Kapitel 2

Grundlagen von C++

Wie am Beispielprogramm in der Einführung zu sehen, besteht ein Programm aus den unterschiedlichsten Elementen und Konstrukten. In den folgenden Abschnitten werden nach und nach die grundlegenden Sprachelemente vorgestellt.

2.1 Sprachkonstrukte

Das erste Konstrukt sind *Kommentare*. Sie haben keinen Einfluß auf den Programmablauf (oder den Computer), sondern sollten lediglich zum besseren Verständnis des Programms eingefügt werden. Als Kommentar wird jener Text angesehen, der rechts der Zeichen '`//`' steht.

Das zweite wesentliche Element sind *Daten*, also z.B. die physikalischen Parameter die man dem Computer zur Berechnung gibt, Konstanten wie die Zahl π, oder Indices $i = 1, ..., n$, die zum Zählen verwendet werden. Man muß dem Computer zuerst mitteilen, welche Daten, Variablen und Konstanten man benötigen wird.

Operatoren und *Funktionen* sind nötig, um Daten zu manipulieren. Beispiele für Operatoren sind die Grundrechenarten, Beispiele für Funktionen sind trigonometrische Funktionen wie Sinus und Cosinus. Funktionen sind nicht nur Sprachkomponenten, sondern können auch vom Programmierer dazu verwendet werden, häufig wiederkehrende Manipulationen zusammenzufassen und dadurch übersichtlich ins Programm einzubauen.
Wichtig ist, daß das sog. Hauptprogramm im Prinzip ebenfalls eine Funktion ist, die immer `main` genannt werden muß. Der Beginn und das Ende einer Funktion sind durch '{' und '}' gekennzeichnet.

Die daneben wichtigsten Sprachelemente sind diejenigen für die Kommunikation zwischen Benutzer und Computer, kurz die der *Ein- und Ausgabe*, die häufig auch mit der aus dem

Angelsächsischen abgeleiteten Abkürzung I/O für Input/Output belegt werden. Im Sprachumfang von C/C++ sind genormte Bibliotheksfunktionen enthalten, die die Ausgabe auf den Bildschirm oder in Dateien regeln. Ebenso wird die Eingabe per Tastatur, Maus oder andere angeschlossene Geräte (externe Meßgeräte, Trackball, etc.) behandelt. Dies wird durch spezielle Bibliotheken erledigt, die je nach der vorhandenen Rechnerhardware und -software verschieden sein können und daher auch nicht allgemein behandelt werden können.

Wie man sieht, sind Dateien eine zweigleisige Möglichkeit mit einem Programm zu kommunizieren und Daten sowohl ein- als auch auszugeben.

Zuletzt seien noch *Organisations*-elemente erwähnt, die nicht direkt zur Programmiersprache gehören, sondern von einer Vorstufe des Compilers verarbeitet werden, dem sog. Präprozessor. Diese Organisationselemente erlauben es, unübersichtliche Programmelemente auszulagern oder Programmteile einfach auszublenden.

Das einfachste denkbare Programm gibt den Text "Hello World!" auf den Bildschirm aus:

