High-Performance Computing Exercise 01

Habib-Ullah Mubarak
6627597
s3091627@stud.uni-frankfurt.de

Eraycan Gökcan
7000210
s9292109@stud.uni-frankfurt.de

Basel Ali
7197202
s3046220@stud.uni-frankfurt.de

May 7, 2025

Exercise 1.1: Hello World? Hello World!

```cpp
#include <iostream>

int main(int argc, char** argv) {
    std::cout << "Hello world!" << std::endl;

    return 0;
}
```

Listing 1: HelloWorld.cpp

Exercise 1.2: Conditional Statements and Loops

```cpp
# include <iostream>

int main(){
    int n;
    std::cout << "Geben Sie eine Zahl n zwischen 0 und 9: ";
    std::cin >> n;

    if (std::cin.fail()){
        std::cout << "Eingabe war keine Zahl" << std::endl;
        return 1;
    }
```

```
12
13      if(0<=n && n <=9){
14          for(int i=0; i<n; i++){
15              std::cout << "Hello world!" << std::endl;
16          }
17      }
18      else{
19          std::cout << "n liegt nicht zwischen 0 und 9." << std::endl;
20      }
21      return 0;
22  }
```

Listing 2: ConditionalStatements.cpp

Das Programm beginnt mit dem Einbinden der Bibliothek `iostream`, die für die Ein- und Ausgabe über die Konsole benötigt wird. In der `main`-Funktion wird zunächst eine Ganzzahlvariable `n` deklariert. Anschließend fordert das Programm den Benutzer auf, eine Zahl zwischen 0 und 9 einzugeben, und liest diese Zahl mit `std::cin` ein. Danach überprüft das Programm, ob bei der Eingabe ein Fehler aufgetreten ist – etwa, weil der Benutzer keine Zahl eingegeben hat. In diesem Fall gibt das Programm eine Fehlermeldung aus und beendet sich mit dem Rückgabewert 1. Wenn die Eingabe erfolgreich war, prüft das Programm, ob n im gültigen Bereich zwischen 0 und 9 liegt. Falls ja, wird mithilfe einer Schleife n-mal die Zeile „`Hello world!`“ auf dem Bildschirm ausgegeben. Liegt die eingegebene Zahl außerhalb dieses Bereichs, wird eine entsprechende Fehlermeldung angezeigt. Am Ende gibt die Funktion den Wert 0 zurück, um den erfolgreichen Abschluss des Programms zu signalisieren. `std::cin.fail()` prüft, ob beim Einlesen mit std::cin ein Fehler aufgetreten ist – zum Beispiel, wenn der Benutzer statt einer Zahl einen Buchstaben eingibt. In diesem Fall wird der Eingabestream in einen Fehlerzustand versetzt, und `cin.fail()` gibt true zurück.

Exercise 1.3: Troubleshooting

i) Factorials

```
1  #include<iostream>
2
3
4  int* GetFactorials(int n){
5
6    /*
7    vorher stand: int a[n];
8    Der Code kompiliert erfolgreich aber damit liegt das Array nur tempor r im
         Stack.
9    Wird automatisch gel scht , sobald die Funktion endet.
10   Ein Zeiger darauf zeigt danach ins Leere.
11   */
12   int *a = new int[n];
13   /*
```

```
14
15    */
16
17    a[0] = 1;
18    for (int i = 1; i < n; ++i)
19      a[i] = i*a[i-1];
20
21    return a;
22  }
23
24
25  int main()
26  {
27    int n = 10;
28    int *a = GetFactorials(n);
29
30    for (int i = 0; i < n; ++i) std::cout << a[i] << std::endl;
31
32    delete[] a;
33
34    return 0;
35  }
```

Listing 3: Factorial.cpp

Nein, der Code funktioniert nicht korrekt. Er kompiliert möglicherweise (je nach Compiler), aber zur Laufzeit ist der Rückgabewert ungültig, da ein Zeiger auf ein lokales Array (`int a[n]`) zurückgegeben wird, das nach dem Funktionsende automatisch gelöscht wird.

