Das obige Programmsegment gibt in Kleinbuchstaben `"hello world!"` aus, wenn der Buchstabe `'a'` übergeben wurde, im Falle von `c` das Wort `WORLD!` und im Falle von `b` die Zeichenfolge `HELLO WORLD!`. Dieses merkwürdig erscheinende Verhalten entsteht dadurch, daß nach Ansprung des passenden `case`-Labels alle folgenden Anweisungen bis zu einem eventuellen `break` durchlaufen werden ("fallthrough"). Da dies häufig nicht das beabsichtigte Verhalten ist, lohnt sich an einer solchen Stelle anstelle des `break` immer ein Kommentar. Das `default`-Label ist optional und wird angesprungen, falls sonst kein passendes in der `case`-Liste gefunden wird.

2.5 Funktionen und Operatoren

Längere Programme enthalten in der Regel Programmteile, die sich mehr oder weniger häufig wiederholen. Diese Teile will man gern an anderer Stelle ausführlich definieren, und im Programm nur durch einen "Platzhalter", also einen *Funktion*saufruf ersetzen. Ein Beispiel ist das Einholen einer Bestätigung durch den Benutzer, ob eine bestimmte numerische Aufgabe das Löschen oder das Umsortieren von Daten ausgeführt werden soll. Weiterhin möchte man häufig aus Gründen der Übersicht bestimmte Programmteile gruppieren und wenn möglich, die dazugehörigen Anweisungen nur noch durch einen Namen kennzeichnen, der ihre "Funktion" charakterisiert.

Das zu diesem Zweck existierende Sprachmittel sind die Funktionen, die in anderen Sprachen auch Prozeduren oder Unterprogramme genannt werden. Eine Funktion ist gekennzeichnet durch Parameter, die eine Art "Eingabe" in den Funktionsblock darstellen und solche, die eine "Ausgabe" darstellen. Im einfachsten Falle erfolgt die Übergabe der Eingabeparameter als Argument an die Funktion und die "Ausgabe" durch einen entsprechenden Rückgabewert.

Außer in wenigen Ausnahmefällen erzwingt C++ die Spezifikation von Argument- und Rückgabetypen von Funktionen. Hier *definiert* `power` eine Funktion, die aus einer Eingabe eines doppeltgenauen Wertes `base` und einer ganzen Zahl `exponent` einen neuen "`double`"-Wert berechnet und zurückgibt. Der Name deutet an, welcher Algorithmus sich dahinter versteckt.

```
// raise 'base' to the power 'exponent', with exponent being integer,
// base being double; accept also negative exponents
double power(double base, int exponent){
   double result  = 1.;    // will multiply this by base 'exponent' times
   bool   neg_exp = false;
   if (exponent < 0 ) {    // for negative exponents we use a single
                           // division at the end of the function
      neg_exp = true;
      exponent = -exponent;
```

```
   }
   for (int i=0; i < exponent; i++ )
      result *= base;       // successive multiplication
   if ( neg_exp )           // have to take the inverse
      result = 1./result;
   return result;           // return the result to the calling program
}
```

Die Namen der Argumente sind wie Variablennamen frei wählbar; unter diesem Namen können sie dann im Funktionskörper angesprochen werden. Der Anweisungsblock des Funktionskörpers wird wie bei Schleifen nicht durch Semikolon abgeschlossen.

Das Schlüsselwort `return` bricht das Unterprogramm an der spezifizierten Stelle ab, legt den Wert des dahinter angegebenen Ausdrucks auf dem Stack ab und kehrt ins Hauptprogramm zurück. Im obigen Fall besteht dieser Ausdruck nur aus der Variable `result`. Runde Klammern werden an dieser Stelle nicht benötigt. Ein Unterprogramm kann mehrere `return` Anweisungen enthalten. Dies kann ein nützliches Mittel sein, um z.B. bei Angabe unpassender Argumenttypen oder bei Problemen mit der Berechnung die Bearbeitung frühzeitig abzubrechen.

