C++ Tutorial

Schnelleinstieg in C++ zum Lösen von Olympiaden-Aufgaben

SWISS OLYMPIAD
IN INFORMATICS
Programmierwettbewerb für Jugendliche

Frühling 2013

Inhaltsverzeichnis

1 Grundlagen **2**
- 1.1 Vorwort . . . 2
- 1.2 Kompilieren und Ausführen . . . 2
- 1.3 Was wir schon über C++ gelernt haben . . . 3
  - 1.3.1 Include und namespace . . . 3
  - 1.3.2 Die main Funktion . . . 3
- 1.4 Programme mit Eingaben . . . 4

2 Judge **5**
- 2.1 Beispielproblem . . . 5
  - 2.1.1 Lösung . . . 6
  - 2.1.2 Lokales Testen . . . 6
  - 2.1.3 Testen unter Windows . . . 7

3 Die Grundlagen von C++ **9**
- 3.1 Datentypen und Operatoren . . . 9
- 3.2 Bedingungen . . . 10
  - 3.2.1 Tücken . . . 11
- 3.3 Schleifen . . . 12

4 Funktionen **14**
- 4.1 Call by value & Call by reference . . . 14

5 Arrays und Vektoren **16**
- 5.1 Arrays konstanter Grösse . . . 16
- 5.2 Vektoren . . . 17
  - 5.2.1 Langsame Vektor-Operationen . . . 18
  - 5.2.2 Mehrdimensionale Vektoren . . . 19

6 Klassen und Structs **20**

7 Datenstrukturen und Algorithmen in der STL **23**
- 7.1 Datenstrukturen . . . 23
  - 7.1.1 Listen, Stacks und Queues . . . 23
  - 7.1.2 Priority Queues, Sets und Maps . . . 23
- 7.2 Algorithmen . . . 25

8 Graphen **27**
8.1 Adjazenzliste 27
8.2 Adjazenzmatrix 28
8.3 Graphenalgorithmen 29

9 Zeitmanagement **31**
9.1 Lesen, Verstehen, Lösen 31
9.2 Debuggen 32
9.3 Laufzeitfehler 33

10 Kleine Hacks für den ambitionierten Olympiaden-Programmierer **36**
10.1 Fortgeschrittenes Verwenden von cout 36
10.2 Makros 37
10.3 Perlen aus der STL 38
10.3.1 Komplexe Zahlen 38
10.3.2 Versteckte Algorithmen 39

11 Danksagungen **40**

Kapitel 1

Grundlagen

1.1 Vorwort

Dieses Dokument wird dir einen Überblick der wichtigsten C++-Prinzipien geben, welche nötig sind, um die Aufgaben der Schweizer und Internationalen Informatik-Olympiaden erfolgreich zu lösen. Es ist kein kompletter C++-Lehrgang sondern ist für Leute gedacht, die schon mit C++ oder einer anderen Programmiersprache umgehen können, aber noch wenige Erfahrungen mit Contest-Programmierung haben. Das Dokument basiert auf dem C++-Tutorial der Vorlesung *Datenstrukturen und Algorithmen* der *ETH Zürich* von *Prof. Peter Widmayer*, wurde ursprünglich von *Dimitrios Leventeas* verfasst und durch das SOI-Team übersetzt und erweitert. Falls du inhaltliche oder stilistische Fehler findest, wären wir dir sehr dankbar, wenn du sie an *sandro@soi.ch* schicken könntest.

Im Folgenden werden wir den `g++` Compiler benutzen um unsere Programme zu kompilieren. Er läuft standardmässig auf Mac und Linux und kann mittels MinGW auch auf Windows benutzt werden. Du bist frei in der Wahl der IDE oder des Editors.

1.2 Kompilieren und Ausführen

Wir erklären die Grundlagen an folgendem *Hallo Welt* Programm.

```
#include <iostream>
using namespace std;

int main()
{
  // Gib Hallo Welt aus.
  cout << "Hallo Welt!" << endl;

  return 0;
}
```

Code 1.1: Ein Hallo-Welt Programm

Speichere das Programm 1.1 in einer Datei Namens `hallo_welt.cpp` und führe folgende zwei Befehle in der Konsole aus.

```
$ g++ -o programm_name hallo_welt.cpp -Wall
$ ./programm_name
```

Der erste Befehl kompiliert dein Programm und erstellt die ausführbare Datei `programm_name`. Der zweite Befehl führt das neu erstellte Programm aus, welches den Text "Hallo Welt!" auf der Konsole ausgeben sollte. Der Name der ausführbaren Datei wird nach dem `-o` Flag spezifiziert. Mit dem `-Wall` Flag schalten wir alle Warnungen ein.

Damit alles reibungslos klappt, musst du dich in der Konsole im selben Ordner befinden, in dem du auch die Datei gespeichert hast. Du kannst mit `pwd` den aktuellen Ordner abfragen und ihn mit `cd` wechseln.

1.3 Was wir schon über C++ gelernt haben

1.3.1 Include und namespace

Das Codestück 1.1 zeigt schon das Skelett, das all unseren Programmen zu Grunde liegen wird. Zu Beginn wird mit `#include` eine externe Bibliotheken eingefügt. In diesem Fall benutzen wir nur die Bibliothek für die *Ein-/Ausgabe*, die wir jedes Mal brauchen werden [1]. Gleich danach folgt der Befehl um den *std namespace* zu benutzen, damit wir vor die Aufrufe der Bibliotheksfunktionen nicht jedes Mal das Kürzel `std::` schreiben müssen.

1.3.2 Die main Funktion

```
int main()
{
    ...
    return 0;
}
```

Die einzige Funktion unseres Programm heisst `main()`, nimmt keine Argumente und gibt eine ganze Zahl (*Integer*) zurück. Die `main()` Funktion existiert in jedem (funktionierenden) C++ Programm und ist der Einstiegspunkt beim Ausführen. Der Rückgabewert wird vom Betriebssystem benutzt um zu entscheiden ob das Programm erfolgreich ausgeführt wurde (0) oder ein Fehler aufgetreten ist.

```
// Gib Hallo Welt aus.
cout << "Hallo Welt!" << endl;
```

Der eigentliche Code der Funktion beginnt mit einem Kommentar und gibt danach die Nachricht auf die Konsole aus. Das Kürzel `cout` repräsentiert die Standardausgabe und

[1] Wir werden noch viele weitere Bibliotheken kennen lernen und manchmal vergisst man deren Namen. Es ist deshalb eine gute Idee die C++ Referenz zu den Favoriten hinzuzufügen.

`endl` beschreibt den Beginn einer neuen Zeile. Wir benutzen den `<<`-Operator um einen Text an `cout` zu "senden".

1.4 Programme mit Eingaben

Wir kompilieren folgenden Code:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string name;

  cout << "Wen moechtest du begruessen?" << endl;
  cin >> name; // Lies einen String.
  cout << "Hallo " << name << "!" << endl;

  return 0;
}
```

Die Ausgabe nach ausführen des Programms sollte wie folgt sein:

```
$ ./programm_name
Wen moechtest du begruessen?
```

Falls du mit "Welt" antwortest, erscheint danach die Nachricht "Hallo Welt!". Merke, dass die Richtung des Operators für `cin` ändert. Wir "senden" den Text von der Standardeingabe an die String-Variable.

Kapitel 2

Judge

2.1 Beispielproblem

Wir werden anhand eines Beispielproblems durch die verschiedenen Schritte des Schreibens, Kompilierens und Ausführens der Lösung führen und schliesslich zeigen, wie die Lösung auf dem Judge eingeschickt wird.

Aufgabe 1. *Gegeben eine natürliche Zahl N, finde die Summe aller geraden Zahlen von 2 bis und mit N.*

Eingabe Die erste Zeile der Eingabe enthält eine einzelne ganze Zahl T, $1 \leq T$ welche die Anzahl der Testfälle angibt. Die folgenden T Zeilen enthalten je einen Testfall gegeben durch eine einzelne ganze Zahl N, $1 \leq N \leq 50000$.

Ausgabe Du musst genau T Zeilen mit je einem Testfall pro Zeile ausgeben. Die i-te Zeile muss eine einzelne ganze Zahl enthalten, die Summe aller geraden Zahlen von 2 bis und mit N.

Beispieleingabe