Die Verwendung von `new int[n]` legt das Array stattdessen im Heap an, nicht im Stack. Speicher im Heap bleibt auch nach dem Ende der Funktion gültig. Dadurch erhält man einen Zeiger auf gültigen Speicher, den man zurückgeben und weiterverwenden kann. Das Programm gibt die Fakultäten der Zahlen von 0 bis 9 aus. Für den Eingabewert $n = 10$ lautet die Ausgabe:

```
1
1
2
6
24
120
720
5040
40320
362880
```

ii) Pointers and Functions

```cpp
#include<iostream>

void piPointer1(float* pi)
{
  *pi = 3.14;
}

// Diese Funktion gibt einen Zeiger auf dynamisch allokierten Speicher im
    Heap zur ck.
float* piPointer2()
{
  float* pi = new float{3.1415}; // mit new wird Speicher im Heap zugewiesen
      -> Ausgabe ist ein Pointer
  return pi;
}

int main() {
  float pi{0};  // Initialisierung von pi, da vorher auf nichts gezeigt da
      Initialisierung fehlte
  float* pi1{&pi}; // Initialisierung des Zeigers pi1 auf die Adresse von pi
  piPointer1(pi1); // Der Wert von pi wird auf 3.14 gesetzt
  std::cout << *pi1 << std::endl; // Ausgabe von pi (3.14)

  float* pi2 = piPointer2(); // Zeiger pi2 auf dynamisch zugewiesenen
      Speicher
  std::cout << *pi2 << std::endl; // Ausgabe von pi2 (3.1415)

  delete pi2; // Speicher im Heap wird freigegeben

  return 0;
}
```

Listing 4: PointersAndFunctions.cpp

Der ursprüngliche Code kompiliert möglicherweise (je nach Compiler), aber zur Laufzeit ist der Rückgabewert ungültig aus zwei Gründen:
Zum Ersten wird in der Funktion `piPointer2` der Zeiger auf eine lokale Variable (`float pi`) zurückgegeben. Diese Variable existiert jedoch nur während der Funktionsausführung und wird nach dem Verlassen der Funktion gelöscht. Das bedeutet, der Zeiger verweist auf einen ungültigen Speicherbereich (den Stack-Speicher, der nicht mehr gültig ist). Dies führt zu undefiniertem Verhalten, wenn versucht wird, auf diesen Speicher zuzugreifen. In unserem abgeänderten Code wird `new float{3.1415}` verwendet, um Speicher auf dem Heap zu reservieren. Der Heap-Speicher bleibt nach dem Verlassen der Funktion gültig und der Zeiger kann sicher verwendet werden.
Des Weiteren wird der Zeiger `pi1` im main-Programm deklariert, aber nicht initialisiert, bevor er in der Funktion `piPointer1` verwendet wird. Ein nicht initialisierter Zeiger könnte auf eine zufällige Speicheradresse zeigen und zu undefiniertem Verhalten führen. Wir haben dafür den Zeiger `pi1` korrekt mit `float pi{0};` initialisiert und dann die Adresse von pi an `piPointer1` übergeben.

Ausgabe:

```
3.14
3.1415
```

iii) Function Arguments

```cpp
#include<iostream>

/*
Wenn man int arg als Parameter ohne & oder *  bergibst , arbeitet man mit einer Kopie.
 nderungen  in der Funktion betreffen nur diese Kopie
Das Original bleibt unver ndert es sei denn man gibt den neuen Wert zur ck und speichert ihn
oder arbeitest mit Zeiger oder Referenz
*/
void number_increment(int arg)
{
  ++arg;
}

/*
arg ist wieder eine Kopie
Diese Kopie wird lokal ver ndert (++arg) und zur ckgegeben
*/
int copy_increment(int arg)
{
  ++arg;

  return arg;
}

/*
Man  bergibt  &number (die Adresse der Variable).
arg ist also ein Zeiger (auf number)
*arg bedeutet: Man geht an die Adresse und greift auf den Wert zu.
++*arg erh ht also direkt den Wert von number.
*/
void pointer_increment(int* arg)
{
  // TODO
  ++*arg;
}

/*
arg ist eine Referenz auf number    kein Zeiger, sondern ein anderen Namen
*/
void reference_increment(int& arg)
{
  // TODO
  ++arg;
}

