C++-Kurs

Stephan Neuhaus

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken







Organisatorisches

Übungsbetrieb durch Übungsaufgaben und Betreuung in den CIP-Pools

Bremser: Sebastian Waßmuth, Stefan Lorenz, Thomas Leichtweis und Christoph Klein

Zeit/Tag	Mo	Di	Mi	Do	Fr
9–11	Vorlesung				Chr + Seb
11–13		Chr + Seb	Th + St		
16–18	Th + St	Vorlesung		Vorlesung	

Sie können sich zwei von drei Vorlesungsterminen aussuchen!

Das erste Programm

```
#include <iostream>

int main(int argc, const char *argv[]) {
  int sum, a, b;
  // Read integers a and b
  std::cout << "Please enter a and b: ";
  std::cin >> a >> b;
  /* Silly comment: Calculate sum of a and b */
  sum = a+b;
  // Silly comment: Output result
  std::cout << "Sum is " << sum << std::endl;
  return 0;
}
```

C++/Java: Groß- und Kleinschreibung werden unterschieden!

PaSCaL/FoRTrAn: gRoß OdEr klEIn iSt EGaL!







Übersetzen und ausführen

Unter Linux mit dem GNU C++-Compiler

```
% g++ -O -Wall -o Firstprog Firstprog.cc
```

Und die Ausführung:

```
% ./Firstprog
Please enter a and b: 3 4
Sum is 7
```







Übersetzungsablauf

1. C++-Präprozessor löst Präprozessoranweisungen auf (`#include` oder `#define`)
2. Compiler übersetzt „reinen“ C++-Code in Assemblertext
3. Assembler kompiliert Assemblertext in Maschinencode
4. Linker linkt Maschinencode plus Bibliotheken zu ausführbarem Programm

All diese Ablaufschritte sind über Kommandozeilenparameter steuerbar

Details später; wichtig bloß, daß es sich um einen mehrschrittigen Prozeß handelt







Datentypen und Operatoren

Datentypen bestehen aus der Menge der möglichen Werte sowie die mit diesen Werten möglichen Operationen.

Beispiel: Datentyp `int` besteht aus einer (implementationsabhängigen) Menge von ganzen Zahlen und den darauf (implementationsabhängig) definierten Operationen +, -, *, % usw.







Boolesche Werte

Ein boolescher Wert (vom Typ `bool`) kann einen der beiden Werte `true` oder `false` haben

Boolesche Werte werden verwendet, um die Ergebnisse logischer Operationen darzustellen:

```
void f(int a, int b) {
  bool b = a == b;  // = is assignment, == is comparison
}
```

Wenn man boolesche Werte in ganze Zahlen konvertiert, bekommt `true` den Wert 1 und `false` den Wert 0

Umgekehrt ist jede von 0 verschiedene Ganzzahl äquivalent zu `true` und 0 ist äquivalent zu `false`

```
bool b = 7;   // 7 is not 0, therefore true
int i = true; // true becomes 1
```







Zeichen

Eine Variable vom Typ char faßt ein Zeichen aus dem Zeichensatz der Implementation:

```
char c = 'a';   // The (implementation-defined) representation of a
char d = '\t';  // A horizontal tab
char e = '\n';  // A newline
char f = '\0';  // The null character
```

Fast überall hat ein char acht bits (256 verschiedene Werte)

In der Regel ist der Zeichensatz eine Untermenge von ISO-646 (z.B. ASCII)

Der Standard garantiert mindestens 256 verschiedene Werte für char







Zeichen

Zeichen sind ganze Zahlen, also umwandelbar in Ganzzahlen

```
char c = 'A';
std::cout << int(c);  // Output is implementation-defined!
char c = 65;
std::cout << c;       // Output is implementation-defined!
std::cout << int(c);  // Output is "65"
```

Ist ein `char` vorzeichenbehaftet oder nicht? „`signed char`“ und „`unsigned char`“ als Varianten

Manchmal findet man in C++-Büchern:

```
char c = 'A';  // c has the value 65
```

Das ist *falsch*. Machen Sie *nie* Annahmen über die Repräsentation von Objekten, die nicht vom Sprachstandard gedeckt werden!







Ganze Zahlen

Ganze Zahlen kommen in C++ in verschiedenen Größen: `short int`, `int` und `long int` (Suffix L) und in zwei verschiedenen Varianten: `signed` und `unsigned` (Suffix U).

Abkürzungen „`unsigned`“ für „`unsigned int`“, „`long`“ für „`long int`“, „`unsigned long`“ für „`unsigned long int`“ usw.

Ganzzahlkonstanten gibt es in vier Varianten

Variante	Präfix	Beispiel
Dezimal	—	2, 63, 83
Oktal	0	02, 077, 0123
Hexadezimal	0x	0x2, 0x3f, 0x53

Die vierte Variante ist eine Zeichenkonstante

Welchen Wert hat `0xffff`?

Kommazahlen (Gleitpunktzahlen)

Gleitpunktzahlen gibt es in drei Größen: `float`, `double` und `long double` (genaue Bedeutung implementationsabhängig)

„Choosing the right precision for a problem where the choice matters requires significant understanding of floating-point computation. If you don't have that understanding, get advice, take the time to learn, or use `double` and hope for the best." (Stroustrup)

Literal	Typ	Literal	Typ	Literal	Typ
1.23	`double`	1.23F	`float`	1.23L	`long double`

Wertebereiche und Größen

Größen von Objekten in C++ sind grundsätzlich immer ein Vielfaches der Größe eines `char`: Definitionsgemäß ist die Größe eines `char` gleich 1.

Der Standard garantiert (und *nicht mehr als das*):

$1 = \texttt{sizeof(char)} \leq \texttt{sizeof(short)} \leq \texttt{sizeof(int)} \leq \texttt{sizeof(long)}$

$1 \leq \texttt{sizeof(bool)} \leq \texttt{sizeof(long)}$

$\texttt{sizeof(char)} \leq \texttt{sizeof(long)}$

$\texttt{sizeof(float)} \leq \texttt{sizeof(double)} \leq \texttt{sizeof(long double)}$

$\texttt{sizeof(N)} = \texttt{sizeof(signed N)} = \texttt{sizeof(unsigned N)}$

$\texttt{bits(char)} \geq 8, \texttt{bits(short)} \geq 16, \texttt{bits(long)} \geq 32$

Ausdrücke

Ausdrücke bestehen aus Operanden, die durch Operatoren verknüpft sind

```
3 + 4         // Expression of type int
'c' + 1       // Expression of type char
~(1UL << 31) // Expression of type unsigned long int
```

Ein Ausdruck hat einen *Wert* und einen *Typ*. Der Typ richtet sich nach dem Typ der Operanden und nach dem Operator.

Wir werden Ausdrücke nicht *en detail* besprechen, sondern stattdessen die offensichtliche Semantik annehmen und in Sonderfällen bestimmte Operatoren besonders besprechen







Ausdrücke für ganze Zahlen

Operator	Beispiel	Bedeutung
arithmetische Operatoren:		
+	+i	unäres Plus
-	-i	unäres Minus
++	++i i++	vor/nachherige Inkrementierung
--	--i i--	vor/nachherige Dekrementierung
+	i+2	binäres Plus
-	i-5	binäres Minus
*	5*i	Multiplikation
/	i/6	Division
%	i % 4	Modulo
=	i = 3+j	Zuweisung

Namen und Deklarationen

Namen (identifier) sind Folgen von Buchstaben und Ziffern, die mit einem Buchstaben beginnen müssen. (Buchstaben sind hier alle Groß- und Kleinbuchstaben und der Unterstrich „_“.)

Einige Zeichenfolgen sind reserviert und können nicht verwendet werden

```
hello  this_is_a_most_unusually_long_name
foo     Foo    fOo      CLASS   _class ___
```

Hier einige Zeichenfolgen, die keine Identifier sind:

```
012     a fool  $sys    $sat
class   foo-bar .name   if
```







Deklarationen und Sichtbarkeit

Eine Deklaration führt einen Namen in einen Sichtbarkeitsbereich ein. Ein Name kann nur nach einer vorherigen Deklaration verwendet werden.

Ein Name heißt *global*, wenn er außerhalb einer Funktion, Klasse oder eines Namespaces (siehe Hauptvorlesung) definiert ist

Sichtbarkeit von globalen Namen ist vom Punkt der Deklaration bis zum Dateiende

Die Deklaration eines Namens in einem Block kann globale Deklarationen ausblenden

Am Blockende erhält ein Name seine frühere Bedeutung







Deklarationen

```
int x;      // Global x
void f() {
  i++;      // Error: name i not visible here

  float x;  // Local x hides global x
  {
    x = 2.0F; // Assign to first local x
    double x; // Second local x hides first local x
    x = 3.0;  // Assign to second local x
    ::x = 3;  // Assign to global x
  }
  x = 4.0F;   // Assign to first local x
  int i = 3;  // Name i introduced here
              // scope of i and first local x ends just befor
              // the closing brace
}
```

Deklarationen

Ein Name ist sichtbar, direkt nachdem er deklariert wurde (aber vor der Initialisierung)

```
int x = 11;

void f() {
  int z = z;  // Perverse: initialize z with its own
              //   (uninitialized) value
  int y = x;  // Use global x without :: operator: y == 11
  int x = 22; // Shadow global x
  y = x;      // Use local x: y == 22
  y = ::x;    // Use global x: y == 11
}
```







Initialisierungen

Explizite Initialisierung durch Angabe eines Initializers

```
int x = 2;  // "= 2" is the initializer for x
```

Globale Objekte werden automatisch mit einer passenden 0 initialisiert

```
int a;     // Same as "int a = 0;"
double d;  // Same as "double d = 0.0"
```

Lokale oder dynamische Objekte werden nicht per Default initialisiert

```
void f() {
  int x;  // x does not have a well-defined value
}
```







Organisatorisches

Zeit/Tag	Mo	Di	Mi	Do	Fr
9–11	Vorlesung				Chr + Seb
11–13		Chr + Seb	Th + St		
16–18	Th + St	Vorlesung			

Nutzen Sie das Angebot, besuchen Sie die Übungen in den CIP-Pools!

http://www.st.cs.uni-sb.de/edu/einst/c++-uebungen.php

http://www.st.cs.uni-sb.de/edu/einst/c++-folien.pdf

Übungen werden ständig aktualisiert

Praktomat-Aufgabe am Ende der nächsten Woche, bitte beachten Sie sie Ankündigungen im Forum!

Aufzählungen (Enumerations)

Aufzählungstypen enthalten eine endliche Liste fester Werte

```
enum Color { RED, GREEN, BLUE };
Color c = RED;
Color d = VIOLET;  // Error: VIOLET not declared
```

Standard garantiert: Aufzählungen sind kompatibel mit `int` und fangen bei 0 an, also hat RED den Wert 0, GREEN den Wert 1 und BLUE den Wert 2.

Ein Aufzählungswert kann explizit initialisiert werden:

```
enum Color { RED = 1, GREEN = 2, BLUE = 4 };
```







Wertebereich einer Aufzählung

Wertebereich eines Aufzählungstyps hält alle Werte der Aufzählung, aufgerundet bis zur nächstgrößeren Zweierpotenz, minus 1. Untere Grenze ist 0, falls der kleinste Enumerator nicht-negativ ist, sonst negative Zweierpotenz

```
enum e1 { dark, light };              // Range: 0:1
enum e2 { a = 3, b = 9 };             // Range: 0:15
enum e3 { min = -10, max = 1000000 }; // Range: -1048576:1048575
```







Konvertierung von und nach Ganzzahlen

Ganzzahlen können explizit nach Aufzählungstypen konvertiert werden:

```
enum flag { x = 1, y = 2, z = 4, e = 8 } // Range: 0:15

flag f1 = 5;          // Error: 5 not of type flag
flag f2 = flag(5);    // OK: flag(5) of type flag and in range
flag f3 = flag(z|e); // OK: flag(12) of type flag and in range
flag f4 = flag(99);  // Undefined: 99 not in range
```

Das unterscheidet sich erheblich von Pascal und Konsorten!