```
3          (Anzahl Testfälle)
5          (erster Testfall N = 5)
10         (zweiter Testfall N = 10)
1234       (dritter Testfall N = 1234)
```

Beispielausgabe

```
6          (Lösung für N = 5)
30         (Lösung für N = 10)
381306     (Lösung für N = 1234)
```

2.1.1 Lösung

Eine von vielen korrekten Lösungen zu der Beispielaufgabe ist Programm 2.1. Es beruht auf folgender Überlegung.

$$\sum_{\substack{i=2,\\ i \text{ gerade}}}^{N} i = \sum_{i=1}^{\lfloor \frac{N}{2} \rfloor} 2 \cdot i = 2 \cdot \sum_{i=1}^{\lfloor \frac{N}{2} \rfloor} i = 2 \cdot \binom{\lfloor \frac{N}{2} \rfloor + 1}{2} = \left\lfloor \frac{N}{2} \right\rfloor \cdot \left(\left\lfloor \frac{N}{2} \right\rfloor + 1 \right)$$

```
#include <iostream>

using namespace std;

int main()
{
  int T, N, sum;

  cin >> T;
  for(int i = 1; i <= T; ++i)
  {
    cin >> N;

    sum = N/2*(N/2 + 1);

    // Gib Loesung aus.
    cout << sum << endl;
  }

  return 0;
}
```

Code 2.1: Korrekte Lösung zum Beispielproblem.

2.1.2 Lokales Testen

Wir haben den Code gespeichert und kompiliert und können ihn nun testen. Dazu speichern wir zuerst die Beispieleingabe in der Datei `sum.in` und die korrespondierende Beispielausgabe in `sum.out`. Um das Programm auf diesen Testfällen zu testen, führen wir es wie folgt aus.

```
$ ./prog_name < sum.in
```

Der Inhalt der Datei `sum.in` wird als Eingabe für unser Programm verwendet. Wir können auf ähnliche Weise die Ausgabe unseres Programms in eine Datei umleiten.

```
$ ./prog_name < sum.in > prog.out
```

Die Eingabe wird von der Datei `sum.in` gelesen und die Ausgabe in die Datei `prog.out` geschrieben.

Nun wollen wir herausfinden, ob die Ausgabe unseres Programms (in Datei `prog.out`) mit der korrekten Ausgabe in Datei `sum.out` übereinstimmt. Falls dem nicht so sein sollte, haben wir einen Fehler in unserem Programm und müssen die Lösung gar nicht erst einschicken. Die Inhalte der beiden Dateien können automatisch mit dem Programm `diff` verglichen werden.

Angenommen unsere Lösung ist buggy und beim Ausführen der Beispieleingabe produziert sie folgende Ausgabe

```
6
30
381406
```

Code 2.2: Falsche Ausgabe in Datei `prog.out`.

Wie oben gesehen, ist die korrekte Antwort

```
6
30
381306
```

Code 2.3: Korrekte Antwort in Datei `sum.out`.

Es kann manchmal schwierig sein, solche Unterschiede von blossem Auge zu sehen. In genau solchen Fällen benutzen wir `diff`.

```
$ diff prog.out sum.out
3c3        (dritte Zeile von prog.out wird verglichen mit dritter Zeile von sum.out)
< 381406   (Zeile 3 von prog.out)
---
> 381306   (Zeile 3 von sum.out)
```

Die Ausgabe enthält alle nicht-übereinstimmenden Zeilen. In unserem Fall unterscheiden sich die beiden Dateien lediglich in der dritten Zeilen. Falls `diff` nichts ausgibt, dann sind die beiden Dateien identisch.

2.1.3 Testen unter Windows

Falls du unter Windows arbeitest, dann empfehlen wir dir Cygwin zu benutzen. Damit kannst du alle oben genannten Programme auch unter Windows laufen zu lassen.

Falls du wirklich Visual Studio (oder eine andere Windows IDE) verwenden willst, wollen wir dich auf einige Stolpersteine hinweisen.

Textdateien auf Windows haben eine andere Konvention bezüglich neuen Zeilen als auf Linux und Mac. Unsere Testdateien werden immer ein einzelnes `\n` Zeichen zum Trennen von Zeilen verwenden.

Zweitens sollte dir bewusst sein, dass nicht alle Compiler genau gleich sind und es dort

subtile Unterschiede gibt. Es kann Fälle geben, wo das genau gleiche Programm zwar unter Visual Studio kompiliert auf unserem Judge jedoch einen Kompilierfehler gibt (weil wir dort `g++` verwenden).

Zu allerletzt solltest du noch nach einer Alternative für das Programm `diff` suchen. Es gibt beispielsweise DiffUtils oder einige Plug-Ins für Editoren wie Notepad++.

Kapitel 3

Die Grundlagen von C++

3.1 Datentypen und Operatoren

Ein Überblick der wichtigsten C++-Datentypen ist in Tabelle 3.1 gegeben. Wir zeigen anhand eines Beispiels wie Variablen der verschiedenen Typen deklariert, initialisiert und mit den Operatoren benutzt werden. Dabei führen wir auch gleich einige nützliche Funktionen der Bibliotheken `cmath`, `cstdlib`, `utility` und `string` ein.

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <utility>
#include <cmath>
using namespace std;

int main()
{
  double d;
  int i, i2;
  string s;
  char c;

  // Bekannte Operatoren Reihenfolge gilt: * / % vor + - vor =
  i = 3 + 4*5 - 6;  // = 17
  i = 17/4;         // = 4 (Ganzzahldivison)
  i = 8%3;          // = 2 (Modulo)
  i = abs(-123);    // = 123 (Absolutwert)

  // Jede der folgenden Zeilen erniedrigt i um den Wert 1.
  i = i - 1;
  i -= 1;
  --i;

  d = -4.0/8.0;       // = -0.5 (Gleitkommadivison)
  d = fabs(-2.0);     // =  2.0 (Absolutwert)
  d = sqrt(2.0);      // =  1.4142... (Wurzel)
  d = pow(2.0, 4.0);  // =  16.0 (Potenzieren)
```

```
  s = "hallo welt";
  c = s.at(0); // = 'h'
  c = c + 2;   // = 'j'

  i = 10; i2 = 20;
  swap(i, i2); // i = 20, i2 = 10

  return 0;
}
```

Code 3.1: Variablen in C++ an einem Beispiel.

Keyword	Typ	Bereich (ungef.)
`int`	Integer	$-2 \cdot 10^9, \ldots, 2 \cdot 10^9$
`double`	Fliesskommazahl	$[-1 \cdot 10^{308}, 1 \cdot 10^{308}]$
`bool`	Wahrheitswert	$true, false$
`char`	Zeichen	ANSI Symbole
`string`	String	-

Tabelle 3.1: Die wichtigsten C++-Datentypen.

3.2 Bedingungen

Zum Berechnen von Wahrheitswerten haben wir eine ganze Reihe an logischen Funktionen: `&&`, `||` und `!` welche jeweils die Funktionen *logisches-und*, *logisches-oder* und *logisches-nicht* ausdrücken. Des weiteren gibt es die Vergleichsfunktionen `<`, `<=`,`>`, `>=`, `==` und `!=`, welche die offensichtlichen Bedeutungen haben. Wahrheitswerte werden oft in *if-then-else* Klauseln gebraucht um Bedingungen auszudrücken. Die grundlegende Form sieht wie folgt aus.

```
if(bedingung)
{
  ...
}
else if(bedingung)
{
  ...
}
else
{
  ...
}
```

Code 3.2: Das Skelett einer *if-then-else* Klausel.

