

Grundgerüst

Grundgerüst	Grundgerüst für einfache Programme
<p>Unser Grundgerüst besteht aus folgenden Komponenten:</p> <ul style="list-style-type: none">• Kommentare• Verfügbarmachung zusätzlicher Funktionalität (via <code>#include <...></code>)• main-Funktion <p>Beachte: Jedes ausführbare Programm braucht die main-Funktion.</p> <p>Beachte: Zusätzliche Funktionalität, wie beispielsweise Eingabe und Ausgabe, ist nicht in der “Grundfunktionalität” von C++ enthalten. Der Programmierer muss also mitteilen, wo diese zusätzliche “Funktionalität” definiert ist. Mittels <code>#include <...></code> kann er dem Compiler sagen, in welcher Befehlssammlung (genannt: Library) diese “Funktionalität” definiert ist. Ein- und Ausgabe sind beispielsweise in <code>iostream</code> definiert.</p>	
<pre>// Informatik - Serie 13 - Aufgabe 4e // Programm: my_program.cpp // Autor: X. M. Mueller (Gruppe F) #include <iostream> int main () { // Your code here... return 0; }</pre>	

Datentypen

<code>int</code>	Datentyp für ganze Zahlen
<pre>int a = 3; int b = a + 4;</pre>	

Operatoren

Programmier-Befehle - Woche 1

*	Multiplikation von zwei <i>R</i> -Werten.
Präcedenz: 14 und Assoziativität: links	
<pre>int a = -3 * 4; int b = 2 * a; // Note: use l-value as r-value</pre>	

=	Zuweisungsoperator. Weist einem L-Wert einen neuen Wert zu.
Präcedenz: 4 und Assoziativität: rechts	
<pre>int a; a = 4; // value 4 a = 3; // value 3</pre>	

>>	(siehe: <code>std::cin</code> unter "Input/Output")
Präcedenz: 12 und Assoziativität: links	

<<	(siehe: <code>std::cout</code> unter "Input/Output")
Präcedenz: 12 und Assoziativität: links	

Input/Output

<code>std::cin >> ...</code>	Eingabe via Terminal (z.B. Tastatureingabe)
Erfordert: <code>#include<iostream></code>	
<pre>int a; std::cin >> a; // stores the user input into a</pre>	

<code>std::cout << ...</code>	Ausgabe in das Terminal
Erfordert: <code>#include<iostream></code>	
Das <code>\n</code> bewirkt einen Zeilenumbruch. Man kann stattdessen auch <code>std::endl</code> verwenden (siehe Beispiel unten).	
<pre>// tell the user to enter a number std::cout << "Enter height in metres: "; int h; std::cin >> h; std::cout << "Your input was: " << h << " m\n"; // line breaks std::cout << "This text is output..." << std::endl; std::cout << "This text is output...\n"; // does the same std::cout << "Twice the same was output...\n";</pre>	

Programmier-Befehle - Woche 2

Datentypen

<code>unsigned int</code>	Datentyp für natürliche Zahlen (inklusive 0)
Literal: <code>...u</code>	
<pre>unsigned int a = 4; // Conversion int --> unsigned int unsigned int b = 4u; // No conversion std::cout << a - 5 << "\n"; // too small (underflow)</pre>	

Operatoren

<code>/</code>	Division
Präcedenz: 14 und Assoziativität: links	
Falls ints oder unsigned ints dividiert werden, so rundet der Operator automatisch zu 0 hin .	
<pre>unsigned int a = 9 / 3; // Result: 3 unsigned int b = 5 / 3; // Result: 1 int c = -3 / 2; // Result: -1</pre>	

<code>%</code>	Modulo. Rest der <i>Ganzzahl</i> division
Präcedenz: 14 und Assoziativität: links	
<code>%</code> gibt es <i>nur</i> für int und unsigned int. Bei negativen Zahlen übernimmt <code>%</code> das Vorzeichen des linken Operanden.	
<pre>int a = 5; int division = a / 3; // Result: 1 int rest = a % 3; // Result: 2 int negative = -5 % -3; // Result: -2</pre>	

Programmier-Befehle - Woche 2

<code>++...</code>	Prä-Inkrement. Erhöht den Wert der Variablen und gibt den <i>neuen</i> Wert zurück.
Präzedenz: 16 und Assoziativität: rechts Sonst gibt es noch: <code>--...</code> Prä-Dekrement	
<pre>int a = 0; int b = ++a; // b gets value 1, // a gets value 1</pre>	

<code>...++</code>	Post-Inkrement. Erhöht den Wert der Variablen und gibt den <i>alten</i> Wert zurück.
Präzedenz: 17 und Assoziativität: links Sonst gibt es noch: <code>...--</code> Post-Dekrement	
<pre>int a = 0; int b = a++; // b gets value 0, // a gets value 1</pre>	

<code>+=</code>	Addiert den rechten Operanden zum linken Operanden.
Präzedenz: 4 und Assoziativität: rechts Sonst gibt es noch: <code>-=...</code> für Subtraktion <code>*=...</code> für Multiplikation <code>/=...</code> für Division <code>%=...</code> für Modulo	

(...)

Programmier-Befehle - Woche 2

(...)

```
int a = 4;
a += 5;    // a gets value 9
```

Generell

```
std::numeric_limits<T>::min()
```

Ermittelt **kleinsten zulässigen Wert** des Datentyps **T**.

Erfordert: `#include<limits>`

Sonst gibt es noch:

```
std::numeric_limits<T>::max()
```

```
int lower_bdd = std::numeric_limits<int>::min();
std::cout << "Enter a number larger than " << lower_bdd << ": ";
int input;
std::cin >> input; // User knows the smallest valid number.
```

Programmier-Befehle - Woche 3

Datentypen

<code>bool</code>	Datentyp für Wahrheitswerte
Literal: <code>true</code> , <code>false</code>	
<pre>bool t = true; if (!t == false) std::cout << "This is output!\n"; if (t) std::cout << "This is output as well!\n";</pre>	

Operatoren

<code>&&</code>	Logisches UND
Präzedenz: 6 und Assoziativität: links	
Kurzschluss-Auswertung: <code>&&</code> wertet <i>immer</i> den <i>linken</i> Operanden zuerst aus. Ist dieser <i>falsch</i> (also <code>false</code>), so wird der <i>rechte</i> Operand <i>nicht mehr</i> ausgewertet.	
<pre>if (3 > 2 && 10 > 11) // no short circuit evaluation std::cout << "Of course not!\n"; int a = 3; if (false && ++a < 2) // short circuit evaluation std::cout << "Of course not!\n"; std::cout << a << "\n"; // Output: 3</pre>	

Programmier-Befehle - Woche 3

	Logisches ODER
<p>Präzedenz: 5 und Assoziativität: links</p> <p>Kurzschluss-Auswertung: wertet auch <i>immer</i> den <i>linken</i> Operanden zuerst aus. Ist dieser <i>wahr</i> (also <i>true</i>), so wird der <i>rechte</i> Operand <i>nicht mehr</i> ausgewertet.</p>	
<pre>if (8 < 3 -2 > -5) // no short circuit evaluation std::cout << "Yes!\n"; int a = 3; if (3 < 8 ++a < 2) // short circuit evaluation std::cout << "Yes!\n"; std::cout << a << "\n"; // Output: 3</pre>	

!	Logisches NICHT
<p>Präzedenz: 16 und Assoziativität: rechts</p>	
<pre>int a; int b; int c; std::cin >> a >> b >> c; // read three int values from user if (! (a <= b && c <= b)) std::cout << "b is not max!\n";</pre>	

<	strikt kleiner
(...)	

