C/C++ Programmierung
Übungsblatt 12

Florian Oetke

1 Fragen zu Typ-Konvertierungen

Erklären Sie kurz in eigenen Worten die folgenden unterschiedlichen Arten von Typkonvertierungen, die wir in der Vorlesung kennengelernt haben, in welchen Fällen sie z.B. angewendet werden können und was es ggf. zu beachten gibt:

1. Implizite Typenkonvertierung
2. `static_cast`
3. `dynamic_cast`
4. `const_cast`
5. `reinterpret_cast`
6. `(T) var`
7. `T(var)`

Beschreiben Sie kurz in eigenen Worten, was es mit der Strict-Aliasing-Regel auf sich hat und was sie uns verbietet bzw. erlaubt.

2 Transform-Algorithmus

Implementieren Sie ein Funktions-Template `my_transform`, das als Funktions-Parameter eine Iterator-Range (begin und end Iterator) sowie eine Funktion erhält, die ein Element als Parameter bekommt und ein neues Element vom selben Typ zurückgibt. Ihr `my_transform` soll über alle Elemente in der Range iterieren und die übergebene Funktion auf jedes der Elemente anwenden und den bisherigen Wert des Elements mit dem Rückgabewert der Funktion überschreiben.

Ihr Funktions-Template sollte mit beliebigen Iteratoren und aufrufbaren Objekten als Parameter funktionieren. Das heißt sowohl mit Iteratoren entsprechender Container als auch mit Pointern und sowohl mit Pointern/Referenzen auf Funktionen als auch mit Lambda-Ausdrücken.

Als Basis für die Implementierung können Sie den im Handout bereitgestellten Programmcode verwenden, der bei einer korrekten Lösung die folgende Ausgabe erzeugen sollte:

```
Aa Ab Ac
3 4 5
```

3 Eigene C-String Hilfsfunktionen

In dieser Aufgabe sollten Sie den Umgang mit C-Strings und Arrays üben, indem Sie einige der Hilfsfunktionen, die wir in der Vorlesung kennengelernt haben, selbst implementieren.

Implementieren Sie die folgenden Funktionen in dem vorgegebenen `c_str` Namespace, ohne Funktionen der Standardbibliothek aufzurufen:

- `strlen`: Bekommt einen C-String als Parameter und gibt dessen Länge als `std::size_t` zurück.
- `strcmp`: Bekommt zwei C-Strings als Parameter und gibt entweder 0 (wenn sie identisch sind), eine negative Zahl (wenn der erste String kleiner ist) oder eine positive Zahl zurück.
- `substr`: Bekommt als Parameter einen Pointer auf ein ausreichend großes C-Array vom Typ `char`, einen C-String und zwei Indices in diesen C-String (vom Typ `std::size_t`). Die Funktion schreibt anschließend in das Array aus dem ersten Parameter einen C-String der aus den Zeichen zwischen den beiden Indices besteht. Denken Sie hierbei daran, dass ein C-String immer mit einem Null-Byte abgeschlossen wird.
- `memcpy`: Bekommt zwei `void*` auf beliebige Speicherbereiche sowie eine Anzahl (`std::size_t`) als Parameter und kopiert entsprechend viele Bytes aus dem Speicherbereich, auf den der zweite Parameter verweist, in den des ersten Parameters. Sie können hierbei davon ausgehen, dass sich die beiden Speicherbereiche nicht überschneiden.

Als Basis für die Implementierung sollten Sie den im Handout bereitgestellten Programmcode verwenden. Bei einer korrekten Lösung sollte Ihr Programm die folgende Ausgabe erzeugen (genaue Ergebnisse von `strcmp` können natürlich abweichen):

```
strlen() : 18
substr() : Beispiel

strcmp("xxx", "xxx") : 0
strcmp("xxx", "xxa") : 23
strcmp("xxx", "xxz") : -2
strcmp("xxx", "xx")  : 120
strcmp("xx", "xxx")  : -120

After memcpy() : ok!
```

4 Dynamisches Array

In dieser Aufgabe sollen Sie die manuelle Speicherverwaltung verwenden um selbst einen einfachen Container zu implementieren.