Die geschweiften Klammern können weggelassen werden, falls danach nur ein Befehl folgt. Des weiteren sind die beiden Teile `else if` und `else` optional. Wahrheitswerte werden

“lazy” von Links nach Rechts ausgewertet, das bedeutet, dass die Berechnung abgebrochen wird, sobald das Resultat eindeutig bestimmt ist.

Angenommen wir haben zwei `bool`-Variablen `a` und `b` und die Bedingung `if(a && b)`. Da von links ausgewertet wird, schaut sich das Programm zuerst `a` an. Falls `a` den Wert `false` hat, wird die ganze Bedingung nie eintreffen egal welchen Wert `b` hält.

Der Code in 3.3 zeigt einige Beispiele wie Bedingungen in C++ benutzt werden können und wie wir “lazy” Auswertung zu unserem Vorteil nutzen können.

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  int i1, i2;
  i1 = 10;
  i2 = 20;

  bool c = ((i1 + i2) == 30) && (i1 == 9 || i2 > 20); // = false

  if(c)
    cout << "Wird nicht ausgefuehrt." << endl;
  else if(!c && i1 + 10 != i2)
    cout << "Wird nicht ausgefuehrt." << endl;
  else
    cout << "Wird ausfgefuehrt." << endl;

  // Lazy Auswertung zu unserem Vorteil nutzen.
  double d = -16.0;
  if(d >= 0.0 && sqrt(d) < 5.0)
  {
    // Diese if-Bedingung ist sicher.
    // Sie wird nie die Wurzel einer negativen Zahl abfragen.
    cout << sqrt(d) << " ist kleiner als 5." << endl;
  }

  return 0;
}
```

Code 3.3: Beispiele von Bedingungen in C++.

3.2.1 Tücken

Wir möchten im Folgenden auf einige oft-gemachte Fehler im Zusammenhang mit Bedingungen hinweisen. Schau dir den Code in 3.4 an und behalte diese Bugs beim Debuggen deiner eigenen Programme im Hinterkopf.

```
#include <iostream>
#include <cstdio>
#include <cmath>
using namespace std;
```

```
int main()
{
  int i = 10;

  // i = 20 ist eine Zuweisung. Wir wollen einen Vergleich: i == 20.
  if(i = 20)
    cout << "Wird ausgefuehrt." << endl;

  // Das Strichpunkt beendet den if-Block.
  if(1 == 2);
    cout << "Wird ausgefuehrt." << endl;

  // Die Genauigkeit von double hat Grenzen.
  double d = sqrt(2.0) * sqrt(2.0);
  if(d == 2.0)
    cout << "Wird nicht ausgefuehrt." << endl;

  // Wie man Fliesskommazahlen richtig vergleicht.
  const double epsilon = 1e-10; // = 0.0000000001
  if(fabs(d - 2.0) < epsilon)
    cout << "Wird ausgefuehrt." << endl;

  return 0;
}
```

Code 3.4: Tücken im Zusammenhang mit Bedingungen.

3.3 Schleifen

Es gibt drei Arten von Schleifen in C++, deren Grundformen im Codestück 3.5 gezeigt sind. Die beliebteste davon ist die *for*-Schleife. Neben der Schleifenbedingung kann optional ein Initialierungsblock angegeben werden, welcher am Start der Schleife ausgeführt wird und ein Inkrementierungsblock, welcher nach jeder Iteration ausgeführt wird.

```
for(initialisierung; bedingung; inkrementierung)
{
  ...
}

while(bedingung)
{
  ...
}

do
{
 ...
} while(bedingung);
```

Code 3.5: Die Skelette der drei C++-Schelifentypen.

Es gibt zwei weitere Befehle, welche im Zusammenhang mit Schleifen äusserst nützlich sind. Ausführen des Befehls `break` innerhalb einer Schleife bricht sie umgehend ab und Analog dazu kann der Befehl `continue` benutzt werden, um ans Ende der aktuellen Iteration zu springen.

Im Codestück 3.6 geben wir eine alternative Lösung zu Aufgabe 1. Sie ist langsamer und nicht so elegant, aber illustriert die besprochenen Schleifenkonzepte.

```
#include <iostream>
using namespace std;

int main()
{
  int testcases, N, sum;

  for(cin >> testcases; testcases > 0; --testcases)
  {
    cin >> N;

    sum = 0;
    for(int i = 1; true; ++i)
    {
      // Ueberspringe ungerade Zahlen.
      if(i%2 == 1)
        continue;

      // Abbrechen, sobald wir ueber N hinauskommen.
      if (i > N)
        break; // Beendet die innere Schleife.

      sum += i;
    }
    // Ausgeben der Loesung.
    cout << sum << endl;
  }

  return 0;
}
```

Code 3.6: Alternative Lösung zu Aufgabe 1.

Kapitel 4

Funktionen

4.1 Call by value & Call by reference

Jede C++-Funktion hat eine bestimmte Anzahl Argumente und maximal einen Rückgabetyp. Das Keyword `void` wird gebraucht, falls die Funktion nichts zurück gibt. Im Normalfall werden beim Ausführen einer Funktion die Argumente kopiert. Das nennt man "*call by value*". Alternativ erlaubt C++ die Deklaration von Argumenten, welche beim Ausführen der Funktion als Referenz übergeben werden. Das nennt sich "*call by reference*" und wird durch das Vorstellen eines `&`-Zeichens vor den Argumenttypen realisiert.

Die Beispiele 4.1 und 4.2 zeigen den Unterschied zwischen "call by value" und "call by reference".

```
void foo(string x)
{
  x = "foo";
}
int main()
{
  string s = "bar";
  foo(s);
  cout << s; // Gibt aus: bar.
  return 0;
}
```

Code 4.1: "Call by value".

```
void foo(string& x)
{
  x = "foo";
}
int main()
{
  string s = "bar";
  foo(s);
  cout << s; // Gibt aus: foo
  return 0;
}
```

Code 4.2: "Call by reference".

Jedes Mal wenn wir eine Funktion deklarieren, müssen wir uns überlegen, ob die Argumente als Werte oder Referenzen übergeben werden sollten. Falls die Änderungen an Argumenten innerhalb der Funktion von der aufrufenden Funktion bemerkt werden müssen, sollten wir "call by reference" wählen. Das Übergeben von Referenzvariablen kann auch dazu benutzt werden, um mehrere Rückgabewerte zu simulieren. Und wir benutzen auch "call by reference", falls der zu übergebende Wert sehr gross ist und wir ihn nicht bei jedem Funktionsaufruf kopieren wollen, da dies zu viel Zeit benötigen würde. In solchen

Fällen kombinieren wir mit dem `const` Keyword. Durch das Hinzufügen von `const` werden Änderungen an der Variable innerhalb der Funktion verunmöglicht. Beispiele aller drei Szenarien sind im Code Segment 4.3 aufgezeigt.