Programmier-Befehle - Woche 3

(...)

Präzedenz: 11 und Assoziativität: links

Sonst gibt es noch:

> **strikt grösser**
<= **kleiner gleich**
>= **grösser gleich**

```
int a;  
int b;  
std::cin >> a >> b; // read two int values from user  
  
if (a < b)  
    std::cout << "a smaller than b\n";
```

==

exakt gleich

Präzedenz: 10 und Assoziativität: links

Sonst gibt es noch:

!= **ungleich**

```
int a;  
int b;  
std::cin >> a >> b; // read two int values from user  
  
if (a == b)  
    std::cout << "a is equal to b\n";
```

Schleifen

for (...) {...}

for-Schleife

Wenn man eine leere Condition als Abbruchbedingung angibt, so wird diese als *wahr* interpretiert.

(...)

Programmier-Befehle - Woche 3

(...)

```
unsigned int n;
std::cin >> n;

// Compute 1 + 2 + 3 + ... + n
unsigned int sum = 0;
for (unsigned int i = 1; i <= n; ++i)
    sum += i;
std::cout << "1 + 2 + ... + n = " << sum << "\n";
```

Generell

if-else	bedingtes Ausführen von Code
Der <code>else</code> -Teil ist optional.	
<pre>int a; int b; std::cin >> a >> b; // read two int values from user // if a < b then output 3; otherwise output 8 if (a < b) std::cout << 3 << "\n"; else std::cout << 8 << "\n";</pre>	

Datentypen

<code>float</code>	Datentyp für Zahlen mit Nachkommastellen (32 Bit)
<p>Literal: ohne Exponent: <code>288.18f</code>, mit Exponent: <code>0.28818e3f</code></p> <p>Der Modulo-Operator <code>%</code> existiert für <code>float</code> nicht.</p>	
<pre>float a = 288.18f; float b = 0.28818e3f / a; // computations work as expected float c; std::cin >> c; // float user input</pre>	

<code>double</code>	größerer Datentyp für Zahlen mit Nachkommastellen (64 Bit)
<p>Literal: ohne Exponent: <code>288.18</code>, mit Exponent: <code>0.28818e3</code></p> <p>Unterschied zu <code>float</code>: <code>double</code> ist genauer (grössere Präzision und grösseres Exponenten-Spektrum), braucht aber mehr Platz im Speicher (<code>float</code>: 32 Bit, <code>double</code>: 64 Bit).</p> <p>Der Modulo-Operator <code>%</code> existiert für <code>double</code> nicht.</p>	
<pre>double a = 288.18; double b = 0.28818e3 / a; // computations work as expected double c; std::cin >> c; // double user input</pre>	

Schleifen

<code>while (...) {...}</code>	while-Schleife
<pre>// Compute number of binary digits for input > 0 unsigned int bin_digits = 0; unsigned int input; std::cin >> input; assert(input > 0); while (input > 0) { input /= 2; ++bin_digits; }</pre>	

<code>do {...} while (...);</code>	do-Schleife
<p>Der Unterschied zur <code>while</code>-Schleife ist, dass der Rumpf der <code>do</code>-Schleife mindestens einmal ausgeführt wird. Sie hat ein <code>“;”</code> am Schluss.</p>	
<pre>int input; do { std::cout << "Enter negative number: "; std::cin >> input; } while (input >= 0); std::cout << "The input was: " << input << "\n";</pre>	

Programmier-Befehle - Woche 4

<code>break</code>	Schleife abbrechen
<pre>double input; int n; std::cin >> input >> n; // Divide input by n numbers // Stop if 0 is entered. for (int i = 0; i < n; ++i) { double k; std::cin >> k; if (k == 0) break; // go straight to Output input /= k; } // Output std::cout << input << " remains\n";</pre>	

<code>continue</code>	zur nächsten Iteration springen
Bei der for-Schleife wird das Inkrement noch ausgeführt .	
<pre>double input; int n; std::cin >> input >> n; // Divide input by n numbers // Skip entered 0's. for (int i = 0; i < n; ++i) { double k; std::cin >> k; if (k == 0) continue; // go straight to ++i input /= k; } // Output std::cout << input << " remains\n";</pre>	

Andere Kontrollanweisungen

<code>switch</code>	Fallunterscheidung
<p>Wird ein case nicht mit einem <code>break</code> abgeschlossen, so werden die darunter liegenden cases auch noch ausgeführt, bis ein <code>break</code> erreicht wird.</p> <p>Die einzelnen Unterscheidungswerte müssen Konstanten sein.</p>	
<pre>std::cout << "Behind which door (1,2,3) is the prize?"; int door_number; std::cin >> door_number; switch (door_number) { case 1: case 3: std::cout << "Wrong choice :-(\n"; break; case 2: std::cout << "You won the prize!\n"; break; default: std::cout << "Error: unknown door number.\n"; } // User inputs 0 --> Error: unknown door number. // User inputs 1 --> Wrong choice :-(// User inputs 2 --> You won the prize! // User inputs 3 --> Wrong choice :-(</pre>	

Generell

<code>{ ... }</code>	Block
<p>Blöcke spielen eine grosse Rolle, wenn es darum geht, wo im Programm eine Variable gültig ist. So ist eine Variable ab ihrer Deklaration bis hin zum Ende des Blocks, in dem sie definiert wurde potentiell gültig.</p>	

(...)

(...)

```
int main () {
    unsigned int a;
    std::cin >> a;
    if (a < 4) {
        std::cout << a << " "; // a exists in nested blocks
        int b = 18;
        std::cout << b << " "; // b exists here too
    } else {
        std::cout << b << " "; // Error: b not declared yet
        int b = 11;
    }
    std::cout << a << " "; // a still exists here
    std::cout << b << "\n"; // Error: b does not exist anymore
    return 0;
}
```

Funktionen

Funktion	Selbstständiger Codeabschnitt
<p>Wichtige Befehle:</p> <p>Definition: <code>int my_fun (bool arg1, float arg2) {...}</code> Rückgabe: <code>return my_val;</code> Aufruf: <code>my_fun(true, 3.75f)</code></p> <p>Der Rückgabewert wird immer zum Rückgabotyp konvertiert.</p> <p>Jede Funktion, die nicht den Rückgabotyp <code>void</code> hat, muss ein return haben.</p>	
<pre>unsigned int bin_digits (unsigned int n) { if (n == 0) return 1; // stops function and returns 1 unsigned int count = 0; do { n /= 2; ++count; } while (n > 0); return count; } int main () { std::cout << bin_digits(3) << "\n"; // Output: 2 std::cout << bin_digits(8) << "\n"; // Output: 4 return 0; }</pre>	

<code>// PRE: ...</code> <code>// POST: ...</code>	Funktionsbeschreibung
<p>PRE-/POST-Conditions gehören vor jede Funktionsdefinition ausser der main-Funktion. (In diesen Programmier-Befehlszusammenfassungen werden sie aber manchmal aus Platzgründen weggelassen.)</p> <p>Man kann beispielsweise <code>assert</code> verwenden, um das Programm abzubrechen, falls die Funktion doch mal mit Argumenten aufgerufen wird, welche die PRE-Condition verletzen.</p>	

(...)