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

int main()
{
   cout << "Hello World!\n" ;  // print a string to the screen
   return 0;
}
```

- Die erste Zeile `#include <iostream>` holt die Definitionen aus der Standard-Bibliothek. Diese ist nötig für die Ein- und Ausgabe von Daten und Text. Diese Zeile muß am Anfang eines jeden Programms stehen, das I/O Operatoren wie z.B. '`cin`', '`cout`', '`<<`' oder '`>>`' enthält.
- Die Anweisung `using std::cout` importiert das Symbol `cout` aus dem Namensraum `std`. Namensräume werden in C++ benutzt, um Algorithmen in verschiedenen Bibliotheken voneinander zu trennen. Ein `cout` aus anderen Namensräumen (evtl. selbst definierte) kann nun nicht mehr verwendet werden. Wir werden häufig auf die C++-Standardbibliothek zurückgreifen, die den Namensraum `std` benutzt. Zur Benutzung von C-Funktionen ist keine `using`-Direktive nötig.
- `main()` ist eine Funktion vom Typ `int`. Jedes Programm besteht aus zumindest dieser einen Funktion, kann aber auch aus einer beliebigen Anzahl zusammengesetzt sein. Dabei verliert `main()` allerdings nie seine besondere Rolle.
- `return 0` gibt den Wert 0 an das aufrufende Programm, also in diesem Fall an das Betriebssystem zurück. Man kann von Null verschiedene Werte verwenden um Fehler oder Erfolg der Funktion anzuzeigen.

- Die Klammern `{}` schließen das Hauptprogramm, eine Funktion, oder allgemeiner, Programm-*Blöcke* ein. Zu jeder Beginn-Klammer `{` gehört eine Ende-Klammer `}`.
- Die Klammern `()` umschließen eventuelle Argumente, die an `main` übergeben werden.
- Die Anführungszeichen `"..."` umschliessen Text (Feld von `char`'s).
- Die Zeichenkombination '`\n`' führt bei der Ausgabe zu einer neuen Zeile.
- `cout` bezeichnet das Standard-Ausgabe Medium, im Normalfall also den Bildschirm. Im ANSI Standard würde man an dieser Stelle `std::cout` verwenden.
- Die Zeichenkombination `<<` wird als Ausgabeoperator benutzt und gibt alles rechts von ihr stehende "nach links", auf `cout`, also auf dem Bildschirm aus.
- *Jede* Anweisung muß mit einem Strichpunkt '`;`' beendet werden, denn eine (optisch) neue Zeile im Programm wird vom Compiler nicht als neues Programmelement erkannt. Man trennt einzelne Befehle also durch '`;`', sollte aber vermeiden, dies innerhalb einer Zeile zu tun. Das Semikolon hat die Funktion der Trennung zweier Anweisungen, daher steht es nicht am Ende von *Anweisungs*blöcken, die durch geschweifte Klammern `{}` gekennzeichnet sind.

2.2 Kommentare

Kommentare werden durch `//` eingeleitet und erstrecken sich bis zum Ende der Zeile. Zeilenumbrüche werden ansonsten wie Leerzeichen behandelt und haben keine syntaktische Relevanz. Die C-Kommentarsyntax `/* comment */` gilt auch weiterhin in C++. `/* */` Kommentare lassen sich nicht schachteln, und sind daher zum "Auskommentieren" größerer Programmbereiche ungeeignet. Falls nämlich im auskommentierten Programmbereich bereits ein C-Kommentar vorhanden gewesen sein sollte, so wird die erste schliessende `*/`-Kombination den gesamten Kommentarbereich beenden und vermutlich dazu führen, daß ein Teil des Programmes wieder eingeblendet wird.

Große Programmbereiche sollten mit Präprozessordirektiven ausgeblendet werden (gekennzeichnet durch `#`). Auch sollte man C++-Kommentare in Zeilen vermeiden, die Präprozessordirektiven enthalten, da der Präprozessor aus C-Zeiten möglicherweise nur C-Kommentarsyntax korrekt behandelt.

```
#if 0       /* get rid of the main program for the moment */
int main()
{            // main is THE keyword, the program starts here
   for ( int i=0 /* loop variable */; i< 10 ; i++ )
   {                                            // for loop
```

```
        ;                                    // do nothing else
    }                                    // end of the for loop
}
#endif                   /* end of getting rid of main... */
```

2.3 Datentypen, Variablen, Konstanten

2.3.1 Grundlegende Datentypen und Deklaration

Die grundlegenden, eingebauten Datentypen von C++ sind `bool, char, int, float, double`. Diese Datentypen werden besonders effizient vom Computer bearbeitet.

Der Typ `bool` wird für die Resultate von Vergleichsoperationen (2.5.3) benutzt. Er kann die Werte `true` oder `false` besitzen. Weiterhin ist er frei in den `int` Datentyp umwandelbar, wobei `true` einer 1 und `false` einer 0 entspricht. Diese Umwandlung stellt die Aufwärtskompatibilität zu alten Programmen her, die statt `bool` einfach `int`'s verwendeten. Einige einfache Deklarationen sind:

