der rechten Seite angegebenen Typ.

```
#include <iostream>
double zahl;
std::cout << "Bitte Zahl eingeben:" << std::endl;
std::cin  >> zahl;

std::cout << "Die eingegebene Zahl lautet " << std::to_string(zahl) << "" << std::endl;
```

Falls die Texteingabe ausnahmsweise nicht perfekt ist, bricht die Operation `std::cin >> ...` den Prozess mit einer Fehlermeldung ab. Wir lernen später wie solche Ausnahmen behandelt werden ohne den Prozess zu beenden.

Texte einlesen III

Der Operator >> wird ebenfalls von links nach rechts ausgewertet und erzeugt bei jeder Auswertung das Objekt **std::cin**. Deshalb können auch mehrere Textzeilen durch wiederkehrende >> eingelesen werden.

```
#include <iostream>
std::string zeile_1;
std::string zeile_2;
std::cin >> zeile_1 >> zeile_2;
```

Es ist nicht ratsam mehrere Textzeilen so einzulesen. Der obige Programmcode ist unnötig missverständlich.

Haben Sie Fragen?

Zusammenfassung

Sie haben gelernt wie man einfache Texte erzeugt, ausgibt und einliest

Die Programmiersprache C++

Arrays in C++

Offene Frage

Unser übergeordnetes Ziel ist es, eine Sequenz von Zahlen zu sortieren

Wie legt man in C++ eine Sequenz von Elementen an?

UNIVERSITÄT BONN

Arrays

Als **Array** bezeichnen wir ein zusammenhängendes Stück Speicher indem Werte desselben Typs abgelegt werden. Ein Array ist **statisch** wenn es eine feste, unveränderbare Größe besitzt und es ist **dynamisch** sonst.

In C++ existieren zwei Arten von Arrays, nämlich C++-Arrays und C-Arrays. Die Programmierung mit C++-Arrays führt zu wesentlich besserem Code. Gleichzeitig führen beide Arrayarten zu Maschinencode der identisch (oder gleich effizient) ist. In diesem Modul sprechen wir von **Arrays** und meinen immer **C++-Arrays**.

C-Arrays spielen (nur) in der expliziten Verwaltung von virtuellem Speicher eine Rolle. Wegen einer Vielzahl an (gewollten aber ggf. unerwarteten) Nebeneffekten und Nutzungseinschränkungen sollen sie nur dann eingesetzt werden, wenn die Arbeit mit expliziten Speicherbereichen unbedingt nötig ist.

Statische Arrays
Deklaration und Initiatlisierung

Jedes statische Array hat einen festen Typ. Beispiel:

```
std::array<int, 3>      // Speichert drei int-Werte
std::array<int, 7>      // Speichert sieben int-Werte
std::array<double, 11>  // Speichert elf double-Werte
// Speichert sieben statische Arrays vom Typ std::array<int,3>
std::array<std::array<int, 3>, 7>
// Speichert ANZAHL viele TYP-Werte
std::array<TYP, ANZAHL>
```

Statische Arrays können durch eine in `{...}` eingefasste Liste aller Werte initialisiert werden. Beispiel:

```
std::array<int, 3> x = {42, 9, 11};
```

UNIVERSITÄT BONN

Statische Arrays
Größe und Elementzugriff

Ein statisches Array weiß wie groß es ist. Die Größe eines Arrays `x` wird durch die Memberfunktion[1] `size()` zurückgegeben.

```
std::array<int, 7> x; // Legt ein int-Array der Größe 7 an.
x.size()              // Wertet zu 7 aus.
```

Um auf die Elemente eines Arrays zuzugreifen (lesend und schreiben) gibt man den Index des Elements innerhalb von eckigen Klammern an. Dabei beginnt man die Indizierung immer mit `0`.

```
std::array<int, 7> x; // Legt ein int-Array der Größe 7 an.
x[0] = 77;            // Setzt das erste Element auf 77.
x[4] = 33;            // Setzt Element vier hinter dem Ersten auf 33.
int y = x[2];         // Liest Element zwei hinter dem Ersten.
```

[1] Wir greifen hier Klassen und Memberfunktionen vorweg. Gedulden Sie sich ein wenig, in zwei Vorlesungen gibt es eine erste, kurze Einführung zu Klassen.