Eine Funktion muß definiert oder deklariert sein, bevor sie im Programm verwendet werden kann. Eine Deklaration ist im wesentlichen die erste Zeile der Funktionsdefinition, ohne ihren Körper. Sie teilt dem Compiler den Namen der Funktion und die Typen der Argumente und des Rückgabewertes mit.

```
double power(double base, int exponent);
```

Anstelle des Anweisungsblocks tritt ein einzelnes Semikolon. Während die Angabe des Argumenttyps zwingend ist, ist die eines Namens für die Argumentvariable dem Benutzer freigestellt. Es ist jedoch guter Stil, hier Namen anzugeben, um die Bedeutung der Argumente klarzustellen.

Deklarationen können im Prinzip beliebig häufig im Programm wiederholt werden, sie sind sozusagen ein Versprechen an den Compiler, daß diese Funktion später oder in einem anderen Programmteil definiert wird. Funktionen sind immer globale Objekte innerhalb eines Namesraumes. Man kann Funktionen nicht wie Variablen innerhalb lokaler Bezugsrahmen definieren, z.B. innerhalb eines geschweiften Klammer-Paares (oder eines begin/end-Blockes wie in PASCAL).

Ein Programm, das von der Tastatur eine Zahl liest, dann `power` benutzt und schließlich das Ergebnis ausgibt, könnte so aussehen:

```
#include <iostream>
using std::cout;
using std::cin;

double power(double base, int exponent);  // declaration of 'power'

int main(){                               // definition of main
  double in;
  cin >> in;                              // read value from keyboard

  cout << "-3rd to 3rd power of " << in << ": ";
  for( int i= -3; i <= 3; i++ ){
    cout << power(in, i) <<  " ";         // call power and use return value
  }
  cout <<  "\n";

  return 0;                               // return to operating system
}

double power(double base, int exponent){  // definition of power
  double result = 1.;
  // ... program text as above

  return result;
}
```

Die Definition einer Funktion ist die Festlegung des Programmtextes, der bei Aufruf der Funktion ausgeführt werden soll. Geht die Definition einer Funktion ihrer Benutzung voraus, so kann sie eine Deklaration ersetzen. Die Spezifikation des Programmsegmentes für die Funktion erfolgt innerhalb eines geschweiften Klammerpaares, das *nicht* durch ein Semikolon abgeschlossen wird.

Die Übergabe von Argumenten an eine Funktion erfolgt "by value," d.h., daß in C vor dem Aufruf der Funktion Kopien der Argumente gemacht werden, die üblicherweise auf dem Stack abgelegt werden. Dann erfolgt der Aufruf der Funktion, die dann nur auf den Kopien dieser Variablen auf dem Stack operiert. Es können so niemals die Werte der Variablen im aufrufenden Programm verändert werden. Möchte man dies erzielen, z.B. in Funktionen, die Werte aus einer Datei oder Benutzereingaben einlesen, so benutzt man Zeiger- oder Referenztypen als Argumente, die wir später in Abschnitt 2.6 besprechen werden.

2.5.1 Funktionen ohne Argumente oder Rückgabewert, void

Das Schlüsselwort `void` wird wie ein Typ benutzt, ist aber kein wirklicher Typ. Er dient dazu, Funktionen zu kennzeichnen, die keinen Rückgabewert aufweisen. Dies ist machmal sinnvoll bei Funktionen, deren Wirkung darin besteht nur etwas auf dem Bildschirm auszugeben oder in einer bestimmten Weise ein Argument zu manipulieren.

```
void error(string message){
  cout << "error occured: " << message << "\n";
}
```

Auch Funktionen ohne Argumente können sinnvoll sein. In C++ wird eine Funktion ohne Argumente durch eine leere Argumentliste angedeutet. Ein typisches Beispiel ist ein Pseudozufallszahlengenerator, der bei jedem Aufruf eine andere Zahl zurückliefert. Die C-Bibiothek stellt einen (schlechten) Generator zur Verfügung, der so deklariert ist (in `stdlib.h`):

```
  void srand(unsigned int seed);
  int  rand(void);
```

Dieser wird so benutzt, daß man zunächst eine Sequenz von Zufallszahlen auswählt, durch Aufruf der Funktion `srand(.)` mit einer positiven ganzen Zahl. Danach erhält man bei jedem Aufruf von `rand()` eine neue, ganze Zahl zurück.