```
bool    test;                    // declare test
test  =(0 < 5);                  // comparison with result true
                                 // test is set to true
int     i(test);                 // i is declared and set to 1
```

Die Initialisierung/Konstruktion einer Variablen kann in der Form *type variable = type_constant* oder äquivalent durch *type variable(type_constant)* erfolgen. Das Gleichheitszeichen an dieser Stelle ist keine Zuweisung, da die Variable gerade erst ins Leben gerufen wird und nicht bereits im Programm existiert.

Der Typ `int` ist der Datentyp, der auf der jeweiligen Maschine vom Compilerschreiber als der für die Verarbeitung integraler Daten als "geeignetster" empfunden wurde. Über seine Größe ist im Standard nichts festgelegt. Mit Hilfe des `sizeof` Operators kann man seine Größe in Einheiten der Größe eines `char` erfragen, der gleichzeitig der kleinste Datentyp ist. Ein `char` wurde vom Compilerschreiber als der für Einzel(schrift)zeichen geeignete Typ gewählt. Verschiedene Rechner unterstützen noch weitere ganzzahlige Datentypen, z.B. neben 32-bit `int` auch 16 oder 64-bit große Variablen. Diese lassen sich ggf. Deklaration von `short` und `long` oder gar (nicht ANSI konform) `long long` ansprechen. Nach diesen Schlüsselworten darf `int` fehlen. Man beachte, daß eine Integer-Variable, immer einen ganzzahligen Wert hat – auch wenn man ihr eine reelle Zahl zuweist. Beispiele für `int`eger Deklarationen sind:

```
bool    test;                      // declare test
```

```
test  =(0 < 5);                // (see above)
int     i(test);               // i is declared and set to 1
int     j=0;                   // j is declared and set to 0
long    l, n, m;               // l, n, and m are declared "long"
short   s1, s2;                // s1 and s2 are declared "short"
cout << sizeof(test) << " "
     << sizeof(i) << " "
     << sizeof(l) << " "
     << sizeof(s1) << "\n";    // check size of: test, i, l, s1
                               // Note: this is not guaranteed
                               // on my machine I get: 8 4 8 2
                               // ANSI should give:    1 (for bool)
i = 37;
j = 3.5674;                    // ATTENTION
cout << i << " " << j << "\n"; // this leads to: 37 3
```

`float` und `double` sind Gleitkommatypen einfacher und doppelter Genauigkeit, die zu numerischen Zwecken geeignet sind. `long` kann zur Modifikation von `double` verwendet werden, um vierfache Genauigkeit zu erhalten (falls diese unterstützt sein sollte). Die exakte Gleitkommarepräsentation ist abhängig von der Hardware-Unterstützung. Es ist heute (1998) üblich, für doppeltgenaue Gleitkommazahlen zumindest die Grundrechenarten innerhalb der CPU in Rechnerhardware durchzuführen. Einfachgenaue Zahlen müssen dazu erst in doppeltgenaue gewandelt werden. Aus diesem Grund und der allgemein hohen Anforderungen numerischer Rechnungen sollte man `float` nur noch dort verwenden, wo dies durch z.B. Speicherplatzbeschränkungen erfordert wird.

C/C++ legt nichts über die wirkliche Repräsentation der Variablen im Speicher fest. Insbesondere ist der Speicherbedarf einer Variablen nicht definiert, es gelten aber immer die Beziehungen:

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

und

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

Im allgemeinen sind auf UNIX-Arbeitsplatzrechnern `int`-Variablen 32, `short` 16 und `char` 8 bit lang. Doppeltgenaue `double`s erfordern 64 bit, `float` 32 bit. Die wirklichen Werte können jedoch im Prinzip sogar von Compiler zu Compiler verschieden sein!

Die folgenden Beispiele deklarieren verschiedene Datentypen.