Implementieren Sie ein Klassen-Template `Dynamic_array<T>`, das ein Array von T-Objekten auf dem Heap verwaltet und eine `size()` Methode sowie einen `operator[](int index)` bereitstellt, um die Größe des Arrays zu ermitteln und Elemente auszulesen. Der `operator[]` soll dabei keine Kopie, sondern eine Referenz zurückgeben und unabhängig davon anwendbar sein, ob der `this`-Pointer `const` ist oder nicht. Das heißt, Sie müssen zwei Varianten des Operators implementieren. Eine, die als `const` markiert ist und entsprechend eine `const`-Referenz zurückgibt sowie eine Variante, bei der dies nicht der Fall ist. Bei einem Zugriff auf einen ungültigen Index sollte eine `std::out_of_range` geworfen werden.

Die Größe des Arrays soll zur Laufzeit beim Erzeugen des `Dynamic_array<T>`-Objekts festgelegt werden können und sich danach nicht mehr ändern. Hierzu benötigen Sie also einen entsprechenden Konstruktor, der die gewünschte Größe als Parameter bekommt und ein entsprechend großes Array auf dem Heap anlegt. Sie können davon ausgehen, dass die Elemente einen Default-Konstruktor haben und direkt initialisiert werden dürfen[1]. Da Sie das Heap-Objekt auch wieder freigeben müssen, benötigen Sie außerdem einen entsprechenden Destruktor.

[1] Das heißt, Sie müssen für diese Aufgabe nicht Placement-New verwenden, sondern können direkt ein Array des entsprechenden Typs erzeugen.

Implementieren Sie außerdem einen Move-Konstruktor und -Zuweisungsoperator, die eine rvalue-Referenz auf ein anderes `Dynamic_array<T>`-Objekt als Parameter bekommen und das Heap-Array dieses anderen Objekts übernehmen, wobei natürlich die Member-Variablen des alten Objekts entsprechend verändert werden müssen, damit das Heap-Array nicht zweimal freigegeben wird. Das Kopieren eines `Dynamic_array<T>`-Objekts wäre etwas komplexer und muss nicht von Ihnen implementiert werden. Sie sollten den entsprechenden Konstruktor und Zuweisungsoperator allerdings explizit mit `= delete` unterdrücken.

Als Basis für die Implementierung sollten Sie den im Handout bereitgestellten Programmcode verwenden. Im Test-Code wird für die Elemente (`T`) der vordefinierte Typ `Log_lifetime<int>` verwendet, der wie ein `int` zugewiesen und ausgegeben werden kann, aber zusätzlich auch alle Aufrufe der Special-Member-Functions ausgibt. Die Ausgabe für eine korrekte Lösung sollte damit so aussehen:

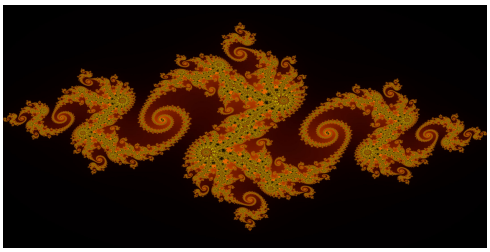
```
array Objekt erzeugen:
  > Log_lifetime()
  > Log_lifetime()
  print : 2
    array[0] = 0
    array[1] = 1

array_2 Objekt erzeugen:
  > Log_lifetime()

Move-Zuweisung:
  > ~Log_lifetime()
  print : 0
  print : 2
    array[0] = 0
    array[1] = 1

process()-Aufruf mit Move-Konstruktion:
  print : 2
    array[0] = 0
    array[1] = -1
  > ~Log_lifetime()
  > ~Log_lifetime()
  print : 0
```

Anhand der Ausgaben der `Log_lifetime<int>`-Klasse können Sie einige häufige Fehler nachvollziehen:

- Wenn Sie mehr Ausgaben haben, kopieren Sie u.U. unnötig Objekte.
- Wenn Sie mehr Konstruktor- als Destruktor-Ausgaben haben, liegt bei Ihnen ein Speicherleck vor. Das heißt, Sie erzeugen Objekte mit `new`, die Sie nicht wieder freigeben.
- Wenn Sie mehr Destruktor- als Konstruktor-Ausgaben haben, geben Sie dasselbe Objekt mehrfach frei. In der Regel, weil Sie `delete` bzw. `delete[]` mehrmals für denselben Pointer aufrufen.