```
#include <iostream>
#include <string>
#include <cmath>
using namespace std;

// Mehrere Rueckgabewerte simulieren.
void to_cartesian(double r, double theta, double& x, double& y)
{
  x = r*cos(theta);
  y = r*sin(theta);
}

// Die Aenderungen sollen sichtbar sein.
void append(string& s)
{
  s += " Welt!";
}

// Kopieren von grossen Datenmengen verhindern.
char get_last_character(const string& s)
{
  return s.at(s.length() - 1);
}

int main()
{
  double x, y;
  to_cartesian(2.0, M_PI/4.0, x, y); // x = 1.414... und y = 1.414...

  string s = "Hallo";
  append(s); // s = "Hallo Welt!"

  char c = get_last_character(s); // = '!'

  return 0;
}
```

Code 4.3: Funktionen in C++ an Beispielen.

Kapitel 5

Arrays und Vektoren

5.1 Arrays konstanter Grösse

Die Arraygrösse wird direkt beim Deklarieren festgelegt und danach nicht mehr verändert. Die Grösse wird explizit in den eckigen Klammern nach dem Variablennamen erwartet. Der Zugriff auf Elemente eines Arrays erfolgt ebenfalls durch eckige Klammern. Merke, dass Arrays fortlaufende Speicherbereiche im RAM sind und darum der Zugriff auf ein beliebiges Element in konstanter Zeit geschieht. Meistens verwenden wir Arrays lediglich um kleine, konstant-grosse Mengen zu repräsentieren. Einige Beispiele sind in 5.1 zusammengefasst.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  // Ein 3-dimensionales Array: Insgesamt 8 Elemente.
  bool b[2][2][2];

  // Direkte Initialisierung eines Arrays.
  string s[2] = {"left", "right"};
  double d[2][3] = { {1.0, 2.0, 3.0}, {4.0, 5.0, 6.0} };
  b[0][1][0] = (d[1][1] == 5.0) && (s[0] == "left"); // = true

  // Traversierung der vier Nachbarn eines Punktes (x,y) in der Ebene.
  int x = 10, y = 20;
  int dx[] = {-1, 0, 1,  0};
  int dy[] = { 0, 1, 0, -1};

  // Gibt aus: (9, 20), (10, 21), (11, 20), (10, 19),
  for(int i = 0; i < 4; ++i)
    cout << "(" << x + dx[i] << ", " << y + dy[i] << "), ";
  return 0;
}
```

Code 5.1: Beispiele mit Arrays konstanter Grösse.

5.2 Vektoren

Oft sind die Grössen unserer Daten nicht konstant. Dann verwenden wir `vector`, welche dynamisch grössen-veränderbare Arrays implementieren und Zugriff auf einzelne Elemente in konstanter Zeit erlauben. Dem Konstruktor eines Vektors kann eine Anfangsgrösse und ein Wert für die initialen Elemente übergeben werden. Falls dieser Wert nicht angegeben wird, werden die Elemente auf den Standardwert des Datentyps gesetzt (Beispielsweise 0 für `int`). Der Zugriff auf Elemente geschieht entweder durch das Aufrufen der Funktion `at` oder mittels den eckigen Klammern. Der Unterschied zwischen den zwei Arten ist, dass die `at`-Funtkion testet, ob der Index innerhalb des gültigen Bereichs liegt und andernfalls einen Fehler ausgibt und das Programm abbricht. Um das Debuggen zu vereinfachen, sollte deshalb immer `at` verwendet werden.

Die Grösse des Vektors kann explizit mit der Funktion `resize` verändert werden. Das kann eine Verschiebung aller Elemente des Vektors zur Konsequenz haben und darum ziemlich lange dauern. Es wird deshalb empfohlen, die Grösse nicht explizit zu ändern, sondern die Funktion `push_back` zu benutzen, welche ein zusätliches Element ans Ende des Vektors anhängt. Es kann zwar immer noch passieren, dass `push_back` eine Verschiebung aller Elemente auslöst, die Funktion garantiert jedoch, dass sie im Durchschnitt nur konstante Zeit braucht. Um direkt das letzte Element abzufragen gibt es die Funktion `back` und um das letzte Element zu entfernen die Funktion `pop_back`.

Die wichtigsten Funktionen von `vector` sind nochmals im Codestück 5.2 zusammengefasst.

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
  // Vektor von string mit Groesse 4 und Anfangswert leeren String.
  // Zugriff auf v[4] wuerde einen Segmentation Fault geben.
  vector<string> v(4, "");
  v[0] = "You"; v[1] = "Are"; v[2] = "How"; v[3] = "World";

  v.push_back("Hello"); // push_back vergroessert den Vektor.

  for(int i = 0; i < (int)v.size(); ++i)
    v[i] += "...";

  while(!v.empty())
  {
    cout << v.back() << endl; // Gib das letzte Element aus.
    v.pop_back();             // Loesche das letzte Element.
  }
  return 0;
}
```

Code 5.2: Vektoren benützen.

5.2.1 Langsame Vektor-Operationen

Wir haben schon gesehen, dass `push_back` und `pop_back` dazu benutzt werden können, um Elemente am Ende von Vektoren einzufügen oder vom Ende zu entfernen. Weiter gibt es die Funktionen `insert` und `erase` welche Einfügen und Entfernen an beliebiger Stelle erlauben. Diese sollten jedoch nur sehr sorgfältig benutzt werden, da jeder einzelne Aufruf alle Elemente des Vektors kopieren muss. Vergleiche die folgenden beiden Codestücke, welche beide den gleichen Vektor mit den Elementen $n, n-1, \ldots, 1$ in absteigender Reihenfolge generieren. Beim Ausführen der beiden Beispiele wird jedoch deutlich, dass die Version mit `push_back` innerhalb einiger Millisekunden terminiert, während die Version mit `insert` etwa 30 Sekunden benötigt.

```
int n = 200000;
vector<int> dec;

// Elemente am Schluss einfuegen.
for(int i = n; i >= 1; --i)
  dec.push_back(i);
```

Code 5.3: Absteigende Sequenz in $\mathcal{O}(n)$.

```
nt n = 200000;
vector<int> dec;

// Elemente am Anfang einfuegen.
for(int i = 1; i <= n; ++i)
  dec.insert(dec.begin(), i);
```

Code 5.4: Absteigende Sequenz in $\mathcal{O}(n^2)$.

Ein weiterer Fehler im Zusammenhang mit Vektoren ist sie mittels "call by value" zu übergeben, obwohl sie innerhalb der Funktion gar nicht verändert werden. Wenn wir Argumente vom Typ Vektor haben, werden wir sie fast immer mittels "call by reference" übergeben wollen. Schau dir folgende zwei Codebeispiele an, welche nochmals verschiedene Arten implementieren, wie eine absteigende Sequenz erstellt werden kann. Wiederum terminiert das linke Beispiel sofort während das rechte etwa eine halbe Minute benötigt. Details zu "call by reference" werden im Kapitel 4 besprochen.

```
// Uebergabe Vektoren als Referenz
void add(vector<int>& cur,
         int i)
{
  cur.push_back(i);

}

int n = 200000;
vector<int> dec;

for(int i = n; i >= 1; --i)
  add(dec, i);
```

Code 5.5: Absteigende Sequenz in $\mathcal{O}(n)$.

```
// Uebergabe Vektoren als Wert
vector<int> add(vector<int> cur,
                int i)
{
  cur.push_back(i);
  return current;
}

int n = 200000;
vector<int> dec;

for(int i = n; i >= 1; --i)
  dec = add(dec, i);
```

Code 5.6: Absteigende Sequenz in $\mathcal{O}(n^2)$.

5.2.2 Mehrdimensionale Vektoren

Mehrdimensionale Vektoren sind lediglich Vektoren vom Typ Vektor. So wird ein zweidimensionaler Vektor vom Typ `string` beispielsweise als `vector<vector<string> >` deklariert. Man merke den Leerschlag zwischen den zwei spitzen Klammern, der benötigt wird um die Deklaration vom `>>`-Operator für `cin` zu unterscheiden.

Die Initialisierung von mehrdimensionalen Vektoren funktioniert nach dem genau gleichen Prinzip wie im eindimensionalen Fall, sieht aber etwas kryptisch aus. Folgendes Codebeispiel zeigt die Benutzung von mehrdimensionalen Vektoren und illustriert auch die Unterschiede der eckigen Klammern und der Funktion `at`.