UNIVERSITÄT BONN

Statische Arrays
Beispiel

Wir legen ein statisches `int`-Array an und verdoppeln zu einem späteren Zeitpunkt alle Einträge.

```cpp
#include <array>  // Muss eingebunden werden um mit Arrays zu arbeiten
int main() {
  std::array<int, 7> x = {1, 2, 3, 4, 5, 6, 7};
  // Berechne wichtige Dinge...

  for (int i = 0; i < x.size(); i++) {
    x[i] = x[i] * 2;
  }
  // Berechne noch mehr wichtige Dinge...

  return 0;
}
```

UNIVERSITÄT BONN

Statische Arrays
Vermeindliche Probleme

Bei im Programmcode definierten statischen Arrays `std::array<TYP, ANZAHL>` wird pro verwendeter Kombination von TYP und ANZAHL ein eigener Arraytyp im ausführbaren Programm definiert. Den genauen Grund verstehen wir nachdem wir **Templateklassen** eingeführt haben.

Als Konsequenz deklariert der folgende Programmcode drei verschiedene Funktionen.

```
std::array<int, 3> verdoppel_die_werte (std::array<int, 3> x);
std::array<int, 4> verdoppel_die_werte (std::array<int, 4> x);
std::array<int, 5> verdoppel_die_werte (std::array<int, 5> x);
```

Dieses Verhalten ist für fortgeschrittene C++-Programmierer:innen im Allgemeinen erwünscht, aber wir verstehen erst bei der Einführung von Templates warum das so ist. In diesem Beispiel führt dieses Verhalten, zusammen mit dem gewählten Ansatz, zu furchtbarem Code.

UNIVERSITÄT BONN

Dynamische Arrays

Dynamische Arrays speichern Werte desselben Typs und haben keine feste Länge.

Dynamisch Arrays werden ähnlich wie statische Arrays deklariert und initalisiert.

```
// int-Array mit drei Elementen
std::vector<int> x = {42, 9, 11};
// int-Array mit 77 Elementen
std::vector<int> y(77);
y[1] = 4; // Setzt das Element mit Index 1 auf 4.
y.size(); // Gibt 77 zurück.
```

Sie lernen später weitere Operationen kennen um dynamische Arrays zu verändern.

UNIVERSITÄT BONN

Dynamische Arrays
Beispiel

Wir schreiben eine Funktion, die ein dynamisches `int`-Array erhält und ein neues dynamisches `int`-Array derselben Länge erstellt welcher die verdoppelten Werte enthält.

```
#include <vector> // Muss eingebunden werden um mit Arrays zu arbeiten

std::vector<int> verdoppel_die_werte (std::vector<int> in) {
  size_t anz_elemente = in.size();
  std::vector<int> out(anz_elemente);
  for (size_t i = 0; i < anz_elemente; i++) {
    out[i] = 2* in[i];
  }
  return out;
}
```

UNIVERSITÄT BONN

Statische oder dynamische Arrays?

Ein ganz wesentlicher Unterschied zwischen statischen und dynamischen Arrays ist, dass die Größe eines statischem Arraysim Typ des Arrays codiert wird und die Größe eines dynamischen Arrays zur Laufzeit bestimmt wird.

Wenn Sie vor der Frage stehen ob ein statisches Array oder ein dynamisches Array besser für Ihren Anwendungszweck geeignet ist dann nehmen Sie im Zweifel immer das dynamische Array. Nur wenn Sie ganz sicher sind, dass Sie eine Speichersequenz brauchen deren Länge vor der Kompilierungszeitpunkt feststeht, können Sie ein statisches Array verwenden.

Aus diesem Grund bezeichnen wir von nun an **dynamische C++-Arrays** vereinfachend als **Array**.

UNIVERSITÄT BONN

Range-Based-For-Loop I

Um durch die Elemente von Arrays (und die später eingeführten sequentiellen Datentypen) zu iterieren, verwenden moderne C++-Programmierer:innen üblicherweise **Range-Based For-Loops**. Die hier vorgeführten For-Schleifen liefern in jedem Schleifendurchlauf eine Kopie des aktuell betrachteten Elements auf dem gearbeitet werden kann.

```
std::vector<int> mein_lieblingszahlen = { /* ... */ };

for (int kopie_der_aktuellen_zahl : mein_lieblingszahlen) {
  std::cout << "Die nächste Lieblingszahl ist: " << kopie_der_aktuellen_zahl << std::endl;
}
```

Wir lernen gleich, wie man die Elemente eines Arrays unmittelbar durchläuft und diese dabei auch direkt verändern kann.