```
#include <string>
using std::string;
```

```
char      c1 = 'x';       // one character: 'x'
char    c2[] = "hello";   // text array with 5 characters
                          //   (note: length is 6 due to added '\0')
string    str("hello");   // the C++ way, arbitrary length strings
                          //   (note: length is 5 - no added '\0')
char      c3 = '\n';      // the new-line character
int       i_var;          // integer variable with name i_var
long int  il_var = 1L;    // long constant
short     is_var;         // short integer variable (int per default)
double    d_var = 34.2;   // real number 34.2 with double precision
```

Bei `char` Variablen ist zu beachten, daß einzelne Zeichen mit einem einfachen `'` eingeschlossen werden können, längere Zeichenketten jedoch mit dem doppelten `"` begrenzt werden müssen. Die Variablen `c1` und `c3` haben beide die Länge Eins, wohingegen die Zeichenkette `"hello"` im "Feld" `c2` abrufbar ist und die Länge Sechs hat, da automatisch ein Zeichenkettenende `'\0'` angefügt wird. (Die Definition `char c2 = 'hello';` führt zu einer Fehlermeldung). Flexibler ist die C++ Methode, `string`s zu benutzen, die den Zeichenkettenterminator `'\0'` nicht benötigen.

Variablendeklaration sind in C++ nicht auf den Anfang eines Blockes beschränkt (wie in C), sondern können überall erfolgen, wo die Variable erstmals an einer Stelle auftaucht und wo auf sie zugewiesen werden kann. Wichtig ist insbesondere die Deklaration innerhalb einer `for(;;)` Schleife

```
double a;
cin >> a;              // input a via keyboard
int    i_trunc = a;    // throw away the fractional part
for ( int i=0; i < i_trunc; i++ )
{
   ...
}
```

Im obigen Programm ist `i` außerhalb der `for(;;)` Schleife (d.h. nach der letzten zugehörigen geschlossenen Klammer, oder der letzten Anweisung des Schleifenkörpers) nicht mehr existent.

Zusammenfassung
(i) Deklarationen können an fast beliebiger Stelle im Programm stehen
(ii) Initialisierung erfolgt in `()` oder `=` Form; es findet *keine Zuweisung* sondern eine Initialisierung statt!
(iii) `sizeof(type)` kann eingesetzt werden, wenn die Größe eines Typs im Programm bekannt sein muß, dies ist für die Portabilität zwischen verschiedenen Rechnern interessant.

2.3.2 Variablenattribute

Neben den Attributen `short`, `long` können ganzzahlige Variablen noch durch `signed` und `unsigned` modifizert werden, die andeuten, daß entweder ganze ($\in \mathcal{Z}$) oder natürliche Zahlen ($\in \mathcal{N}_0$) repräsentiert werden sollen. Gleichermaßen kann auch hier das `int` der Typdeklaration entfallen.

2.3.3 Lebensdauer und Lage der Variablen im Systemspeicher

`auto` Falls keine weitere Angabe erfolgt, sind Variablen `auto`matic. Dies bedeutet, daß sie angelegt werden, sobald das Programm über die Zeile läuft, in der sie deklarariert sind. Sobald das Programm den Block verlässt, in dem sie deklariert sind, wird der zugehörige Speicher wieder freigegeben. Der Block ist im allgemeinen das innerste die Variable umschließende Paar geschweifter Klammern, oder wie im Fall der in den ()-Bereichen von Anweisungen wie `for(;;)` deklarierten Variablen, das Ende dieser Anweisung. In diesem Fall wird der Speicherplatz freigegeben, sobald diese Anweisung vollständig abgearbeitet ist. Variablen dieser Art werden vom Compiler meisst im sogenannten Stack-Bereich des Programmes angelegt.
Der Inhalt einer solchen Variable ist zufällig, wenn sie nicht explizit im Programm initialisiert wurde. Da dies zu subtilen Fehlern führen kann, sollte man solche Variablen immer entweder durch eine explizite Initialisierung auf einen passenden Wert setzen oder möglichst unmittelbar vor der ersten Verwendung deklarieren (so daß sichtbar ist, daß die Variable angemessen – z.B. in einer Eingabe – verwendet wird).

`static` Das Schlüsselwort `static` bewirkt, daß eine Variable während der gesamten Programmlebensdauer existiert. Diese Variablen werden vom Compiler vor Programmbeginn meist auf dem heap-Speicher angelegt und, sofern nicht anders angegeben, auf den Wert 0 initialisiert. Eine `static` Variable innerhalb einer Funktion hat daher den Inhalt 0, wenn die Funktion das erste Mal angesprungen wird und behält zwischen Funktionsaufrufen den jeweils zuletzt zugewiesenen Wert. Eine automatische Variable wird hingegen jedes Mal neu angelegt und ggf. initialisiert. Die folgende Funktion zählt, wie häufig sie aufgerufen wurde, und gibt diesen Wert zurück:

```
int f(){
  static int count;  // 0 at first function call
  return ++count;    // increment before use as return value
                     // (i.e. returns one on the first call)
}
```

`const` Variablen, die `const` erklärt werden, dürfen nach der Initialisierung nicht mehr verändert werden. Dies kann dazu dienen, in einen Namensraum Konstanten einzuführen. Der Compiler kann diese u.U. bereits zur Compilezeit wegoptimieren. Diese Technik ist der unter C weitverbreiteten Gewohnheit, solche Ersetzungen durch den Präprozessor machen zu lassen, vorzuziehen, da der Präprozessor keine Typenüberprüfung vornehmen kann. Die konsequente Benutzung von `const` erlaubt dem Compiler weitere Optimierungen, so daß man Variablen und Funktionsargumente sorgfältig daraufhin überprüfen sollte, ob es möglich, sie als `const` zu erklären.

```
const int    Size = 100;
const double Pi   = 3.1415926;   // When including cmath on can
                                 // use the predifined constant M_PI
```

`volatile` dient dazu, den Compiler darauf hinzuweisen, daß eine Variable ihren Wert ändern kann, auch ohne daß eine Zuweisung oder ein anderweitiger Programmzugriff erfolgt ist. Dies Verhalten ist typisch für hardware-Register, die im Adreßraum des Programmes liegen. Der Compiler muß in diesem Fall von zu agressiver Optimerung abgehalten werden.

2.3.4 Felder

Felder dienen der Zusammenfassung mehrerer Variablen gleichen Typs in einem zusammenhängenden Speicherbereich. Auf die einzelnen Elemente des Feldes kann mit Variablen eines ganzzahligen Typs als Index zugegriffen werden. Sie erlauben einen direkten Zugriff auf große Datenmengen und werden in Programmiersprachen verwendet, um z.B. Matrizen zu implementieren, viele Daten gleichen Typs und gleicher Bedeutung zu speichern und z.B. einfache Such-Operationen zu ermöglichen. Häufig (aber nicht immer) ist das Vorkommen einer indizierten Variable in einer mathematisch-formalen Beschreibung eines Problems ein Hinweis darauf, daß diese in einem Computerprogramm durch ein Feld repräsentiert werden kann.

Felder werden in C/C++ mittels `[]` deklariert. Gleichzeitig dient dieser Operator auch zum Zugriff auf Feldelemente. Felder beginnen in C/C++ immer mit dem Index 0, der letzte gültige Index hat daher den Wert der um Eins verringerten Größe des Feldes. Mehrdimensionale Felder werden durch mehrfache, nicht verschachtelte `[]`-Paare deklariert.

Felder haben in C/C++ *feste* Größen, d.h., daß die in Klammern stehende Feldgröße ein Ausdruck sein muß, der nur Konstanten oder zum Zeitpunkt der Compilation feststehende `const` deklarierte Variablen umfaßt.

```
#include <string>
```

```
using std::string;

// array of 20 int's
int  a[20];

// array of three strings, initialized to some values:
string  stra2[] = { "Matthias", "Stefan", "Oliver" };