2.5.2 Bibliotheken

Wie wir oben gesehen haben, reichen Deklarationen dem Compiler als Information aus, um ein Programm zu übersetzen, und er erzeugt an der entsprechenden Stelle im Programm einen Funktionsaufruf. Allerdings muß dann natürlich später die Information hinzugefügt werden, welche Anweisungen denn wirklich mit diesem Aufruf ausgeführt werden sollen. Mit anderen Worten, wir müssen die übersetzten Funktionsdefinitionen "anbinden" (Computerdeutsch: linken). Dies geschieht durch Angabe des Namens der Programmbibliothek, oder der Objektdatei (`.o`), in der diese enthalten sind.

Ein klassisches Beispiel liefern die Funktionen der Mathematikbibliothek, hier `sqrt()`. Diese sind deklariert in `cmath` und ihre übersetzten Definitionen befinden sich in `libm.a`. Ein Programm, das eine solche Funktion benutzt, sieht vielleicht so aus:

```
// myprog.cc:
```

```
#include <cmath>            // declaration of (C) math functions
#include <iostream>         // declaration of (C++) I/O functions and objects
using std::cin;
using std::cout;

int main(){                 // program to compute the sqrt of a number
                            //  entered from the terminal
  double  d;
  cin >> d;
  if ( d < 0. )
    cout << "cannot compute the square root of negative numbers\n";
  else
                            // compute sqrt(d) and use value
    cout << "sqrt(" << d << ") = " << sqrt(d) << "\n";
}
```

Eine UNIX-Kommandozeile, um dieses Programm zu übersetzen und mit den Funktionen der Mathematikbibliothek zu linken, lautet:

```
g++ myprog.cc -o myprog -lm
```

Hier steht `-l` für “library” und `m` für die Datei `libm.a`, die die mathematischen Funktionen enthält; man findet die hinter `-l` anzugebenden Zeichen durch Weglassen von `lib` und der Endung `.a`

2.5.3 Operatoren

Operatoren liefern einen wesentlichen Anteil zur Lesbarkeit eines Programmes. Während wir ohne weiteres einen Ausdruck der Form `3 * z + 5` lesen können, bereitet die Schreibweise `plus(mult(3,z),5)` wesentlich größere Schwierigkeiten. Wir unterscheiden binäre Operatoren mit zwei Argumenten, wie die Multiplikations und Additionsoperatoren, und unäre mit einem, wie den Adressoperator `&` oder das unäre `-`. Seltener sind ternäre Operatoren (siehe Abschnitt 2.4.2).

Arithmetische Operatoren

Die arithmetischen Operatoren sind `+, -, *, /` und `%`. Der Operator `%` (modulo) bildet den Rest der ganzzahligen Division des linken mit dem rechten Operanden (beide ganzzahlig). Wenn beide positiv sind, ist das Ergebnis positiv und echt kleiner als der rechte

Operand. Für einen oder zwei negative Operanden ist das Vorzeichen des Ergebnisses implementationsabhängig. Die Kombination mit einem Gleichheitszeichen (Zuweisungsoperator) erspart die Wiederholung des linken Operanden:

```
int i=3, j=7;       // initialize
i    = 3 + j % i;   // i = 3 + (j mod i),   i is set to 4
i   += 3 * 4 + j;   // i = i + (3 * 4 + j), i is set to 23
i   /= 5;           // integer division, i is now 4
i   /= 5;           // integer division, i is now 0 !
```