Haben Sie Fragen?

Zusammenfassung

Sie haben gelernt wie Sie dynamische und statische C++-Arrays anlegen und verwenden

Dynamische C++-Arrays spielen die wichtigste Rolle

Eventankündigungen

Linux-Install-Party (auch auf virutellen Maschinen)
Morgen 17:00 Uhr, B-IT Hörsaal (0.109)

Einsteiger-CTFs
Donerstag 17.11.2022 ab 18:15 Uhr Raum 0.016
Donerstag 08.12.2022 ab 18:15 Uhr Raum 0.016
Donerstag 12.01.2023 ab 18:15 Uhr Raum 0.016

Die Programmiersprache C++

Stackframes und lokale Variablen im Speicher

Offene Frage

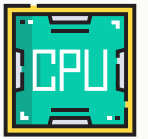
Wie werden die, in einem C++-Programm verarbeiteten Informationen, im Speicher organisiert?

UNIVERSITÄT BONN

Erinnerung: Computermodell

In unserem Computermodell verfügt jeder Prozess über folgende Ressourcen:

Mindestens eine CPU sowie einen virtuelle Adressraum mit 2^{47} Bytes.



0x7FFF FFFF FFFF

0x0000 0000 0000

Das durch C++ beschriebene Modell legt unter anderem fest, welche grundlegenden Datentypen existieren und wie diese Datentypen durch Bytes dargestellt werden.

Offene Fragen

Wie werden Variablen im virtuellen Adressraum organisiert?

Wie werden grundlegende Datentypen als Bytefolge repräsentiert?

Wie werden Funktionsparameter und Returnwerte im virtuellen Adressraum realisiert?

UNIVERSITÄT BONN

Der Stackspeicher im Allgemeinen

In C++ werden (die meisten) Variablen mithilfe eines **Stackspeichers** im virtuellen Adressraum organisiert.

Im Allgemeinen ist ein Stack ein *abstrakter Datentyp* zur Organisation von Elementen der über die folgenden beiden Operationen verfügt:

- Push: Legt ein neues Element auf dem Stack ab
- Pop: Entfernt das zuletzt abgelegte Element (das noch nicht entfernt wurde)

Man bezeichnet einen Stack auch als **Last-In-First-Out (LIFO)**.

UNIVERSITÄT BONN

Der Stackspeicher in C++

In C++ ist der Stackspeicher in sogenannte **Stackframes** eingeteilt[2]. Bei jedem **Funktionsaufruf** wird ein neuer Stackframe auf dem Stack abgelegt. Im Stackframe sind alle Variablen enthalten, die in der Funktion definiert werden.

Wenn die Funktion zurückkehrt (also beendet wird) dann wird der aktuelle Stackframe entfernt[3].

[2] Die x86-64-Architektur verwendet auch einen Stack zur Verwaltung von Speicher. Wenn man zwischen C++ und x86-64 hin und her wechselt, ist es wichtig zu wissen, wann man über den C++-Stack und wann man über den x86-64-Stack spricht.

[3] Auf x86-64-Niveau sind die im Stackframe abgelegten Werte noch zugänglich bis sie überschrieben werden. Das liegt an Optimierungsgründen. Das kann ausgenutzt werden, um Prozesse anzugreifen.