Objekte und Lvalues

Wir können Objekte erzeugen, die keine Namen haben (z.B. Stringkonstanten) und merkwürdigen Ausdrücken etwas zuweisen

Daher brauchen wir einen Begriff für „irgendwas im Speicher“ (Das ist der einfachste und fundamentalste Befriff für „Objekt“)

Ein *Objekt* ist ein zusammenhängender Speicherbereich und ein *Lvalue* ist ein Ausdruck, der ein Objekt bezeichnet.

Ursprung von Lvalue: „Something that can appear on the left side of an assignment“

Z.B. ist `"abc"` ein konstanter Lvalue, der eine Zeichenkette bezeichnet. Da er konstant ist, kann man ihm aber nichts zuweisen, sonst gäbe es Ausdrücke wie `"abc" = "xyz";`

Pointer

Wenn „T“ ein Datentyp ist, ist „T*“ der Typ „Zeiger auf T“, d.h. eine Variable von Typ „T*“ kann die Adresse eines Objekts von Typ „T“ speichern

```
char c = 'a';
char *pc = &a;  // p points to c
```

Man sagt dann „p zeigt auf c“. Graphisch:

Die fundamentale Operation, die man mit einem Pointer machen kann ist, ihn zu *dereferenzieren*. Der so entstehende Ausdruck hat als Wert das Objekt, auf den der Zeiger zeigt

```
char c = 'a'; char *pc = &a;
char d = *pc;   // Dereferencing pc, d == 'a'
```

Null

Null (0) ist vom Typ `int`, kann aber auch als Konstante für jeden Ganzzahl- oder Gleitkommatyp verwendet werden

Kein gültiges Objekt hat die Adresse 0, daher kann 0 auch als Pointerkonstante verwendet werden, um einen Pointer zu bezeichnen, der auf kein Objekt zeigt

In C wird der Null-Zeiger als Makro NULL definiert, um den Null-Zeiger zu bezeichnen; wegen der strengeren Typprüfung von C++ führt aber die Verwendung des Literals 0 zu weniger Problemen

Wenn man unbedingt NULL definieren will:

```
const int NULL = 0;
```

Arrays

Wenn „T“ ein Datentyp ist, ist „T[n]“ der Typ „Array von n Elementen vom Typ T“. Die Elemente werden von 0 bis $n-1$ indiziert

```
float v[3];   // Array of three floats v[0], v[1], v[2]
char *a[32];  // Array of 32 pointers to char a[0]...a[31]
```

Die Anzahl der Elemente in einem Array, die *Arraygrenze*, muß ein konstanter Ausdruck sein. Bei Variablen Arraygrenzen sollte man besser einen `vector` nehmen (später)

```
void f(int i) {
  int v1[i];          // Error: array size (i) not constant
  vector<int> v2(i); // OK
}
```







Array-Initialisierungen

Arrays werden mit einer Werteliste initialisiert

```
char a[2] = { 'a', 'b', 0 };  // Error: too many initializers
char b[3] = { 'a', 'b', 0 };  // OK
int c[8] = { 1, 2, 3, 4 };    // OK, assume 0 for indices 4..7
int d[] = { 1, 2, 3, 4 };     // OK, implicit array size 4
```

Arrays können aber nicht zugewiesen werden

```
char b[2];
b = a;                        // Error: no array assignment
```







String-Konstanten

Eine String-Konstante ist eine Zeichenkette, die in doppelten Hochkommata (") eingeschlossen ist und enthält die Zeichen zwischen den Hochkommata plus ein abschließendes Null-Zeichen

```
sizeof("Bohr") == 5
```

Der Typ einer String-Konstanten ist „Array von genügend vielen konstanten char“ (also im obigen Beispiel `const char[5]`)

Arrays von `char` und Pointer auf `char` können mit Zeichenketten-Konstanten initialisiert werden

```
char *p = "Bohr";      // OK, but (silently) discards const
p[1] = 'e';            // Error: assignment to const
char a[] = "Bohr";     // OK, copies characters
a[1] = 'e';            // OK
```



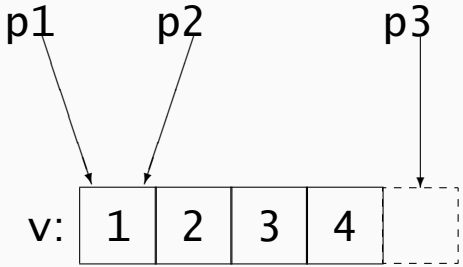




Pointer auf Array-Elemente

Der Name eines Arrays kann als Pointer auf das erste Element verwendet werden

```
int v[] = { 1, 2, 3, 4, }; // Comma OK after last element
int* p1 = v;               // Pointer to first element (implicit)
int* p2 = &v[0];           // Pointer to first element (explicit)
int* p3 = &v[4];           // Pointer to one beyond last element
```



Man darf einen über's Ende hinaus gehen, aber nicht weiter, und man darf diesen Pointer auch nicht dereferenzieren (man braucht ihn für Vergleiche, dazu später)

Pointerarithmetik

Wenn p und q zwei Pointer sind, die auf Elemente eines Arrays vom Typ T zeigen, dann ist der Wert des Audruck q - p die Anzahl der Elemente zwischen den beiden

Wert positiv: q zeigt hinter p, Wert negativ: p zeigt hinter q

Addition von p und einer Ganzzahl n: Zählt von p aus n Elemente weiter

```
int v1[10];
int v2[10];
int i1 = &v1[5] - &v1[3]; // i1 == 2
int i1 = &v1[5] - &v2[3]; // undefined, not in same array
int* p1 = v2 + 2;         // p1 == &v2[2], *p1 == v2[2]
int* p2 = v2 - 2;         // p2 undefined
int* p3 = v2 + 10;        // p3 defined, *p3 undefined(!)
int* p4 = v2 + 11;        // p4 undefined
```







Strukturen

Ein Array ist eine Zusammenfassung von Elementen *gleichen* Typs. Eine Struktur ist eine Zusammenfassung von Elementen *unterschiedlichen* Typs

```
struct address {
  char* name;      // "Stephan Neuhaus"
  char* street;    // "Stuhlsatzenhausweg"
  long number;     // 123
  int zip[5];      // { 6, 6, 2, 3, 4 }
  char* city;      // "Saarbrüecken"
};                 // <-- Note semicolon!
```

Diese Deklaration definiert einen neuen Typen namens `address`; Variablen vom Typ `address` werden genau wie andere Variablen deklariert und der Punkt-Operator greift auf die Felder (engl. *members*) zu.







Strukturen

Ist p ein Pointer auf eine Struktur vom Typ T, dann ist `p->member` eine Kurzform für `(*p).member`

```
void f() {
  address a1;
  a1.name = "Stephan Neuhaus"; // Dot operator

  address a2 = {
    "Stephan Neuhaus",
    "Stuhlsatzenhausweg",
    123,
    { 6, 6, 2, 3, 4 },
    "Saarbrücken",              // <-- comma OK here
  };

  address *p = &a2;
  p->number = 234;              // Struct pointer dereference operator
}
```

Beispiel

```
struct group {
  char* name;
  member *my_members;  // Error: member not defined
};

struct member {
  char* name;
  group *my_groups;
};

struct No_good {
  No_good x;           // Error: can't determine size of No_good
};
```







Vorwärtsdeklaration

```
struct member;          // Forward declaration: to be defined later

struct group {
  char* name;
  member *my_members;   // Now OK
  member m1;            // Error, size unknown here
};

struct member {
  char* name;
  group *my_groups;
};

struct No_good {
  No_good *x;           // OK
};
```







Beispiel: Verkettete Liste

Definition: Eine Verkettete int-Liste ist entweder *leer* oder hat ein erstes int-Element, gefolgt von einer verketteten Liste

Das ist eine rekursive Definition

Rekursive Definition: Rekursive Datenstruktur

```
struct IntItem {
  int x;             // Element's value
};

struct List {
  List* next;
  IntItem element;
};
```







Beispiel: Verkettete Liste

```
int main(int argc, const char* argv[])
{
  IntItem a = {1};
  IntItem b = {2};
  IntItem c = {3};
  List myList[3];

  myList[0].next = &myList[1];  myList[0].element = &a;
  myList[1].next = &myList[2];  myList[1].element = &b;
  myList[2].next = 0;  myList[2].element = &c;

  List *head = myList;

  std::cout << head->element->x
    << " " << head->next->element->x
    << " " << head->next->next->element->x
    << std::endl;
  return 0;
}
```







Dynamischer Speicher mit new

Speicher dynamisch anfordern und Freigeben mit new und delete

```
void f() {
  IntItem *p = new IntItem;  // Create new IntItem on the heap
  p->x = 2;
  delete p;
  // Can't use *p unless reinitialization
}

void g() {
  IntItem *p;
  {
    IntItem *q = new IntItem;
    q->x = 2;
    p = q;
  }
  // q out of scope here, but object pointed to by q still exists
  delete p;  // Delete object formerly pointed to by q
}
```







Dynamische Listen richtig dynamisch

```
int main(int argc, const char* argv[])
{
  List *head, *p;

  head = new List; p = head;
  p->element = new IntItem; p->element->x = 1;

  p->next = new List; p = p->next;
  p->element = new IntItem; p->element->x = 2;

  p->next = new List; p = p->next;
  p->element = new IntItem; p->element->x = 3;
  p->next = 0;

  std::cout << head->element->x
    << " " << head->next->element->x
    << " " << head->next->next->element->x
    << std::endl;
  return 0;               // Bug: must delete list elements
}
```







Freigeben von Listenelementen

```
int main(int argc, const char* argv[])
{
  // ... see above

  delete head->next->next->element;
  delete head->next->next;
  delete head->next->element;
  delete head->next;
  delete head->element;
  delete head;

  return 0;
}
```







Fehlerhafte Freigabe

Nachdem ein Speicherbereich mit `delete` freigegeben wurde, ist der Inhalt des Speicherbereichs undefiniert

```
int main(int argc, const char* argv[])
{
  // ...

  // Bug: don't do this:
  delete head;
  delete head->element;
  delete head->next;
  delete head->next->element;
  delete head->next->next;
  delete head->next->next->element;

  return 0;
}
```







Funktionen

Funktionen dienen dazu, Abstraktionen zu schaffen und Code wiederzuverwenden

Funktionen erhalten *Parameter* bevor sie *aufgerufen* werden und liefern einen Wert *zurück*

Beispiel aus der Mathematik: Sinus-Funktion, erhält als Parameter eine reelle Zahl und liefert als Wert eine reelle Zahl. In C++:

```
double sin(double x) {
  double ret = 0.0;

  // Code to calculate sin x and store in ret
  return ret;
}
```

Besonderer Rückgabetyp: `void` gibt nichts zurück

Funktionsaufrufe

Funktionsaufrufe unterbrechen den Programmfluß an der Stelle des Aufrufs, führen den Code der Funktion aus und kehren dann an die Stelle des Aufrufs zurück