Programmier-Befehle - Woche 5

(...)

```
// POST: return value is a^4
int power_4 (unsigned int a) {
    return a*a*a*a;
}

// PRE: width >= 0 and height >= 0
// POST: returns the rectangle area given by width and height
double area (double width, double height) {
    assert(width >= 0 && height >= 0);
    return width * height;
}
```

Datentypen

<code>void</code>	Datentyp für Funktion ohne Rückgabe.
void-Funktionen haben keinen Rückgabewert, aber sinnvollerweise einen Effekt (z.B. Textausgabe im Beispiel unten).	
<pre>void print_account (double assets, double interest) { std::cout << "Your assets: " << assets << "\n" << "Your interest: " << interest << "\n"; }</pre>	

Programmier-Befehle - Woche 6

Funktionen

Deklaration	Erweiterung des Gültigkeitsbereiches der Funktion
<p>Eine Funktion kann im Programm nur in Zeilen verwendet werden, welche nach der ersten Deklaration der Funktion kommen.</p> <p>Die Definition der Funktion ist immer gleichzeitig auch eine Deklaration.</p>	
<pre>void B (int i); // separate declaration void A (int i) { if (i <= 0) return; // stop calls std::cout << "A"; B(i/2); // use of B although its definition happens below } void B (int i) { if (i <= 0) return; // stop calls std::cout << "B"; A(i/2); }</pre>	

Standardbibliothek

<code>std::pow</code>	Potenzieren
<p>Erfordert: <code>#include<cmath></code></p>	
<pre>double a = std::pow(2.5, 2); // Computes 2.5 ^ 2 == 6.25</pre>	

<code>std::sqrt</code>	Quadratwurzel
------------------------	---------------

(...)

Programmier-Befehle - Woche 6

(...)

Erfordert: `#include<cmath>`

IEEE 754 garantiert, dass der (mathematisch) exakte Wert auf die nächste repräsentierbare Zahl gerundet wird.

```
double a = std::sqrt(14.0625); // Result: 3.75
```

`std::abs`

Absolutbetrag

Erfordert: `#include<cmath>`

```
double a = std::abs(-3.5); // Result: 3.5
```

`std::min`

Minimum zweier Argumente

Erfordert: `#include<algorithm>`

Sonst gibt es noch:

`std::max` **Maximum** zweier Argumente

Wichtig ist, dass beide Argumente **vom selben Typ** sind. Sonst geht es **nicht**.

```
double z;  
std::cin >> z;  
std::cout << std::min(z, 1.0); // min of z and 1  
  
std::cout << std::min(z, 1); // Error: z is double, 1 is int
```

Programmier-Befehle - Woche 6

Generell

<code>namespace</code>	Katalogisierung von Befehlen
<p>Mit <code>namespaces</code> kann man Funktionen, Typen, etc. katalogisieren (z.B. bezüglich Projekten). Beispielsweise werden viele der “offiziellen” Funktionen dieser Vorlesung im <code>namespace ifmp</code> zusammengefasst. So können Sie diese Funktionen einfach von Ihren eigenen Funktionen unterscheiden.</p> <p>Ausserdem kann man bei grösseren Projekten mit <code>namespaces</code> verschiedenste Namenskonflikte verhindern (z.B. bei gleich benannten Funktionen).</p>	
<pre>namespace ifmp { // namespace called ifmp // POST: "Hi" was written to the terminal void output_func () { // this function is in namespace ifmp std::cout << "Hi"; } } int main () { ifmp::output_func(); // use output_func from namespace ifmp return 0; }</pre>	

<code>assert</code>	sofortiges Stoppen des Programms bei Verletzung einer Bedingung (zu Testzwecken)
<p>Erfordert: <code>#include<cassert></code></p> <p>Wenn das fertige Programm veröffentlicht werden soll, kann man die <code>assert</code>-Befehle bequem deaktivieren.</p>	
<pre>int a; int b; std::cin >> a >> b; // read two int values from user assert(b != 0); // prevent division by 0 std::cout << a / b << "\n";</pre>	

Datentypen

<code>const ...</code>	<i>Schreibzugriff</i> auf Variable verbieten
<p>Gemeint ist natürlich der Schreibzugriff <i>nach</i> der Initialisierung.</p> <p><code>const</code> gibt es auch für Referenzen, siehe unten.</p>	
<pre>int a = 3; const int b = 4; a = 5; // valid b = 3; // not valid since b is const int c = -2 * b; // valid since just WRITE-access to b is // forbidden by "const"</pre>	

Referenzen	Alias für bestehende Variable
<p>Referenzen können nur Variablen ihres zugrundeliegenden Typs referenzieren. Sonst gibt es einen Fehler.</p> <p>Ausserdem können Referenzen nur mit L-Werten initialisiert werden (also Werten mit einer Adresse im Speicher).</p> <p>Funktionen, bei denen die Argumente Referenztyp haben, können ihre Aufrufargumente ändern. Das ist eine sehr mächtige Anwendung von Referenzen. Siehe beispielsweise die Funktion <code>swap</code> aus der Vorlesung.</p>	
<pre>// Usage int a = 3; int& b = a; // reference to a std::cout << b << "\n"; // Output: 3 a = 18; std::cout << b << "\n"; // Output: 18 b = 25; std::cout << a << "\n"; // Output: 25 // Issues int& c = 3; // Error: 3 is not an lvalue (3 has no address) bool d = false; int& e = d; // Error: d is bool, e wants to reference an int</pre>	

Programmier-Befehle - Woche 7

const Referenzen	const-Alias für bestehende Variable
<p>Im Prinzip funktionieren <code>const Referenzen</code> so wie normale Referenzen, bloss dass der Schreibzugriff auf das Ziel der Referenz <i>via diese Referenz</i> verboten ist.</p> <p>Ein weiterer Unterschied ist, dass <code>const Referenzen</code> R-Werte beinhalten können. Dann wird jeweils ein temporärer Speicher für den R-Wert erstellt, der solange gültig ist, wie die <code>const Referenz</code> selbst. Dies erlaubt beispielsweise, eine Funktion bezüglich Call-by-Reference trotzdem mit R-Werten aufzurufen.</p> <p>Zu beachten ist auch, dass man keine nicht-const Referenz mit einer const Referenz initialisieren darf.</p>	
<pre>double a = 3.0; double& b = a; // non-const reference const double& c = a; // const reference c = 4.0; // Error: write-access forbidden a = 5.0; // this works, a can be changed through itself b = 6.0; // this works, a can be changed through non-const refs std::cout << c << "\n"; // Output: 6.0, read-access is allowed. double& d = c; // Error: non-const ref from const ref not allowed const double& e = 5.0; // this works for const references.</pre>	

Programmier-Befehle - Woche 7

Vektoren	“Massenvariable” eines bestimmten Typs
<p>Erfordert: <code>#include<vector></code></p> <p>Wichtige Befehle:</p> <p>Definition: <code>std::vector<int> my_vec =</code> <code>std::vector<int>(length, init_value);</code></p> <p>Zugriff: <code>my_vec.at(2) = 8 * my_vec.at(3);</code></p> <p>neues Element hinten: <code>my_vec.push_back(5)</code></p> <p>(Anstatt <code>int</code> gehen natürlich auch andere Typen.)</p> <p>Statt der Syntax</p> <pre>std::vector<int> my_vec = std::vector<int>(...)</pre> <p>zur Deklaration und Initialisierung eines Vektors, kann alternativ auch eine der folgenden verwendet werden:</p> <pre>auto my_vec = std::vector<int>(...) // 1st alternative std::vector<int> my_vec(...) // 2nd alternative</pre> <p>In Alternative 1 kann die Typdeklaration der Variable durch das Schlüsselwort <code>auto</code> ersetzt werden, da der Typ durch die rechte Seite eindeutig bestimmt werden kann. In Alternative 2 ist es sozusagen andersherum: Da der Typ der neuen Variablen bekannt ist, muss er bei der Initialisierung nicht zwingend wiederholt werden. Alle drei Formen sind auch für andere Typen nutzbar, d.h. nicht speziell für Vektoren. Mehr zu Objektinitialisierung (und Konstruktoraufrufen) erfahren Sie im Kapitel zu Klassen.</p>	
<pre>int len; std::cin >> len; // Assume here: len > 2 // my_vec: 0, 0, 0, ..., 0 std::vector<int> my_vec = std::vector<int>(len, 0); my_vec.at(1) = 3; // my_vec: 0, 3, 0, ..., 0</pre>	