```

```
46
47 int main () {
48   int number = 0;
49
50   std::cout << "Number is: " << number  << std::endl;
51
52   // 1st case:
53   number_increment(number);
54   // Der Funktionsaufruf number_increment(number) erzeugt eine Kopie von
         number
55   // Es wird ein neuer Speicherplatz f r arg angelegt.
56   // arg = 0 (Kopie des Werts von number)
57   // ++arg erh ht diese Kopie     arg = 1
58   // Funktion endet     arg wird gel scht (Speicher freigegeben)
59   // number bleibt unver ndert bei 0
60   std::cout << "Number is: " << number  << std::endl;
61
62   // 2nd case:
63   number = copy_increment(number);
64   // Man   berschreibt   number mit dem R ckgabewert von arg
65   std::cout << "Number is: " << number << std::endl;
66
67   // 3rd case:
68   pointer_increment( &number /* TODO */ );
69   // geht an die Adresse von number und erh ht direkt den Wert von number
70   std::cout << "Number is: " << number << std::endl;
71
72   // 4th case:
73   reference_increment( number /* TODO */ );
74   //   nderungen   an arg sind direkte   nderungen   an number
75   // arg ist jetzt nur ein anderer Name f r number
76   // ++arg = ++number     number = 3
77   std::cout << "Number is: " << number << std::endl;
78
79   return 0;
80 }
```

Listing 5: FunctionArguments.cpp

Zu Beginn wird die Variable number auf 0 gesetzt. In der ersten Funktion `number_increment` wird number per Call-by-Value übergeben. Das bedeutet, dass nur eine Kopie von number verändert wird, die ursprüngliche Variable bleibt unverändert – daher bleibt der Ausgabewert 0.

Im zweiten Fall wird number an `copy_increment` übergeben, ebenfalls per Call-by-Value. Hier wird jedoch der Rückgabewert der Funktion verwendet, der die erhöhte Kopie enthält. Dieser Rückgabewert wird number zugewiesen, weshalb number nun den Wert 1 hat.

Im dritten Fall wird number über einen Zeiger an die Funktion `pointer_increment` übergeben (Call-by-Pointer). Die Funktion dereferenziert den Zeiger und erhöht so direkt den Wert von number. Dadurch wird number auf 2 erhöht.

Im vierten Fall wird number per Referenz an `reference_increment` übergeben (Call-by-Reference). Dadurch kann die Funktion direkt auf die originale Variable zugreifen und sie verändern. number wird dadurch auf 3 erhöht.

Ausgabe:

```
Number is: 0
Number is: 0
Number is: 1
Number is: 2
Number is: 3
```

iv) Templates

```cpp
#include <iostream>

/*
Die Funktion funktioniert f r jeden Datentype T, z.B. int, double, float,
    short, char
*/
template <typename T>
T GetMax (T a, T b) {
  T result;

  if (a > b) result = a;
  else result = b;

  return result;
}


int main () {
  int i=5, j=6;
  double l=9.2, m=2e+9;

  /*
  Wir gehen davon aus, dass die Ausgabe 6 sein wird,
  weil 6 > 5 ist und kein Casting stattfindet
  */
  std::cout << "int: " << GetMax<int>(i, j) << std::endl;
  // AUSGABE: 6  ==> Erwartung erf llt

  /*
  Wir erwarten, dass die Ausgabe 2e+09 ist,
  weil 2e+09 > 9,2 ist und kein Casting stattfindet
  */
  std::cout << "double: " << GetMax<double>(l, m) << std::endl;
  // AUSGABE: 2e+09  ==> Erwartung erf llt

  /*
  Wir erwarten, dass die Ausgabe 2e+09 sein wird,
```

```
37    weil 2e+09 > 9,2 ist und float eine Gr   e von 32 Bits hat, was ausreicht,
38    um 2e+09 im IEEE 754-Standard darzustellen.
39    */
40    std::cout << "float:␣" <<  GetMax<float>(1, m) << std::endl;
41    // AUSGABE: 2e+09  ==> Erwartung erf llt
42
43    /*
44    Wir erwarten, dass die Ausgabe 2e+09 sein wird,
45    weil 2e+09 > 9 ist und int hat auch eine Gr   e von 32 Bits,
46    was ausreicht, um 2e+09 darzustellen, und 9,2 wird in 9 umgewandelt
47    */
48    std::cout << "int:␣" <<  GetMax<int>(1, m) << std::endl;
49    // AUSGABE: 2000000000  ==> Erwartung im Grunde erf llt
50
51    /*
52    Wir wissen nicht, wie die Ausgabe aussehen wird,
53    weil wir nicht genau wei  , wie der Overflow gehandhabt werden soll
54    */
55    std::cout << "short:␣" <<  GetMax<short>(1, m) << std::endl;
56    // AUSGABE: 9
57    /*
58    Die Ausgabe ist 9, weil short eine Gr   e von 16 Bits hat,
59    die nicht ausreicht um 2e+09 darzustellen und nur die ersten 16 Bits
60    von 2e+09 an die neue Speicherstelle   bertragen   werden,
61    die in diesem Fall -27648 (im Zweierkomplement) und 9 > -27648 darstellen
62    */
63
64    /*
65    Wir wissen nicht, wie die Ausgabe aussehen wird,
66    weil wir nicht genau wei  , wie der Overflow gehandhabt werden soll
67    */
68    std::cout << "char:␣" <<  GetMax<char>(1, m) << std::endl;
69    // AUSGABE:  <== HT (horizontal tab)
70    /*
71    Die Ausgabe ist   (9), weil char eine Gr   e von 8 Bits hat, was nicht
          ausreicht,
72    um 2e+09 darzustellen und nur die ersten 8 Bits von 2e+09 an die neue
          Speicherstelle   bertragen   werden,
73    die in diesem Fall 0 darstellen, so dass 9 > 0 und 9 der ASCII-Code f  r HT
           (horizontaler Tabulator) ist
74    */
75
76    return 0;
77  }
```

