11_blackbox: Entwicklung einer Blackbox

Dipl.-Ing. Dr. Günter Kolousek

1 Allgemeines

- **Es gelten die gleichen Richtlinien wie beim ersten Beispiel!!!**

2 Aufgabenstellung

In diesem Beispiel geht es darum, eine Blackbox zu entwickeln, die immer die letzten n Messwerte eines Sensors speichern soll.

3 Anleitung

1. Für diese Anwendung eignet sich die Datenstruktur eines Ringpuffers. Allerdings soll dieser Ringpuffer den letzten Werte überschreiben, wenn der Ringpuffer voll ist und ein weiterer Wert hinzugefügt werden soll.

   Das API dieses Ringpuffers soll vorerst so aussehen:

   ```
   class Ringbuffer {
     public:
       std::optional<double> get();
       void put(double val);
       size_t size();
     private:
       // eigentlich kein Teil des APIs (zumindest bis jetzt)
       static constexpr size_t CAPACITY{5};
   };
   ```

   Implementiere diesen Ringpuffer in einem eigenem header-only Modul in einem eigenem Unterverzeichnis `ringbuffer` und entwickle auch gleich die entsprechenden Unit-Tests.

   Für die eigentliche Implementierung hast du `std::array` zu verwenden und die Größe kannst du vorerst fix annehmen (siehe API). Weiters soll die Implementierung mit zwei Indizes, je einen zum Lesen und je einen zum Schreiben auskommen. Damit bleibt zwangsläufig immer ein freier Platz übrig. Damit muss die eigentliche Größe des verwendeten `array` um eins größer sein als die angegebene Kapazität, damit der Benutzer unsers APIs wirklich die angegebene Anzahl an Gleitkommazahlen in den Ringpuffer schreiben kann!

   Du weißt, dass im TDD (Test Driven Development) immer **zuerst** Unit-Tests geschrieben werden, die natürlich zuerst fehlschlagen und danach erst die Implementierung vorgenommen wird, sodass die Unit-Tests danach erfolgreich durchlaufen.

2. Entwickle jetzt ein Programm `blackbox`, das von `stdin` zeilenweise jeweils Gleitkommazahlen (im gängigen Format) parst und diese Zahlen jeweils in den Ringpuffer einspeist.

   Wird das Programm beendet indem der Eingabestrom geschlossen wird, dann wird der Inhalt des Ringpuffers zeilenweise auf `stdout` ausgegeben. Geht der Eingabestrom in einen Fehlerzustand (nicht EOF), dann ist das Programm mit einer entsprechenden Fehlermeldung ebenfalls zu beenden.

   Vergiss bei Konvertieren in eine Zahl auf die folgenden Fälle zu achten:

   - Es kann sich um eine ungültige Zahl wie z.B. "a" oder "1a" handeln.
   - Die Zahl kann sich nicht im gültigen Bereich der darstellbaren Gleitkommazahlen befinden.

   Fehlermeldungen sollen nach `stderr` geschrieben werden (u.a. damit diese von den normalen Ausgaben unterschieden werden können).

3. Damit unsere Simulation etwas realitätsnäher wird, entwickle jetzt ein zweites Programm mit dem Namen `inputsim` (oder wie auch immer), das 10 zufällige Werte im Intervall $[0.0, 10)$ erzeugt und danach zeilenweise auf `stdout` ausgibt.

4. Starte jetzt beide Programme und verknüpfe den `stdout` von `inputsim` mit dem `stdin` von `blackbox`.

5. Ok, gehen wir zurück zu unserem Ringpuffer, der ja ganz gut funktioniert, aber halt nur mit `double`-Werten umgehen kann. Weiters haben wir die Kapazität auch von Haus aus begrenzt und müssten jedes Mal den Ringpuffer umschreiben, wenn wir eine andere Größe benötigten. Beachte, dass die Größe nicht zur Laufzeit geändert werden kann, da wir ein `array` verwenden!

   Wir werden diesen jetzt erweitern, dass dieser mit weitgehend beliebigen Datentypen und einer beliebigen Größe verwendet werden kann!

   Dazu greifen wir auf Templates zurück! Stelle die Klasse `Ringbuffer` auf ein Template um, das mit einem formalen Typparameter und einem formalen Parameter vom Typ `size_t` parametrisiert wird.

   Eine Instanz von Ringpuffer sollte dann folgendermaßen angelegt werden können:

   ```
   Ringbuffer<double, 5> buffer;
   ```

6. Wir können jetzt einen Ringpuffer mit beliebigen Typen anlegen. Was ist wenn wir in unseren Ringpuffer nicht nur Werte eines bestimmten Typs schreiben wollen, sondern für mehrere verschiedene Typen gleichzeitig? Welche Möglichkeiten bieten sich uns?

   - `union` wäre prinzipiell möglich, aber mit diesem Thema sind wir schon fertig…
   - `std::variant` kennen wir schon.
   - `std::any` kennen wir nicht, aber das Prinzip ist so ähnlich wie `std::variant`, wenn auch die Syntax eine andere ist.
   - Will man mehrere Werte zusammenfassen wäre ein `struct` die natürliche Möglichkeit und kann mit den anderen angeführten Möglichkeiten kombiniert werden.
   - Eine Basisklasse definieren und von dieser beliebige Typen ableiten
   - JSON wäre auch noch eine Möglichkeit, wenn wir mit den Basistypen von JSON auskommen bzw. wir eigene Typen nach JSON serialisieren und von JSON deserialisieren können.

   Mit JSON treffen wir sicherlich keine schlechte Wahl für die gegebene Aufgabenstellung: einfach und interoperabel. Bzgl. der Performanz sicher kein Rennpferd und bzgl. der Datenübertragung über ein Netzwerk nicht die optimale Lösung, aber das sind für diese Anwendung keine relevanten Kriterien.