Programmier-Befehle - Woche 7

<code>char</code>	Datentyp für Zeichen
<p>Literal: <code>'a'</code> für Zeichen (<i>einfache</i> Anführungszeichen) Literal: <code>"Hello World"</code> für Strings (<i>doppelte</i> Anführungszeichen)</p> <p><code>chars</code> können sehr einfach zu <code>int</code> hin und her umgewandelt werden. (Der resultierende <code>int</code>-Wert ist auf den meisten Plattformen eine entsprechende Zahl gemäss ASCII-Code, siehe Vorlesungshandout 7, Slide 45.)</p>	
<pre>char ch = 'd'; int i = ch; // convert char --> int (here: 'd' --> 100) ++ch; // increase to 101 which is 'e' ++i; std::cout << (ch == i) << "\n"; // compare 101 == 101 // Read single character from user: std::cin >> ch;</pre>	

Operatoren

<code>my_vec.at(...)</code>	Vektor-Zugriff (at-Methode)
<p>Präcedenz: 17 und Assoziativität: links</p> <p>Nicht vergessen: Indizes beginnen bei 0 und nicht 1</p>	
<pre>// Directly Initialise vector with given values std::vector<int> a = {8, 9, 10, 11}; std::cout << a.at(0); // outputs 8 a.at(3) = 5; // a is 8, 9, 10, 5</pre>	

Datentypen

Vektoren (mehrdim.)	mehrdimensionale “Massenvariable” eines bestimmten Typs
<p>Erfordert: <code>#include<vector></code></p> <p>Wichtige Befehle:</p> <p>Definition: <code>std::vector<std::vector<int> ></code> <code>my_vec (n_rows, std::vector<int>(n_cols, init_value))</code></p> <p>Zugriff: <code>my_arr.at(1).at(1) = 8 * my_arr.at(0).at(2);</code> (Anstatt <code>int</code> gehen natürlich auch andere Typen.)</p>	
<pre>std::vector<std::vector<int> > my_vec (2, std::vector<int>(4, 0)); my_vec.at(1).at(2) = 3; // my_vec becomes // 0, 0, 0, 0 // 0, 0, 3, 0</pre>	

Typ-Alias	Abkürzung eines Typnamens
<p>Typnamen können sehr lang werden. Dann hilft die Deklaration eines Typ-Alias:</p> <p>Syntax: <code>using Name = Typ;</code></p>	
<pre>#include <iostream> #include <vector> using imatrix = std::vector<std::vector<int>>; // POST: Matrix 'm' was output to standard output void print(const imatrix& m); int main() { imatrix m = ...; print(m); }</pre>	

Programmier-Befehle - Woche 8

<code>std::string</code>	komfortablerer Datentyp für Zeichen										
<p>Erfordert: <code>#include<string></code></p> <p>Vorteile:</p> <table><tr><td>variable Länge:</td><td><code>std::string my_str (n, 'a');</code> (n kann variabel sein)</td></tr><tr><td>Länge abfragen:</td><td><code>my_str.length()</code></td></tr><tr><td>vergleichbar:</td><td><code>text1 == text2</code></td></tr><tr><td>hintereinander hängen:</td><td><code>text1 += text2</code></td></tr><tr><td>bequemer Output:</td><td><code>std::cout << my_str;</code></td></tr></table>		variable Länge:	<code>std::string my_str (n, 'a');</code> (n kann variabel sein)	Länge abfragen:	<code>my_str.length()</code>	vergleichbar:	<code>text1 == text2</code>	hintereinander hängen:	<code>text1 += text2</code>	bequemer Output:	<code>std::cout << my_str;</code>
variable Länge:	<code>std::string my_str (n, 'a');</code> (n kann variabel sein)										
Länge abfragen:	<code>my_str.length()</code>										
vergleichbar:	<code>text1 == text2</code>										
hintereinander hängen:	<code>text1 += text2</code>										
bequemer Output:	<code>std::cout << my_str;</code>										
<pre>std::string my_word (5, 'a'); // initialize my_word as aaaaa std::string ref (5, 'z'); my_word += ref; // append ref to my_word. // Afterwards my_word: aaaaazzzzz // Afterwards ref: zzzzz std::cout << my_word.length() << "\n"; // output: 10 my_word.at(3) = 'b'; // change my_word to aaabazzzzz if (my_word == ref) { // false std::cout << "not output\n"; } std::cout << my_word << "\n"; // output whole string at once</pre>											

Funktionen

Rekursion	Selbstaufruf einer Funktion
<p>Jeder rekursive Funktionsaufruf hat seine eigenen, unabhängigen Variablen und Argumente. Dies kann man sich sehr gut anhand des in der Vorlesung gezeigten Stacks vorstellen (<code>fac</code> ist im Beispiel unten definiert):</p> <pre>graph TD n1["n = 1 1! = 1"] n2["n = 2 2 · 1! = 2"] n3["n = 3 3 · 2! = 6"] n4["n = 4 4 · 3! = 24"] main["std::cout << fac(4)"] n1 -- "fac(1)" --> n2 n2 -- "2" --> n3 n3 -- "6" --> n4 n4 -- "24" --> main</pre>	
<pre>// POST: return value is n! unsigned int fac (const unsigned int n) { if (n <= 1) return 1; return n * fac(n-1); // n > 1 }</pre>	

Datentypen

struct	Container für Datentypen
<p>Wichtige Befehle:</p> <p>Definition: <pre>struct str_name { int mem1; bool mem2; int mem3; };</pre></p> <p>Objekt erstellen: <pre>str_name obj1;</pre></p> <p>mit Startwerten: <pre>str_name obj2 = {3, true, 4};</pre></p> <p>aus anderem Objekt: <pre>str_name obj3 = obj2;</pre></p> <p>Zugriff auf Member: <pre>obj1.mem1</pre></p> <p>Die <i>Definition</i> eines Structs hat ein <code>;</code> am Schluss.</p> <p>Nur der Zuweisungsoperator (=) wird automatisch erstellt (und kopiert dann die Member einzeln). Die anderen Operatoren (z.B. ==, !=, ...) muss man selbst passend überladen (siehe Eintrag operator...).</p> <p>Bei der Default-Initialisierung eines Objekts des Typs <code>str_name</code> werden alle Member einzeln default-initialisiert. Für fundamentale Typen (<code>int</code>, <code>float</code>, usw.) bedeutet das, dass sie <i>uninitialisiert</i> sind, bis man ihnen nachträglich einen Wert zuweist. Das führt zu Problemen, falls man ihren Wert vorher schon ausliest.</p>	
<pre>struct candidate { std::string name; // Name of the participant int age; // Her/his age }; int main () { // Initialization candidate mary; // default-initialisation std::cout << mary.age; // Undefined behavior mary.name = "Mary"; mary.age = 43; std::cout << mary.age; // Problem gone: mary.age is 43 candidate bob = {"Bob", 28}; // using starting values candidate fred = bob; // using other object fred.name = "Fred"; return 0; }</pre>	