Dabei werden die *formalen Parameter* aus der Funktionsdefinition durch die *aktuellen Parameter* aus dem Funktionsaufruf ersetzt

```
double sin(double x) {
  double ret = 0.0;
  // Code to calculate sin x and store in ret
  return ret;
}
int main(int argc, const char *argv[]) {
  // ...
  double s1 = 2.0*sin(0.3421);  // x gets the value 0.3421
  double s2 = 2.0*sin(0.1243);  // x gets the value 0.1243
}
```

Parameterübergabe

In C++ werden Parameter per Wert übergeben

```
void inc(int x) {
  x = x + 1;
}

int main(int argc, const char *argv[]) {
  // ...
  int i = 1;
  inc(i);
  // i still has the value 1
}
```

Will man den Wert des Parameters verändern, muß man mit Zeigern oder Referenzen arbeiten







Parameterübergabe per Zeiger

```
void inc(int* x) {
  *x = *x + 1;
}

int main(int argc, const char *argv[]) {
  // ...
  int i = 1;
  inc(&i);
  // i now has the value 2
}
```







Parameterübergabe per Referenz

```
void inc(int& x) {
  x = x + 1;
}

int main(int argc, const char *argv[]) {
  // ...
  int i = 1;
  inc(i);
  // i now has the value 2
}
```







Ablauf eines C++-Programms

Erster Anlaufpunkt bei der Ausführung eines C++-Programms ist die (!) Funktion namens `main()`

Funktion `main()` erhält Zugriff auf Kommandozeilenparameter

Funktion `main()` ruft andere Funktionen auf und führt selbst Berechnungen durch

Der Rückgabewert von `main()` entscheidet in der Aufrufumgebung (shell) über Erfolg oder Mißerfolg des Programms (0 bedeutet Erfolg, alles andere Mißerfolg)

Wenn nichts anderes da steht, werden Anweisungen eine nach der anderen bearbeitet, bis die `return`-Anweisung in `main()` erreicht ist







Ausdrücke und Anweisungen

Beispiel anhand eines Taschenrechnerprogramms (Stroustrup)

Alle interessanten Anweisungen werden an einem realistischen Beispiel besprochen

Es ist aber Aufmerksamkeit vonnöten, um dem zu folgen!

Eingabe:

```
r = 2.5
area = pi * r * r
```

Ausgabe:

```
2.5
19.635
```







Struktur des Taschenrechners

- Eingabefunktion, die die Eingabe entgegennimmt und in sogenannte *Token* zerlegt
- Parser, der die zerlegte Eingabe auswertet;
- Symboltabelle, die die Bedeitung von Namen wie `pi` und `area` speichert
- Treiberprogramm, das alles zusammenhält

Token sind Typ-Wert-Paare, wie sie im Compilerbau verwendet werden. Token-Typen sind z.B. „Zahl“, „Name“, „Linke Klammer“ usw. Token-Werte können sein: „`123`“, „`foo`“, „`(`“.

In unserem Beispiel könnte die Zeile „`area = 2*pi*pi`“ in die Tokenfolge (Name, "`area`"), (Gleich, =), (Zahl, 2), (Mal, *), (Name, "`pi`"), (Mal, *), (Name, "`pi`") zerlegt werden.







Grammatik

```
program:
  END                          // END is end-of-input
  | expr_list END

expr_list:
  expression PRINT             // PRINT is semicolon
  | expression PRINT expr_list

expression:
  expression + term | expression - term | term

term:
  term / primary | term * primary | primary

primary:
  NUMBER
  | NAME
  | NAME = expression
  | - primary
  | ( expression )
```







Auswertungs-Strategie

Die hier verwendetet Strategie heißt *recursive descent*, also rekursiver Abstieg

- Jede Produktion in der Grammatik erhält ihre eigene Funktion, die andere Funktionen aufruft
- Terminale Symbole (`NUMBER`, `NAME`, `(` usw.) werden von einer lexikalischen Analyse erkannt
- Sobald beide Operanden eines Ausdrucks erkannt sind, wird die Operation sofort berechnet







Definition von Token im Taschenrechner

```
enum Token_value {
  NAME,          NUMBER,        END,
  PLUS = '+',    MINUS = '-',   MUL = '*',  DIV = '/',
  PRINT = ';',   ASSIGN = '=',  LP = '(',   RP = ')',
};
Token_value cur_token = PRINT;
```

- Die Aufzählungswerte hätte man nicht festlegen müssen (ist aber bequem für Leute mit Debuggern)
- Dier Initialwert für `cur_token` ist absichtlich PRINT (später)







Parser-Funktionen

Beim Lesen von Token aus der Eingabe kann es vorkommen, daß man mal ein Token zuviel liest. Dieses Token muß dann gegebenenfalls beim nächsten Versuch erneut gelesen werden und darf nicht überschrieben werden

Jede Parser-Funktion erhält ein bool als Argument, in dem festgelegt wird, ob das nächste Token bereits vorliegt oder eingelesen werden muß

Jede Funktion entspricht einer Produktion in der Grammatik

Die Funktionen werden in der Regel nicht viel tun, sondern oft bloß andere Funktionen aufrufen







Funktion expr()

```
double expr(bool get) {   // Add and subtract
  double left = term(get);
  bool finished = false;

  while (!finished) {
    switch (cur_token) {
    case PLUS:
      left += term(true);
      break;
    case MINUS:
      left -= term(true);
      break;
    default:
      finished = true;
      break;
    }
  }
  return left;
}
```







Kombinierte Berechnung/Zuweisung

```
var += value;
var -= value;
var *= value;
// etc
```

- Hat denselben Effekt wie `var = var + value;`, aber `var` wird nur einmal ausgewertet
- Typische Indikatoren: „Ich will die Variable erhöhen, verdoppeln, halbieren“, „Ich muß von x den Wert $f(x)/f'(x)$ abziehen“
- Kontraindikatoren: „Ich muß x durch $1/f(x)$ ersetzen“







Die `switch`-*Anweisung*

```
switch (selector) {
case label1: statement
case label2: statement
...
default: statement
}
```

- Der *Selektor* wird ausgewertet und der Reihe nach mit `label1` usw. verglichen
- Es wird der Code beim passenden *case-Label* ausgeführt
- Eine *break-Anweisung* wird benutzt, um die case-Anweisung zu verlassen (sonst *fall-through*)
- Passt keins der case-Labels: `default`-Zweig
- Die case-Labels müssen alle verschieden und alle ganzzahlig sein

Die `while`*-Anweisung*

```
while (condition)
  statement
```

- Die *Bedingung* wird ausgewertet
- Wenn `true`, dann wird die Anweisung ausgeführt
- Das ganze wird solange wiederholt, bis die Bedingung `false` wird
- Ist die Bedingung gleich zu Anfang `false`, wird die Anweisung gar nicht ausgeführt

```
int i = 0;
while (i < 10) {
  std::cout << i << std::endl;
  i += 1;
}
```

Funktion term()

```
double term(bool get) {   // Multiply and divide
  double left = prim(get);
  bool finished = false;
  while (!finished) {
    switch (cur_token) {
    case MUL:
      left *= prim(true); break;
    case DIV:
      if (double d = prim(true)) {
        left /= d;
      } else {
        left = error("division by zero");
        finished = true;
      }
      break;
    default:
      finished = true; break;
    }
  }
  return left;
}
```







Die `if`*-Anweisung*

```
if (condition)
  statement

if (condition)
  statement
else
  statement
```

- Die *Bedingung* wird ausgewertet
- Wertet sie zu `true` aus, wird der erste Zweig (*if-* oder auch *then-Zweig*) ausgeführt
- Wertet sie zu `false` aus, und gibt es einen zweiten Zweig (*else-Zweig*), so wird dieser ausgeführt
- Hier: Implizite Umwandlung von `double` zu `bool`: Alles außer `0.0` ist `true`, nur `0.0` ist `false`







Funktion `prim()`, *Teil 1*

Die Funktion `prim()` funktioniert analog zu `expr()` und `term()`, aber wir kommen jetzt etwas in die Tiefe, wo ein bißchen echter Arbeit getan wird

```
double prim(bool get) { // Handle primaries
  double ret = 0.0;

  if (get) get_token();

  switch (cur_token) {
  case NUMBER:          // Floating-point constant
    ret = number_value;
    get_token();
    break;
```







Funktion `prim()`, *Teil 2*

```
  case NAME: {
      double &v = table[string_value];
      if (get_token() == ASSIGN) v = expr(true);
      ret = v;
    }
    break;
  case MINUS:
    ret = -prim(true);
    break;
  case LP: {
      ret = expr(true);
      if (cur_token != RP) ret = error("')' expected");
      get_token();     // Eat ')'
    }
    break;
  default:
    ret = error("Primary expected");
    break;
  }
  return ret;
}
```







Symboltabelle

Symboltabelle ist eine `map`

```
#include <map>

map<string, double> table;
```

Das ist das erste Beispiel eines *Template-Typs*, eines *generischen* Datentyps

Die Standard C++-Bibliothek hat eine Reihe solcher Typen bereits vordefiniert, sie heißt auch Standard Template Library

Eine `map` speichert Schlüssel-Wert-Paare (key-value-pairs) und bildet einen Schlüssel auf einen gespeicherten Wert ab

Der Typ `map` macht etwas magisches, so daß man auf das Abbild eines Schlüssels k mit `table[k]` zugreifen kann







Symboltabelle: Beispiel

Angenommen, der Benutzer gibt `radius = 6378.388;` ein

Der Taschenrechner wird dann folgendes ausführen:

```
double& v = table["radius"];
// expr() calculates value to be assigned
v = 6378.388;
```

Der Wert von `table["radius"]` ist eine *Referenz* und kein *Wert*, damit man das wie ein Array auch auf der linken Seite des Gleichheitszeichens verwenden kann (zuweisbarer Lvalue, siehe zweite Vorlesung)







Vorwärtsdeklaration

Die Funktion `expr()` ruft `term()` auf, die ruft `prim()` auf und die wiederum `expr()`

Jeder Name muß vor Verwendung deklariert werden

Bei zyklischen Abhängigkeiten geht das aber nicht

Dieser Zyklus muß also aufgebrochen werden: Durch Vorwärtsdeklarationen

```
double expr(bool);
```

Vor der Definition von `prim()` wird gesagt: „Achtung, da gibt's später eine Funktion `expr()`, die ein `bool` als Parameter bekommt und einn `double` zurückliefert“







Eingabebehandlung I: EOF, „Eigenwerte“

```
Token_value get_token() {
  Token_value ret = END;
  char ch = 0;
  std::cin >> ch;

  switch (ch) {
  case 0: break;  // Return END token
  case ';':
  case '*':
  case '/':
  case '+':
  case '-':
  case '(':
  case ')':
  case '=':
    ret = Token_value(ch);
    break;
```







Eingabebehandlung III: Zahlen, Namen

```
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
    cin.putback(ch);
    cin >> number_value;
    ret = NUMBER;
    break;
  default:
    if (isalpha(ch)) {
      cin.putback(ch);
      cin >> string_value;
      ret = NAME;
    } else {
      error("bad token");
      ret = PRINT;
    }
    break;  // Not strictly needed
  }
  curr_token = ret;
  return ret;
}
```







„Fehlerbehandlung“

```
int no_of_errors;

double error(const std::string &message) {
  no_of_errors++;
  std::cerr << "error: " << message << std::endl;
  return 1.0;
}
```







Treiberprogramm

```
int main(int argc, const char *argv[]) {
  table["pi"] = 3.1415926535897932385;
  table["e"] = 2.7182818284590452354;

  while (cin) {
    get_token();
    if (cur_token == END)
      break;
    if (cur_token == PRINT)
      continue;
    std::cout << expr(false) << std::endl;
  }

  return no_of_errors;
}
```







Beispiele

```
% gcc -O -Wall -o Calc Calc.cc -lstdc++ # For gcc 2.95
% gcc -O -Wall -o Calc Calc.cc          # For gcc 3.x
% ./Calc
r = 2;
2
area = pi * r * r ;
12.5664
a=2 = 3;
3
a=2 ;
3
%
```



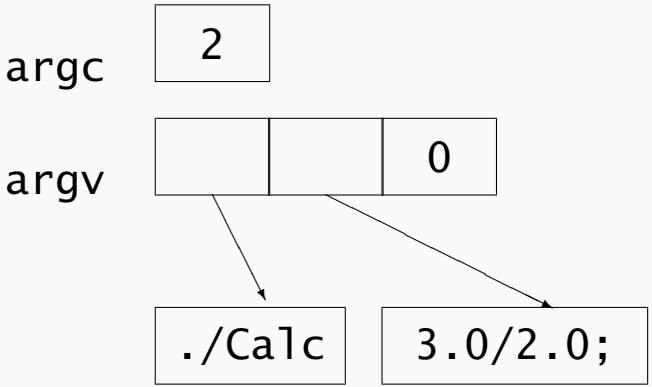




Kommandozeilenargumente

Die Funktion `main()` erhält Kommandozeilenargumente in `argc` und `argv`. Der Parameter `argv` enthält dabei den größten gültigen Index in `argv`.