Baue daher die beiden Hauptprogramme dementsprechend um, dass das Programm `inputsim` zufälligerweise jeweils eine ganze Zahl, eine Gleitkommazahl oder einen boolschen Wert in der Schleife ausgibt. Die Blackbox soll diese entsprechend parsen und wieder ganz normal ausgeben.

Zufälligerweise? Konsultiere die cppreference unter `Numerics library->Pseudo-random number generation->uniform_real_distribution`!

7. Soweit sogut, jetzt wollen wir nicht nur ganze Zahlen, Gleitkommazahlen oder einen boolschen Wert zwischen den Programmen austauschen, sondern direkt Messwerte kommunizieren.

   Dazu definieren wir uns ein `struct` mit dem Namen `Measurement` im Namensraum `measurement` in einem eigenem Header `measurement.h`. Solch ein Meßwert soll eine eigene `id` (`string`) ein Datum `date` (vorerst `string`) und einen Wert `value` (`double`) haben.

   `inputsim` soll jetzt so angepasst werden, dass zufälligerweise auch eine Instanz von `Measurement` generiert wird und dann an `stdout` als JSON ausgegeben wird. Dazu muss im Namensraum `measurement` noch eine Funktion `void to_json(json& j, const Measurement& m)` wobei im Rumpf `j` auf `json{{"id", m.id},...};` gesetzt wird.

   Testen!

8. Ok, in der Blackbox wird jetzt ein entsprechender JSON Wert gespeichert. Das ist auch gut so, nur bei der Entnahme aus der `blackbox` sollte auch wieder eine Instanz von `Measurement` generiert werden. Kommt auf dieser Seite eine Zahl an, dann sollte auch eine Zahl aus dem Ringpuffer genommen werden, kommt eine boolscher Wert an, dann sollte dieser auch in ein `bool` konvertiert werden.

   Die Konvertierung eines JSON Wertes in einen Wert eines benutzerdefinierten Wertes geschieht so, dass eine Funktion `void to_json(json& j, const Measurement& m)` (wieder im entsprechenden Namensraum) definiert wird, die für jedes Feld einen Aufruf folgender Gestalt aufweist:

   ```
   j.at("id").get_to(m.id);
   ```

   Jeder empfangene/eingelesene Wert soll danach auch wieder auf `stdout` ausgegeben werden! Natürlich macht es keinen Sinn, zuerst in ein `double`, einen `bool` oder eine Instanz von `Measurement` zu konvertieren und diese danach *nur* auszugeben, aber wir wollen ja lernen wie dies funktioniert, nicht wahr?

   Zur Erkennung, um welchen Typ es sich handelt, bietet das `json`-Objekt die Methoden `is_object()`, `is_number()` und `is_boolean()` an. Mittels `j.get<measurement::Measurement>()...` kannst du auf den entsprechenden Wert zugreifen.

   Zur Ausgabe eine Instanz von `Measurement` benötigst du wieder einen überladenen Operator `operator»`, eh klar.

9. Das ist ja sowei fein, aber das aktuelle Datum und die aktuelle Zeit sind ja derzeit gar nicht aktuell sondern fix als Testwert kodiert. Auch die `id` als auch der Wert ist als Testwert kodiert.

   a) Die aktuelle Zeit bekommst du folgendermaßen:

      ```
      time_t datetime_now = chrono::system_clock::to_time_t(chrono::system_clock::now());
      ```

      Um diesen `time_t` Wert noch in einen String zu wandeln muss man (derzeit) noch auf einen Stream und einen entsprechenden IO Manipulator zurückgreifen:

      ```
      std::stringstream buf;
      buf << std::put_time(std::localtime(&datetime_now), "%F %T");
      date = buf.str();
      ```

      Schaue dir in der Referenz die entsprechenden Funktionen an.

   b) Die Id sollte möglichst eindeutig sein. Die Frage ist in welchem Kontext diese eindeutig sein soll:

- Nur über alle `Measurement`-Instanzen hinweg?
- Innerhalb eines Programmaufrufes?
- Über alle Programmaufrufe (auf einem Node) hinweg?
- Weltweit über alle Programmaufrufe?
- Weltweit? Erfüllt eine Id diese Anforderung, dann spricht man in der Regel von einer GUID (globally unique id) oder einer UUID (universally unique id).

Der Einfachheit halber reicht uns, dass die Id eindeutig sein soll über allle `Measurment`-Instanzen innerhalb eines Programmaufrufes. Damit wird die Angelegenheit wiederum sehr einfach, da wir lediglich einen Zähler benötigen, den wir z.B. bei 0 oder 1 zu zählen beginnen lassen.

c) Der Wert sollte einfach ein zufälliger Wert im Intervall $[0, 10]$ sein.

Schreibe einen Konstruktor für die Klasse `Measurement`, die diese beiden Felder entsprechend den hier angeführten Anweisungen initialisiert. Damit können wir in `inputsim` leicht diese Instanzen anlegen und

10.

11.

4 Übungszweck dieses Beispiels

- Ringpuffer implementieren
- `std::array` verwenden
- Wiederholung der Konvertierung in Zahlen
- Wiederholung der Behandlung von `cin`
- Ausgeben von Fehlermeldungen auf `stderr`
- Zufallszahlen erzeugen
- Mehrere Prozesse von der Konsole starten und mittels Pipe verknüpfen
- Templates entwickeln
- JSON vertiefen und benutzerdefinierte Typen in und von JSON konvertieren
- Auf die aktuelle Zeit zugreifen und diese in einen String konvertieren
- eindeutige Ids (für einen Programmaufruf) generieren