Thema, Ziele: C++ and ROMability, Placement `new`, Dynamic Memory Management

Alle Aufgaben wurden mit dem GNU-Compiler g++ v3.4.4 durchgeführt.

Aufgabe 1: C++ and ROMability

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

Solange die Strings völlig identisch sind (Formatstring), wird der String nur einmal im Speicher abgelegt. Bei Teilstrings werden bei jeder Optimierungsstufe beide Strings vollständig gespeichert (doppelt). Ab O1 wird das printf() durch puts() ersetzt. Bei dieser speziellen Anwendung spart man sich dadurch den Formatstring (hier ist g++ smart).

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

Solange von beiden Tabellen nur gelesen wird, könnten beide Tabellen geROMt werden, bzw. sogar nur einfach gespeichert werden, da der Inhalt identisch ist.

Diese kurzen Tabellen werden sogar auf dem Stack abgespeichert. Sobald sie grösser sind, werden sie ebenfalls im Datenbereich gespeichert.

Interessant sind vor allem noch die folgenden Deklarationen mit `static`. `static` bewirkt, dass die Tabellen auf jeden Fall im Datenbereich gespeichert werden, d.h. keinesfalls mehr auf dem Stack. `const` bewirkt, dass die Tabellen in den read-only Bereich gehen.

```
static const int table1[] = {1, 2, 3};
static int table2[] = {1, 2, 3};
```

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

Integerkonstanten, die mit `#define` oder `enum` definiert sind, jedoch nicht gebraucht werden, ergeben weder Code noch Daten. Integerkonstanten, die mit `const` definiert sind, werden auch dann gespeichert, wenn sie gar nicht verwendet werden, damit die Konstanten vorhanden sind, falls eine andere Compilationseinheit diese Konstanten verwenden würde. Innerhalb des .cpp-Files der Konstantendefinition werden diese Konstanten ebenfalls mittels Immediate-Adressierung verwendet.

Die beste Variante, um Integerkonstanten zu definieren, ist eindeutig mittels `enum`. `#define`s haben die bekannten Makroprobleme, `const`s brauchen unnötigerweise Speicher.

d) siehe `./Loesung/A1-3`

Gleitpunktkonstanten können nicht mittels Immediate-Adressierung verwendet werden, d.h. es ist immer eine Konstante im Datenbereich vorhanden. Konstanten, die mit `#define` definiert sind, jedoch nicht gebraucht werden, ergeben weder Code noch Daten. Gleitpunktkonstanten, die mit `const` definiert sind, werden auch dann gespeichert, wenn sie gar nicht verwendet werden, damit die Konstanten vorhanden sind, falls eine andere Compilationseinheit diese Konstanten verwenden würde. Allerdings sind diese Konstanten immer nur einfach vorhanden.

Konstanten, die mit `#define` definiert wurden, sind in jeder Compilationseinheit separat, d.h. mehrfach vorhanden.

Die beste Variante, um Gleitpunktkonstanten zu definieren, ist nicht so eindeutig festzulegen. `#define`s haben die bekannten Makroprobleme und benötigen allenfalls mehrfach Speicher für denselben Wert, jedoch nur, wenn sie wirklich verwendet werden. `const`s brauchen leider auch Speicher, wenn sie gar nicht verwendet werden, allerdings nur einmal. Solange nicht zu viele `const`s auf Vorrat definiert werden, schlage ich vor, auch hier auf `#define`s zu verzichten und `const`s zu verwenden.

e) siehe `./Loesung/A1-4`

Die vtbl's werden in den Abschnitt `rdata`, d.h. read-only data gelegt. Bei einem Embedded System mit ROM könnte dieser Bereich ins ROM gelegt werden.

Aufgabe 2: Placement `new`

a) siehe Eclipseprojekt in `./Loesung/Placement`

b)

c)

0	0x55	U	32	0	64	0x6e	n	96	0		
1	0		33	0	65	0x86		97	0		
2	0		34	0	66	0x1b		98	0		
3	0		35	0	67	0xf0	p	99	0		
4	0xb		36	0	68	0xf9	y	100	0x6c	l	
5	0xf6	v	37	0	69	0x21	!	101	0x45	E	
6	0x54	T	38	0	70	0x9		102	0		
7	0xa3	#	39	0	71	0x40	@	103	0		
8	0x12		40	0	72	0		104	0		
9	0xda	Z	41	0	73	0		105	0		
10	0x45	E	42	0	74	0		106	0		
11	0		43	0	75	0		107	0		
12	0		44	0	76	0		108	0		
13	0		45	0	77	0		109	0		
14	0		46	0	78	0		110	0		
15	0		47	0	79	0		111	0		
16	0		48	0	80	0x48	H	112	0		
17	0		49	0	81	0x53	S	113	0		
18	0		50	0	82	0x52	R	114	0		
19	0		51	0	83	0		115	0		
20	0		52	0	84	0		116	0		
21	0		53	0	85	0		117	0		
22	0		54	0	86	0		118	0		
23	0		55	0	87	0		119	0		
24	0		56	0	88	0xa5	%	120	0		
25	0		57	0	89	0xa5	%	121	0		
26	0		58	0	90	0xa5	%	122	0		
27	0		59	0	91	0		123	0		
28	0		60	0	92	0x49	I	124	0		
29	0		61	0	93	0x4d	M	125	0		
30	0		62	0	94	0x45	E	126	0		
31	0		63	0	95	0x53	S	127	0		

d) Untersuchung und Diskussion:
Die Memory Map weist keine unerwarteten Werte auf. Zu vermerken sind allenfalls die Bytes 85-87 und 97-99, welche aufgrund des Alignments nicht belegt werden.

Aufgabe 3: Fixed-size Pool

a) siehe `./Loesung/FixedPool`

Beachten Sie den Referenzparameter in

```
void deallocate(void*& ptr) throw()
```

Dadurch ist es ohne Doppelpointer möglich, `ptr` auf Null zu setzen.

b) siehe `./Loesung/FixedPoolCeil`

Die einzige Änderung muss in der Elementfunktion

```
void* allocate()
```

vorgenommen werden. Die Operator `!=` auf Zeile 39 muss durch den `>`-Operator ersetzt werden:

```
if (bytes > elemSize)  // objects smaller than elemSize are allowed
  throw HeapSizeMismatch();
```

Aufgabe 4: Block Allocator

siehe `./Loesung/BlockAllocation`