Beispiel: `./Calc "3.0/2.0;"`



Benutzung, Teil 1

```
#include <sstream>

std::istream* input;

int main(int argc, const char *argv[]) {
  bool input_ok = true;
  int ret = 0;

  switch(argc) {
  case 1: input = &std::cin; break;
  case 2: input = new std::istringstream(argv[1]); break;
  default: error("too many arguments"); input_ok = false; break;
  }
```







Benutzung, Teil 2

```
  if (input_ok) {
    table["pi"] = 3.1415926535897932385;
    table["e"] = 2.7182818284590452354;

    while (*input) {
      get_token();
      if (cur_token == END) break;
      if (cur_token == PRINT) continue;
      std::cout << expr(false) << std::endl;
    }
    ret = no_of_errors;
  } else
    ret = 1;

  return ret;
}
```

Die `for`*-Anweisung*

```
for (initializer; condition; update)
  statement
```

Lediglich eine verkürzende Schreibweise für eine bestimmte `while`-Anweisung:

```
initializer;
while (condition) {
  statement;
  update;
}
```

Mit `break` verläßt man die Schleife

Mit `continue` leitet man die nächste Iteration ein

Alle drei Teile (initializer, condition, update) dürfen leer sein

Endlosschleife: `for ( ; ; )`







Auswertungsreihenfolge

Auswertungsreihenfolge von Unterausdrücken innerhalb eines Ausdrucks ist in der Regel *undefiniert*!

```
int x = f(2) + g(3);  // unknown whether f() or g() evaluated first
```

Grund: Compiler hat mehr Optimierungsmöglichkeiten, wenn er die Auswertungsreihenfolge selbst bestimmen darf. Andererseits:

```
int i = 1;
v[i] = i++;
```

Kann ausgewertet werden entweder als `v[1] = 1` oder als `v[2] = 1` oder noch merkwürdigere Dinge

Compiler *können* warnen, *müssen* aber nicht







Garantierte Auswertungsreihenfolge

Operatoren || und && verknüpfen logische Ausdrücke:

Der Ausdruck a || b hat den Wert `true`, wenn a zu `true` auswertet, sonst den Wert von b

Der Ausdruck a && b hat den Wert `false`, wenn a zu `false` auswertet, sonst den Wert von b

Garantierte links-nach-rechts-Auswertung

```
address* p = 0;

// ...
if (p != 0 && p->zip[0] == 6) { // p->zip[0] accessed only if p != 0
  // ...
}
```







Auswertungsreihenfolge forcieren

Klammern können zur Forcierung benutzt werden

```
x = a*b/c;   // means (a*b)/c
x = a*(b/c); // force a*(b/c)
```

Regel: “[The expression] a*(b/c) may be evaluated as (a*b)/c only if the user cannot tell the difference.”

Bei Ganzzahlen gibt es Überläufe

```
x = (3 * 1234567890) / 1234567890; // 0 on a 32-bit machine
x = 3 * (1234567890 / 1234567890); // 3 on a 32-bit machine
```

Bei Gleitkommazahlen sind die Operatoren ohnehin nicht assoziativ

```
x = (0.1 * 0.3) / 0.7; // 0.042857142857142858 on an IEEE 754 machine
x = 0.1 * (0.3 / 0.7); // 0.042857142857142864 on an IEEE 754 machine
```







Operator-Präzedenz I

Wird a && b || c als (a && b) || c oder als
a && (b || c) ausgewertet?

Präzedenzregeln decken die meisten Fälle „intuitiv“ ab

Manchmal jedoch nicht:

```
if (i&mask == 0)  // oops: means i & (mask == 0)
```

Wenn Sie viele Klammern benutzen müssen, zerlegen Sie vielleicht besser Ihren Ausdruck!

Machen Sie sich keine Sorgen um die Effizienz Ihres Prograsmms: Der Compiler wird in der Regel solche zerlegten Ausdrücke wieder optimieren

Setzen Sie stattdessen auf Lesbarkeit!







Operator-Präzedenz II

Das hier kompiliert zwar, funktioniert aber nicht:

```
if (10 <= x < 15) // take if (x <= 10 && x < 15) instead
```

Auch das hier funktioniert (wahrscheinlich) nicht wie geplant

```
if (a = 7) // oops: means if ((a = 7) != 0)
```

Typisch jedoch:

```
while (cin >> c) { /* ... */ }

/* The above is a shorthand notation for this: */

cin >> c;
while (c) {
  // ...
  cin >> c;
}
```







Bitoperationen

Sie können sich einen `int` auch als Abfolge von Bits vorstellen. C++ hat Operatoren, mit denen Sie bits an- uns ausknipsen können

```
unsigned int bit_value = 0x1257;    // 0001 0010 0101 0111
unsigned int mask = 0x3122;         // 0011 0001 0010 0010

unsigned int b = bit_value | mask;  // 0011 0011 0111 0111
unsigned int c = bit_value & mask;  // 0001 0000 0000 0010
unsigned int d = ~mask;             // 1100 1110 1101 1101
unsigned int e = bit_value & ~mask; // 0000 0010 0101 0101
unsigned int f = bit_value ^ mask;  // 0010 0011 0111 0101
```







Inkrement und Dekrement

C++ kennt je zwei Inkrement- und Dekrement-Operatoren: Ist `lvalue` ein zuweisbarer Lvalue, dann bedeutet `++lvalue` per Definition dasselbe wie `lvalue += 1`

Dekrementierung analog mit `--` statt `++`

Beide Operatoren kann man als Präfix- oder Postfix-Operatoren verwenden

Der Wert von `++lvalue` (`lvalue++`) ist der *neue* (*alte*) Wert von `lvalue`

Die Operatoren funktionieren selbstverständlich auch bei Zeigern auf Array-Elemente

Die Operatoren funktionieren besonders gut, um in Schleifen Variablen hoch- oder herunterzuzählen







Kopieren von Zeichenketten I

Erste Fassung:

```
void cpy(char *p, const char *q) {
  for (int i = 0; i <= strlen(p); i++) p[i] = q[i];
}
```

Zweite Fassung (nur noch eine Iteration über q)

```
  for (int i = 0; q[i] != 0; i++) p[i] = q[i];
  p[i] = 0; // Terminating null character
```

Dritte Fassung (Elimination von `i`)

```
  while (*q != 0) {
    *p = *q;
    p++;
    q++;
  }
```

Kopieren von Zeichenketten II

Vierte Fassung (Postfix-Inkrement)

```
while (*q != 0) {
  *p++ = *q++;
}
*p = 0;   // Terminating null character
```

Fünfte Fassung (Wert von *p++ = *q++ ist *q)

```
while ((*p++ = *q++) != 0) { }
```

Sechste Fassung (Vergleich mit 0, leerer Block)

```
while (*p++ = *q++) ;    // Empty loop body
```







Kopieren von Zeichenketten III

```
void cpy(char *p, const char *q) {
  while (*p++ = *q++) ;    // Empty loop body
}
```

Ist diese Version schwerer zu lesen? Nicht für erfahrene Programmierer!

Ist diese Version effizienter? Wahrscheinlich nicht (mit Ausnahme der ersten Version)

Effizienteste Art, eine Zeichenkette zu kopieren:

```
#include <string.h>

strcpy(p, q);  // char* strcpy(char*, const char*);
```







Erneut dynamischer Speicher

Rekapitulation: Dynamischer Speicher wird mit new erzeugt und mit `delete` wieder freigegeben

Man kann auch Arrays dynamisch erzeugen:

```
int* s = new int[100]; // Create array of 100 int-s
// use s
delete[] s;            // Note []
```

Um ein Objekt mit `delete` oder `delete[]` freigeben zu können, muß die Freigabefunktion ermitteln können, wieviel Speicher vorher alloziert wurde. Daher wird in der Regel ein mit new erzeugtes Objekt etwas mehr Speicher verbrauchen, als ein statisch erzeugtes.







Deklarationen in Anweisungen I

Es ist guter Stil, Objekte erst dann zu deklarieren, wenn sie auch wirklich benötigt werden

Das geht mit C++, aber nicht mit C

```
int f() {
  int x;    // Declare and define x
  // 100 lines later...
  for (x = 0; x < 100; x++) // ...
}
```

Besser (nur in C++):

```
int f() {
  // 100 lines of code...
  for (int x = 0; x < 100; x++) // ...
  // Can't use x after the loop
}
```







Deklarationen in Anweisungen II

Manchmal muß man den Wert der Iterationsvariable nach Ausführung der Schleife noch kennen:

```
int i;
for (i = 0; i < n; i++) {
  if (a[i] == b[i])
    break;
}
if (i < n)
  std::cout << "Match at i = " << i << std::endl;
```

Solche Deklarationen funktionieren auch in `if`-Anweisungen:

```
if (double d = prim())
  x /= d;
else
  error("division by zero");
// Can't use d here: out of scope
```

Default-Argumente

Eine allgemeine Funktion benötigt oft mehr Argumente, als zur Abarbeitung einfacher Fälle nötig sind

Beispiel: Funktion, die einen `int`-Wert ausgibt und das in verschiedenen Basen tun kann

```
void print(int value int base = 10); // default base is 10

void f() {
  print(31);    // print(31,10)
  print(31,10); // same
  print(31,16); // print in hex
  print(31,2);  // print in binary
}
```

Unspezifizierte Anzahl an Argumenten I

Manche Funktionen haben eine unspezifizierte Anzahl an Argumenten unspezifizierten Typs (z.B. `printf`)

In diesem Fall wird die Liste der Argumente mit „`...`“ abgeschlossen:

```
int printf(const char*, ...);
```

Zum Beispiel:

```
printf("Hello, world\n");
printf("My name is %s %s\n", first_name, second_name);
printf("%d + %d = %d\n", 2, 3, 2+3);
```

So eine Funktion braucht Informationen, die dem Compiler nicht bekannt sind (hier: Format-String mit besonderen Zeichenfolgen (%s, %d usw)







Unspezifizierte Anzahl an Argumenten II

```
#include <cstdarg>
#include <stdio.h>

void error(const char *message, ...) {
  va_list args;
  fprintf(stderr, "Error: ");
  va_start(args, message);
  vfprintf(stderr, message, args)
  va_end(args);
  putc('\n', stderr);
}

void f() {
  int expected;
  int got;

  // ...
  if (expected != got)
    error("%d expected, got %d", expected, got);
}
```