Operatoren

<code>operator...</code>	Einen Operator überladen.
<p>Operator-Überladung wird zum Beispiel verwendet, um Operatoren (+, -, *, etc.) auf eigenen Structs zu definieren.</p> <p>Mittels dem <code>operator...</code> Keyword ist es ebenfalls möglich, den Operator auszuführen. Das sollte man aber vermeiden, da damit der Code unlesbar wird.</p>	
<pre>struct rational { int n; int d; // INV: d != 0 }; // POST: return value is the sum of a and b rational operator+ (const rational a, const rational b) { rational result; result.n = a.n * b.d + a.d * b.n; result.d = a.d * b.d; return result; } // POST: return value is the sum of a and b rational operator+ (const rational a, const int b) { rational b_rat; b_rat.n = b; b_rat.d = 1; // b_rat is b/1 return a + b_rat; // Use operator+ for two rationals (above) } int main () { rational r = {1, 2}; rational s = {3, 4}; rational t = r + s; // first overload std::cout << t.n << "/" << t.d << "\n"; // Output: 10/8 rational u = r + 3; // second overload std::cout << u.n << "/" << u.d << "\n"; // Output: 7/2 return 0; }</pre>	

Datentypen

<code>class</code>	Datencontainer mit Kapselung
<p>Eine Klasse besteht aus Daten und Funktionen, genannt Member, und erlaubt deren Kapselung via Zugriffskontrolle: Auf Member im privaten Teil (<code>private</code>) einer Klasse kann nur durch die Klasse selbst, d.h., deren Member-Funktionen zugegriffen werden.</p> <p>Zugriff von ausserhalb der Klasse muss über öffentliche (<code>public</code>) Member erfolgen. Per default sind die Member einer Klasse privat.</p> <p>Einziger Unterschied gegenüber <code>structs</code>: Member in <code>structs</code> sind per default öffentlich (<code>public</code>).</p> <p>Deklarationsreihenfolge von Membern ist irrelevant.</p>	
<pre>class my_class { public: // public section double some_public_member; private: // private section double some_private_member; }; ... my_class inst; inst.some_public_member = 1.0; inst.some_private_member = 0.0; // ERROR: cannot access private // members directly</pre>	

Memberfunktion	Funktionalität auf Klassen
<p>Memberfunktionen stellen Funktionalität auf einer Klasse bereit. Sie ermöglichen den kontrollierten Zugang zu den privaten Daten und privaten Memberfunktionen. Die <i>Deklaration</i> einer Memberfunktion erfolgt immer in der Klassendefinition, die <i>Definition</i> der Memberfunktion ist auch extern möglich (ermöglicht vorkompilierte Libraries). Dann muss allerdings die Zugehörigkeit zur Klasse explizit erwähnt werden mittels der ::-Schreibweise.</p> <p>Der Aufruf einer Memberfunktion ist <code>obj.mem_func(arg1, arg2, ..., argN)</code>. Der Teil <code>obj.</code> kann weggelassen werden, falls aus der Class heraus auf einen Member des aufrufenden Objekts (siehe Eintrag *this) zugegriffen wird.</p>	
<pre>// Internal Definition vs. External Definition class Insurance { public: void set_rate_i (const double v) { rate = v; } // int. void set_rate_e (const double v); ... private: double rate; ... }; void Insurance::set_rate_e (const double v) {rate = v;} // ext. ----- // Call from Inside vs. Call from Outside class Insurance { public: double get_rate () { if (!is_up_to_date) update_rate(); // from inside return rate; } double get_cost () {return get_rate() * ...;} // from inside ... // e.g. stuff which sets the data members private: bool is_up_to_date; double rate; double update_rate () { rate = ...; } }; ... Insurance insurance; ... std::cout << insurance.get_rate(); // from outside</pre>	

<code>const</code> Memberfunktion	Unverändernde Memberfunktion
<p>Das <code>const</code> bezieht sich auf <code>*this</code>. Es verspricht, dass durch die Funktionsausführung das implizite Argument nicht im Wert verändert wird.</p>	
<pre>class Insurance { public: double get_value() const { return value; // same: return (*this).value; } ... // e.g. members which set the data members private: double value; };</pre>	

Konstruktor	Datencontainer Initialisierung
<p>Konstruktoren sind spezielle Memberfunktionen einer Klasse, die den Namen der Klasse tragen. Sie werden bei der Variablendeklaration aufgerufen.</p> <p>Sie werden analog zu Funktionen überladen und bei der Variablendeklaration wie eine Funktion aufgerufen. Damit das funktioniert, muss der Konstruktor öffentlich (<code>public</code>) sein.</p> <p>Spezielle Konstruktoren sind der Default-Konstruktor (kein Argument), welcher automatisch erzeugt wird, falls eine Klasse keinen Konstruktor definiert, und der Konversions-Konstruktor (genau ein Argument), welcher die Definition benutzerdefinierter Konversionen ermöglicht.</p>	

(...)

(...)

```
class Insurance {
public:
    Insurance(double v, int r) // general constructor
        : value (v), rate (r) // initialize data members
        { update_rate(); }
    Insurance()                // default constructor
        : value (0), rate (0) // initialize data members
        { }
    // other members
private:
    double value;
    double rate;
    void update_rate();
};
...
// General Constructor
Insurance i1 (10000, 10);
// default-Constructor, direct call
Insurance i3; // identical:    Insurance i3 = Insurance();
...
-----
class Complex {
public:
    // Conversion Constructor (float --> Complex)
    Complex(const float i) : real (i), imag (0) { }
private:
    float real;
    float imag;
};
```

Iteratoren

Iterator (auf Vektor)	Iterieren über einen Vektor.
<p>Erfordert: <code>#include<vector></code></p> <p>Wichtige Befehle (gelte <code>std::vector<int> a (6, 0);</code>):</p> <p>Definition: <code>std::vector<int>::iterator itr = ...;</code></p> <p>Iterator auf a.at(0): <code>a.begin()</code></p> <p>Past-the-End-Iterator: <code>a.end()</code></p> <p>Zugriff auf Iterator: <code>itr = otr_itr // Iterator gets new target.</code></p> <p>Zugriff auf Target: <code>*itr = 5 // Target gets new value 5.</code></p> <p>Vergleich: <code>itr == otr_itr // Same target?</code> <code>itr != otr_itr // Different targets?</code></p> <p>Anstelle des <code>...</code> in der Definition eines Iterators müssen andere Iteratoren stehen (z.B. <code>a.begin()</code>).</p> <p>Um lange Zeilen zu vermeiden, siehe Eintrag Typ-Alias.</p> <p>Der <code>*</code> Operator, um auf das Targets eines Iterators zuzugreifen, wird auch Dereferenz-Operator genannt.</p>	
<pre>// Example for vectors. // Read 6 values into a vector std::cout << "Enter 6 numbers:\n"; std::vector<int> a (6, 0); for (std::vector<int>::iterator i = a.begin(); i < a.end(); ++i) std::cin >> *i; // read into object of iterator // Output: a.at(0)+a.at(3), a.at(1)+a.at(4), a.at(2)+a.at(5) for (std::vector<int>::iterator i = a.begin(); i < a.begin()+3; ++i) { assert(i+3 < a.end()); // Assert that i+3 stays inside. std::cout << (*i + *(i+3)) << ", "; }</pre>	

<code>const</code> (Iterator)	kein Schreibzugriff auf das Objekt
<p>Vorsicht: Einen <code>const</code>-Iterator erzeugt man mittels <code>std::vector<int>::const_iterator ...</code> und nicht mittels <code>const std::vector<int>::iterator ...</code></p> <p>Die zweite Version erzeugt einen Iterator, den man nicht herumschieben kann. In dieser Vorlesung gehen wir aber nur auf die Iteratoren näher ein, welche den Schreibzugriff auf <i>das Objekt</i> verbieten (erste Variante oben).</p>	
<pre>std::vector<int> a (6, -8); // a is: -8 -8 -8 -8 -8 -8 std::vector<int>::const_iterator itr = a.begin() + 3; *itr = 4; // NOT valid itr = a.begin(); // valid (itr now points to a.at(0))</pre>	