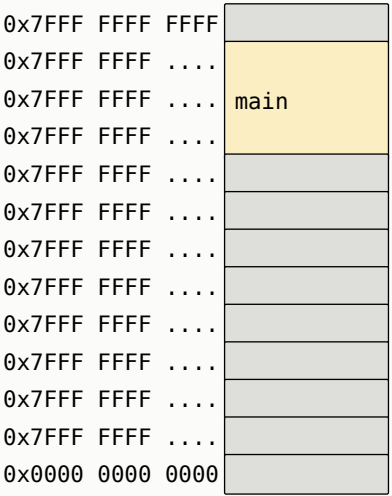
UNIVERSITÄT BONN

Stackframes am Beispiel

```
void g() {
  ...
}

void f() {
  ...
  g();
  ...
}

int main() { ←
  f();
  return 0;
}
```



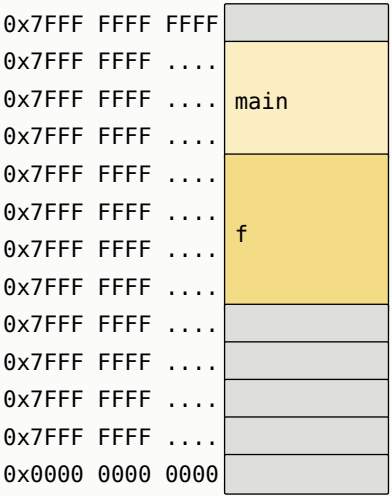
UNIVERSITÄT BONN

Stackframes am Beispiel

```
void g() {
  ...
}

void f() {    ←
  ...
  g();
  ...
}

int main() {
  f();        ←
  return 0;
}
```



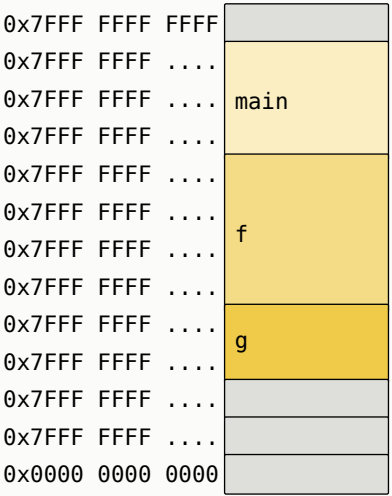
UNIVERSITÄT BONN

Stackframes am Beispiel

```
void g() {    ←
  ...
}

void f() {
  ...
  g();        ←
  ...
}

int main() {
  f();        ←
  return 0;
}
```



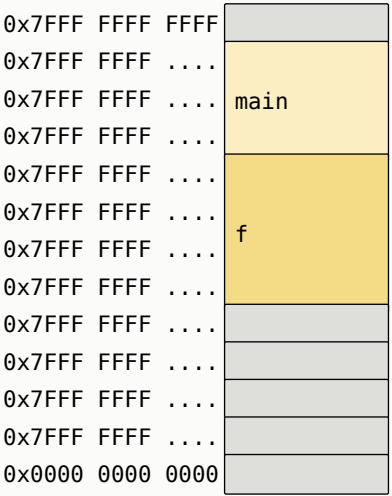
UNIVERSITÄT BONN

Stackframes am Beispiel

```
void g() {
  ...
}

void f() {
  ...
  g();
  ...
}

int main() {
  f();
  return 0;
}
```



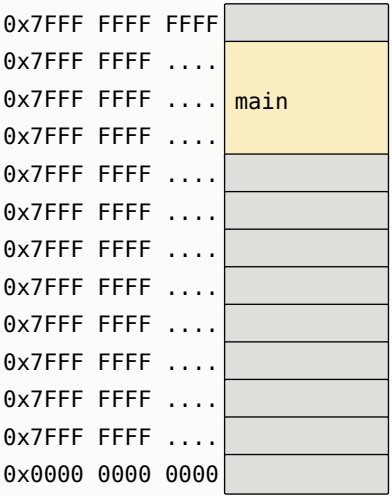
UNIVERSITÄT BONN

Stackframes am Beispiel

```
void g() {
  ...
}

void f() {
  ...
  g();
  ...
}

int main() {
  f();
  return 0;  ←
}
```



Haben Sie Fragen?

UNIVERSITÄT BONN

Zur Bytedarstellung von Datentypen

Jeder grundlegende Datentyp wird im Speicher als zusammenhänger Block von Bytes repräsentiert. Wie das im Detail funktioniert, ist für diese Vorlesung nicht wichtig. Wir schauen uns dennoch ein Beispiel an.