Der Typ eines Operatorausdrucks hängt von den Typen der Argumente ab. Wenn beide Operanden vom Typ `T` sind, so ist auch das Resultat wieder vom Typ `T`. Bei verschiedenen Typen entscheidet der "bessere" Typ (sagen wir `U`), und es wird der `T` Operand erst in `U` gewandelt, bevor die Operation durchgeführt wird. Das Ergebnis des Ausdrucks ist dann vom Typ `U`. C/C++ führt folgende "type promotions" durch:

```
bool -> char -> short -> int -> long
             -> float -> double -> long double
signed -> unsigned  (!)
```

Insbesondere die Verwandlung von vorzeichenbehafteten Größen nach rein positiven führt häufig zu unliebsamen Überraschungen. Der Ausdruck `3U - 5` wird in `3U - 5U` gewandelt und ergibt ein implementierungsabhängiges Resultat, denn das Ergebnis der Subtraktion kann nicht in einer `unsigned int` Variablen gespeichert werden. Diese Regeln der automatischen Typ-Umwandlung gelten sinngemäß auch für die weiteren Operatoren.

Die unären Operatoren `++` und `--` gibt es in postfix und prefix-Form, und sie bewirken, daß der Operand um Eins inkrementiert, bzw. dekrementiert wird. In der postfix-Form ist der Wert der Operation der des Operanden *vor* der In/Dekrementierung, in der prefix-Form der Wert nach der Operation.

```
int i=5, j;
j = i++;        // j is 5 now (i before increment, i is incremented to 6
double a[20];
i=j;                             // i=j=5
std::cout << a[++j] << a[j++];   // prints a[6] twice,    j is now 7
std::cout << a[i++] << a[++i];   // prints a[5] and a[7], i is now 7

// j++ = i;     // ERROR: cannot assign to a temporary
// i = (j++)++; // ERROR: cannot apply ++ to a temporary
```

Vergleichs- und logische Operatoren

Für Operationen zwischen zwei integralen Datentypen stehen bitweise, logische und Schiebe-Operatoren zur Verfügung, für die C++ zur Unterstützung beschränkter Zeichensätze auch Schlüsselwörter reserviert. Auch diese Operatoren (siehe Tafel 2.1) lassen sich mit dem Zuweisungsoperator `=` verbinden.

`|`	`bitor`	bitweises oder
`&`	`bitand`	bitweises und
`^`	`xor`	bitweises ausschließendes oder
`<<`		Linksschieben, n-fach
`>>`		Rechtsschieben, n-fach
`~`	`compl`	Komplement, bitweises not, unär
`|=`	`or_eq`	bitweises oder, Zuweisung nach links
`&=`	`and_eq`	bitweises und, Zuweisung
`^=`	`xor_eq`	bitweises ausschließendes oder, Zuweisung
`<<=`		Linksschieben, n-fach mit Zuweisung
`>>=`		Rechtsschieben, n-fach mit Zuweisung

Tabelle 2.1: Bitmanipulationsoperatoren

Die logischen Operatoren sind `&&` (oder auch `and`) und `||` (`or`) und die Negation `!`. Die Operatoren `&&` und `||` haben die besondere Eigenschaft, daß die Auswertung eines Ausdruckes abgebrochen wird, sobald dessen Wahrheitswert feststeht (Sequencing). Diese Eigenschaft teilen sie mit dem Komma-Operator `,` — letzterer wird manchmal benutzt, um komplexe `for`-Schleifen zu konstruieren. Sein Wert ist der Wert des rechtsstehenden Ausdruckes.

```
double    a[20];
unsigned ind[5];

// safe, even if some ind[i] >= 20, since the last expression will not
//  be evaluated in that case
for (int i=0; i < 5 && ind[i] < 20 && a[ind[i]] >= 0 )
   sqrt(a[ind[5]]);

// sequence operator used to combine two expressions
int i, j;
for ( i=0, j=2; i < 18; i++, j++ )
   a[i] = a[j];

 // bizarre but legal use of ,
 i = 5*i, 3;    // i is set to 3, 5*i is computed, but discarded
```