```
// 20 mal 10 Vektor von Strings mit Anfangswert "Hallo".
vector<vector<string> > X(20, vector<string>(10, "Hallo"));

// Gueltiger Zugriff.
X[15][5] += " Welt!";

// Ungueltiger Zugriff aendert das Element X[6][3].
X[5][15] = "Unerwartet";
cout << X[6][3] << endl; // Gibt aus "Unerwartet"

// Laufzeitfehler. Index out of bounds.
X.at(5).at(15) = "Unerwartet";
```

Code 5.7: Mehrdimensionaler Vektor vom Typ `string`.

Kapitel 6

Klassen und Structs

Wir benutzen Klassen um Daten zu bündeln. Innerhalb der Klasse werden zwei Arten von Elementen unterschieden: *Felder* und *Funktionen*. In Feldern werden Daten gespeichert, auf welche die Funktionen arbeiten können. Mittels "*access-modifiers*" werden die Elemente einer Klasse in einen `public` und einen `private` Bereich aufgeteilt, welcher angibt ob die Elemente von ausserhalb der Klasse abrufbar sind oder nicht. Für Klassen ist der "access-modifier" standardmässig `private` und für Structs ist er standardmässig `public`. Das ist der einzige Unterschied zwischen Klassen und Structs. Traditionell werden Structs verwendet, wenn lediglich Daten repräsentiert werden sollen ohne dass wir irgendwelche Funktionen auf ihnen definieren wollen. Das kommt von der Art wie Structs in C verwendet werden. Wir werden im folgenden Klassen und Structs jedoch beliebig mischen.

Der optionale Konstruktor einer Klasse ist eine Funktion ohne Rückgabewert und hat den gleichen Namen wie die Klasse selbst. Er wird benutzt um die Felder der Klasse zu initialisieren. Des weiteren kann ein korrespondierender Destruktor implementiert werden, der aufgerufen wird sobald das Objekt gelöscht wird. Der Destruktor hat den selben Namen wie die Klasse mit einer vorgestellten Tilde (`~`).

Klassen können auf zwei Arten instanziert werden: Entweder explizit durch das Aufrufen des Konstruktors oder mittels des `new` Keywords. Letztere Variante gibt einen *Zeiger* auf die Speicherzelle zurück, an der die neue Instanz abgelegt wurde. Merke, dass diese Speicherzelle bestehen bleibt bis wir die Instanz explizit mittels des Befehls `delete` löschen. Zeiger können selbst in Variablen gespeichert werden, der korrespondierende Typ der Variable ist der Klassenname mit einem Nachgestellten Stern(`*`).

Explizite Instanzen werden automatisch gelöscht sobald die Variable nicht mehr aktuell ist (die Funktion, in der die Variable deklariert wurde, terminiert).

Um auf Elemente von Objekten zuzugreifen, verwenden wir den Punkt (`.`), falls es sich um eine explizite Instanz der Klasse handelt oder den Pfeil (`->`), falls das Objekt mittels eines Zeigers dereferenziert wird.

Die Unterschiede zwischen expliziten Instanzen und Instanzierung mittels `new` werden im folgenden Codestück zusammengefasst. Ein ausführlicheres Beispiel mit Klassen und

Structs ist im Codestück 6.3 gegeben.

```
struct Foo
{
  int bar;
  // Konstruktor.
  Foo()
  {
    bar = 123;
  }

  // Destruktor.
  ~Foo()
  {
    // Hier gibts nichts zu tun.
  }
};
```

```
int main()
{
  // Explizite Instanzierung.
  Foo foo = Foo();
  // Equivalent:
  // Foo foo;

  // Ein Feld abrufen.
  foo.bar = 321;

  // foo automatisch geloescht,
  // sobald Funktion beendet.

  return 0;
}
```

Code 6.1: Explizite Instanzierung.

```
int main()
{
  // Instanzierung mittels new.
  Foo* foo = new Foo();



  // Ein Feld abrufen.
  foo->bar = 321;

  // Loesche foo.
  delete foo;

  return 0;
}
```

Code 6.2: Instanzierung mittels `new`.

```
#include <iostream>
#include <cmath>

using namespace std;

// Konstruktoren und Destruktoren sind optional.
struct Point
{
  int x, y;
  double length()
  {
    return sqrt(x * x + y * y);
  }
};

class Rectangle
{
public:
  // Konstruktor
  Rectangle(int x_, int y_, int width_, int height_)
```

```
  {
    width = width_;
    height = height_;

    position = new Point();
    position->x = x_;
    position->y = y_;
  }


  // Destruktor
  ~Rectangle()
  {
    delete position;
  }

  int area()
  {
    return width * height;
  }

  // Die Funktion kann ausserhalb der Klasse implementiert werden.
  double distance();

private:
  int width, height;
  Point* position;
};

// Benutzen des ::-Operators (scope resolution),
// Wir implementieren die Funktion 'distance' der Klasse Rectangle.
double Rectangle::distance()
{
  return position->length();
}

int main()
{
  // Rechteck an Position (3,4) mit Breite 5 und Hoehe 6.
  Rectangle r = Rectangle(3, 4, 5, 6);

  cout << "Distanz: " << r.distance() << endl;
  cout << "Flaeche: " << r.area() << endl;

  return 0;
}
```

Code 6.3: Beispiel von Structs und Klassen.

Kapitel 7

Datenstrukturen und Algorithmen in der STL

Die *Standard Template Library (STL)* ist ein Teil von C++ und bietet Implementationen von vielen nützlichen Datenstrukturen und Algorithmen. Wir haben die Bibliotheken *strings*, *vectors* und *iostream* schon mehrfach verwendet und werden nun einige weitere praktische Verwendungen der STL sehen.

7.1 Datenstrukturen

7.1.1 Listen, Stacks und Queues

Manchmal benutzen wir einen Vektor obwohl wir eigentlich Elemente nur von einer Seite hinzufügen oder entfernen. Für solche Fälle gibt es `queue`, `stack` und `dequeue` (double-ended queue) in der STL. Ihre Benutzung ist recht intuitiv, sofern du weisst was ein Stack und eine Queue sind. Wir verweisen auf C++ reference für die Details.

7.1.2 Priority Queues, Sets und Maps

Neben den unsortierten Datenstrukturen hat die STL auch einige sortierte Datenstrukturen. So ist unter dem Namen `priority_queue` beispielsweise ein Heap implementiert, der Funktionen zum Einfügen und Entfernen des Maximums in jeweils logarithmischer Zeit bereitstellt.

Die beiden Datenstrukturen `set` und `map` implementieren beide balancierte Binärbäume. Beide erlauben Zugriff auf beliebige Elemente in logarithmischer Zeit. Der einzige Unterschied zwischen den beiden ist, dass `map` ein Mapping von Schlüsseln auf Werte enthält, während `set` lediglich Schlüssel speichert.

Wenn wir sortierte Datenstrukturen zusammen mit eigenen Klassen benutzen wollen,

müssen wir dafür sorgen, dass Objekte vergleichbar sind. Das geschieht durchs Überladen des Vergleichsoperators der Klasse. Ein Beispiel ist in 7.1 zu sehen.

Zu guter Schluss gibt es auch noch `multiset` und `multimap`, welche mehrfaches Vorkommen der gleichen Schlüssels erlauben.

