Praktikum 1: Klassen, `const` & `static`-Attribute

Lernziele

- Entwickeln von komplexeren Datenstrukturen
- Anwenden von Klassentemplates
- Verwenden einer `static`-Variablen zum Erzeugen einer eindeutigen ID
- Einlesen von längeren Texten auf der Konsole

1 Klasse `Ticket`

Ein **Ticketsystem**[1] ist eine Software, um Anfragen oder Fälle zu sammeln, klassifizieren und abzuarbeiten. Typischerweise enthält eine Ticket Informationen über die eigentliche Anfrage, eine eindeutige Ticketnummer und den (Benutzer-)Namen des Mitarbeiters, der dieses Ticket bearbeitet. Darüber hinaus hat ein Ticket einen bestimmte Status, beispielsweise „offen", falls es in Bearbeitung ist, oder „geschlossen", falls die Bearbeitung abgeschlossen ist.

Die zu implementierende Klasse `Ticket` soll ein solches Ticket darstellen, die in Aufgabe 2 zu implementierende **Priority Queue**[2] dient zur Speicherung der Tickets, die ein bestimmter Mitarbeiter zu bearbeiten hat.

Die Klasse `Ticket` soll über folgende Attribute und Methoden verfügen:

- **`string text`** enthält den Beschreibungstext des Tickets
- **`string owner`** speichert den Besitzer des Tickets, also den Benutzer der dieses Ticket erzeugt hat
- **`Status status`** bezeichnet den aktuellen Status des Tickets. Der Datentyp `Status` soll ein Aufzählungstyp (`enum`) mit folgenden Werten sein:
  - open
  - closed
  - duplicate
- **`int id`** ist die fortlaufende Ticketnummer. Verwenden Sie ein **zusätzliches statisches** (`static`) privates Attribut um in Konstruktor eine eindeutige ID zu erzeugen. Dazu initialisieren Sie das statische Attribut mit dem Wert 1. Im `Ticket`-Konstruktor kopieren Sie den Wert des statischen Attributes in das Attribut `id` und erhöhen den Wert des statischen Attributes um 1.

- **`Ticket()`** kann einfach leer gelassen werden (Je nachdem, wie Sie Aufgabe 2 lösen, kann es sein dass Sie für `Ticket` einen Standardkonstruktor benötigen)
- **`Ticket(string, string)`** besitzt zwei Parameter für `owner` und `description`, setzt die `id` entsprechend und setzt den Status des Tickets auf open
- **`string getText()`, `string getOwner()`, `int getId()`, `Status getStatus()`** geben die entsprechenden Attributwerte zurück.
- **`string getShort()`** soll nur den **ersten Satz** (bis zum ersten „`.`") des Beschreibungstextes zurück geben. Verwenden Sie hierzu (beispielsweise) die beiden `string`-Funktionen `find` und `substr`[3]
- **`string getStatusAsString()`** konvertiert den Status des Tickets in einen String und gibt diesen zurück
- **`void print()`** Gibt alle Informationen über das Ticket formatiert auf dem Bildschirm aus

1.1 `const`-correctness

Methoden von Klassen können das „Versprechen" abgeben, **keine** Attribute der Klasse zu verändern.
Identifizieren Sie alle Methoden, auf die dies zutrifft! Markieren Sie die betreffenden Methoden anschließend als `const`, sowohl in der `.h` als auch in der `.cpp` Datei wie folgt:

```
//Beispiel.h
class Beispiel
{
  public:
    void print() const;
}
```

```
//Beispiel.cpp
#include "Beispiel.h"
void Beispiel::print() const
{
  //...
}
```

[1] `https://de.wikipedia.org/wiki/Issue-Tracking-System`
[2] dt. etwa „priorisierte Warteschlange"
[3] Recherchieren Sie gegebenenfalls die Funktionsweise auf `http://cplusplus.com`

2 Priority Queue

Eine **Priority Queue** ist eine spezielle Datenstruktur, bei der für den Zugriff nur zwei Funktionen existieren, ähnlich wie bei den Datenstrukturen `Queue` oder `Stack`.
Eine „normale" Queues (dt. „Warteschlange") funktioniert in etwa so:

- Eine Funktion `queue` oder `push` fügt ein neues Element **hinten** in der Warteschlange ein
- Eine weitere Funktion `dequeue` oder `pop` entnimmt der Warteschlange ein Element. Dies wird dabei von **vorne** entnommen

Das Prinzip ist das gleiche wie bei einer „wirklichen" Warteschlange: Hinten wird angestellt, vorne wird bedient.