Die (arithmetischen) Vergleichsoperatoren sind `==`, `!=`, `<`, `<=`, `>` und `>=`. Der Wert eines logischen Ausdruck und das Resultat einer Vegleichsoperation in C++ sind vom Typ `bool`, d.h. entweder `true` oder `false`. Dieser Typ ist frei in `int` konvertierbar und liefert dann eine 1 (true) oder 0 (false).

Weitere Operatoren

Der einzige ternäre Operator ist `?:`, der zur Selektion von Subausdrücken (gleichen Typs) dient. Falls der Ausdruck links von `?` wahr ist, wird der erste, sonst der zweite Ausdruck verwendet (siehe Abschnitt 2.4.2).

```
  int i;
  std::cout << (i > 0) ? i : 0;     // never prints a negative number
```

Der Operator `()` bewirkt Funktionsaufruf, `[]` dient als Indizierungsoperator für Zeiger und Felder, ein Punkt `'.'` zur Auswahl eines Elementes aus einer Struktur und `->` dem gleichen Zweck, wenn ein Pointer auf eine Struktur gegeben ist.

```
struct Person {
  string name;
  int age;
};

void f(){
  Person    p;
  Person *p_p = &p;
  p.age    = 25;
  p_p->age = 25;  // equivalent
}
```

Die unären Operatoren `*` und `&` dienen zum Finden des Wertes einer Zeigervariablen, bzw. zum Finden der Adresse einer (beliebigen) Variablen (siehe Kapitel 2.6).

Operatorausdrücke werden nach den in C üblichen Regeln für Rang und ggf. bei Mehrfachvorkommen mit der in Tafel 2.2 zusammengefaßten Assoziativität behandelt:

2.5.4 inline Funktionen

Folgende Operationen finden bei einem Funktionsaufruf und beim Rückkehren aus einer Funktion statt:

<table>
<tr><th>Operator</th><th>Auswertung von</th></tr>
<tr><td><code>::</code></td><td>links nach rechts</td></tr>
<tr><td><code>() [] -> . X++ X-- typeid XXX_cast<TYPE>()</code></td><td>links nach rechts</td></tr>
<tr><td><code>! ~ ++X --X + - * & (TYPE) sizeof new delete</code></td><td>rechts nach links</td></tr>
<tr><td><code>.* ->*</code></td><td>links nach rechts</td></tr>
<tr><td><code>* / %</code></td><td>links nach rechts</td></tr>
<tr><td><code>+ -</code></td><td>links nach rechts</td></tr>
<tr><td><code>>> <<</code></td><td>links nach rechts</td></tr>
<tr><td><code>< <= > >=</code></td><td>links nach rechts</td></tr>
<tr><td><code>== !=</code></td><td>links nach rechts</td></tr>
<tr><td><code>&</code></td><td>links nach rechts</td></tr>
<tr><td><code>^</code></td><td>links nach rechts</td></tr>
<tr><td><code>|</code></td><td>links nach rechts</td></tr>
<tr><td><code>&&</code></td><td>links nach rechts</td></tr>
<tr><td><code>||</code></td><td>links nach rechts</td></tr>
<tr><td><code>||</code></td><td>links nach rechts</td></tr>
<tr><td><code>= *= /= %= += -= <<= >>= &= |= ^=</code></td><td>rechts nach links</td></tr>
<tr><td><code>?:</code></td><td>rechts nach links</td></tr>
<tr><td><code>throw</code></td><td>–</td></tr>
<tr><td><code>,</code></td><td>links nach rechts</td></tr>
</table>

Tabelle 2.2: Bindungsstärke (Priorität) und Assoziativität von Operatoren

1. Sichern des lokalen Programmkontextes (z.B. Variablen, die in CPU-Registern gehalten wurden, Rücksprungadresse) auf den Stack.