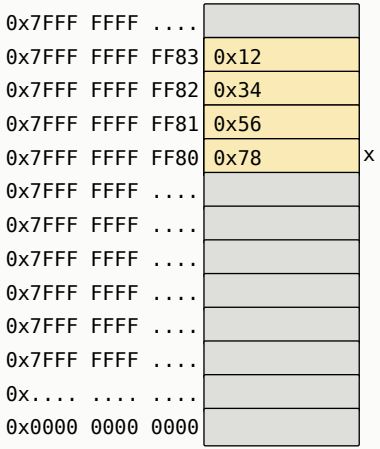
Ein `unsigned int` stellt eine natürliche Zahl `n` durch 4 Byte dar. Dazu stellt man die natürliche Zahl als Hexadezimalzahl mit 8 Ziffern dar, z.B. `n = 0x12345678`. Jeweils zwei Ziffern werden durch ein Byte repräsentiert, z.B. `0x12 0x34 0x56 0x78`.

Auf einer x86-64 Architektur hat das Byte `0x78` die niedrigste Adresse `adr`, das Byte `0x56` hat die Adresse `adr+1` usw.. Diese (willkürlich gewählte) Bytereihenfolge nennt man **little-endian**.

Darstellung einer Variable im Stackframe

Wir legen hier eine `unsigned int`-Variable im Stackframe einer Funktion `f` an.

```
void f() {
  unsigned int x;
  x = 0x12345678;
  ...
}
```



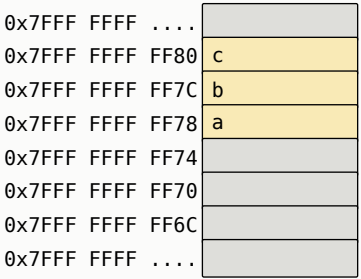
Haben Sie Fragen?

Parameter und lokale Variablen

Beim Funktionsaufruf wird ein neuer Stackframe angelegt indem alle Funktionsparameter[4] und die lokalen Variablen verwaltet werden.

```
void f(int a, int b, int c) {
  unsigned int x;
  unsigned int y;
  unsigned int z;
  ...
}

int main() {
  f(1,2,3);        ←
}
```



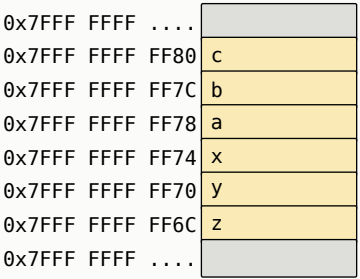
[4] Wir betrachten hier die Calling Convention **cdecl**. Aus Optimierungsgründen werden bei anderen Calling Conventions bis zu sechs Parameter über Register übergeben und ggf. nicht im Stack abgelegt. Diese Optimierung vernachlässigen wir hier. Außerdem vernachlässigen wir abgelegte Statusinformationen.

Parameter und lokale Variablen

Beim Funktionsaufruf wird ein neuer Stackframe angelegt indem alle Funktionsparameter[4] und die lokalen Variablen verwaltet werden.

```
void f(int a, int b, int c) {
  unsigned int x;
  unsigned int y;
  unsigned int z; ←
  ...
}

int main() {
  f(1,2,3);       ←
}
```



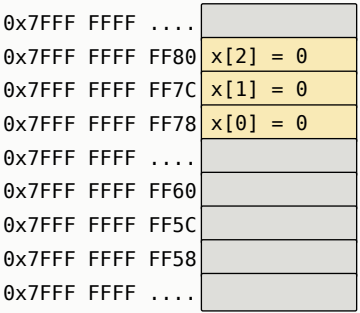
[4] Wir betrachten hier die Calling Convention **cdecl**. Aus Optimierungsgründen werden bei anderen Calling Conventions bis zu sechs Parameter über Register übergeben und ggf. nicht im Stack abgelegt. Diese Optimierung vernachlässigen wir hier. Außerdem vernachlässigen wir abgelegte Statusinformationen.