Kommentare

Wie man nicht kommentieren sollte:

```
// Variable "v" must be initialized

// Variable "v" must be ued only by function f()

// Call function "init()" before calling any other function

// Call function "cleanup()" at the end of the program

// Don’t use function "weird()"

// Function "f()" takes two arguments
```







Getrennte Übersetzung

Aufteilung eines Programmierprojekts auf mehrere Dateien:

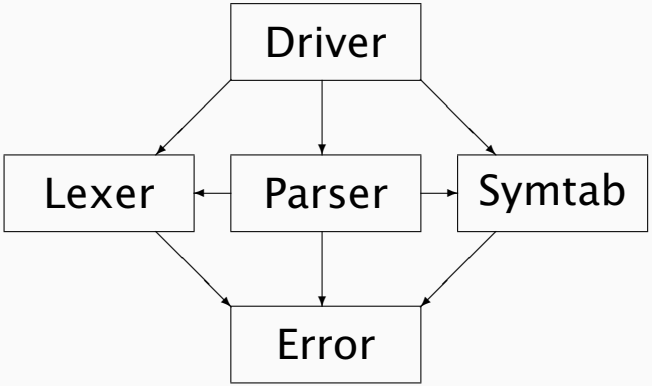
- Verschiedene Datein können von verschiedenen Leuten bearbeitet werden
- Funktionen und Daten können zu logischen Einheiten gruppiert werden
- (Nicht-)Abhängigkeiten zwischen Modulen können ausgenutzt werden, um die Übersetzungszeit zu veringern

C++ bietet Ihnen Unterstützung, um das zu erreichen:

- Die `#include`-Anweisung
- Die `extern` und `static`-Deklarationen
- Trennung zwischen Deklaration und Definition

Beispiel: Taschenrechner

Taschenrechner besteht aus lexikalischer Analyse, Parser, Driver, Symboltabelle und Fehlerbehandlung, also fünf Module



Aufteilung: Deklarationen in Header-Dateien (Endung `.h`), Definitionen in Quell- (Source-)Dateien (Endung `.cc`)

Error

```
// Error.h
#include <string>

extern int no_of_errors;
extern double error(const std::string &message);
```

Implementierung

```
//Error.cc
#include <iostream>

#include "Error.h"

int no_of_errors;

double error(const std::string &message) {
  no_of_errors++;
  std::cerr << "error: " << message << std::endl;
  return 1.0;
}
```







Lexer.h

```
#include <string>

enum Token_value {
  NAME,         NUMBER,         END,
  PLUS = '+',   MINUS = '-',    MUL = '*',  DIV = '/',
  PRINT = ';',  ASSIGN = '=',   LP = '(',   RP = ')',
};

extern Token_value cur_token;
extern double number_value;
extern std::string string_value;

extern Token_value get_token();
```







Lexer.cc

```
#include <cctype>

#include "Lexer.h"
#include "Error.h"

Token_value cur_token = PRINT;
double number_value;
std::string string_value;

Token_value get_token() {
  Token_value ret = END;
  // ...
  return ret;
}
```







Parser.h

```
double expr(bool);
```







Parser.cc

```
#include "Lexer.h"
#include "Symtab.h"
#include "Parser.h"
#include "Error.h"

static double prim(bool get) { // Handle primaries
  double ret = 0.0;
  // ...
  return ret;
}
static double term(bool get) {   // Multiply and divide
  double left = prim(get);
  // ...
  return left;
}

double expr(bool get) {   // Add and subtract
  double left = term(get);
  // ...
  return left;
}
```







Symtab.h, Symtab.cc

```
#include <map>
#include <string>

extern map<std::string, double> table;
```

Implementierung

```
#include "Symtab.h"

map<std::string, double> table;
```







Makefile

```
LOADLIBES = -lstdc++
CXXFLAGS = -O -Wall

Main: Main.o Error.o Lexer.o Parser.o Symtab.o

.PHONY: clean
clean:
rm -f *.o Main

Error.o: Error.cc Error.h
Lexer.o: Lexer.cc Lexer.h Error.h
Main.o: Main.cc Parser.h Symtab.h Error.h Lexer.h
Parser.o: Parser.cc Lexer.h Symtab.h Parser.h Error.h
Symtab.o: Symtab.cc Symtab.h
```







Übersetzungslauf

```
% make
g++ -O -Wall   -c -o Main.o Main.cc
g++ -O -Wall   -c -o Error.o Error.cc
g++ -O -Wall   -c -o Lexer.o Lexer.cc
g++ -O -Wall   -c -o Parser.o Parser.cc
g++ -O -Wall   -c -o Symtab.o Symtab.cc
cc   Main.o Error.o Lexer.o Parser.o Symtab.o -lstdc++  -o Main
% ./Main
2 + 3;
5
a = 3 + 2;
5
m = a * a ;
25
%
```







Translation Units und Linkage

Die Datei, die entsteht, wenn in einer C++-Datei alle `#include`-Anweisungen bearbeitet wurden, heißt *Translation Unit*

Das Programm, das die `#include`-Anweisungen bearbeitet, verändert das C++-Programm noch weiter, z.B. durch *Makros*, die wir aber nicht besprechen werden

Von einem Namen, der außerhalb seiner Translation Unit (TU) sichtbar ist, sagt man, er habe *external linkage*

Ein Name, der nur innerhalb seiner TU sichtbar ist, sagt man, er habe *internal linkage*







Linkage

Ein Objekt kann beliebig oft deklariert, aber nur einmal definiert werden

Bei mehrfacher Deklaration müssen die Deklarationen exakt übereinstimmen

```
// File1.cc
int x = 1;
int b = 1;
extern int c;
static int d = 1;

// File2.cc
int x;  // Means: int x = 0;
extern double b;
extern int c;
static int d = 3;
```







Fehler in vorigem Beispiel

- Variable x zweimal definiert
- Variable b zweimal verschieden deklariert
- Variable c zweimal deklariert, aber nie definiert

Deklaration von d hat internal linkage und ist daher OK!







Mehr Fehler

```
//File1.cc
int x;
extern int g();
int f() { return g(x); }

// File2.cc
int x;
int g() { return f(); }
```







Fehler in vorigem Beispiel

- Variable x zweimal definiert
- Funktion `f()` in `File2.cc` nicht deklariert

Achtung: Aufruf von `f()` wäre in C *kein* Fehler!

Aufruf von `g()` aus `f()` wegen vorheriger korrekter Deklaration OK







Default Linkage

```
const int x = 4;   // Default: internal linkage
struct a {
  char b;
};                 // Default: internal linkage
extern const int y = 5; // Force external linkage on const
int c;             // Default: external linkage (with definition)
static int d;      // Force internal linkage (with definition)
extern int e;      // Declaration with ext. linkage (no definition)
extern int f = 1;  // Definition with external linkage
```







Inline-Funktionen: Motivation

Funktionen werden in der Regel aufgerufen:

- Argumente werden ausgewertet
- Argumente werden auf dem Stack abgelegt
- Die Rücksprungadresse wird auf dem Stack abgelegt
- Die Funktion wird angesprungen
- Auf dem Stack werden lokale Variablen angelegt
- Der Funktionscode wird ausgeführt
- Der Rückgabewert wird auf dem Stack abgelegt
- Die lokalen Variablen werden abgebaut
- Die Rücksprungadresse wird geladen

Funktionsdefinition

```
int max(int a, int b) {
  if (a > b) return a;
  else return b;
}
```

Assembler-Code (i386-Architektur):

```
max__Fii:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%edx
        movl 12(%ebp),%eax
        cmpl %eax,%edx
        jle .L6
        movl %edx,%eax
.L6:    leave
        ret
```







Funktionsaufruf

```
extern void g(int);
extern int max(int a, int b);

void f(int a, int b) {
  g(max(a,b));
}
```

Assembler:

```
f__Fii: pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
        addl $-12,%esp
        addl -8,%esp
        pushl 12(%ebp)
        pushl 8(%ebp)
        call max__Fii
        pushl %eax
        call g__Fi
        leave
        ret
```







Als `inline`*-Funktion: C++*

```
extern void g(int);

inline int max(int a, int b) {
  if (a > b) return a;
  else return b;
}

void f(int a, int b) {
  g(max(a,b));
}
```







Als `inline`*-Funktion: Assembler*

```
f__Fii:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
        movl 8(%ebp),%edx
        addl $-12,%esp
        movl 12(%ebp),%eax
        cmpl %eax,%edx
        jle .L7
        movl %edx,%eax
.L7:    pushl %eax
        call g__Fi
        leave
        ret
```







Tipps und Tricks

- `inline`-Funktionen haben internal linkage, daher müssen sie in jeder TU neu definiert werden
- Mehrfachdefinitionen nicht OK (normalerweise kein Problem)
- Nur für kleine und nichtrekursive Funktionen
- `inline` ist nur ein Hinweis für den Compiler, den dieser ignorieren darf
- “Premature optimization is the root of all evil” (D. E. Knuth), daher Optimierung bitte erst, nachdem Sie mit Messungen (also wirklichen, echten Messungen!) herausgefunden haben, daß Ihr Programm an dieser Stelle zu langsam ist

Exceptions

Bei Programmen, die aus mehreren Modulen bestehen, muß man Fehlerbehandlung in zwei Klassen aufteilen:

- Das Erzeugen und Weiterleiten von Fehlermeldungen, die nicht am Ort behandelt werden können
- Die behandlung von Fehlern, die anderswo entdeckt und erzeugt wurden

Der Autor einer Bibliothek kann Fehlerzustände entdecken, hat aber in der Regel keine Ahnung, was das richtige Vorgehen ist (Meldung, Aufforderung zur Neueingabe, Abbruch, . . .)

Der Benutzer einer Bibliothek weiß, wie mit Fehlern umzugehen ist, hat aber keine Ahnung, wie man sie entdeckt (sonst würden sie ja nicht in der Bibliothek gefunden werden)







Beispiel: Taschenrechner

Im Taschenrechner findet eine *enge Kopplung* zwischen dem Modul für die Fehlerbehandlung und allen anderen Modulen statt

Das geht nur, wenn die anderen Module gründliches Wissen darüber haben, was das Fehlerbehandlungsmodul macht

Dieser Lxus ist aber nicht immer vorhanden

In C++ gibt es einen Mechanismus, um Fehlermeldung und Fehlerbehandlung voneinander zu trennen: Ausnahmen







throw

```
struct RangeError {
  int i;
  RangeError(int i) { this->i = i; } // Constructor
};

char to_char(int i) {
  int min_char = ...; // implementation-defined constant
  int max_char = ...; // implementation-defined constant

  if (i >= min_char && i <= max_char)
    return i;
  else
    throw RangeError(i);
}
```

Die Funktion stellt eine Fehlerbedingung fest und hofft, daß der Aufrufer damit klarkommt und den Fehler behandeln kann







Exception Handler mit `catch`

```
void g(int i) {
  try {
    char c = to_char(i);
    // Use c
  } catch (RangeError e) {
    std::cerr << "Out of range for characters: " << e.i;
  }
}
```

- Kann nur nach Block mit `try` oder direkt nach einem anderen Exception Handler deklariert werden
- Parameter deklariert die Art von Fehler, die gefangen wird
- Wenn kein passender Exception Handler gefunden wird, wird die Exception weitergeworfen
- Wenn überhaupt kein passender Handler gefunden wird, terminiert das Programm

Ausführungsreihenfolge

```
struct SyntaxError { /* ... */ };
struct DivisionByZeroError { /* ... */ };

try {
  expr(false);
  // We get here if and only if expr() didn't cause an exception
} catch (SyntaxError) {
  // Handle syntax error
} catch (DivisionByZeroError) {
  // Handle attempt to divide by zero
}
// We get here if
//   a) expr() didn't cause an exception or
//   b) expr() did cause an exception, it was caught by some
//      exception handler and the exception handler did not
//      return, throw an exception itself or alter the flow
//      of control in another way
```

Exception Handler verschachtelt

```
struct XXII { /* ... */ };

void f() {
  // ...
  try {
    // ...
  } catch (XXII) {
    try {
      // Something complicated
    } catch (XXII) {
      // Complicated handler code failed
    }
  }
  // ...
}
```







Wenn der Speicher ausgeht. . .

```
void f() {
  while (true) {
    new char[10000];
  }
  std::cout << "Done" << std::endl;
}
```

Wird nie „Done“ ausgeben!

```
void f() {
  try {
    while (true) {
      new char[10000];
    }
  } catch (bad_alloc) {
    std::cerr << "Memory exhausted" << std::endl;
  }
  std::cout << "Done" << std::endl;
}
```







set_new_handler()

```
void out_of_store() {
  std::cerr << "new failed: out of memory" << std::endl;
  throw bad_alloc();
}

int main(int argc, const char *argv[]) {
  set_new_handler(out_of_store);
  while (true) {
    new char[10000];
  }
  std::cout << "Done" << std::endl;
  return 0;
}
```

Wenn Sie das versuchen wollen, stellen Sie bitte sicher, daß Sie alleine auf der Maschine sind

Alternativ verwenden Sie das `ulimit`-Kommando der shell
Siehe dazu die manpage von `bash(1)`.