Bereichsbasierte for-Schleife	
<p>Sequenzielle Iteration mittels eines Iterators über einen <code>std::vector<int></code> (const-Iterator möglich; andere Container möglich):</p> <pre>std::vector<int> v(3); // v == 0, 0, 0 for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) { std::cout << *it; // 000 }</pre> <p>Kann alternativ auch wie folgt geschrieben werden:</p> <pre>for (int i : v) std::cout << i; // 000</pre> <p>Wird dann zu Iterator-basierter Schleife übersetzt.</p> <p>Modifizierender Zugriff ist auch möglich:</p> <pre>for (int& i : v) i += 3; for (int i : v) std::cout << i; // 333</pre>	

Datentypen

set	Datentyp für Mengen (jedes Element kommt nur einmal vor).
<p>Erfordert: <code>#include<set></code></p> <p>Wichtige Befehle (Sei <code>b = some_vec.begin(); e = some_vec.end();</code>):</p> <p>Definition: <code>std::set<int> my_set (b, e);</code> (Initialisiert <code>my_set</code> mit den Werten im Bereich <code>[b,e)</code>.)</p> <p>Die Iteratoren der sets funktionieren wie die Iteratoren der Vektoren, aber:</p> <p>Keine: <code>[], +, -, <, >, <=, >=, +=, -=</code> Zum Verschieben nur: <code>++..., ...++, --..., ...--</code>, <code>=</code> Zum Vergleichen nur: <code>==, !=</code></p>	
<pre>// Determine All Occurring Numbers std::cout << "Enter 100 numbers:\n"; std::vector<int> nbrs (100); for (int i = 0; i < 100; ++i) std::cin >> nbrs.at(i); std::set<int> uniques (nbrs.begin(), nbrs.end()); // Output using Sit = std::set<int>::iterator; for (Sit i = uniques.begin(); i != uniques.end(); ++i) std::cout << *i << " "; // This does not work: for (int i = 0; i < uniques.end() - uniques.begin(); ++i) std::cout << uniques.at(i);</pre>	

Standard-Funktionen

<code>std::fill(b, p, val)</code>	Wert <code>val</code> in einen Bereich <code>[b,p)</code> einlesen
Erfordert: <code>#include<algorithm></code>	
<pre>// Goal: Generate vector: 4 4 4 2 2 std::vector<int> vec (5, 4); // vec: 4 4 4 4 4 std::fill(vec.begin()+3, vec.end(), 2); // vec: 4 4 4 2 2</pre>	

<code>std::find(b, p, val)</code>	<code>val</code> suchen im Bereich <code>[b,p)</code>
Erfordert: <code>#include<algorithm></code>	
Zurückgegeben wird ein Iterator auf das <i>erste</i> gefundene Vorkommnis.	
Wenn <code>std::find</code> nicht fündig wird, gibt es den Past-the-End-Iterator <code>p</code> zurück. (Beachte: Past-the-End ist bezüglich Bereich <code>[b,p)</code> gemeint.)	
<pre>using Vit = std::vector<int>::iterator; std::vector<int> vec (5, 2); vec.at(3) = -7 // Goal: Find index of -7 in vec: 2 2 2 -7 2 Vit pos_itr = std::find(vec.begin(), vec.end(), -7); std::cout << (pos_itr - vec.begin()) << "\n"; // Output: 3</pre>	

<code>std::sort(b, e)</code>	Bereich <code>[b, e)</code> sortieren
Erfordert: <code>#include<algorithm></code>	
<code>std::sort</code> funktioniert nur, wenn Random-Access Iteratoren für <code>b</code> und <code>e</code> übergeben werden. Somit funktioniert <code>std::sort</code> z.B. für Felder und Vektoren, aber nicht z.B. für Sets.	

(...)

Programmier-Befehle - Woche 10

(...)

```
std::vector<int> vec = {8, 1, 0, -7, 7};  
std::sort(vec.begin(), vec.end()); // vec: -7 0 1 7 8
```

`std::min_element(b, p)`

Iterator auf Minimum im Bereich [b,p)

Erfordert: `#include<algorithm>`

Wenn das Minimum nicht eindeutig ist, so wird ein Iterator auf das erste Vorkommen zurückgegeben.

```
// Goal: Make sure that all inputs are > 0  
std::vector<int> vec (10, 0);  
for (int i = 0; i < 10; ++i)  
    std::cin >> vec.at(i);  
  
assert( *std::min_element(vec.begin(), vec.end()) > 0 );  
    // Note: We have to dereference the (r-value-)iterator.
```

Zeiger

Zeiger (generell)	Adresse eines Objekts im Speicher								
<p>Wichtige Befehle:</p> <p>Definition: <code>int* ptr = address_of_type_int;</code> (ohne Startwert: <code>int* ptr = nullptr;</code>)</p> <p>Zugriff auf Zeiger: <code>ptr = otr_ptr // Pointer gets new target.</code></p> <p>Zugriff auf Target: <code>*ptr = 5 // Target gets new value 5.</code></p> <p>Adresse auslesen: <code>int* ptr_to_a = &a; // (a is int-variable)</code></p> <p>Vergleich: <code>ptr == otr_ptr // Same target?</code> <code>ptr != otr_ptr // Different targets?</code></p> <p>(Anstatt <code>int</code> gehen natürlich auch andere Typen.) (Eine <code>address_of_type_int</code> kann man durch einen anderen Zeiger oder auch mittels dem Adressoperator <code>&</code> erzeugen (siehe Beispiel unten).)</p> <p>Der Wert des Zeigers ist die Speicheradresse des Targets. Will man also das Target via diesen Zeiger verändern, muss man zuerst "zu der Adresse gehen". Genau das macht der Dereferenz-Operator <code>*</code>.</p> <p>Beispiel: (Gelte <code>int a = 5;</code>)</p> <table><tr><td>Wert von <code>a</code>:</td><td>5</td></tr><tr><td>Speicheradresse von <code>a</code>:</td><td>0x28fef8</td></tr><tr><td>Wert von <code>a_ptr</code>:</td><td>0x28fef8</td></tr><tr><td>Wert von <code>*a_ptr</code>:</td><td>5</td></tr></table> <p>Ein Zeiger kann immer nur auf den entsprechenden Typ zeigen. (z.B. <code>int* ptr = &a;</code> Hier muss <code>a</code> Typ <code>int</code> haben.)</p>		Wert von <code>a</code> :	5	Speicheradresse von <code>a</code> :	0x28fef8	Wert von <code>a_ptr</code> :	0x28fef8	Wert von <code>*a_ptr</code> :	5
Wert von <code>a</code> :	5								
Speicheradresse von <code>a</code> :	0x28fef8								
Wert von <code>a_ptr</code> :	0x28fef8								
Wert von <code>*a_ptr</code> :	5								
<pre>int a = 5; int* a_ptr = &a; // a_ptr points to a a_ptr = a; // NOT valid (same as: a_ptr = 5;) // 5 is NOT an address. a_ptr = &a; // valid *a_ptr = 9; // a obtains value 9 std::cout << "a == " << a << "\n"; // Output: a == 9 std::cout << "a == " << *a_ptr << "\n"; // Output: a == 9</pre>									