Rückgabe von Werten

Wir betrachten am Beispiel wie die Rückgabe von Werten umgesetzt wird.

```
std::array<int, 3> f() {
  std::array<int, 3> t;
  ...
  return t;
}

int main() {
  std::array<int, 3> x; ←
  ...
  x = f();
}
```



UNIVERSITÄT BONN

Rückgabe von Werten

Wir betrachten am Beispiel wie die Rückgabe von Werten umgesetzt wird.

```
std::array<int, 3> f() {
  std::array<int, 3> t; ←
  ...
  return t;
}

int main() {
  std::array<int, 3> x;
  ...
  x = f();              ←
}
```

0x7FFF FFFF	
0x7FFF FFFF FF80	x[2] = 0
0x7FFF FFFF FF7C	x[1] = 0
0x7FFF FFFF FF78	x[0] = 0
0x7FFF FFFF	
0x7FFF FFFF FF60	t[2] = 0
0x7FFF FFFF FF5C	t[1] = 0
0x7FFF FFFF FF58	t[0] = 0
0x7FFF FFFF	

UNIVERSITÄT BONN

Rückgabe von Werten

Wir betrachten am Beispiel wie die Rückgabe von Werten umgesetzt wird.

```
std::array<int, 3> f() {
  std::array<int, 3> t;
  ...
  return t;              ←
}

int main() {
  std::array<int, 3> x;
  ...
  x = f();               ←
}
```

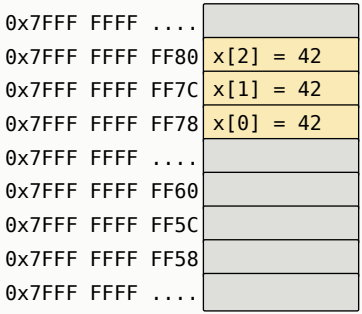
0x7FFF FFFF	
0x7FFF FFFF FF80	x[2] = 0
0x7FFF FFFF FF7C	x[1] = 0
0x7FFF FFFF FF78	x[0] = 0
0x7FFF FFFF	
0x7FFF FFFF FF60	t[2] = 42
0x7FFF FFFF FF5C	t[1] = 42
0x7FFF FFFF FF58	t[0] = 42
0x7FFF FFFF	

Rückgabe von Werten

Wir betrachten am Beispiel wie die Rückgabe von Werten umgesetzt wird.

```
std::array<int, 3> f() {
  std::array<int, 3> t;
  ...
  return t;
}

int main() {
  std::array<int, 3> x;
  ...
  x = f();              ←
}
```



Haben Sie Fragen?

Zusammenfassung

Variablen die in Funktionen definiert werden, liegen auf „dem Stack“

Wie grundlegende Datentypen durch Bytefolgen dargestellt werden, hat weiterführende Gründe und ist hier nicht so wichtig

Wir haben erfahren wie Funktionsparameter und Returnwerte realisiert werden

Die Programmiersprache C++

Der Heapspeicher

Offene Frage

Wie werden dynamische Arrays gespeichert?

UNIVERSITÄT BONN

Motivation

Der Stackframe wird zum Beginn der Funktionsausführung angelegt. Insbesondere ist der Stackframe einer gegebenen Funktion immer gleichgroß. Für dynamische Arrays (und weitere Typen die wir später kennen lernen) ist es deshalb unmöglich die richtige Menge von Speicher auf dem Stack zu reservieren.

Als Beispiel betrachten wir eine Funktion die ein dynamisches Array anlegt welches eine durch die Nutzer bestimmte Größe hat.

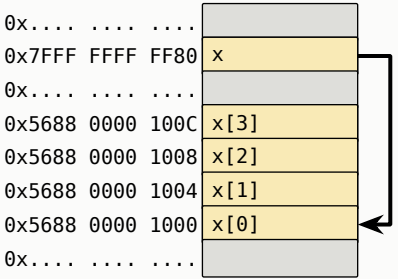
```
void f() {
  int groesse = lies_userinput();
  std::vector<int> x (groesse);
  ...
}
```

UNIVERSITÄT BONN

Heapspeicher und Referenzen

Für Typen deren Größe erst zur Laufzeit bestimmt wird, wird der zugehörige Speicherplatz auf dem **Heapspeicher** verwaltet[5]. Auf dem Stack wird gleichzeitig eine zugehörige **Referenz** abgelegt die auf den reservierten Speicher im Heapspeicher verweist.

```
int main() {
  std::vector<int> x;
  ...
}
```



[5] Der Heapspeicher hat keine wesentliche Gemeinsamkeit mit der Datenstruktur **Heap**. Ein passenderer, aber längerer Name für diesen Speicher wäre wohl **Memory Pool für dynamische Speicherstrukturen**.

