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

Aufgabe 1: Effizienzbetrachtungen bei Minimalprogrammen

a) siehe `./Loesung/A1-0`

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

In den folgenden Tabellen finden Sie die Codegrösse in Bytes.

Optimierung O…	gcc v3.4.4	gcc v4.5.3	g++ v3.4.4	g++ v4.5.3
0	44'004	49'528	44'004	49'528
1	44'004	49'528	44'004	49'528
2	44'004	49'528	44'004	49'528
3	44'004	49'528	44'004	49'528
s	43'814	49'528	43'814	49'528

Bei einem Minimalprogramm bestehen zwischen gcc und g++ keinerlei Unterschiede, auch der generierte Assemblercode ist identisch.

b) siehe `./Loesung/A1-1`

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

Optimierung O…	g++ v3.4.4	g++ v4.5.3
0	514'107	50'909
1	514'002	50'909
2	513'951	50'917
3	513'951	50'832
s	513'951	50'917

Die Optimierungsstufe hat keinen grossen Einfluss. Obwohl nur ein `#include <iostream>` gemacht wird, ohne dass auch eine Funktion wirklich genutzt wird, nimmt die Codegrösse beim g++ v3.4.4 enorm zu. Die Version g++ v4.5.3 hat sich diesbezüglich stark verbessert. Es ist klar ersichtlich, dass bei nicht gut optimierenden Compilern die Streamlibrary die Hauptverantwortliche für die Codegrösse ist.

c) siehe `./Loesung/A1-1`

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

Optimierung O…	g++ v3.4.4	g++ v4.5.3
0	514'143	51'997
1	514'038	52'041
2	513'987	52'049
3	513'987	50'832
s	513'987	50'917

Gegenüber Teilaufgabe b) nimmt der Code bei allen Optimierungsstufen leicht zu, beim g+ v3.4.4 konstant um je 36 Bytes zu.

d) siehe `./Loesung/A1-2`

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

Optimierung O…	g++ v3.4.4	g++ v4.5.3
0	44'252	49'812
1	44'252	49'812
2	44'252	49'812
3	44'252	49'812
s	44'062	49'812

Die Optimierungsstufe hat wiederum keinen grossen Einfluss. Interessant ist, dass hier der neue Compiler sogar etwas schlechter geworden ist.

e) Eine Hauptursache für die grösseren ausführbaren C++ - Programme bei Verwendung eines schlecht optimierenden Compilers ist die Streamlibrary. Wenn nur wenige Funktionen dieser Library benötigt werden, so empfiehlt sich unter Umständen der Einsatz der `printf()` - Funktion. Diese kostet nur 1'200 Bytes gegenüber ca. 470'000 Bytes bei der Streamlibrary. Als erste Benchmark wird ab und zu ein Hello World - Programm in C und eines in C++ genommen und daraus wird der falsche Schluss gezogen, dass C++ viel mehr Code ergebe als C. In Embedded Systems wird die Streamlibrary ohnehin häufig nicht benötigt. Der Grundbedarf für ein C - und ein C++ - Programm ist völlig identisch, C++ - Programme sind nicht generell grösser.

Aufgabe 2: inline-Methoden

a) just do it

b) Bei Optimierungsstufe `0` sind keine Funktionen inline (siehe `./Loesung/A2-1`). Man erkennt das daran, dass bei den einzelnen Funktionen Labels definiert sind und mit einem Return (`ret`) abgeschlossen werden. Die Funktionen werden mit einem call-Befehl aufgerufen (siehe folgenden Ausschnitt)

```
        .file                   "main.cpp"
* ...
_main:
* ...
        call                    __ZN9RectangleC1Edd
        leal                    -24(%ebp), %eax
        movl                    %eax, (%esp)
        call                    __ZN9Rectangle4getAEv
        fstpl                   -32(%ebp)
        leal                    -24(%ebp), %eax
        movl                    %eax, (%esp)
        call                    __ZN9Rectangle7getAreaEv
* ...
__ZN9Rectangle4getAEv:
        pushl                   %ebp
        movl                    %esp, %ebp
        movl                    8(%ebp), %eax
        fldl                    (%eax)
        popl                    %ebp
        ret
* ...
```