<code>const</code> (Zeiger)	Zeiger Konstantheit
Es gibt zwei Arten von Konstantheit: kein Schreibzugriff auf Target: <code>const int* a_ptr = &a;</code> kein Schreibzugriff auf Zeiger: <code>int* const a_ptr = &a;</code>	
<pre>int a = 5; int b = 8; const int* ptr_1 = &a; *ptr_1 = 3; // NOT valid (change target) ptr_1 = &b; // valid (change pointer) int* const ptr_2 = &a; *ptr_2 = 3; // valid (change target) ptr_2 = &b; // NOT valid (change pointer) const int* const ptr_3 = &a; *ptr_3 = 3; // NOT valid (change target) ptr_3 = &b; // NOT valid (change pointer)</pre>	

Programmier-Befehle - Woche 11

<code>*this</code>	Zugriff auf implizites Argument
<p>Memberfunktionen einer Klasse haben ein implizites Argument, nämlich das aufrufende Objekt. Und <code>this</code> ist ein Zeiger darauf. Via <code>*this</code> kann man darauf zugreifen.</p> <p>Bei Zugriffen von innerhalb einer Klasse aus auf Daten-Member oder Member-Funktionen wird das implizite Argument automatisch verwendet. Man muss es dann also nicht unbedingt explizit angeben (siehe Eintrag Memberfunktion). Man muss <code>*this</code> aber mindestens explizit verwenden, falls z.B. eine Referenz auf das implizite Argument zurückgegeben werden soll.</p>	
<pre>// General example class Human { public: void set (const int a) { age = a; } // or (*this).age = a; void print1 () const { std::cout << (*this).age; } void print2 () const { std::cout << age; } // equivalent private: int age; }; ... Human me; me.set(175); me.print1(); // 175 me.print2(); // 175 ----- // Another example class Complex { public: // Note: In most applications // a reference should be returned. Complex& operator+= (const Complex& b) { real += b.real; imag += b.imag; return *this; } ... // other members private: float real; float imag; };</pre>	

Dynamische Datentypen

<code>new</code>	Objekt mit dynamischer Lebensdauer erstellen.
<p>Mit <code>new</code> wird ein Objekt erstellt, indem der nötige Speicherplatz reserviert wird, und dann ein gegebener <code>Konstruktor</code> aufgerufen wird.</p> <p>Der Rückgabewert von <code>new</code> ist ein <code>Pointer</code> auf das neu erstellte Objekt.</p>	
<pre>Class My_Class { public: My_Class (const int i) : y (i) { std::cout << "Hello"; } int get_y () { return y; } private: int y; }; ... My_Class* ptr = new My_Class (3); // outputs Hello My_Class* ptr2 = ptr; // another pointer to the new object std::cout << (*ptr).get_y(); // Output: 3 ...</pre>	

Operatoren

<code>&</code>	Adressoperator (siehe: <i>Adresse auslesen</i> unter <i>Zeiger</i> (generell), Summary 8)
Präcedenz: 16 und Assoziativität: rechts	

<code>*</code>	Dereferenz-Operator (siehe: <i>Zugriff auf Target</i> unter <i>Zeiger</i> (generell))
Präcedenz: 16 und Assoziativität: rechts	

<code>...->...</code>	Auf einen Member eines Objekts zugreifen , auf das ein Pointer gegeben ist.
<code>ptr->mem</code> macht das Selbe wie <code>(*ptr).mem</code> .	
<pre>struct my_class { // POST: "Hi! " is written to std::cout void say_hi () const { std::cout << "Hi! "; } }; ... my_class obj; my_class* ptr = &obj; // just a pointer to obj obj.say_hi(); // direct access ptr->say_hi(); // using -> (*ptr).say_hi(); // equivalent</pre>	

Dynamische Datentypen

<code>new, delete</code>	Objekt mit dynamischer Lebensdauer erstellen.
<p>Mit <code>new</code> wird ein Objekt erstellt, indem der nötige Speicherplatz reserviert wird, und dann ein gegebener <code>Konstruktor</code> aufgerufen wird. Bei <code>delete</code> wird zuerst ein Destruktor aufgerufen, bevor der Speicherplatz freigegeben wird.</p> <p>Der Rückgabewert von <code>new</code> ist ein <code>Pointer</code> auf das neu erstellte Objekt. Wird mit <code>delete</code> ein Objekt gelöscht, so sollte man immer <i>alle</i> <code>Pointer</code>, die auf das Objekt zeigen, auf <code>nullptr</code> setzen.</p> <p>Jedes <code>new</code> braucht ein <code>delete</code>. Sonst existieren die erstellten Objekte bis zum Ende des Programms, was je nach Laufdauer eine grosse Speicherverwendung ist.</p>	
<pre>Class My_Class { public: My_Class (const int i) : y (i) { std::cout << "Hello"; } int get_y () { return y; } private: int y; }; ... My_Class* ptr = new My_Class (3); // outputs Hello My_Class* ptr2 = ptr; // another pointer to the new object std::cout << (*ptr).get_y(); // Output: 3 delete ptr; ptr = nullptr; ptr2 = nullptr; // has to be done !separately! ...</pre>	

Memory Management mit Klassen

Copy-Konstruktor	Kopier-Initialisierung
Der Copy-Konstruktor ist der Konstruktor, dessen Argumenttyp const My_Class& ist.	
<pre>class stack { public: stack(); // constructor stack(const stack& s); // copy constructor private: lnode* topn; }; stack::stack (const stack& s) : topn(nullptr) { if (s.topn == nullptr) return; topn = new lnode(s.topn->value, nullptr); lnode* prev = topn; for (lnode* n = s.topn->next; n != nullptr; n = n->next) { lnode* copy = new lnode(n->value, nullptr); prev->next = copy; prev = copy; } }</pre>	

Destruktor	Klasse abbauen
<p>Der Destruktor einer Klasse ist eine eindeutige Memberfunktion die automatisch aufgerufen wird wenn, die Speicherdauer eines Klassenobjekts endet (z.B. beim Aufrufen von <code>delete</code> oder wenn der Gültigkeitsbereich des Objekts endet.).</p> <p>Falls eine Klasse keinen Destruktor deklariert wird automatisch einer erzeugt. Der automatisch erzeugte Destruktor ruft den Destruktor aller Membervariablen auf.</p>	
<pre>class stack { public: stack(); // constructor stack(const stack& s); // copy constructor ~stack(); // destructor private: lnode* topn; }; stack::~~stack() { while (topn != nullptr) { lnode* t = topn; topn = t->next; delete t; } }</pre>	

Programmier-Befehle - Woche 12

operator=	Kopier-Zuweisung
<p>Eng verwandt mit <code>operator=</code> ist der Copy-Konstruktor. Der Unterschied ist, dass der Copy-Konstruktor nur bei der Initialisierung aufgerufen wird, <code>operator=</code> hingegen nur <i>nach</i> der Initialisierung. z.B.</p> <pre>my_class a(5, 6), c(4, 4); // Call a general constructor my_class b = a; // Call copy-constructor c = b; // Call operator=</pre> <p><code>operator=</code> muss gegebenenfalls anders als der Copy-Konstruktor implementiert werden. Ein Beispiel sind Klassen, welche Pointer auf dynamisch generierte Objekte als Member haben. Dann muss bei <code>operator=</code> meistens zuerst das aktuell vorhandene Objekt gelöscht werden, bevor die Kopie erstellt werden kann. Dies ist beispielsweise beim Stack aus der Vorlesung relevant.</p> <p><code>operator=</code> gibt im Normalfall eine Referenz auf seinen linken Operanden zurück.</p> <p>Faustregel: Meistens führt <code>operator=</code> die Aufgaben des Copy-Konstruktors und Destructors aus.</p>	
<pre>class stack { public: stack(); // constructor stack(const stack& s); // copy constructor ~stack(); // destructor stack& operator=(const stack& s); // assignment operator private: lnode* topn; }; stack& stack::operator= (const stack& s) { if (this != &s) { // no self assignment stack copy = s; // copy constructor std::swap(topn, copy.topn); // copy now contains old topn pointer } // copy goes out of scope. Destructor will be called which clean up everything. }</pre>	

Dreierregel	Rule of Three
Wenn eine Klasse einen Destruktor, Copy-Konstruktor, oder den Kopier-Zuweisungs operator definiert, dann sollte sie alle drei definieren.	