2. Kopieren der Funktionsargumente (Pointer oder Klassen) aus dem lokalen Kontext auf den Stack, um Überschreiben aus der Funktion heraus zu verhindern.

3. Anspringen der Einsprungadresse der Funktion.

4. Ausführen der Anweisungen in der Funktion. Berechnung des Rückgabewertes.

5. Ablegen/Kopieren des Rückgabewertes auf den Stack.

6. Rücksprung in das aufrufende Programm, Wiederherstellen des lokalen Kontextes

Die Schritte 1, 5 und 6 stellen dabei Aufwand dar, der vermieden werden kann, indem man den Funktionsaufruf umgeht. Für kurze Funktionen, die in einem Programm sehr häufig aufgerufen werden, können diese Schritte einen signifikanten Anteil der Gesamtausführungszeit der Funktion darstellen. Diesen Zusatzaufwand umgeht man durch sogenanntes *Inlining* der Funktion; gewissermassen wird dabei der Funktionstext an der entsprechenden Stelle im ablaufenden Programm eingefügt. In C hat man an dieser Stelle häufig Makros verwendet, die aber den Nachteil unintuitiver Nebeneffekte haben.

```
#define SQR(a) ((a)*(a))
int   b=2,c;
c   = SQR(b++);  // c = ((a++) * (a++));
                 // c is now 2*3 = 6, b is 4 after 2(!) increments
```

Der Schritt 2 bleibt in der Regel nötig, weil natürlich auch weiterhin keine Variablen unkontrolliert verändert werden dürfen.

Wenn man in C++ im Header-file (siehe Abschnitt 2.10) eine Funktion *definiert* und mit dem Schlüsselwort `inline` kennzeichnet, so versucht der Compiler, den Funktionsaufruf durch Inlining wegzuoptimieren.

```
inline int sqr(int a){ return a*a; }
int  b=2, c;
c   = sqr(b++);  // b++ is evaluated once, hence c is now 4, b is 3
```

Da der Nachteil des Inlining vergrößerte Programme sind, sollte man mit diesem Mittel sparsam umgehen. Lange Funktionen gehören in Implementierungsdateien und nicht in die header-Dateien. Klassen-Elementfunktionen, die im Klassenkörper definiert werden, behandelt der Compiler automatisch als inline. Zusammenfassung:
(i) inlining ist gut für kurze Funktionen, die effizient gemacht werden müssen;
(ii) inline Funktionen müssen im header definiert werden;
(iii) Elementfunktionen, die im header definiert sind, sind per default inline.

2.5.5 Defaultargumente

Häufig stellt sich die Frage, wie man eine Funktion "erweitern" kann, ohne die Funktionalität eines bestehenden Programms, das diese Funktion verwendet, zu verändern, oder wie man eine Funktion, die zur vollständigen Charakterisierung mehrere Argumente benötigt, einfach bedienbar gestalten kann. Diese Aufgaben können mit Defaultargumenten gelöst werden. Beispielsweise könnte eine Funktion zum Löschen eines Feldes so aussehen:

```
void clear(int size, int *array){
  for (int i=0; i < size; i++ )
     array[i] = 0;
}
```

Diese läß sich später leicht zu einer Funktion erweitern, die das Feld mit einem beliebigen Wert füllt:

```
void clear(int size, int *array, int value=0){
  for (int i=0; i < size; i++ )
     array[i] = value;
}
```

Ein Defaultargument muß einen konstanten Wert besitzen, der zur Compilezeit feststeht und eindeutig ist. Das bedeutet, daß dieser in jeder Übersetzungseinheit nur einmal spezifiziert werden darf. Wenn sichergestellt ist (z.B. mit include guards im header-file), daß der Compiler nur eine Deklaration der Funktion sieht, dann ist der beste Platz, um den Defaultwert festzulegen, in einer Funktionsdeklaration in einem header-file. Ansonsten sollte dieser Wert bei der Definition der Funktion festgelegt werden.