c) Der Debugger-Output zeigt, dass bei Optimierungsstufe `0` alle Funktionen mit `call` aufgerufen werden.

```
(gdb) disassem
Dump of assembler code for function main:
   0x00401170 <+0>:     push   %ebp
   0x00401171 <+1>:     mov    %esp,%ebp
=> 0x00401173 <+3>:     and    $0xfffffff0,%esp
   0x00401176 <+6>:     sub    $0x40,%esp
   0x00401179 <+9>:     call   0x401250 <__main>
   0x0040117e <+14>:    fldl   0x402080
   0x00401184 <+20>:    fstpl  0xc(%esp)
   0x00401188 <+24>:    fldl   0x402088
   0x0040118e <+30>:    fstpl  0x4(%esp)
   0x00401192 <+34>:    lea    0x20(%esp),%eax
   0x00401196 <+38>:    mov    %eax,(%esp)
   0x00401199 <+41>:    call   0x401770 <_ZN9RectangleC1Edd>
   0x0040119e <+46>:    lea    0x20(%esp),%eax
   0x004011a2 <+50>:    mov    %eax,(%esp)
   0x004011a5 <+53>:    call   0x4017a4 <_ZNK9Rectangle4getAEv>
   0x004011aa <+58>:    fstpl  0x38(%esp)
   0x004011ae <+62>:    lea    0x20(%esp),%eax
   0x004011b2 <+66>:    mov    %eax,(%esp)
   0x004011b5 <+69>:    call   0x4011c8 <_ZNK9Rectangle7getAreaEv>
   0x004011ba <+74>:    fstpl  0x30(%esp)
   0x004011be <+78>:    mov    $0x0,%eax
   0x004011c3 <+83>:    leave
   0x004011c4 <+84>:    ret
```

d) Ab Optimierungsstufe 1 sind die impliziten inline-Funktionen alle inline, auch der Konstruktor (siehe Zeile `0x00401199 <+41>:` in Aufgabe c). Einzig die Funktion `getArea()` wird immer noch aufgerufen, da diese Funktion in einer anderen Objectdatei liegt (siehe `./Loesung/A2-2/*.s`). Der Assemblerdump zeigt das:

```
Dump of assembler code for function main:
   0x00401170 <+0>:     push   %ebp
   0x00401171 <+1>:     mov    %esp,%ebp
=> 0x00401173 <+3>:     and    $0xfffffff0,%esp
   0x00401176 <+6>:     sub    $0x20,%esp
   0x00401179 <+9>:     call   0x401230 <__main>
   0x0040117e <+14>:    flds   0x402080
   0x00401184 <+20>:    fstpl  0x10(%esp)
   0x00401188 <+24>:    flds   0x402084
   0x0040118e <+30>:    fstpl  0x18(%esp)
   0x00401192 <+34>:    lea    0x10(%esp),%eax
   0x00401196 <+38>:    mov    %eax,(%esp)
   0x00401199 <+41>:    call   0x4011a8 <_ZNK9Rectangle7getAreaEv>
   0x0040119e <+46>:    fstp   %st(0)
   0x004011a0 <+48>:    mov    $0x0,%eax
   0x004011a5 <+53>:    leave
   0x004011a6 <+54>:    ret
```

e) Wenn das bestehende `main()` genommen wird, so optimiert der Compiler ab Optimierungsstufe 1 alle Variablen weg, da sie nicht verwendet werden (siehe `./Loesung/A2-3/main.s`).

```
        .file                   "main.cpp"
        .def                    ___main;    .scl    2;   .type  32;   .endef
        .text
        .align 2
.globl _main
        .def                    _main;      .scl    2;   .type  32;   .endef
_main:
        pushl                   %ebp
        movl                    %esp, %ebp
        subl                    $24, %esp
        andl                    $-16, %esp
        movl                    $16, %eax
        call                    __alloca
        call                    ___main
        movl                    $0, %eax
        leave
        ret
```

In der beiliegenden Lösung (siehe `./Loesung/A2-4`) werden die Daten deshalb auf `cout` geschrieben oder die Variablen als `volatile` deklariert, d.h. die Berechnungen können nicht wegoptimiert werden. Die Methoden können bei der Deklaration, bei der Definition oder bei beiden Stellen mit `inline` gekennzeichnet werden.

f) siehe `./Loesung/A2-5`

Eine neue Methode `getB()` wurde eingeführt, die nicht `inline` ist. Dies ist der einzige Call. Dies kann auch im gdb-Dump nachgeprüft werden (`getB()` gibt es, `getA()` nicht):

```
(gdb) disass _ZNK9Rectangle4getBEv
Dump of assembler code for function _ZNK9Rectangle4getBEv:
   0x004012a0 <+0>:     push   %ebp
   0x004012a1 <+1>:     mov    %esp,%ebp
   0x004012a3 <+3>:     mov    0x8(%ebp),%eax
   0x004012a6 <+6>:     fldl   0x8(%eax)
   0x004012a9 <+9>:     pop    %ebp
   0x004012aa <+10>:    ret
   0x004012ab <+11>:    nop
End of assembler dump.
(gdb) disass _ZNK9Rectangle4getAEv
No symbol "_ZNK9Rectangle4getAEv" in current context.
```