Smart Pointers

<code>std::shared_ptr<T></code>	Shared Pointer
<p>Ein <code>std::shared_ptr<T></code> ist ein Zeiger auf ein Objekt der Klasse T, der sich die Anzahl der Zeiger die auf dieses Objekt zeigen merkt. Sobald die Anzahl der Zeiger auf das Objekt auf 0 fällt wird das Objekt automatisch gelöscht.</p>	
<pre>class SomeClass { public: SomeClass(); void f(); }; // Create a shared pointer std::shared_ptr<SomeClass> s1 = std::make_shared<SomeClass>(); // Shared pointers can be used like regular pointers s1->f(); { // Create another shared pointer pointing to new object std::shared_ptr<SomeClass> s2(new SomeClass()); // Shared pointers can be assigned s1 = s2; // Point s1 to the same object as s2 // The object s1 used to point to will now be deleted. // We can check how many pointers point to the same object std::cout << s1.use_count(); // This prints "2". std::cout << s2.use_count(); // This also prints "2". } // s2 went out of scope. The reference count decreases. std::cout << s1.use_count(); // Now, this prints "1". s1 = nullptr; // Now, 0 pointers point to the object s1 used to point to. // The object is deleted.</pre>	

<code>std::unique_ptr<T></code>	Unique Pointer
<p>Ein <code>std::unique_ptr<T></code> ist der einzige Zeiger auf ein bestimmtes Objekt der Klasse T. Kein anderer Zeiger darf auf dasselbe Object zeigen.</p>	
<pre>class SomeClass { public: SomeClass(); void f(); }; // Create two unique pointers for two different objects std::unique_ptr<SomeClass> s1 = std::make_unique<SomeClass>(); std::unique_ptr<SomeClass> s2(new SomeClass()); // Unique pointers can be used like regular pointers s1->f(); // Unique pointers do not have a copy constructor or a regular assignment operator. std::unique_ptr<SomeClass> s3 = s1; // This causes a compiler error! s2 = s1; // This also causes a compiler error! // Ownership of an object can be moved from one unique pointer to another std::unique_ptr<SomeClass> s3 = std::move(s1); // Now, s1 does not own the object anymore s1->f(); // This causes a runtime error!</pre>	

Datentypen

Array	“Massenvariable” eines bestimmten Typs
<p>Wichtige Befehle:</p> <p>Definition: <code>int my_arr[5] = {2, 3, 8, -1, 3};</code> Zugriff: <code>my_arr[2] = 8 * my_arr[3];</code></p> <p>(Anstatt <code>int</code> gehen natürlich auch andere Typen.) (Die Definition kann auch ohne Initialisierung erfolgen: <code>int my_arr[5];</code>)</p> <p>Wie bei Vektoren beginnen die Indizes bei 0. Allerdings muss der Programmierer bei Arrays selber sicherstellen, dass die Indizes nicht über den Array hinausgehen, weshalb wir von der Verwendung von Arrays abraten und stattdessen Vektoren empfehlen.</p> <p>Zuweisungen (ausser Initialisierung), Vergleiche, etc. müssen elementweise erfolgen. Sie können nicht direkt gemacht werden.</p> <p>Die Länge des Arrays muss zum Kompilierzeitpunkt eindeutig bestimmbar sein. (z.B. Literal oder <code>const</code>-Variable, die mittels Literal eingelesen wurde, etc.)</p>	
<pre>int a[10]; // Accessing an array: for (int i = 0; i < 10; ++i) a[i] = i; // a becomes {0 1 2 ... 9} a[10] = 2; // NOT allowed, index 10 outside a[-4] = 2; // NOT allowed, index -4 outside // Copying an array: int b[10] = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11}; for (int i = 0; i < 10; ++i) a[i] = b[i]; // Have to do it element-wise a = b; // NOT valid: direct array-copying is forbidden</pre>	

Dynamische Datentypen

<code>new ...[], delete[]</code>	Ranges mit dynamischer Lebensdauer und Länge erstellen.
<pre>int n; std::cin >> n; int* range = new int[n]; // Read in values to the range for (int* i = range; i < range + n; ++i) std::cin >> *i; delete range; // ERROR: must say: delete[] delete[] range; // This works</pre>	

Zeiger-Arithmetik

Zeiger	Iterieren										
<p>Wichtige Befehle:</p> <table><tr><td>Zeiger:</td><td><code>int* ptr = new int[6];</code></td></tr><tr><td>temporärer Shift:</td><td><code>ptr + 3</code> <code>ptr - 3</code></td></tr><tr><td>permanentener Shift:</td><td><code>++ptr</code> <code>ptr++</code> <code>--ptr</code> <code>ptr--</code> <code>ptr += 3</code> <code>ptr -= 3</code></td></tr><tr><td>Distanz bestimmen:</td><td><code>ptr1 - ptr2</code></td></tr><tr><td>Position vergleichen:</td><td><code>ptr1 < ptr2</code> (Sonst: <code><=</code>, <code>></code>, <code>>=</code>, <code>==</code>, <code>!=</code>)</td></tr></table> <p>Achtung: Die grünen Shifts erzeugen einen neuen (temporären) Zeiger und verschieben <code>ptr</code> nicht. Die violetten Shifts verschieben aber <code>ptr</code>.</p>		Zeiger:	<code>int* ptr = new int[6];</code>	temporärer Shift:	<code>ptr + 3</code> <code>ptr - 3</code>	permanentener Shift:	<code>++ptr</code> <code>ptr++</code> <code>--ptr</code> <code>ptr--</code> <code>ptr += 3</code> <code>ptr -= 3</code>	Distanz bestimmen:	<code>ptr1 - ptr2</code>	Position vergleichen:	<code>ptr1 < ptr2</code> (Sonst: <code><=</code> , <code>></code> , <code>>=</code> , <code>==</code> , <code>!=</code>)
Zeiger:	<code>int* ptr = new int[6];</code>										
temporärer Shift:	<code>ptr + 3</code> <code>ptr - 3</code>										
permanentener Shift:	<code>++ptr</code> <code>ptr++</code> <code>--ptr</code> <code>ptr--</code> <code>ptr += 3</code> <code>ptr -= 3</code>										
Distanz bestimmen:	<code>ptr1 - ptr2</code>										
Position vergleichen:	<code>ptr1 < ptr2</code> (Sonst: <code><=</code> , <code>></code> , <code>>=</code> , <code>==</code> , <code>!=</code>)										
<pre>// Read 6 values into an array std::cout << "Enter 6 numbers:\n"; int* a = new int[6]; int* pTE = a+6; for (int* i = a; i < pTE; ++i) std::cin >> *i; // read into array element // Output: a[0]+a[3], a[1]+a[4], a[2]+a[5] for (int* i = a; i < a+3; ++i) { assert(i+3 < pTE); // Assert that i+3 stays inside. std::cout << (*i + *(i+3)) << ", "; }</pre>											