```
#include <iostream>
#include <queue>
#include <set>
#include <map>
#include <string>
using namespace std;

struct Coordinate
{
  int x, y;
  Coordinate(int x_, int y_)
  {
    x = x_; y = y_;
  }

  // Ueberladen des Vergleichsoperators.
  // Vergleiche x-Werte und benutze y-Werte bei identischen x-Werten.
  bool operator<(const Coordinate& other) const
  {
    return (x < other.x) ||
           (x == other.x && y < other.y);
  }
};

int main()
{
  // Ein Heap von Koordinaten.
  priority_queue<Coordinate> pq;
  pq.push(Coordinate(1,2));
  pq.push(Coordinate(3,4));
  pq.push(Coordinate(5,6));

  // Die Koordinaten sortiert ausgeben (Maximum zuerst).
  while(!pq.empty())
  {
    cout << pq.top().x << "," << pq.top().y << endl;
    pq.pop();
  }

  // Set von ganzen Zahlen.
  set<int> s;
  s.insert(5);
  s.insert(4);
  s.insert(5);
  s.insert(4);
  cout << s.size() << endl; // Gibt aus: 2

  // Mapping von Koordinaten auf strings.
  map<Coordinate, string> m;
  m[Coordinate(1,3)] = "Untere linke Ecke";
```

```
  m[Coordinate(5,5)] = "Obere rechte Ecke";

  // Vor dem Zugriff wird getestet, ob das Element existiert.
  Coordinate c(5,5);
  if(m.find(c) != m.end())
    cout << m[c] << endl;

  return 0;
}
```

Code 7.1: Priority Queues, Sets und Maps.

7.2 Algorithmen

Die Bibliothek `algorithm` implementiert einige äusserst nützliche Funktionen. Wir werden oft die Funktion `sort` zum sortieren eines Vektors oder Arrays verwenden. Schau dir den Code in 7.2 für ein Beispiel an.

Viele algorithmische Funktionen der STL haben gewisse Anforderungen an die Eingabe und es liegt an dir sicherzustellen, dass diese erfüllt sind. Die Funktionen `binary_search` und `unique` gehen beispielsweise davon aus, dass der übergebene Vektor sortiert ist. Falls das nicht gilt, kann die Funktion ein unerwartetes Ergebnis zurückgeben.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Eigene Vergleichsfunktion definieren.
bool compare(const double& a, const double& b)
{
  return a > b;
}

int main()
{
  // Einen Vektor sortieren.
  vector<int> v(5);
  v[0] = 1;
  v[1] = 5;
  v[2] = 2;
  v[3] = 4;
  v[4] = 3;

  sort(v.begin(), v.end());

  // Gibt aus: 1 2 3 4 5
  for(int i = 0; i < (int)v.size(); ++i)
    cout << v[i] << " ";
  cout << endl;

  // Ein Array sortieren
```

```
  double d[5] = {4.0, 5.0, 3.0, 1.0, 2.0};
  sort(d, d+5);

  // Mit der selbst-definierten Vergleichsfunktion sortieren.
  sort(d, d+5, compare);

  // Gibt aus: 5 4 3 2 1
  for(int i = 0; i < 5; ++i)
    cout << d[i] << " ";
  cout << endl;

  return 0;
}
```

Code 7.2: Benutzung von `sort` der STL.

Kapitel 8

Graphen

Für dieses Kapitel solltest du mit Graphen vertraut sein und wissen, wie sie in Programmen repräsentiert werden können. Wir zeigen lediglich typische Implementierungen in C++.

8.1 Adjazenzliste

Die n Knoten werden von 0 bis $n-1$ nummeriert. Für ungewichtete Graphen werden die Nachbarn eines jeden Knotens in einem `vector<int>` gespeichert. Da wir eine solche Liste pro Knoten brauchen, wird der ganze Graph als `vector<vector<int> > graph` repräsentiert. Das Element `graph[i][j]` beschreibt die Knotennummer des j-ten Nachbars von Knoten i.

Um einen gewichteten Graphen zu repräsentieren reicht eine einzelne Zahl pro Kante nicht aus. Wir könnten eine Klasse definieren mit zwei Felder für den Nachbarsknoten und das Kantengewicht. Alternativ kann die Datenstruktur `pair` der STL benutzt werden.

Der Code in 8.1 geht davon aus, dass ein gewichteter Graph wie folgt von der Standardeingabe gelesen wird: Die erste Zeile enthält zwei ganze Zahlen n und m (getrennt durch ein Leerzeichen) – die Anzahl Knoten und Kanten des Graphen. Jede der folgenden m Zeilen enthält drei ganze Zahlen a, b und c ($0 \leq a \neq b < n$, $c \geq 0$), getrennt durch Leerzeichen. Die drei Zahlen beschreiben eine Kante von Knoten a zu Knoten b mit Gewicht c.

```
#include <iostream>
#include <vector>

using namespace std;

// Eine Kante hat ein Ziel und ein Gewicht.
struct Edge
{
  int to, weight;
  Edge(int to_, int weight_)
```

```
  {
    to = to_;
    weight = weight_;
  }
};


// Einen Alias fuer den Graphtypen definieren.
typedef vector<vector<Edge> > AdjList;

int main()
{
  int n, m ;
  cin >> n >> m ;
  AdjList graph(n) ;

  // Kanten einlesen.
  for(int i = 0; i < m; ++i)
  {
    int a, b, c;
    cin >> a >> b >> c;

    // Kante hinzufuegen.
    graph[a].push_back(Edge(b, c));
  }

  return 0;
}
```

Code 8.1: Reading a graph into an adjacency list.

8.2 Adjazenzmatrix

Obwohl wir meistens mit Adjazenzlisten arbeiten werden, ist es wichtig auch Adjazenzmatrizen zu kennen, die vor allem für sehr *dichte* Graphen geeignet sind. Ein Graph ist dicht, falls fast alle Kanten vorhanden sind. In diesem Fall sind die meisten Knoten zu fast allen anderen Knoten verbunden und anstatt alle Nachbarn in einer Liste zu speichern, können wir in einer Matrix einfach angeben, welche Nachbarn existieren. Ein ungewichteter Graph wird so als `vector<vector<bool> > graph` implementiert, wobei der Eintrag `graph[i][j]` angibt, ob eine Kante von i nach j existiert. Für gewichtete Graphen verwenden wir `vector<vector<int> > graph`, wobei der Eintrag gerade das Gewicht der Kante ist, oder eine vordefinierte Konstante, falls die Kante nicht existiert.

Das Codestück 8.2 liest einen Graphen nach obigem Format ein und speichert ihn in einer Adjazenzmatrix.

```
#include <iostream>
#include <vector>

using namespace std;


// Einen Alias fuer den Graphtypen definieren.
typedef vector<vector<int> > AdjMatrix;

int main()
{
  int n,m;
  cin >> n >> m;


  // n x n Matrix.
  // Der Wert -1 repraesentiert fehlende Kanten.
  AdjMatrix graph(n, vector<int>(n, -1));

  // Kanten einlesen.
  for(int i = 0; i < m; ++i)
  {
    int a, b, c;
    cin >> a >> b >> c;

    // Kante hinzufuegen.
    graph[a][b] = c;
  }

  return 0;
}
```

Code 8.2: Speichern eines Graphen in einer Adjazenzmatrix.

8.3 Graphenalgorithmen

Je nach dem welche Graphenrepräsentation gewählt wurde, müssen die Algorithmen unterschiedlich implementiert werden und werden unterschiedliche Laufzeiten haben. Als Beispiel zeigen wir eine rekursive Implementierung einer *Tiefensuche(DFS)*, welche die von 0 aus erreichbaren Knoten zählt. DFS hat eine Laufzeit von $\mathcal{O}(n^2)$ auf einer Adjazenzmatrix und $\mathcal{O}(n+m)$ auf einer Adjazenzliste[1]

[1]Wobei wir hier n für die Anzahl Knoten und m für die Anzahl Kanten verwenden.

```
// Zaehlt Anzahl von x
// erreichbarer Knoten.
int dfs(const AdjList& g,
        vector<bool>& visit,
        int x)
{
 int c=1;
 for(int i=0; i<g[x].size(); ++i)
  if(!visit[g[x][i].to])
  {
    visit[g[x][i].to] = true;
    c+=dfs(g, visit, g[x][i].to);
  }
  return c;
}

int main()
{
  // Liest Graph als AdjList...
  vector<bool> visit(n, false);
  cout << dfs(graph, visit, 0)
       << endl;
  return 0;
}
```

Code 8.3: DFS mit Adjazenzliste.

```
// Zaehlt Anzahl von x
// erreichbarer Knoten.
int dfs(const AdjMatrix& g,
        vector<bool>& visit,
        int x)
{
 int c=1;
 for(int i=0; i<g.size(); ++i)
  if(g[x][i]!=-1 && !visit[i])
  {
    visit[i] = true;
    c+=dfs(g, visit, i);
  }
  return c;
}

int main()
{
  // Liest Graph als AdjMatrix...
  vector<bool> visit(n, false);
  cout << dfs(graph, visit, 0)
       << endl;
  return 0;
}
```

Code 8.4: DFS mit AdjMatrix.