Listing 6: templates.cpp

Exercise 1.4: I Also Like to Live Dangerously

Wir haben erwartet, dass die Ausgabe einfach `p1[i] = i p2[i] = i` für alle i von 0 bis 9 ist. Unter Windows funktioniert das tatsächlich so, das Programm läuft ohne Fehlermeldung und die Ausgabe stimmt mit den Erwartungen überein. Unter Linux ist das aber anders:

Dort wird z.B. `p2[3]` überschrieben, nachdem in `p1[15]` der Wert `1015` geschrieben wurde. Das zeigt, dass p1 und `p2` im Speicher sehr nah beieinander liegen. Außerdem bekommen wir eine Fehlermeldung beim Beenden des Programms: double free or corruption (out). Der Grund dafür ist wahrscheinlich, dass durch `p1[11] = 0111` Speicher überschrieben wird, der gar nicht zu `p1` gehört und ein Problem mit dem Speicherverwaltungsbereich des Heaps entsteht. Wenn wir die problematischen Zeilen (15 und 17) auskommentieren, läuft das Programm zwar ohne Fehlermeldung, aber `p2[3]` bleibt trotzdem falsch, denn es wurde vorher schon überschrieben. Man greift auf Speicher zu, den man nicht reserviert hat, und das kann auf anderen Systemen oder bei kleinen Änderungen zu komplett unvorhersehbarem Verhalten führen, deshalb ist undefiniertes Verhalten gefährlich und sollte unbedingt vermieden werden.
Ausgabe:

```
p1[0] = 0 p2[0] = 0
p1[1] = 1 p2[1] = 1
p1[2] = 2 p2[2] = 2
p1[3] = 3 p2[3] = 1015
p1[4] = 4 p2[4] = 4
p1[5] = 5 p2[5] = 5
p1[6] = 6 p2[6] = 6
p1[7] = 7 p2[7] = 7
p1[8] = 8 p2[8] = 8
p1[9] = 9 p2[9] = 9
double free or corruption (out)
Aborted (core dumped)
```

Exercise 1.5: Fun with Pointers

Im Code wird das gleiche Array auf zwei Arten ausgegeben: Beim ersten Durchlauf wird einfach der Indexzugriff `arr[idx]` verwendet – klassisch über das Array. Beim zweiten Durchlauf wird ein Pointer `ptr` benutzt, der auf das Array zeigt. Der wird dann inkrementiert und über `*ptr` dereferenziert, um die Werte zu holen. Beides liefert den gleichen Output, aber der Pointer-Walk zeigt nochmal schön, wie sich die Adresse verändert.

i) Memory Locations:

Die Adresse vom ersten Element ist nicht fest, die kann sich bei jedem Programmstart ändern – hängt davon ab, wo gerade Platz ist im Speicher. Die Adressen der Elemente im Array steigen immer um 4 (weil int meistens 4 Bytes braucht). Also liegt `arr[1]` genau 4 Byte nach `arr[0]`, dann `arr[2]` nochmal 4 weiter usw. Der Pointer `ptr` selbst hat eine

eigene Adresse und zeigt halt auf das Array. Man sieht im zweiten Loop auch `&ptr` – das ist die Adresse vom Pointer selbst, nicht die vom Inhalt.