Haben Sie Fragen?

Zusammenfassung

Dynamische Arrayvariablen werden durch eine Speicherbereich auf dem Heapspeicher und eine Referenz auf dem Stack realisiert.

Die Programmiersprache C++

Call by Value und Call by (const) Reference

Offene Frage

Bei der Übergabe von Variablen an eine Funktion geschieht die Übergabe als Kopie, denn nur der Wert der Variable wird übergeben

(Wie) ist es möglich, innerhalb der Funktion auf der übergebenen Variablen zu operieren?

UNIVERSITÄT BONN

Motivation

Wir betrachten zunächst zwei Beispiele.

```
void ausgeben(const std::vector<int> in) { ... } // Gibt Array auf der Standardausgabe aus.
```

Bei der Funktion `ausgeben` wird zu Beginn der Funktion eine Kopie des Arrays erzeugt. Im Anschluss wird die Kopie elementweise ausgegeben. Das Anlegen der Kopie scheint unnötig aufwendig.

```
std::vector<int> betrag(std::vector<int> in) { ... }  // Ersetzt jeden Eintrag durch dessen Betrag
```

Bei der Funktion `betrag` wollen wir den elementweisen Betrag eines gegebenen Arrays bestimmen. In unserer Anwendung würde es ausreichen direkt auf dem Array zu arbeiten. Allerdings erstellt die Funktion zu Beginn eine Kopie des Arrays, berechnet auf dieser den elementweise Betrag und gibt das Ergebnis als Rückgabewert zurück. Das Anlegen der Kopie und die Rückgabe scheint unnötig aufwendig und unpraktisch.

UNIVERSITÄT BONN

Call by Value und Call by Reference

Die **Übergabe von Werten** an die zugehörigen Funktionsparametergeschieht per **Call by Value** wenn die Funktionsparameter jeweils als Kopie aus den übergebenen Werten hervorgehen. Als Konsequenz führt eine Veränderung der Funktionsparameter nicht zu einer Veränderung der „übergebenen Variablen“.

Die **Übergabe von Variablen** an die zugehörigen Funktionsparametergeschieht per **Call by Reference** wenn die Funktionsparameter jeweils als Referenz auf die übergebenen Variablen entsteht. Als Konsequenz führt eine Veränderung der Funktionsparameter stets zu einer Veränderung der übergebenen Variable. Es ist hierbei nur möglich Variablen zu übergeben aber nicht allgemeine Werte.

UNIVERSITÄT BONN

Call by Reference in C++

In C++ wird (ohne Weiteres) die Übergabe von Werten an Funktionsparameter via Call by Value realisiert. Um anzugeben dass ein Parameter die Übergabe als Call by Reference realisieren soll, wird dem Parametertyp das Referenzsymbol & angehangen[6].

```
void swap_by_ref(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main() {
  int a = 3;
  int b = 5;
  swap_by_ref(a,b);
  std::cout << std::to_string(a) << ";" << std::to_string(b) << std::endl; // Gibt 5;3 aus.
}
```

[6] Allgemeiner ist es bei der Variablendeklaration möglich durch die Angabe des Referenzsymbol eine Referenzvariablen zu deklarieren. Rein praktisch finden Referenzvariablen aber nur bei Funktionsparametern und **Range-Based For-Loops** Anwendung. Dazu später mehr.

UNIVERSITÄT BONN

Beispiel für Call by (const) Reference

Es gibt gute Gründe einzelne Funktionsparameter via Call by Reference zu übergeben. Die vorangegangene Swapfunktion ist ein wiederkehrendes Beispiel.

In einem anderen prominenten Beispiel wird ein (großer) Parameter innerhalb der Funktion nur zum Auslesen verwendet. Hier scheint es unnötig zu sein, den (großen) Wert in den Stackframe zu kopieren. In diesem Fall bietet sich die Übergabe als **Call by Const Reference** an, das ist die Übergabe einer Referenz auf einen `const`-Parameter.

```
void ausgeben(const std::vector<int>& raus) {
  for (int i; i < raus.size(); i++) {
    std::cout << raus[i] << std::endl;
  }
  raus[0] = 55; // Compilerfehler
}
```