Kapitel 9

Zeitmanagement

Lösen von Olympiadenaufgaben kann zeitaufwendig sein und normalerweise hast du während eines Contests lediglich etwa fünf Stunden Zeit. Es ist deshalb wichtig, diese Zeit optimal zu nutzen.

In diesem Kapitel wollen wir einige Strategien aufzeigen, wie man die Aufgaben angehen sollte.

9.1 Lesen, Verstehen, Lösen

Der erste und weitaus wichtigste Schritt geschieht ohne Computer: Ganz am Anfang steht nämlich das sorgfältige und korrekte Lesen der Aufgabe. Einzelne Sätze in der Aufgabenstellung, wie zum Beispiel "*der Graph enthält keine Zyklen*" können den Unterschied zwischen einem einfachen und einem unlösbaren Problem ausmachen. Falls du also an einem Punkt nicht mehr weiter kommst, lies die Aufgabe nochmals, denn vielleicht hast du ja etwas überlesen.

Nachdem du das Problem gelesen und verstanden hast, geht es ans Lösen auf Papier. Bevor du überhaupt anfängst ans Programmieren zu denken, musst du genau wissen was du implementieren willst. Eine der grössten Zeitfallen ist zu früh mit Programmieren anzufangen. Falls du schon mit Implementieren beginnst, obwohl du deine Lösung nicht bis ins letzte Detail durch gedacht hast, kann es sein, dass du später stundenlang eine falsche Lösung am Debuggen bist. Die Formel lautet lieber 5 Minuten mehr auf Papier überlegen als danach 2 Stunden am PC einen Fehler zu suchen.

Eine gesunde Einstellung ist: Falls du keinen guten Grund hast, warum deine Lösung richtig sein sollte, dann ist sie es wahrscheinlich auch nicht. Versuche immer Gegenbeispiele zu konstruieren und dir zu überlegen, wie du die Korrektheit deiner Lösung beweisen würdest.

Zum Schluss ist es auch wichtig dir zu überlegen, wie du deinen Algorithmus *implementieren* kannst. Das heisst, du solltest schon auf Papier wissen, welche Datenstrukturen du genau verwenden willst und ob sie die benötigten Operationen schnell genug implemen-

tieren können.

9.2 Debuggen

Du hast lange und hart über deine Lösung nachgedacht und hast sie implementiert. Der nächste Schritt ist testen. Führe deine Lösung auf die Beispiele in der Aufgabenbeschreibung aus und kreiere möglichst viele kleine Testfälle mit etlichen Spezialfällen von Hand. Oft hilft es auch eine zweite langsame Lösung (beispielsweise Bruteforce) zu implementieren und dann grössere Testfälle von einem Zufallsprogramm erstellen zu lassen. Um die Ausgaben von grösseren Beispielen zu überprüfen solltest du das Programm `diff` verwenden, welches im Kapitel 2 erklärt wird.

Sobald du einen Fall gefunden hast, bei dem deine Lösung nicht korrekt ist, geht es ans debuggen. Falls die Eingabe klein-genug ist, führe deinen Algorithmus von Hand aus und vergleiche ihn mit den Werten, welche dein Lösungsprogramm ausgibt. Ändere dazu dein Programm so ab, dass es möglichst viele Zwischenresultate ausgibt.

Wenn dein Programm lediglich auf grossen Testfällen fehlschlägt ist das debuggen schwieriger. Versuche wiederum einige Zwischenresultate auszugeben und sie zu überprüfen. Alternativ kannst du die Bibliothek *assert* verwenden. Sie lässt dich eine Bedingung ausdrücken, von welcher du sicher bist, dass sie gelten muss. Du musst lediglich `assert(bedingung);` als Befehl einfügen, wo du sicher bist, dass `bedingung` wahr sein muss. Falls zu einem Zeitpunkt diese Bedingung dort `false` ist, wird das Programm automatisch abgebrochen und gibt die Zeilennummer der entsprechenden Zeile aus, und du erhältst damit einen Startpunkt um nach Fehlern zu suchen.

Betrachte den Beispielcode in 9.1. Es ist eine falsche Lösung zu Aufgabe 1 mit zusätzlichen Debug-Ausgaben und Assert-Bedingungen.

```
#include <iostream>
#include <cassert>

using namespace std;

int main()
{
  int testcases, N, sum;

  for(cin >> testcases; testcases > 0; --testcases)
  {
    cin >> N;
    cout << "DEB: Beginne neuen Testfall. N = " << N << endl;

    for(int i = 1; i <= N; ++i)
    {
      if(i%2 == 1)
        continue;

      cout << "DEB: Noch da i = " << i << " sum(alt) = " << sum;
      sum += i;
```

```
      cout << " sum(neu) = " << sum << endl;

      assert(sum%2 == 0);  // Die Summe muss immer gerade sein.
    }
    // Loesung ausgeben.
    cout << sum << endl;
  }

  return 0;
}
```

Code 9.1: Debuggen einer Lösung von 1.

Falls der Code mit der Beispieleingabe testen, erhalten wir folgende Ausgabe

```
DEB: Beginne neuen Testfall. N = 5
DEB: Noch da i = 2 sum(alt) = 0 sum(neu) = 2
DEB: Noch da i = 4 sum(alt) = 2 sum(neu) = 6
6
DEB: Beginne neuen Testfall. N = 10
DEB: Noch da i = 2 sum(alt) = 6 sum(neu) = 8
DEB: Noch da i = 4 sum(alt) = 8 sum(neu) = 12
...
```

Die zweitletzte Zeile offenbart das Problem. Die Variable `sum` ist am Anfang nicht 0, obwohl ein neuer Testfall angefangen hat. Wir haben vergessen `sum` zu Beginn jedes Testfalls auf null zu initialisieren.

Normalerweise gilt: Je mehr Debug-Ausgabe, desto besser. Wir müssen danach aber aufpassen, dass wir vor dem Einsenden der Lösung jede Debug-Zeile wieder entfernen, da der Judge nicht zwischen Debug-Zeilen und normaler Ausgabe unterscheiden kann.

9.3 Laufzeitfehler

Obwohl wir meistens ohne Debugger auskommen, ist er vor allem bei der Analyse von Laufzeitfehlern nützlich. Ein typisches Szenario ist die Verwendung von `at`. Wir beginnen gleich mit einem Beispiel:

```
#include <iostream>
#include <vector>

using namespace std;

const int SIZE = 100;
vector<int> hash_table(SIZE,-1);

// Element in Hashtabelle einfuegen.
void insert(int element)
{
  hash_table.at(element % SIZE) = element;
}
```

```
int main()
{
  insert(123456);
  insert(-4242);
  insert(654321)
  return 0;
}
```

Code 9.2: (Schlechte) Implementierung einer Hashtabelle.

Wenn wir den Code in 9.2 kompilieren und starten, wird die Ausführung mit folgender Nachricht abgebrochen:

```
terminate called after throwing an instance of 'std::out_of_range'
  what():  vector::_M_range_check
Aborted
```

Wir schliessen daraus, dass ein Zugriff auf ein Vektor-Element ungültig war. Um mehr Informationen über den Fehler zu finden, benutzen wir den Debugger `gdb`.