Orthodox Canonical Class Form

Problem: Sie wollen einen Datentyp schreiben, den Sie genauso verwenden können, wie die Basis-Datentypen `int` usw, d.h., Sie wollen solche Datentypen:

- zuweisen können (mit =)
- per Wert an Funktionen übergeben

Ihr Datentyp enthält außerdem möglicherweise Zeiger auf Objekte, die einer Referenzzählung unterliegen und Sie möchten ihn der Gleichförmigkeit wegen anderen Datentypen gleichstellen

Lösung: Sie bedienen sich eines Entwurfsmusters, das *Orthodox Canonical Class Form* genannt wird

Beispiel

```
class ComplexNumber {
public:
  // Default constructor, required by OCCF
  ComplexNumber();
  // Copy constructor, required by OCCF
  ComplexNumber(const ComplexNumber &);
  // Assignment operator, required by OCCF
  ComplexNumber& operator=(const ComplexNumber &);
  // Destructor, required by OCCF
  ~ComplexNumber();

  ComplexNumber(double, double);
  ComplexNumber operator+(const ComplexNumber &) const;
  double abs() const;

private:
  double re;  // Real part
  double im;  // Imaginary part
};
```







Zuweisungsoperator

```
ComplexNumber& ComplexNumber::operator=(const ComplexNumber &copy) {
  re = copy.re;
  im = copy.im;
  return *this;
}

// ...

void f() {
  ComplexNumber x(1,2);  // 1+2i
  ComplexNumber y = x;   // Use copy constructor
  ComplexNumber z;
  z = x;                 // Use ComplexNumber::operator=
  ComplexNumber a, b, c;
  a = b = c = x;         // Use return value of operator=
}
```

Addition

```
ComplexNumber
ComplexNumber::operator+(const ComplexNumber &add) const {
  return ComplexNumber(re + add.re, im + add.im);
}

// ...

void f() {
  ComplexNumber a, b, c;
  ComplexNumber a[] = new ComplexNumber[100];

  c = a + b;  // Shorthand for c = a.operator+(b)

  for (ComplexNumber sum(0,0), i = 0; i < 100; i++)
    sum = sum + a[i];
}
```







Besser

```
class ComplexNumber {
public:
  // ...
  ComplexNumber& operator+=(const complexNumber &);
};

ComplexNumber& ComplexNumber::operator+=(const ComplexNumber &add) {
  re += add.re; im += add.im;
  return *this;
}

void f() {
  ComplexNumber a[] = new ComplexNumber[100];

  // Fill a[]..., then:

  for (ComplexNumber sum(0,0), i = 0; i < 100; i++)
    sum += a[i];
}
```







Operatoren-Überladung

+	-	*	/	%	^	\&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	->*	,
->	[]	()	new	new[]	delete	delete[]

Template-Typen

Sie kennen alle die Operationen für einen Stack:

- new erzeugt einen leeren Stack
- `push(item)` schiebt `item` auf den Stack
- `pop()` entfernt das oberste Element vom Stack
- `top()` gibt das oberste Element, ohne es zu entfernen

Sowas können Sie auch in C++ schreiben







*Spezial-Stack (*int*)*

```
class IntStack {
public:
  static const int stackSize = 100;
  IntStack() { t = 0; mem = new int[stackSize]; }
  ~IntStack() { delete[] mem; }
  void push(int item) { if (top < stackSize) mem[t++] = item; }
  int pop() { if (t > 0) return mem[--t]; else return -1; }
  int top() { if (t > 0) return mem[t-1]; else return -1; }
private:
  int t;
  int* mem;
};
```







Probleme

- Wenn Sie einen `double`-Stack brauchen, müssen Sie den gesamten Code duplizieren
- Durch Duplizierung können sich Fehler einschleichen
- Eine Vielzahl von im wesentlichen gleichen Codeteilen zu warten, ist sehr aufwendig

C++ löst dieses Problem durch Unterstützung für generische Programmierung

Hier können Sie Typen Parametrisieren und als Parameter andere Typen angeben







Generischer Stack

```
class StackEmpty {};
class StackFull {};

template<class C> class Stack {
public:
  static const int stackSize = 100;
  Stack() { t = 0; mem = new C[stackSize]; }
  ~Stack() { delete[] mem; }
  void push(const C& item) {
    if (t < stackSize) mem[t++] = item; else throw StackFull();
  }
  C &pop() {
    if (t > 0) return mem[--t]; else throw StackEmpty();
  }
  const C& top() const {
    if (top > 0) return mem[top-1]; else throw StackEmpty();
  }
private:
  int t;
  C* mem;
};
```







Eigenschaften

- Im Konstruktor werden die Default-Konstruktoren von C aufgerufen
- Im Destruktor werden die Destruktoren von C aufgerufen
- Es wird in `push()` automatisch der Zuweisungsoperator für C aufgerufen







Standard C++-Bibliothek

Was sollte alles in der Standard C++-Bibliothek enthalten sein?

Idealerweise sollte ein Programmierer alle interessanten, wichtigen und hinreichen allgemeinen Klassen, Funktionen, Templates etc. in einer Bibliothek finden

Die Frage ist aber nicht: „Was sollte in *irgendeiner* Bibliothek sein?“, sondern: „Was sollte in der *Standard*-Bibliothek enthalten sein“

„Alles!“ ist also eine vernünftige Approximation auf die Antwort auf die erste Frage, aber nicht auf die zweite.

Inhalt der C++ Bibliothek

- Spracheigenschaften (Speichermanagement)
- Implementierungsabhängigkeiten (größter `float`-Wert)
- Funktionen, die implementierungsunabhängig nicht optimal implementierbar sind (`sqrt()`, `memmove()`)
- Nicht-Elementare Abstraktionen, die Programmierer zwecks Portabilität verwenden sollten (Maps, Sortierfunktionen, I/O)
- Framework und Konventionen zur benutzergesteuerten Erweiterung der Bibliotheksleistungen („<<“)
- Gemeinsame Grundlage für weitere Bibliotheken
- Dienste, die üblicherweise in Bibliotheken enthalten sind (Zufallszahlen)







Organisation

- Alle Dienste sind Bestandteil des Namensraums `std`
- Werden über Header angeboten
- Standard Header, der mit `c` beginnt, ist äquivalent zu einem Standard-C-Header
- Jeder Standard-C-Header <X.h> hat einen äquivalenten C++-Header <cX>, der dieselben Namen im Namensraum `std` definiert.







Container

`<vector>`	eindimensionales Array von T
`<list>`	doppelt verkettete Liste von T
`<deque>`	Warteschlange mit zwei Enden von T
`<queue>`	einfache Warteschlange von T
`<stack>`	Stack von T
`<map>`	assoziatives Array von T
`<set>`	Menge von T
`<bitset>`	Array von booleschen Werten

- `multimap` und `multiset` befinden sich in `<map>`, bzw. `<set>`
- `priority_queue` findet sich in `<queue>`

Allgemeine Hilfen

`<utility>`	Operatoren und Paare
`<functional>`	Funktionsobjekte
`<memory>`	Speicherverwaltung für Container
`<ctime>`	C-artige Datums- und Zeitfunktionen







Iteratoren

`<iterator>` Iteratoren und Unterstützung dafür







Algorithmen

`<algorithm>`	Später...
`<cstdlib>`	binäre Suche, sortieren

Typische Algorithmen können auf beliebige Folgen von Elementen beliebigen Typs angewandt werden (was genau eine Folge ist: später)

Dazu gehören: Sortierung, Kopieren, Iterieren, Umkehren, usw.

Wenn Sie also irgndwann mal eine Funktion schreiben sollten, die etwas in dieser Art macht, schauen Sie besser erst mal in die STL.







Diagnose

`<exception>`	Klasse für Ausnahmen
`<stdexcept>`	Standard-Ausnahmen
`<cassert>`	Zusicherungen à la C
`<cerrno>`	Fehlernummern à la C







Strings

`<string>`	String von T
`<cctype>`	Zeichenklassifikation
`<cwctype>`	Klassifikation von „weiten“ Zeichen
`<cstring>`	String-Funktionen à la C
`<cwchar>`	String-Funktionen à la C für „weite“ Zeichen
`<cstdlib>`	Noch mehr String-Funktionen à la C

Ein/Ausgabe

`<iosfwd>`	Vorwärtsdeklarationen von I/O-Diensten (*)
`<iostream>`	Standard Ein/Ausgabedienste und -Klassen
`<ios>`	Basisklassen (*)
`<streambuf>`	Ein/Ausgabe-Puffer (*)
`<istream>`	Eingabeströme
`<ostream>`	Ausgabeströme
`<iomanip>`	Manipulation von E/A-Strömen (*)
`<sstream>`	String-Ströme
`<cstdlib>`	Zeichenklassifizierung
`<fstream>`	Ströme von/nach Dateien
`<cstdio>`	`printf()` und Konsorten
`<cwchar>`	`printf()` und Konsorten für „weite“ Zeichen

Ein Strom ist wie eine Wurst...