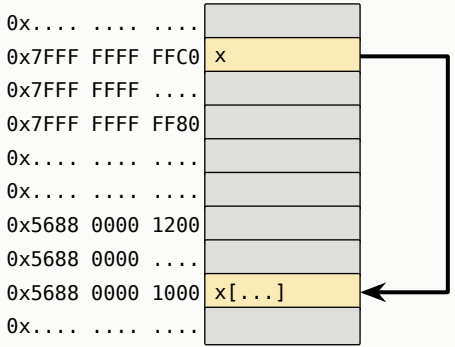
UNIVERSITÄT BONN

Call by Value im Speicher

Wir übergeben zunächst den Wert eines Array via Call by Value.

```
void f(const std::vector<int> z)
{
  ...
}

int main() {
  std::vector<int> x; ←
  ...
  ausgeben(x);
}
```



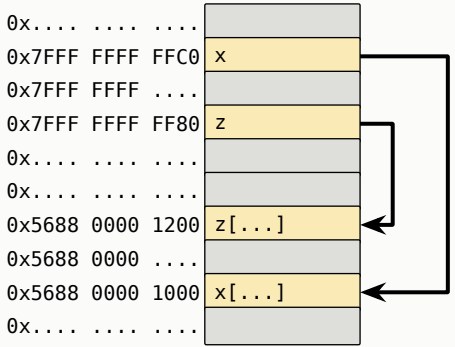
UNIVERSITÄT BONN

Call by Value im Speicher

Wir übergeben zunächst den Wert eines Array via Call by Value.

```
void f(const std::vector<int> z)
{
  ...          ←
}

int main() {
  std::vector<int> x;
  ...
  ausgeben(x); ←
}
```



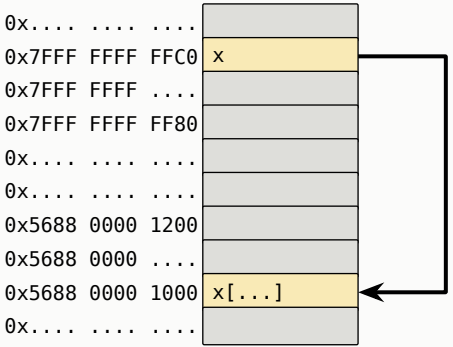
UNIVERSITÄT BONN

Call by Reference im Speicher

Wir übergeben nun eine Arrayvariable via Call by Reference.

```
void f(const std::vector<int>& z)
{
  ...
}

int main() {
  std::vector<int> x; ←
  ...
  ausgeben(x);
}
```



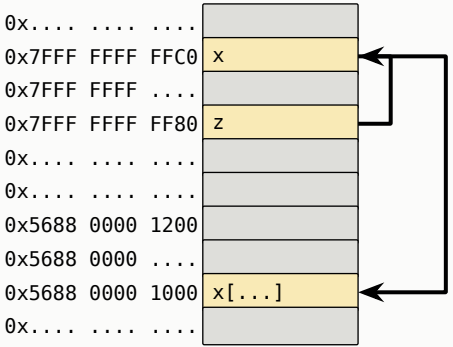
UNIVERSITÄT BONN

Call by Reference im Speicher

Wir übergeben nun eine Arrayvariable via Call by Reference.

```
void f(const std::vector<int>& z)
{
  ...          ←
}

int main() {
  std::vector<int> x;
  ...
  ausgeben(x); ←
}
```



UNIVERSITÄT BONN

Range-Based-For-Loop II

Range-Based For-Loops können ebenfalls die Elemente als Referenz oder als const-Referenz verarbeiten.

```
std::vector<int> zahlen = { 2, 3, 5, 7 };

for (int& zahl : zahlen) {
  zahl = zahl* 2
}

for (const int& zahl : zahlen) {
  std::cout << zahl << " -> ";
} // Druckt "4 -> 6 -> 10 -> 14 -> "
```

Haben Sie Fragen?

Zusammenfassung

Lokale Variablen und Funktionsparameter werden in einem, zur Funktion gehörendem Stackframe organisiert

Dynamische Datentypen werden (auch) im Heapspeicher organisiert

Typischerweise werden Parameter per Call By Value übergeben; um Variablen an Funktionen (zum Lesen) zu übergeben, verwendet man Call By (const) Reference