ii) Size of Arrays:

`sizeof(arr)` gibt die Größe des ganzen Arrays in Bytes zurück, `sizeof(arr[0])` die Größe eines Elements. Division ergibt also die Anzahl an Elementen. Das klappt aber nur bei statisch deklarierten Arrays. Wenn man mit `new` ein Array anlegt oder nur einen Pointer übergibt, dann kennt `sizeof` die Länge nicht mehr – es weiß dann nur noch, dass da ein Pointer ist, also kommen dann z.B. 8 raus (Pointer-Größe auf 64-bit-Systemen).

Exercise 1.6: The Power of References

Das Originalprogramm war langsam, weil in jeder Iteration ein riesiger `std::vector<int>` (Größe: 10 Millionen Elemente) kopiert wurde, sobald die Funktion `CalculateSum()` aufgerufen wurde. Der Grund: Die Parameterübergabe war per Value (`std::vector<int> values`), was heißt, es wurde jedes Mal ein komplett neuer Vektor angelegt und befüllt – und das hundertmal.
Wir haben den Parameter mit einem einzigen Zeichen geändert:
`int CalculateSum(std::vector<int>& values)`
Das & sorgt dafür, dass der Vektor per Referenz übergeben wird – also die Funktion greift direkt auf das bestehende Objekt im Speicher zu.

Ausgabe ohne &:

```
887459712 calculated in 53.2361
```

Ausgabe mit &:

```
887459712 calculated in 34.8268
```

Weil das Kopieren von großen Objekten längere Laufzeit beansprucht – gerade bei `std::vector`, der intern Speicher auf dem Heap benutzt. Pass-by-Reference vermeidet das komplett.

Exercise 1.7: Debugger

i) Write a How-to Guide

- **Kompilieren mit Debug-Flag:**
  Beim Kompilieren mit `g++` sollte das Flag `-g` verwendet werden:

```
g++ -g -o my_program my_source.cpp
```

  Dieses Flag sorgt dafür, dass zusätzliche Debugging-Informationen in das Binary eingebettet werden.

- **Programm mit GDB starten:**

```
gdb my_program
```

- **Breakpoints setzen:**
  - Nach Funktionsnamen: `break main`
  - Nach Datei und Zeilennummer: `break my_file.cpp:15`

  Breakpoints ermöglichen es, das Programm an bestimmten Stellen anzuhalten, um den Zustand zu analysieren.

- **Ausführung starten:**

```
run
```

- **Programmanalyse während Ausführung:**
  - `next` – Nächste Zeile ausführen (ohne in Funktionen zu springen)
  - `step` – Nächste Zeile ausführen (inkl. Funktionssprung)
  - `continue` – Bis zum nächsten Breakpoint fortfahren
  - `print <variable>` – Wert einer Variablen anzeigen

- **GDB beenden:**

```
quit
```

ii) Debugger-Lauf mit Arrays.cpp

Screenshot

```
achilles:Aufgabe> g++ -g Arrays.cpp -o ArraysDebug
achilles:Aufgabe> gdb ArraysDebug
GNU gdb (Fedora Linux) 16.2-2.fc41
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ArraysDebug...
(gdb) run
Starting program: /home/users3/lio/s3091627/Documents/HL Prak 2025/Blatt 1/Aufgabe/ArraysDebug

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
p1[0] = 0 p2[0] = 0
p1[1] = 1 p2[1] = 1
p1[2] = 2 p2[2] = 2
p1[3] = 3 p2[3] = 1015
p1[4] = 4 p2[4] = 4
p1[5] = 5 p2[5] = 5
p1[6] = 6 p2[6] = 6
p1[7] = 7 p2[7] = 7
p1[8] = 8 p2[8] = 8
p1[9] = 9 p2[9] = 9
[Inferior 1 (process 946977) exited normally]
```

Aufgrund der Aufgabenstellung (Zeile 15 und 17 auskommentieren) haben wir vermutet, dass das Programm möglicherweise abstürzt oder fehlerhafte Werte ausgibt. Das Programm lief jedoch fehlerfrei durch und gab alle Werte korrekt aus. Die Debug-Ausgabe unterschied sich nicht von der normalen Ausführung. Das Entfernen der besagten Zeilen hatte vermutlich keinen Einfluss auf die Funktionalität.