5 Julia Fraktal

In dieser Aufgabe sollen Sie, wie in der Vorlesung gezeigt, eine Binärdatei mithilfe eines `struct` schreiben. Ihr fertiges Programm sollte eine BMP-Datei erzeugen, die Sie mit einem beliebigen Bildbearbeitungsprogramm öffnen können und die ein Julia-Fraktal enthalten sollte, welches (abhängig von ihren Parametern) z.B. wie folgt aussehen könnte.



Als Basis für die Implementierung sollten Sie den im Handout bereitgestellten Programmcode verwenden, der bereits die Binärdatei öffnet und die Farbwerte der einzelnen Pixel berechnet. Definieren Sie ein `struct` welches dem unten beschriebene Binärformat entspricht und ergänzen Sie die beiden Funktionen um den entsprechenden Code, der den Header und die einzelnen Pixelwerte in die Datei schreibt.

Die folgende Tabelle beschreibt die für uns relevanten Informationen, die in einer BMP-Datei gespeichert werden. Wir treffen hierbei einige vereinfachende Annahmen[2] und verwenden nur wenige Features des Formats, weswegen die meisten Werte konstant sind und der Header ebenfalls immer eine konstante Größe von 54 Byte hat.

Eine BMP-Datei beginnt immer mit zwei Bytes, die den ASCII-Werten für `` `B` `` und `` `M` `` entsprechen, gefolgt von einige 32 und 16 Bit Zahlenwerten, die Positionen/Größen angeben, sowie weiteren Optionen, die wir hier nicht verwenden und deren Bits entsprechend alle auf 0 gesetzt werden können. Der letzte Teil der BMP-Datei sind dann schließlich die Bilddaten selbst, die aus je einem Byte für den blauen, grünen und roten Farbwert des Pixels bestehen, also Werte zwischen 0 und 255 erlauben.

Beachten Sie, dass wir die Endianness hier ignorieren können, da das Format little-endian vorschreibt, was auch der Byte-Reihenfolge unseres Systems entspricht.

Im vorgegebenen Handout müssen die Pixelwerte in der Schleife in `write_data` geschrieben werden, während die übrigen Bytes von der Funktion `write_header` geschrieben werden sollen. Beachten Sie, dass der Header am Ende einige Null-Bytes enthält, welche Sie entweder (z.B. als Array) mit in Ihr `struct` aufnehmen oder manuell hinter das `struct` in die Datei schreiben müssen. Achten Sie bei der Lösung der Aufgabe außerdem darauf, dass Ihr `struct` kein unbeabsichtigtes Padding enthält.

Anzahl Bytes	**Beschreibung**	**Wert**
1	Signatur, um zu signalisieren, dass es sich um eine	B
1	BMP-Datei handelt	M
4	Größe der gesamten Datei in Byte	Breite * Höhe * 3 + 54
4	Reserviert (für spätere Erweiterungen)	0
4	Position (in Byte vom Anfang der Datei) an der die eigentlichen Farbwerte der Pixel anfangen	54
4	Größe des nun folgenden Device-Independent Bitmap (DIB) Headers	40
4	Breite des Bildes in Pixeln	...
4	Höhe des Bildes in Pixeln	...
2	Anzahl der Farbebenen	1
2	Anzahl der Bits pro Pixel	8*3
24	Weitere Optionen (Kompression, Farbpaletten, ...), die wir nicht verwenden	0
Breit*Höhe*3	Farbwerte der einzelnen Pixel, mit je einem Byte pro Farbe (0-255). In der Reihenfolge Blau, Grün, Rot	...

[2] Unter anderem, dass die Breite*3 ein Vielfaches von 4 ist, wodurch kein zusätzliches Padding zwischen den Farbwerten der Pixel notwendig ist.