Zuerst kompilieren wir das Programm mit dem `-g` Flag, welches den Quellcode in die ausführbare Datei einschliesst. Danach starten wir das Programm mittels `gdb` und führen es im Debugger aus. Sobald der Laufzeitfehler eintritt, wird das Programm gestoppt und wir können in `gdb` durch verschiedenste Befehle wie `bt`, `frame`, `print`, `list` die zum gestoppten Zeitpunkt aktiven Funktionen einsehen und wechseln, sowie Variablen und Quellcode ausgeben.

```
$ g++ -g -o programm_name hash_tabelle.cpp -Wall
$ gdb programm_name
GNU gdb (Ubuntu/Linaro 7.3-0ubuntu2) 7.3-2011.08
...
(gdb) run
...
Program received signal SIGABRT, Aborted.
0x00007ffff75523e5 in __GI_raise (sig=6) at ../nptl/sysdeps/unix/sysv/linux/raise.c:6
64 ../nptl/sysdeps/unix/sysv/linux/raise.c: No such file or directory.
in ../nptl/sysdeps/unix/sysv/linux/raise.c
(gdb) bt
...
#8  0x0000000000400a5b in std::vector<int, std::allocator<int> >::at ...
#9  0x00000000004008a2 in insert (element=-4242) at hash_tabelle.cpp:11
#10 0x00000000004008c6 in main () at hash_tabelle.cpp:17
(gdb) frame 9
#9  0x00000000004008a2 in insert (element=-4242) at hash_tabelle.cpp:11
11   hash_table.at(element % SIZE) = element;
(gdb) print element
$1 = -4242
(gdb) print SIZE
$2 = 100
```

```
(gdb) print element%SIZE
$3 = -42
(gdb) list
6 const int SIZE = 100;
7 vector<int> hash_table(SIZE,-1);
8
9 void insert(int element)
10 {
11   hash_table.at(element % SIZE) = element;
12 }
13
14 int main()
15 {
(gdb)
```

Das sind nur die grundlegenden Funktionen eines Debuggers aber sollten für unsere Zwecke meist ausreichen.

Kapitel 10

Kleine Hacks für den ambitionierten Olympiaden-Programmierer

10.1 Fortgeschrittenes Verwenden von cout

Das Objekt `cout` kann durch mehrere Funktion der Bibliothek `iomanip` konfiguriert werden. So kann beispielsweise durch `setprecision` die Genauigkeit beim Ausgeben von Fliesskommazahlen geregelt werden oder mittels `setbase` das Zahlensystem geändert werden[1].

Um selbst-definierte Klassen mit `cout` zusammen zu verwenden, muss der `<<`-Operator überladen werden und so eine Nachricht basierend auf den Feldern der Klasse erstellt werden.

Diese ganzen Operationen können auch zur String-Manipulation benutzt werden. Anstatt das Objekt `cout` zu verwenden können wir selbst eine Instanz von `stringstream` aus der Bibliothek `sstream` generieren und mittels der Funktion `str` den daraus entstehenden Text nutzen.

Das Codestück in 10.1 zeigt Beispiele für all die besprochenen Fälle.

```
#include <iostream>
#include <sstream>
#include <iomanip>
#include <string>
#include <cmath>

using namespace std;

string to_hex(int value)
{
  stringstream s;
  s << setbase(16) << value;
  return s.str();
}
```

[1] Funktioniert lediglich für Basis 8, 10 und 16

```
int main()
{
  cout << "Die Konstante PI ist " << M_PI << endl;
  cout << "Genauer: " << setprecision(15) << M_PI << endl;

  int val = 123;
  cout << val << " in Hex:  " << to_hex(val) << endl;
  return 0;
}
```

Code 10.1: Verwendung von `cout`.

10.2 Makros

Makros sind Funktionen, die zur Kompilierzeit ausgeführt werden. Sie können dazu benutzt werden um oft benützte Konstrukte abzukürzen und so beim eigentlichen programmieren ein bisschen Zeit zu sparen. Vor allem für Programmierwettbewerbe an denen von zu Hause teilgenommen werden kann, ist das natürlich extrem nützlich. Makros beginnen durch `#define` und ähneln danach Funktionen.

Es gibt auch eine ganze Reihe von vordefinierten Makros. Die sind jedoch teilweise Compiler abhängig und für uns ist eigentlich nur `__LINE__` interessant, das benutzt werden kann um informativere Debug-Nachrichten zu erstellen.

Falls wir lediglich einen Alias für einen Datentyp erstellen möchten, können wir anstelle eines Makros auch `typedef` verwenden. Wir geben im Codesegment 10.2 einige Beispiele von oft gesehenen Makros und Verwendungen von `typedef` und wie sie im Code benutzt werden können.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// Makros.
#define PB push_back
#define MP make_pair
#define ALL(c) (c).begin(),(c).end()
#define FORALL(i,c) \
         for (typeof((c).begin()) i = (c).begin(); i != (c).end(); ++i)
#define FOR(i,m) for (int i = 0; i < m; i++)
#define DEBUG(x) cout << __LINE__ << ": " #x " = " << (x) << endl
// #define DEBUG // Kommentar entfernen beim Einsenden.

// Typedefs.
typedef long long ll;
typedef pair<int, int> PII;
typedef pair<ll, ll> PLL;
typedef vector<int> VI;
```

```
typedef vector<ll> VL;
typedef vector<VI> VII;
typedef vector<VL> VLL;


int main()
{
  VI vec;

  FOR(i,100)
    vec.PB(-i);

  sort(ALL(vec));

  FORALL(i,vec)
    DEBUG(*i);

  return 0;
}
```

Code 10.2: Makros für Programmierwettbewerbe.

10.3 Perlen aus der STL

10.3.1 Komplexe Zahlen

Die STL hat mit `complex` eine Bibliothek für komplexe Zahlen. Obwohl diese selbst bei Olympiaden-Aufgaben äusserst selten vorkommen ist die Bibliothek nützlich, da wir sie für 2D-Geometrie-Aufgaben benutzen können. Ein Punkt in der Ebene wird als komplexe Zahl repräsentiert, wobei die reelle Achse die X- und die imaginäre Achse die Y-Koordinate darstellt.

```
#include <iostream>
#include <complex>

using namespace std;

#define X real()
#define Y imag()
typedef complex<double> point;

int main()
{
  point p1(15.0,18.0), p2(0,0);
  p1.X -= 100.0;
  p1 *= 5;
  p2 += p1;

  // Distanz zum Nullpunkt.
  cout << p2 << endl;
  cout << "Distanz: " << abs(p2) << endl;
```

```
  cout << "Winkel: " << arg(p2) << endl;
  return 0;
}
```

Code 10.3: Geometrie mit komplexen Zahlen.

10.3.2 Versteckte Algorithmen

Wir möchten abschliessen mit der Aufforderung, sich die Bibliothek `algorithm` nochmals genauer anzuschauen. Wir haben schon einige Beispiele wie `sort` gesehen, es gibt jedoch viele versteckte Perlen, welche in gewissen Situationen sehr hilfreich sein können. Wir verweisen an dieser Stelle auf die Dokumentationen der Funktionen `nth_element`, `lower_bound`, `binary_search` und `next_permutation`.

C++ ist eine Sprache bei der sich in den letzten Jahren viel getan hat. Abhängig vom Compiler und dessen Version, stehen dir noch viele weitere nützliche Funktionen aus dem *C++ Technical Report 1* (versteckt im Namespace `tr1`) oder sogar aus dem neuen *C++11* zur Verfügung. Speziell erwähnt seien hier vor allem *Hashmaps* (`unordered_map`) und *Regular Expressions*.

Kapitel 11

Danksagungen

Die Schweizer Informatikolympiade bedankt sich bei der Gruppe von *Prof. Peter Widmayer* für die Erlaubnis zur Benutzung der ursprünglichen Version des Tutorials, sowie bei folgenden Teilnehmern, die durch ihre Hinweise und das Finden von Fehlern das Tutorial weiter verbessert haben.

- Florian Schroeder
- Benjamin Schmid