I18N

`<locale>`	Unterschiede in Schreibweisen
`<clocale>`	Unterschiede in Schreibweisen à la C

In Deutschland beginnt die dezimale Entwicklung von π mit $3,1415$, in englischsprachigen Ländern mit 3.1415

In Deutschland schreibt man für tausend „1.000“, in englischsprachigen Ländern „1,000“ oder „1’000“

C++-Unterstützung

`<limits>`	numerische Grenzen
`<climits>`	numerische Grenzen à la C (Ganzzahlen)
`<cfloat>`	numerische Grenzen à la C (Gleitkommazahlen)
`<new>`	dynamische Speicherverwaltung
`<typeinfo>`	Laufzeit-Typinformation
`<exception>`	Ausnahmebehandlung
`<cstddef>`	C-Bibliotheks-Unterstützung
`<cstdarg>`	Funktionsargumentlisten variabler Länge
`<csetjmp>`	Wollen Sie nicht wirklich wissen. . . (*)
`<cstdlib>`	Programmbeendigung
`<ctime>`	Systemuhr
`<csignal>`	Signalverarbeitung à la C

Wer `setjmp()` kennt: Das ist inkompatibel mit Exceptions

Numerik

<complex>	komplexe Zahlen
<valarray>	numerische Vektoren und deren Operationen
<numeric>	verallgemeinerte numerishe Operationen
<cmath>	numerische Funktionen à la C
<cstdlib>	Zufallszahlen

Aus historischen Gründen sind `abs()` und `div()` in `<cstdlib>` statt in `<cmath>` zu finden







Container

Ein Container ist ein Objekt, das andere Objekte (tyoischerweise mehrere) beinhaltet

Operationen: Hinzufügen, löschen, iterieren, suchen

Die STL bietet ein *sehr* reichhaltiges Angebot an Klassen und Methode zum Thema Container an

Viel mehr, als wir hier behandeln können

Bedeutsame *kurze* Beispiele sind schwer zu finden

Daher Referenz-Charakter der folgenden Folien







vector I

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

int intVector() {
  std::vector<int> v = std::vector(); // Vector with size 0

  int i;
  while (std::cin >> i)
    v.push_back(i); // Add to rear, grows as needed

  for (std::vector<int>::iterator p = v.begin(); p != v.end(); ++p)
    std::cout << *p << std::endl;

  v.clear();        // Erase all elements in v
}
```







Element-Zugriff I

```
int oddElementSum(const std::vector<int>& v) {
  int sum = 0;
  for (int i = 1; i < v.size(); i += 2)
    sum += v[i];
  return sum;
}

void f(const std::vector<int>& v) {
  int firstElement = v.front();
  int lastElement = v.back();
  const int firstElementConst = v.front(); // also as const
  const int lastElementConst = v.back(); // also as const

  try {
    int checkedElement = v.at(2); // third element, checked access
  } catch (out_of_range) {
    std::cerr << "Vector access out of range" << std::endl;
  }
}
```







Konstruktoren I

```
void f() {
  std::vector<int> v;         // Standard constructor, size 0
  std::vector<int> w(10);     // Initial size 10
  std::vector<int> x(10, 1); // Initial size 10, initial value 1
  std::vector<int> y = x;     // Copy constructor
  std::vector<int> z;

  z = x;                      // Assignment
  z.assign(10, 1);            // Make z contain 10 copies of 1
}
```







Konstruktoren II

```
class Num {
public:
  Num(long);
  // No default constructor!
};

void f(int i, const list<char> &l) {
  vector<Num> v1(1000);    // Error: No default constructor for Num
  vector<Num> v2(1000, 1); // Use constructor Num::Num(long)
  vector<char> v3(-1);     // Easy for compiler to warn against
  vector<char> v4(i);      // OK but tricky

  vector<char> v5(l.begin(), l.end());
  char[] url = "http://www.st.cs.uni-sb.de/edu/einst/";
  vector<char> v6(&p[0], &p[sizeof(p) - 1]);
  vector<char> v7(p, p + sizeof(p) - 1); // Equivalent to above
}

void g(const list<char>& l) {
  f(-1, l);      // Trick f() into accepting large positive value
}
```







Konstruktoren III

```
void f1(vector<int>& v);       // Common style
void f2(const vector<int>& v); // Common style
void f3(vector<int> v);        // Rare, usually beginner's mistake!

void h() {
  vector<int> v(10000);        // Ten-thousand-element vector

  f1(v);                       // Pass reference: fast
  f2(v);                       // Pass reference: fast
  f3(v);                       // Copy argument: slow
}
```







Iteratoren I

```
void f(const vector<int>& v, const vector<double>& w) {
  int max = v.front();
  for (std::vector<int>::iterator p = v.begin(); p != v.end(); ++p) {
    if (*p > max)
      max = *p;
  }

  // Calculate sum both ways
  double sum1 = 0.0;
  for (std::vector<double>::iterator p = w.begin(); p != w.end(); ++p)
    sum1 += *p;

  double sum2 = 0.0
  for (std::vector<double>::reverse_iterator p = w.rbegin();
       p != w.rend(); ++p)
    sum2 += *p;
}
```







Iteratoren II

Rückwärtsiteratoren werden genauso behandelt wie Iteratoren, d.h., sie haben ihre eigenen `rbegin()` und `rend()`-Methoden und werden hoch(!)gezählt

```
void f(const vector<double>& w) {
  double sum2 = 0.0;

  // Bugs: reverse iterator begins at rbegin(), not at end(),
  // ends at rend(), not begin(), is incremented, not decremented
  for (std::vector<double>::reverse_iterator p = w.end();
       p != w.begin();
       --p)
    sum2 += *p;
}
```







Strings

Ein String ist eine Folge von Zeichen

“Experience shows that it is impossible to design the perfect *string*. People’s taste, expectations, and needs differ too much for that.” (Stroustrup)

In Java ist ein String *immutable*, d.h. kann nach der Erzeugung nicht mehr verändert werden

In C++ können Stringteile nach Erzeugung noch geändert werden

Ein String kann prinzipiell beliebige Zeichen enthalten (ASCII, Unicode, UTF-8, . . .)







Zeichen

> “Character” itself is an interesting concept. Consider the character *C*. The *C* that you see as a curved line on the screen, I typed into my computer many months ago. There, it lives as the numeric value 67 in an 8-bit byte. It is the third letter in the Latin alphabet, the usual abbreviation of the sixth atom (Carbon), and, incidentally, the name of a programming language.”

Ein Zeichen kann in verschiedenen Zeichensätzen verschiedene Werte haben

Der C++-Ansatz ist es, irgend einen Zeichentyp in Strings zu verwenden (Details zu komplex)

Es ist `std::string` äquivalent zu `std::basic_string<char>`







Elementzugriff

Wie `vector` mit `[]` und `at()`

```
void f(const string& s) {
  char c = s[0];                // Unchecked
  char d = s[s.length() - 1]; // Unchecked
  char d = s[s.size() - 1];   // Same as length(), unchecked
  c = s.at(0);                  // Checked
  d = s.at(100);                // Checked
}
```

Kein `front()` und kein `back()`

Beim Datentyp `string` ist `&s[0]` *nicht* dasselbe wie `s` (also nicht wie bei Arrays)

Strings beginnen ihre Indizes bei 0 und reichen bis `length() - 1`







Konstruktoren

```
void f(char* p, const vector<char>& v) {
  string s0;              // Empty string
  string s00 = "";        // Also the empty string

  string s1 = 'a';        // Error: No conversion from char to string
  string s2 = 7;          // Error: No conversion from int to string
  string s3(7);           // Error: No constructor taking one int argument

  string s4(7, 'a');      // Seven copies of 'a': "aaaaaaa"

  string s5 = "Frodo";    // Copy of "Frodo"
  string s6 = s5;         // Copy of s5

  string s7(s5, 3, 2);    // s5[3] and s5[4], i.e., "do"
  string s8(p + 7, 3);    // p[7], p[8], and p[9]
  string s9(p, 7, 3);     // string(string(p), 7, 3), expensive

  string s10(v.begin(), v.end()); // copy all characters from v
}
```







Strings: Fehler I

Oft werden Strings nur gelesen, geschrieben, kopiert etc, was ohne Probleme geht

Wenn wir aber anfangen, auf Teile von Strings zuzugreifen, kann es Schwierigkeiten geben

Die meisten Funktionen zur Stringmanipulation bekommen eine Anfangsposition und eine Zeichenanzahl

Eine Anfangsposition jenseits des Stringendes ruft `out_of_range` Exceptions hervor

Eine Zeichenanzahl, die mehr Zeichen umfaßt, als im String vorhanden sind, meint „den gesamten Rest“ (und ist kein Fehler!)

Strings: Fehler II

```
void f() {
  string s = "Snobol4";
  string s2(s, 100, 2); // start beyond end: throw out_of_range
  string s3(s, 2, 100); // character count too large: ok
  string s4(s, 2, string::npos); // copy everything from s[2] to end
}

void g(string& s) {
  string s5(s, -2, 3); // bad position: throw out_of_range error
  string s6(s, 3, -2); // large count: ok
}
```

Der Typ, der verwendet wird, um Positionen und Zeichenanzahlen anzugeben, ist ein `unsigned` Typ: Eine negative Zahl ist also nur eine verwirrende Art, eine große positive Zahl anzugeben







Strings: Fehler III

Alle `string` Objekte haben eine Länge kleiner als `string::npos`

Versucht man, einen String zu konstruieren, dessen Länge größer als das ist, wird eine `length_error` Exception geworfen

```
void f() {
  string s(string::npos, 'a');  // throws length_error
}
```







Zuweisung

```
void g() {
  string s1 = "Knuth";
  string s2 = "Donald";

  s1 = s2;     // s1 contains copy(!) of s2
  s1[1] = 'a'; // s1 now "Danald", s2 still "Donald"

  s1 = 'a';    // OK, even though construction not allowed
  s1 = "a";    // OK, too

  s2 += 'o';   // s2 now "Donaldo"
}
```







Konvertierung in C-Strings

```
void f() {
  string s = "equinox";        // s.length() == 7
  const char* p1 = s.data();   // p1 points to seven(!) characters
  printf("p1 = %s\n", p1);     // Bad, missing 0 terminator
  p1[2] = 'a';                 // Error: p1 points to const array
  s[2] = 'a';
  char c = p1[1];              // Bad, access after modification

  const char* p2 = s.c_str(); // p2 points to eight(!) characters
  printf("p2 = %s\n", p2);     // OK, c_str() adds terminator
}

char* c_string(const string& s) {
  char* p = new char[s.length() + 1]; // Note: + 1
  // s.copy(p, n, m = 0): Copy at most n characters from s into p,
  // starting at s[m]
  s.copy(p, string::npos);
  p[s.length()] = 0;                  // Note: add terminator
  return p;
}
```

Strings: Vergleiche

```
void f(const string& s1, const string& s2,
       const char* p1, const char* p2) {
  // == works as you would expect...
  if (p1 == s1 && s1 == p2)
    // ...