2.5.6 Überladen von Funktionen

Im Gegensatz zu C und einigen anderen Sprachen sind in C++ die Argumenttypen zusätzlich zum Funktionsnamen relevant bei der Auswahl der aufzurufenden Funktion. Es ist möglich, eine `print`-Funktion für `int` zu haben, eine für `double` und eine weitere für `struct Date` {...}. Die Mehrfachverwendung des Funktionsnamens erlaubt, diesen von Typinformation freizuhalten und sich auf die Funktion zu beschränken. Als C-Gegenbeispiel können hier `sprintf` und `fprintf` dienen, in denen `f` und `s` den Typ des ersten Argumentes angeben. Diese Häßlichkeit (die darüberhinaus noch zu Fehlern führen kann, wenn beispielsweise der `FILE *` einmal durch `char *` ersetzt wird) wird in C++ vermieden; der Compiler wählt automatisch die richtige Funktion aus.

Man kann sich vorstellen, daß der Compiler dafür sorgt, daß gewissermaßen programmintern komplexere Namen für Funktionen verwendet werden, die neben dem Funktionsnamem auch noch die Argumente enthalten.

```
void swap(int &a, int &b){   // internal name e.g. swap_int_int
  int tmp = a;
  a = b; b = tmp;
}

void swap(double &a, double &b){     // swap_double_double
  double tmp = a;
  a = b; b = tmp;
}

void f(){
  double a, b;
  int    i, j;
```

```
    // ...
    swap(a, b);  // double version
    swap(i, j);  // int    version
  }
```

Um zu verhindern, daß diese internen Namen auch für Funktionen erzeugt werden, die aus C-Bibliotheken geladen werden, müssen diese für den Compiler sichtbar als `extern "C"` deklariert werden.

```
  extern "C" double sqrt(double);
```

Dieser Mechanismus erlaubt dem Compiler sowohl das Erzeugen der richtigen Namen als auch die Berücksichtigung anderer Eigenarten der Sprache, deren Prozeduren der Linker an das Programm binden soll. Bei einem Funktionsaufruf kann z.B. die Reihenfolge, in der Parameter auf dem Stack abgelegt werden eine andere als die in C++ sein.

Durch Überladen läßt sich ein ähnlicher Effekt erzielen wie mit Defaultargumenten, nämlich, daß nachträglich weitere Versionen der "gleichen" Funktion mit erweiterten Argumentlisten bereitgestellt werden können. Nachteil des Überladens zu diesem Zweck ist, daß dabei der Programmcode dupliziert wird, was zu größeren Programmen führt und größere Disziplin bei der Unterhaltung des verdoppelten Codes erfordert (Fehler (bugs) müssen in beiden Versionen entfernt werden).

Falls `@` für einen eingebauten Operator steht, so erlaubt C++ auch, diesen als `operator@()` anzusprechen. Der Ausdruck `3 * z + 5` könnte also auch, weniger intuitiv, geschrieben werden als `operator+(operator*(3,z),5)`.

In dieser Syntax lassen sich Operatoren genau wie Funktionen überladen. Dies kann dazu dienen, eine natürliche Syntax für benutzerdefinerte Typen wie z.B. Matrizen oder komplexe Zahlen einzuführen.

2.6 Zeiger, Zeiger-Feld-Dualität, und Referenzen

2.6.1 Zeiger

Vom Standpunkt des Computers betrachtet ist jede erreichbare Speicherzelle im (virtuellen) Speicher durch eine eindeutige Zahl gekennzeichnet (Teile des virtuellen Adressbereiches können auf Harddisk gehalten sein, andere können im RAM liegen, wieder andere, gegenwärtig unbenutzte, müssen vom Betriebssystem gar nicht repräsentiert sein). Auf regulären PC's liegen diese Adressen im Wertebereich normaler `int` Variablen, auf anderen