Draft der C++-Vorlesung vom 3. Mai 2024

Klassen §9

Klasse Komplex cnt.

Rekapitulation zu Beispiel *v_ 6a*:

- Member: Real-, Imaginäranteil
- Konstruktoren (mehrere!):
- Destruktor (genau einer)
- Zuweisungsoperator
- Methoden (Funktionen der Klasse):
  - Setter
  - Getter (`const` Method)
  - `operator+` für `a+b` oder `a+2` (`const` Method)
  - `operator+=`
  - abs (`const` Method)
- Member (Daten der Klasse)
  - **private** $\longrightarrow$ Datenkapselung (encapsulation)
- Funktionen:
  - Ausgabe: `operator<<`
  - Eingabe: `operator>>`
  - `operator+` für `2+a`
  - allgemein: binärer Operator (wie `operator+`) als Funktion, aber `operator+=` als Methode implementieren.
    - Funktion nutzt Getter/Setter für Zugriff auf Member, oder als
    - `friend`-Funktion in Klasse deklarieren.

Demo `vector<Komplex>` mit parameterlosem Konstruktor und zu Vektorausgabe am Beispiel *v_ 6a_ vektor*:

- `vector<Komplex> vv;` $\longleftarrow$ `Komplex::Komplex()` wird benötigt.
- Demo der (Nicht-)Initialierung im parameterlosem Konstruktor.
- Intialisierung des Vektors mit initializer list mit Elementen `Komplex(1.1,2.3)` oder `{1.1,2.3}` .
- Ausgabeoperator (function) für `vector` (allg. mit Templates).

Aktuelles C++17 in *v_6a_cpp17/*

- Rule of Five via `= default`
- Constructor forwarding
- Header Guarding: `#pragma once`
- `vector<Komplex>`; Vektorausgabe; Name der Variablen (Macro)
- sort (`#include <algorithm>`) benötigt `bool operator<(Komplex a, Komplex b)` .
- sort über λ-Funktion als Sortierkriterium: `[ ]( ){ }`
  - [ ] Caption, d.h. Transfer von gültigen Variablen/Instanzen in die Funktion.
  - ( ): übliche Parameterliste
  - { }: Funktionskörper
  - `[](Komplex lhs, Komplex rhs) -> bool { return lhs.abs()<rhs.abs(); }`

zu Aufg. 19: λ-Function *Bisect/Bisect_3_lambda.cpp*

- `double f(const double x) { return sin(x) - 0.5*x; }` wird in
  `double x0 = Bisect3(f, a, b, EPS);` als auszuwertende Funktion übergeben.
- dasselbe als λ-Function:
  `double x0 = Bisect3([](const double x) { return sin(x) - 0.5*x; }, a, b, EPS);`

Jetzt wollen wir ein Polynom $p(x) = \sum_{k=0}^{n} a_k x^k$ an Bisect übergeben:

- Gegeben: Auswertungsfunktion für $p(x)$
  `double eval(vector<double> const &a, double x)`
- Wie übergeben wir den zusätzlichen Koeffizientenvektor a an Bisect?
  ==> via Caption in der λ-Function.
- `double x0 = Bisect3([a](const double x) { return eval(a,x); }, a, b, EPS);`

Klasse Student *v_ 7a/* und *v_ 7a_ set/*

Wir beginnen mitd em Entwurf:

- Member: Familienname, Matrikelnr. (const?), Studienkennzahl (eine; mehrere?)
- Methoden:
  - Konstruktor(en); Nutzung default/delete
    - Konstruktor mit einer SKZ (1. Interface)
    - Konstruktor mit SKZ-Vektor/initializer list (2. Interface, später).
  - Ausgabeoperator
  - Getter/Setter (Namensänderung?)
  - Hinzufügen/Entfernen von Studien
- `const` Member $\Longrightarrow$ `operator=` nicht mehr möglich aber Copy-Constructor erlaubt.
- Getter sind `const` Methoden.
- Ausgabeoperator `operator<<` ist `friend`
  (erlaubt Zugriff auf `private` member in Funktion).
- Änderung: `int _skz` $\longrightarrow$ `vector<int> _skz`
  $\Longrightarrow$ `const vector<int>& Get_SKZ() const {return _skz;}`
  $\Longrightarrow$ zusätzlicher Konstruktor (2. Interface)
- Add_SKZ:
  - Prüfung auf bereits eingeschriebenes Studium mit Algorithmus `find` .
  - Falls neues Studium (nichts gefunden), dann Methode `push_back` .
- Del_SKZ:Prüfung:
  - Überhaupt für das Studium eingeschrieben? mit Algorithmus `find` .
  - Falls dafür eingeschrieben (gefunden), dann Methode `erase` .
- Nutzung von `set` statt `vector` erspart das Suchen in `_skz` im Bsp. *v_ 7a_ set/*:
  - Vorteile von `set`:
    - Keine mehrfach vorkommenden Elemente möglich.
      (*Sets are containers that store unique elements following a specific order.*)
    - $\mathcal{O}(\log \mathcal{N})$-Komplexität bei Suche da als geordeter Baum gespeichert.
      (ist auch mit sortiertem Vektor und Algorithmus `upper_bound` erreichbar)
  - Nachteile von `set`:
    - Pro Element (key) müssen zusätzlich 3 pointer (24 Byte) gespeichert werden.
    - Bei jedem Hinzufügen von Elementen wird zuerst nach dem Eintrag gesucht und dann sortiert eingefügt.
    - Kein wahlfreier Zugriff `[ ]` möglich, nur via Iterator.