  // ...or does it?
  if (p1 == p2)
    // ...

  int comparisonResult = s1.compare(s2);
  if (comparisonResult > 0)      // s1 > s2
    // ...
  else if (comparisonResult < 0) // s1 < s2
    // ...
  else                           // s1 == s2
    // ...
}
```







Strings: Groß/Kleinschreibung

```
int cmpNocase(const string& s1, const string& s2) {
  string::const_iterator p1 = s1.begin();
  string::const_iterator p2 = s2.begin();

  while (p1 != s1.end() && p2 != s2.end()
         && toupper(*p1) != toupper(*p2)) {
    p1++;
    p2++;
  }

  if (p1 != s1.end() && p2 != s2.end())
    return toupper(*p1) < toupper(*p2) ? -1 : 1
  else if (s1.size() == s2.size())
    return 0;
  else
    return s1.size() < s2.size() ? -1 : 1;
}
```







Strings: Einfügen

```
string complete_name(const string& first_name,
                     const string& last_name) {
  string s = first_name;
  s += ' ';
  s += last_name;
  return s;
}

// Possibly inefficient:
string complete_name2(const string& first_name,
                      const string& last_name) {
  string s = family_name;
  s.insert(s.begin(), ' ');
  return s.insert(0, first_name);
}

string complete_name3(const string& first_name,
                      const string& last_name) {
  return first_name + ' ' + last_name;
}
```







Strings: Suchen und Finden

```
void f() {
  string s = "accdcde";
  string::size_type i1 = s.find("cd");                // i1 = 2
  string::size_type i2 = s.rfind("cd");               // i2 = 4
  string::size_type i3 = s.find_first_of("cd");       // i3 = 1
  string::size_type i4 = s.find_last_of("cd");        // i4 = 5
  string::size_type i5 = s.find_first_not_of("cd"); // i5 = 0
  string::size_type i6 = s.find_last_not_of("cd");  // i6 = 6
}
```







Strings: Ersetzen

```
void f() {
  string s = "but I have heard it works even if you don't "
             "believe in it";  // String literal concatenation

  s.erase(0,4);                          // Remove initial "but "
  s.replace(s.find("even"), 4, "only");
  s.replace(s.find("don't"), 5, "");   // Erase (replace by nothing)
}

string g() {
  string s = "Mary had a little lamb";
  s.replace(s.find("lamb"), 4, "fun");
  s.replace(s.find("a little", 8, "no");
  return s.substring(s.find(' '), string::npos);
}
```







Zeichenklassifizierung I

```
int isalpha(int);  // Letter: 'A'..'Z', 'a'..'z' in C locale
int isupper(int);  // Uppercase letter: 'A'..'Z' in C locale
int islower(int);  // Lowercase letter: 'a'..'z' in C locale
int isdigit(int);  // decimal digit: '0'..'9'
int isxdigit(int); // hexadecimal digit: '0'..'9', 'a'..'f', 'A'..'F'
int iscntrl(int);  // Control characters (ASCII 0..31 and 127)
int isspace(int);  // ' ', '\t', '\v', return newline formfeed
int ispunct(int);  // Punctuation: none of the above
int isgraph(int);  // isalpha() | isdigit() | ispunct()
int isalnum(int);  // isalpha() | isdigit()
int isprint(int);  // printable: ASCII ' '..'~'

int tolower(int);  // Lower-case equivalent
int toupper(int);  // Upper-case equivalent

// Buggy: Don;t do this (can;t assume 'a'..'z' contiguous
int is_letter(int c) {
  return ('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z');
}
```







Strings: Parametrisierung

```
typedef basic_string<unsigned char> Ustring;
struct JapaneseChar { /* ... */ };
typedef basic_string<JapaneseChar> Jstring;

Ustring first_word(const Ustring& us) {
  Ustring::size_type pos = us.find(' ');
  return Ustring(us, 0, pos);
}

Jstring first_word(const Jstring& js) {
  Jstring::size_type pos = js.find(' ');
  return Jstring(js, 0, pos);
}

template<class S> S first_word(const S& s) {
  typename S::size_type pos = s.find(' ');
  return S(s, 0, pos);
}
```







Stilfragen I: Kommentare

```
//Funktion ggT
int ggT(int a,int b)
{
   while (a%b!=0) //Wenn Rest ungleich 0
   {
      int c=a;   // nach definition tauschen
      a = b;
      b = c%b;   // y nach definition setzten
   }
   return b;     // Rückgabe von b
}

int main()
{
// Variablen initialisieren
int ggt,x=0,y=0;
// Schleifen Variable
bool ende = false;

// ...
}
```







Stilfragen

- Kommentare sollen etwas sagen, das *nicht* im Text steht
- Kommentare sollten *konsistente Groß/Kleinschreibung* aufweisen
- Kommentare sollten *korrekt* sein
- Kommentare sollten *korrekt geschrieben* sein







Kommentierung: Besser

```
// Falls beide Zahlen 0 sind, mache Schluss
while (input1 != 0 || input2 != 0){
// ...
}
```

Kommentar sagt etwas, das nicht offensichtlich ist (Anwendung deMorganscher Regeln nötig, um das herauszufinden)







Ohne Kommentierung

```
int ggt(int m, int n) {  // Annahme: m > 0, n > 0
  int rest = m % n;
  while (rest != 0) {
    m = n;
    n = rest;
    rest = m % n;
  }
  return n;
}

int main(int argc, const char *argv[]) {
  int input1, input2;
  // ...
  while (!(input1 == 0 && input2 == 0)) {
    // ...
  }
  return 0;
}
```







Einrückstile

```
#include <iostream>

using namespace std;


int ggt (int x, int y);

int main(int argc, char* argv[])
{
  int x, y;


  while (true)
  {
    cout << "ggT? ";

    cin >> x >> y;

    if (x == 0 && y == 0)
      break;
```








```
    if (x == 0)
    {
      cout << "ggT: ung?ltiger Wert" << endl;
      continue;

    }

    if (y == 0)
    {
      cout << "ggT: ung?ltiger Wert" << endl;
      continue;

    }

    cout << "ggT(" << x << ", " << y << ") = " << ggt (x, y) << endl;
  }

}


int ggt (int x, int y)
{
  int tmp;

  if (x < 0)
    x = -x;
```








```
  if (y < 0)
    y = -y;

  while (y != 0)
  {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```







Einrückung

Es gibt viele gute Einrückstile

Nehmen Sie irgendeinen, aber nehmen Sie einen!

Populäre Stile: K&R, GNU

Rücken Sie bitte *nie* mit Tabs ein, sondern *immer* mit Leerzeichen (Grund: Nicht alle Editoren haben dieselben TAB-Settings)

Seien Sie konsistent: Wenn Sie nach einem `if` einrücken, tun Sie das also immer







Einrückung: Schlechter Stil

```
int main()
{
  int x, y, z;
  while (true)
    {
  cout << "ggT? ";
  cin >> x >> y;
  if(x == 0 && y == 0) return 0;
  if(x<= 0 || y <= 0) { cout << "ggT: ungueltiger Wert" << endl;continue;
  }
  z=ggT(x,y);
   cout << "ggT("<< x<< ", "<< y<< ") = "<< z <<   endl;
    }
}
   int ggT(int x, int y)
 {
   while (x!=y)
    if(x > y) x -= y;
    else      y -= x;
  return x;
}
```







Einrückung: Verbesserung

```
int main()
{
  while (true) {
    int x, y;

    cout << "ggT? ";
    cin >> x >> y;

    if (x == 0 && y == 0)
      return 0;

    if (x <= 0 || y <= 0) {
      cout << "ggT: ungueltiger Wert" << endl;
      continue;
    }

    cout << "ggT(" << x << ", " << y << ") = " << ggT(x, y) << endl;
  }
}
```







```
int ggT(int x, int y)
{
  while (x!=y) {
    if(x > y)
      x -= y;
    else
      y -= x;
  }
  return x;
}
```







Einrückung: Anderer Stil

```
int main()
{
  while (true)
    {
      int x, y, z;

      cout << "ggT? ";
      cin >> x >> y;

      if (x == 0 && y == 0)
        return 0;
      if (x <= 0 || y <= 0)
        {
          cout << "ggT: ungueltiger Wert" << endl;
          continue;
        }

      z = ggT (x,y);
      cout << "ggT(" << x << ", " << y << ") = " << z << endl;
    }
}
```







```
int ggT (int x, int y)
{
  while (x!=y)
  {
    if(x > y)
      x -= y;
    else
      y -= x;
  }
  return x;
}
```







K&R-Stil

Öffnende geschweifte Klammer auf selber Zeile wie Anweisung (Ausnahme: Funktionsdefinitionen)

Einrücktiefe pro Level: 4 Leerzeichen

Mehrere Deklarationen pro Zeile möglich

`public:`-Labels etc in `class`-Deklaration nicht eingerückt

Methodennamen nur in Kleinbuchstaben durch Unterstriche getrennt







K&R-Stil: Beispiele

```
class MyString {
public:
  MyString();
  MyString& append(const MyString& a);
  MyString& append_multiple(const MyString& a, const MyString& b);
};

MyString& MyString::append_multiple(const MyString& a,
                                    const MyString& b)
{
    if (a.length() == 0) {
        some_function(b);
        return b;
    } else if (b.length() > 0)
        return a;

    while (a.some_method(1, 1, some_really_long_argument,
                         another_argument)) {
        a++, b++;
        call_some_other_function(another_argument);
    }
```








```
    for (i = some_really_long_initializer();
         some_really_long_condition_involving_i(i);
         i++)
        ;         // Empty loop body

    switch(2) {
    case 1:
        some_method();
        break;

    case 2:
        some_other_method();
        break;

    default:
        yet_another_method();
        break;
    }

    cout << "a has the value " << a << " and b has the value "
         << b << endl;
    return b;
}
```







GNU-Stil

In der Regel öffnende geschweifte Klammer auf eigener Zeile, um 2 Leerzeichen eingerückt

Blöcke um weitere 2 Zeichen eingerückt

Leerzeichen vor Funktionsargumenten

Methoden in Kleinschreibung mit Unterstrichen







GNU-Stil: Beispiele

```
class MyString
{
public:
  MyString ();
  MyString & append (const MyString & a);
  MyString & append_multiple (const MyString & a, const MyString & b);
};

MyString &
MyString::append_multiple (const MyString & a,
                           const MyString & b)
{
  if (a.length () == 0)
    {
      some_function (b);
      return b;
    }
  else if (b.length () > 0)
    {
      return a;
    }
```







```
while (a.some_method (1, 1, some_really_long_argument,
                      another_argument))
  {
    a++, b++;
    call_some_other_function (another_argument);
  }

for (i = some_really_long_initializer ();
     some_really_long_condition_involving_i (i); i++)
  ; // Empty loop body

switch (2)
  {
  case 1:
    some_method ();
    break;

  case 2:
    some_other_method ();
    break;

  default:
    yet_another_method ();
    break;
  }
```

```
  cout << "a has the value " << a
       << " and b has the value " << b << endl;

  return b;
}
```







Java-inspirierter Stil

Öffnende Klammer immer auf selber Zeile wie Code

Einrücktiefe 2 oder 4

Methodennamen gemischte Groß/Kleinschreibung

`case`-Labels auf Höhe der `switch`-Anweisung







Java-Stil: Beispiel

```
class MyString {
public:
  MyString ();
  MyString& append(const MyString& a);
  MyString& appendMultiple(const MyString& a, const MyString& b);
};

MyString&
MyString::appendMultiple (const MyString& a,
                          const MyString& b) {
  if (a.length() == 0) {
    someFunction (b);
    return b;
  } else if (b.length() > 0)
    return a;

  while (a.someMethod (1, 1, someReallyLongArgument,
                       anotherArgument)) {
    a++, b++;
    callSomeOtherFunction(anotherArgument);
  }
```








```
  for (i = someReallyLongInitializer();
       someReallyLongConditionInvolvingI(i);
       i++)
    ; // Empty loop body

  switch (2) {
  case 1:
    someMethod ();
    break;

  case 2:
    someOtherMethod();
    break;

  default:
    yetAnotherMethod();
    break;
  }

  cout << "a has the value " << a
       << " and b has the value " << b << endl;

  return b;
}
```







Vorbereitung auf Praktomat I

Gegeben ist ein array $a[0 \ldots n-1]$. In diesem Array sind die Werte $a[0]$ bis $a[k]$ $(0 \leq k < n)$ bereits belegt und wir wollen an Position $0 \leq j \leq k$ ein Element *einfügen*

Dazu müssen wir die Elemente j bis k um eine Position nach oben *verschieben*

Erster Versuch:

```
for (int i = j; i <= k; i++)
  a[i+1] = a[i];
```

Zweiter Versuch:

```
for (int i = k; i >= j; i--)
  a[i+1] = a[i];
```







Vorbereitung auf Praktomat II

Gegeben ist ein array $a[0 \ldots n-1]$. In diesem Array soll das Maximum der Werte von $a[j]$ bis $a[k-1]$ gefunden werden $(j < k)$

```
int max = a[j];
for (int i = j + 1; i < k; i++) {
  if (a[i] > max)
    max = a[i];
}
```

Lediglich den Index finden:

```
int maxIndex = j;
for (int i = j + 1; i < k; i++) {
  if (a[i] > a[maxIndex])
    max = i;
}
```