Die Besonderheit der **Priority Queue** ist, dass jedem Element eine **Priorität** zugeordnet ist. Jedes neue Element wird beim Einfügen an die „richtige" Stelle in der Queue eingefügt, so dass sich **vor** dem neuen Element kein weiteres mit höherer Priorität befindet. Üblicherweise ist 0 die **höchste** Priorität.

Folgendes Beispiel zeigt eine Priority Queue für `char`-Daten:

```
Queue q;
q.push('a', 0); //a mit Prioritaet 0 einfuegen
q.push('b', 8); //Inhalt: a b
q.push('c', 7); //Inhalt: a c b
q.push('e', 6); //Inhalt: a e c b;
q.push('g', 100); //Inhalt: a e c b g;
q.push('f', 1); //Inhalt: a f e c b g;
q.push('f', 4); //Inhalt: a f f e c b g;
cout << q.pop() << endl; //Ausgabe: a, Inhalt: f f e c b g
```

Implementieren Sie eine Klasse `PrioQueue` für Objekte des Typs `Ticket`!
Als Grundlage zur Speicherung der Elemente können Sie wählen zwischen:

- `vector`
- Array (dynamisch oder mit fester Größe)
- Eine Weiterentwicklung des angehängten Quellcodes „verkettete Liste"
- Eine Weiterentwicklung des angehängten Quellcodes „Queue"

Zusatzaufgabe: Verfassen Sie die Klasse als Template!

3 Anwendung

Schreiben Sie eine kleine Anwendung mit einem Menü, welches es einem Benutzer gestattet, eine (leere) Priority Queue für Tickets anzulegen. Darüber hinaus sollte das Menü (in etwa) über folgende Möglichkeiten verfügen:

- Neues Ticket erfassen und in die Queue speichern. Lesen Sie hierzu alle benötigten Werte von der Tastatur ein
- Alle Tickets löschen
- Nächstes Ticket aus der Queue anzeigen
- Programm beenden

3.1 Einlesen von langen Zeichenketten von der Tastatur

Wie Ihnen bekannt sein sollte, liest der Operator $>>$, mit dem Sie normalerweise `string`-Daten von der Tastatur einlesen nur bis zum nächsten **Zwischenraumzeichen**. Das bedeutet, dass keine ganzen Sätze eingelesen werden können, da diese Leerzeichen enthalten. Hierzu muss eine andere Funktion verwendet werden: **`getline`**

Diese ist im Header `string` enthalten, und nimmt zwei Parameter:

- Einen **Eingabestrom** wie beispielsweise `cin`, von dem gelesen wird
- Eine Variable vom Typ `string`, in die geschrieben wird

Die Funktion liest so lange Eingaben von der Tastatur, bis das Zeilenendezeichen `'\n'` gefunden wird. Dieses wird aus dem Eingabestrom extrahiert und verworfen.

Wichtig: Der $>>$-Operator extrahiert das Zwischenraumzeichen **nicht**. Wenn Sie also an irgendeiner Stelle im Programm

ein `getline` nach einem >> verwenden, findet dieses `getline` noch das Zeilenendezeichen im Tastaturpuffer vor. Daher sollten Sie **immer** ein `cin.ignore(255, '\n')` **vor** dem Aufruf von `getline` tätigen!

Laden Sie sich am besten die Datei `getline.cpp` herunter, schauen Sie sich das Beispiel an, und probieren Sie was passiert wenn Sie die die `cin.ignore`-Zeilen auskommentieren.