  Dank **Datenkaspelung** bleiben die Interfaces der Methoden (fast) unverändert
  $\Longrightarrow$ Hauptprogramm *v_ 7a_ set/main.cpp* bleibt **unverändert**.

Nutzung der STL mit Container der Klasse Student *v_ 7b/*

Initialisierung und Ausgabe:

- `vector<Student>` über initializer list definieren;
  unterschiedliche Konstruktoren benutzt. `main.cc:43-46`
- Demo bei Loop: Index-Loop $\longrightarrow$ Iterator-Loop $\longrightarrow$ Range-For `main.cc:48-64`

Sortieren der Einträge:

- `sort` mit Operator `bool Student::operator<(Student)` `main.cc:67`
- Sortieren mit Funktion `bool num_studs(Student,Student)` `main.cc:72`
  binäre Funktion (2 Argumente vom Typ `Student`) `main.cc:92ff`
- Ersetzen `num_studs` als 3. Argument von `sort` direkt durch λ-Fkt. `main.cc:73`

Finden bestimmter Einträge im Vektor, z.B. Student(en) mit `cSKZ==865`.

- Ungeeignet: `find` findet nur identische Studenten via `operator==`.
- `find_if` benötigt eine **unäre** Funktion (1 Argument vom Typ `Student`)
  der Form: `bool unaer_fkt(Student)` .
  Analog bei `count_if` und ähnlichen Algorithmen `XXX_if`
- Problem: Wie kann Parameter `cSKZ` zur unären Funktion transferriert werden?
  - alt: Via Functorskonstruktor einer Structur: `main.cc:79; 28-36`
  - neu: Via Caption in λ-Funktion `main.cc:83-85`

Templates §10

Motivation

- [Haase, §10.1]: Jeweils 3 Funktionen mit identischem Funktionskörper, aber unterschiedlichen Datentypen bei den Input-Parametern.
  Die 3 Funktionen haben jeweils unterschiedliche Parameterlisten und damit unterschiedliche *Signaturen*.
- Auch mit höheren Datentypen wie `vector<int>`, `vector<float>`, ...
  Zeigen in Beispiel *v_ 8a/*
- $\Longrightarrow$ wie eine Schablone (engl. Template)

Die erste Templatefunktion §10.1

Mit Fkt. `float max_elem(const vector<float>& x)` im Bsp. *v_ 8a/*:

- Ersetze alle (passenden) Typdeklarationen `float` durch den *Templateparameter* `T` (oder einen anderen Bezeichner).
- Deklariere unmittelbar **vor** der Funktion den Templateparameter:
  `template <class T>`
- Das ganze im Templatefile *tfkt.tcc* speichern (auch *tfkt.tpp*, *tfkt.hpp*, *tfkt.h* üblich) und **als Headerfile inkludieren**.
- In *main.cpp* die 6 Funktionsdefinitionen kommentieren und Zeile 1 aktivieren.
- Erläuterung implizite und explizite Templateargumente demonstrieren und an Tafel erläutern [Haase, §10.1.2].
- Spezialisierung von Templates (Extrawurst für bestimmte Datentypen) [Haase, §10.1.3].

Eine Template-Funktion ist eigentlich eine Familie von Funktionen.

Größte Fibonaccizahl im Zahlbereich *v_ 8b/*

Grundidee: Was ist die größte Fibonaccizahl $f_n := f_{n-1} + f_{n-2}$ für den übergebenen Datentyp?
Lösungsidee:

- Alle f_n sind nichtnegativ.
- Damit muß gelten: $f_n >= f_{n-1}$.
  - Falls diese Bedingung nicht mehr gilt, dann wurde offenbar der (Integer-) Zahlbereich bei der Addition $f_{n-1} + f_{n-2}$ überschritten (wir fangen wieder links am Zahlenstrahl an).
  - Somit ist f_{n-1} die größe im Datentyp noch darstellbare F-Zahl.
- Template-Funktion `T fibo(T& n)` realisiert dies und
  (Loop in Funktion als Struktogramm erläutern)
  Template-Parameter `T` wird implizit beim Aufruf aus dem Datentyp von `n` der Parameterliste ermittelt.

