Draft der C++-Vorlesung vom 26.April 2024

Bsp. Graph

Ein Graph wie der angegebene[1], unvollständige Graph, besteht aus Knoten (knots/vertices) welche über Kanten (edges) verbunden sind.
Im Beispielcode graph[2] wird die Kantenbeschreibung obigen Graphs eingelesen und daraus für jeden Knoten dessen sämtliche Knotennachbarn ermittelt.

1. Jede Kante besitzt stets genau 2 Knoten, jedoch ist die Anzahl der Kanten unbekannt. Somit ist z.B. folgende Datenstruktur zur Speicherung der Kantenbeschreibung sinnvoll:
   `vector<array<int,2>> edges;` // main.cpp:15
   D.h., jede Kante speichert die Indizes der sie definierenden Knoten.

2. Das Einlesen der Kantenbeschreibung erfolgt über die Funktion
   `read_edges_from_file(name, edges);` // main.cpp:16
   welche vom Vorlesungsbeispiel file_io[3] übernommen und adaptiert wurde. Die einzigen Änderungen zum Original sind `graph.cpp:18` und die Signatur der Funktion.

3. Die Knoten in obigem Graphen besitzen eine unterschiedliche Anzahl von Nachbarknoten welche mittels
   `auto n2n=get_node2nodes(edges);` // main.cpp:30
   aus der Kantenbeschreibung bestimmt werden. Das Schlüsselwort **auto** legt für die Variable `n2n` denselben Datentyp wie den Rückgabetyp der Funktion fest.

4. Bislang wissen wir noch nicht wieviele Knoten unser Graph besitzt. Unter Annahme einer fortlaufenden Numerierung wird diese Knotenanzahl in `graph.cpp:43-51` bestimmt.

5. Damit können wir eine Datenstrukur festlegen, welche für jeden Knoten dessen noch zu bestimmende Nachbarnknotenindizes jeweils in einem dynamisch wachsenden Vektor speichern wird. Mit
   `vector<vector<int>> n2n(nnode);` //n2n(nnode,vector<int>()) // graph.cpp:54
   wird die Anzahl der Nachbarknoten stets mit 0 initialisiert.

6. Der Algorithmus in `graph.cpp:54-61` liest aus der Kantenbeschreibung für jede Kante k die Indizes ihrer beiden Knoten und hängt diese wechselseitig beim anderen Knoten als Nachbarknoten an, die Vektoren der Nachbarn wachsen also dynamisch.
   ```
   {
       const int v0 = edges[k][0];
       const int v1 = edges[k][1];
       n2n[v0].push_back(v1);       // add v1 to neighborhood of v0
       n2n[v1].push_back(v0);       // and vice versa
   }
   ```
   Die aktuelle Anzahl der Nachbarn für Knoten k ist hierbei `n2n[k].size()`.

7. Abschließend werden die ermittelten Indizes der Nachbarknoten in `graph.cpp:65` für jeden Knoten aufsteigend sortiert [Haase, §11.3.2].

---

[1] `http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS24/graph_1.pdf`
[2] `http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS24/graph.zip`
[3] `http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS24/file_io.zip`

Klassen §9

Motivierung am Bsp. Graph
graph Klassisch $\longrightarrow$ *graph_3* objektorientiert

- Eigenschaften (engl. properties) der Klasse
  - Daten $\longrightarrow$ Member der Klasse
  - Funktionen $\longrightarrow$ Methods der Klasse
- Variable (`double p(0.2);`)
  $\longrightarrow$ Instanz/Objekt der Klasse (`graph g1("g_2.txt");`)
- Datenkapselung (engl. encapsulation) via `private` (´`protected`, `public`)
- Vererbung (engl. inheritance): Basisklasse, abgeleitete Klassen, Klassenhierarchie IS-A und HAS-A Eigenschaften abgeleiteter Klasse.
- virtuelle Methoden: abstrakte und konkrete Klassen
- Polymorphismus

Eigene Klasse implementieren:

- Grobentwurf der Klasse: Member (Daten) , Methoden (Funktionalität)
- Codeblocks: File → New → Class. [kurze Demo]
  $\longrightarrow$ *komplex.h. komplex.cpp*

Klasse Komplex

Komplexe Zahlen gibt es bereits in `<complex>`, die eigene Klasse dient zur Demonstration der Implementieriungsdetails dahinter.

Was soll die eigene Klasse `Komplex` für $x + yi$ enthalten?

- Realanteil x, Imaginäranteil y Member
- Deklaration ohne Initialisierung: `Komplex b` parameterloser Konstruktor
- Deklaration mit Initialisierung: `Komplex a(2.3,4.2)` Parameterkonstruktor
- Deklaration mit Initialisierung aus Realteil: `Komplex a(2.3)` Parameterkonstruktor
- Deklaration mit Objektinit.: `Komplex c(a)` Copykonstruktor
- Zuweisung: `b = a` Zuweisungsoperator `operator=` [Method]
- Addition: `a+b` Operator `operator+` [Method/Function]
- Ausgabe: `cout << c` Ausgabeoperator `operator<<` [Function]
- Getter-/Settermethoden
- Betrag: `abs(c)` [Method]

Für Lecturer:
Verzeichnis *v_6a/* in *v_6a_demo/* kopieren und *main.cpp*:11-27 kommentieren, desgleichen *komplex.cpp*:7-76.
Danach schrittweise die Funktionalität zuschalten.

Weiter wie im Skript §9.
Compiler option `-Weffc++`[4].

- Member: Real-, Imaginäranteil
- Konstruktoren (mehrere!):
  - Member Initialization List
  - ohne Parameter (Standardinitialisierung!?)
  - mit Parameter(n); optionale Argumente?
  - Copy-Constructor
  - [Move-Constructor]
  - Rule-of-Five[5]
- Destruktor (genau einer)
- Zuweisungsoperator
  - Copy-
  - [Move-]
- Methoden:
  - Setter
  - Getter (`const` Method)
  - `operator+` für `a+b` oder `a+2` (`const` Method)
  - `operator+=`
  - abs (`const` Method)
- Funktionen:
  - Ausgabe: `operator<<`
  - Eingabe: `operator>>`
  - `operator+` für `2+a`

Aktuelles C++17 in *v_6a_cpp17/*

Fun with ASCII[6]: *char_sum.cpp*

- How to achieve 103% performance?
- { HARDWORK, KNOWLEDGE, ATTITUDE, BULLSHIT };
- `magic=96` $\longrightarrow$ `magic=0` : BILLGATES (the third!)

Literatur

[Haase] Gundolf Haase: Einführung in die Programmierung mit C++ (2024), *www*[7].

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

[4] https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/compiler-options/compiler-option-details/compiler-diagnostic-options/weffc-qeffc.html
[5] https://en.cppreference.com/w/cpp/language/rule_of_three
[6] http://www.torsten-horn.de/techdocs/ascii.htm
[7] http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/Script/html/script_programmieren.pdf