**Thema, Ziele:** Effizienzbetrachtungen C vs. C++, Inlining

## Aufgabe 1: Effizienzbetrachtungen bei Minimalprogrammen

In dieser Aufgabe sollen Minimalprogramme auf deren Grösse untersucht werden. Insbesondere sollen die GNU-Compiler gcc, bzw. g++ mit unterschiedlichen Optionen untersucht werden.

Für diese Aufgabe sind folgende Optionen der GNU-Compiler nützlich:

| | |
|---|---|
| `-E` | Precompile only, der Output wird auf stdout geschrieben |
| `-S` | Assembleroutput, ohne Objectfile erzeugen |
| `-c` | nur compilieren |
| `-O0` | keine Optimierung |
| `-O1` | Optimierungsstufe 1 (siehe g++ - Help für Details) |
| `-O2` | Optimierungsstufe 2 |
| `-O3` | Optimierungsstufe 3 |
| `-Os` | Optimierung auf Codegrösse |

**Hinweis**: Verwenden Sie für diese Untersuchungen die Console, Eclipse bietet keine Vorteile.

a) Nehmen Sie das minimale C++-Programm ohne jedes Include, d.h. den folgenden Code:

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

Compilieren Sie dieses Programm mit gcc und mit g++ mit unterschiedlichen Optimierungsstufen. Betrachten Sie jeweils das entstandene Assemblerfile und notieren Sie die Grösse des ausführbaren Programms.

b) Fügen Sie ein `#include <iostream>` an, ohne irgendeine Funktion davon zu nutzen.

```
#include <iostream>
int main(void)
{
  return 0;
}
```

Compilieren Sie dieses Programm mit g++ vorerst als Precompile only. Dadurch erhalten Sie einen Eindruck von der Anzahl Zeilen, die der Compiler jedesmal zu übersetzen hat. Compilieren Sie anschliessend mit unterschiedlichen Optimierungsstufen. Betrachten Sie jeweils das entstandene Assemblerfile und notieren Sie die Grösse des ausführbaren Programms. Im Verzeichnis `./bin` erhalten Sie eine ältere Version des C++-Compilers. Führen Sie Ihre Untersuchungen auch mit diesem Compiler durch.

c) Erweitern Sie das Programm wie folgt.

```
#include <iostream>
int main(void)
{
  std::cout << "Hello World";
  return 0;
}
```

Compilieren Sie dieses Programm mit g++ mit unterschiedlichen Optimierungsstufen. Betrachten Sie jeweils das entstandene Assemblerfile und notieren Sie die Grösse des ausführbaren Programms.

d) Ersetzen Sie nun die iostream-Bibliothek und cout durch stdio und printf().

```
#include <cstdio>
int main(void)
{
  printf("Hello World");
  return 0;
}
```

Compilieren Sie dieses Programm mit g++ vorerst als Precompile only. Compilieren Sie dieses Programm wiederum mit g++ mit unterschiedlichen Optimierungsstufen. Betrachten Sie jeweils das entstandene Assemblerfile und notieren Sie die Grösse des ausführbaren Programms.

e) Welche Schlussfolgerungen ziehen Sie aus Ihren Untersuchungen?

## **Aufgabe 2:** inline-Methoden

Bei sehr kurzen Methoden (Einzeiler) ist der Overhead eines Funktionsaufrufs recht gross. Durch Inlining kann der Funktionsaufruf vermieden werden, der Code (Funktionsrumpf) wird vom Compiler direkt anstelle des Funktionsaufrufs gesetzt. Der Compiler wird üblicherweise keinen Inlinecode erzeugen, falls die Funktion rekursiv aufgerufen wird oder falls ein Pointer auf diese Funktion verwendet wird. Eine weitere Schwierigkeit ergibt sich, wenn die Inlinefunktionen in mehreren Sourcefiles verwendet werden sollen, der Linker kann aus einer bestehenden Objektdatei kaum Inlinecode erzeugen.

Für diese Aufgabe sind zusätzlich die folgenden Optionen der GNU-Compiler nützlich:

`-Winline` Erzeugt eine Warnung, falls von einer mit `inline` spezifizierten Funktion kein Inlinecode erzeugt werden konnte.

a) Wenn bei Klassendeklarationen der Code direkt definiert wird, sind diese Funktionen implizit inline. Verwenden Sie den im Verzeichnis `./Vorgabe` zur Verfügung gestellte C++-Code. Wie Sie feststellen, ist die Methode `getArea()` nicht inline.
Compilieren Sie dieses Programm, ein Makefile steht zur Verfügung.

b) Betrachten Sie die erzeugten Assemblerfiles. Welche Methoden sind inline, welche nicht?
(Hinweis: betrachten Sie dazu die call-Befehle)?

c) Wenn Sie die einzelnen `*.s`-Dateien betrachten, können Sie nicht feststellen, ob der Linker allenfalls auch noch optimieren kann. Relevant ist einzig, wie der Code im `*.exe`-File aussieht. Verwenden sie den Debugger ***gdb*** von der Kommandozeile, um dies festzustellen.
Starten Sie die Debugsession mit dem Befehl **`gdb main.exe.`** Anschliessend müssen Sie das Programm starten mit **`start`**. Den disassemblierten Code sehen Sie, wenn Sie **`disassem`** eintippen. Im folgenden sehen Sie den Ausschnitt einer Debugsession.

```
$ gdb main.exe
GNU gdb (GDB) 7.5.50.20120815-cvs (cygwin-special)
...
(gdb) start
Temporary breakpoint 1 at 0x401173
Starting program: /cygdrive/c/work/main.exe
[New Thread 7568.0x23a0]
[New Thread 7568.0x2654]

Temporary breakpoint 1, 0x00401173 in main ()
(gdb) disassem
Dump of assembler code for function main:
   0x00401170 <+0>:     push   %ebp
   0x00401171 <+1>:     mov    %esp,%ebp
=> 0x00401173 <+3>:     and    $0xfffffff0,%esp
   ...
```

d) Wahrscheinlich haben Sie festgestellt, dass alle Methoden mit einem Call aufgerufen werden. Ändern Sie die Optimierungsstufe bis alle Methoden ausser getArea() inline sind.

e) Sie möchten nun sowohl die impliziten Inlinefunktionen ins cpp-File zügeln, als auch von `getArea()` Inlinecode erhalten. Sie müssen die Funktionen mit inline kennzeichnen. Häufig werden die Implementationen der Inlinefunktionen auch in ein separates File, z.B. *.icc ausgelagert. Dieses wird am Ende des Headerfiles included. ("semi-ugly-include").

f) Testen Sie, ob alles richtig funktioniert, indem Sie ein Projekt mit mehreren cpp-Files erstellen, welche die Inlinefunktionen verwenden, d.h. `main.cpp` plus ein weiteres File. Betrachten Sie die Assemblerfiles, es sollten keine Calls auf Memberfunktionen mehr geben.