- Die Template-Funktion `void check_fibo()` kann den Template-Parameter nicht aus der Parameterliste ableiten, daher muß der Template-Parameter explizit beim Aufruf angegeben werden: `check_fibo<short int>();`
- Spezialisierung für `check_fibo<signed char>();` nötig, da `char` nicht als Zahl ausgegeben wird:
  `template <> void check_fibo<unsigned char>()`
  Demonstrieren, es gibt auch kürzere Lösungen.
- T-Fkt.: `void check_fibo2()`
  - Überprüft auf `char`-Datentypen über Anzahl der Bytes für Speicherung
    `const bool is_char = sizeof(T)<2;`
  - Flexibler Datentyp für `tmp1` in
    `auto tmp1 =  is_char?static_cast<short>(f1):f1;`
  - Nutzt `is_integral<T>::value` um zur Compilezeit auf einen Integer-Datentyp für den Template-Parameter T zu überprüfen.
    Demo mit `check_fibo2<double>();` und Abbruch der Compilierung.
    Erfordert: `#include <type_traits>`

Funktionen (und Klassen) können mehr als einen Templateparameter besitzen, z.B.:

```
template <class T, class S>
S myfunc(T const a, list<S> const &b);
```

Vorteile von Templatefunktionen:

- Algorithmische Verbesserungen müssen nur einmal implementiertt werden.
- Vermeidet Copy-Paste Fehler im Vergleich zu mehreren Funktionen.
- Kleine Funktionen werden werden automatisch *inline* in aufrufender Funktion eingebaut. ´

Templateklasse §10.2

Analog zu Templatefunktionen wird eine Familie von Klassen erzeugt.

```
template<class T>
class X
{
...                          // Definition der Klasse X<T>
};
```

Erst mit der Belegung des Templateparameters wird eine Klasse erzeugt und ein Objekt dieser Klasse deklariert.

```
{
   X<int>    ai;
   X<float> fi;
   vector< X<char> > vc;
};
```

Umformulierung der Klasse `Komplex` in eine Template-Klasse `Komplex<T>`: *v_ 8b/*

1. `template <class T>` vor die Klassendeklaration schreiben. (**.h*)
2. Den zu verallgemeinernden Datentyp durch den Template-Parmeter `T` ersetzen. (**.h, *.tcc*)
   Achtung, vielleicht wollen Sie nicht an allen Stellen diesen Datentyp ersetzen.
3. Ersetze in allen Parameterlisten, Returntypen und Variablendeklarationen den Klasstentyp `Komplex` durch die Template-Klasse `Komplex<T>`. (**.h, *.tcc*)
4. Vor jede Methodenimplementierung `template <class T>` schreiben. (**.tpp*)
   Desgleichen vor Funktionen, welche die Klasse `Komplex::` als Parameter benutzen. (**.tpp*)
   Bei `friend`-Funktionen muß man `template <class T>` auch im Deklarationsteil vor der Funktion angeben. (**.h*)
5. In der Methodenimplementierung `Komplex::` durch `Komplex<T>::` ersetzen. (**.tpp*)
   Achtung: `Komplex::Komplex(...)` $\longrightarrow$ `Komplex<T>::Komplex(...)` , dasselbe beim Destruktor.
6. Im Headerfile das Sourcefile includieren, also `#include "komplex.tcc"`, oder gleich alles in das Headerfile schreiben (nicht empfohlen). (**.h*)

Weitere Topics:

- Mehrere Template-Parameter [Haase, §10.2.2].
- `friend`-Funktionen und Template-Klasse [Haase, §10.2.4].

Einschränkungen an Template-Datentyp §10.3

- Überprüfung via Type Traits [Haase, §10.3.1]. *v_ 8c_ cpp17/*
- C++20: *Concepts* für Typüberprüfung [Haase, §10.3.2]. *v_ 8c_ cpp20/ komplex2.h*

Literatur

[Haase] Gundolf Haase: Einführung in die Programmierung mit C++ (2024), *www*[1].

[Stroustrup10] Bjarne Stroustrup: Einführung in die Programmierung mit C++. Pearson Studium, München (2010).

[1] `http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/Script/html/script_programmieren.pdf`