// two dimensional array of 20x30 elements of int type
int  ia[20][30];
// initialization in a for loop
for ( int i=0; i < 20; ++i )
  for ( int j=0; j < 30; ++j )
    ia[i][j] = i * j;
```

Wir werden später noch sehen, daß zwischen Feldern und den Zeigervariablen (Pointern) in C/C++ ein sehr enger Zusammenhang besteht, der uns an der Stelle auch erlauben wird, "dynamische" Felder anzulegen, d.h. solche, deren Größe erst während der Laufzeit des Programmes, zum Beispiel durch eine Benutzereingabe, ermittelt wird und ggf. auch dynamisch geändert werden kann.

2.4 Kontrollstrukturen

Für die Umsetzung von Algorithmen in einer bestimmten Programmiersprache ist es essentiell, daß diese Sprache Kontrollstrukuren aufweist, die in der Lage sind, bestimmte Anweisungsblöcke nur unter bestimmten Bedingungen oder mehrfach zu durchlaufen. Die in C (und C++) meistverwendeten Kontrollstrukturen sind das `if`-Statement zur bedingten Ausführung und die `for`-Schleife zur Wiederholdung von geblockten Anweisungen. Daneben existieren noch die abweisenden `while()`- und die nichtabweisenden `do{...}while()` Schleifen, das `switch()...case:`-Statement, das aufgrund des Inhalts einer Variablen einen auszuführenden Block anspringt und die unbedingte Sprunganweisung `goto`, die sich manchmal sinnvoll dazu verwenden läßt, aus tief verschachtelten Schleifen herauszuspringen.

2.4.1 if-Anweisung

Die `if`-Anweisung dient dazu, bestimmte Programmteile nur unter bestimmten Bedingungen zu durchlaufen. Eine solche Anweisung kann, muß aber keinen `else`-Teil enthalten, der durchlaufen wird, wenn der Bedingungsteil `false` ist. Der `else` Teil kann aus beliebig vielen `else if` und einem `else` bestehen. Sowohl `if`, `else if`, als auch `else`-Zweige können entweder einzelne Anweisungszeilen (abgeschlossen durch Semikolon) oder in Klammerpaare `{}` gesetzte Anweisungsblöcke sein. Ein `else`-Block bezieht sich immer auf die gerade vorstehende, aktuelle `if`-Anweisung. Im folgenden Beispiel ist die zweite `if`-Anweisung ein einzelnes Statement im `else`-Teil der ersten.

```
#include<iostream>
using std::cout;
#include<cmath>
// ...

  // solution of quadratic equations  x^2 + p x + q = 0

  double p= 2., q= 3.;            // or other values
  double sol1, sol2;
  double p= 2., q= 3.;            // or other values
  double sol1, sol2;
  double root_arg = 0.25*p*p - q; // calculate the argument for the root

  if ( root_arg > 0. ) { // begin of if-block
    sol1 = 0.5*p + sqrt(root_arg);
    sol2 = 0.5*p - sqrt(root_arg);
  }                               // end if-block
  else if ( root_arg < 0. )       // 'error' condition, 'else' part
    cout << "no real solution exists";
  else                            // refers to last if(root_arg < 0.)
    sol1 = sol2 = 0.5 * p;        // assignment from right to left

  if ( sol1 > 0. )
    cout << "graph cuts positive x-axis";
                                  // ok, else branch missing
```

2.4.2 Ternärer ?:-Operator

Als Einschub und der Vollständigkeit halber soll hier der ternäre Operator `?:` behandelt werden. Ternär bedeutet, daß der Operator drei Argumente benötigt. Dieser kann dazu dienen, innerhalb von Ausdrücken Teilausdrücke bedingt auszuwerten.

```
inline int abs(int number) {
  // compute absolute value
  return number > 0 ? number : -number;
```

```
}
```

Ist der vor dem Fragezeichen stehende Ausdruck `true`, so wird der Teilausdruck vor dem Doppelpunkt, sonst derjenige nach dem Doppelpunkt ausgewertet. Der resultierende Typ der beiden alternativen Ausdrücke muß gleich sein. Aufgrund der geringen Priorität dieses Operators und zur Verdeutlichung der Zusammengehörigkeit der Teilausdrücke sollte man sich angewöhnen, großzügig Klammern zu setzen.

```
int i1   =       3 > 4 ? 0 : 1;    //  i1   is 1
int i2   = 3 * ( 3 > 4 ? 0 : 1 );  //  i1_3 is 3
int i3   = 3 *   3 > 4 ? 0 : 1;    //  i2_3 is 0, since * binds
                                   //    stronger than >
```

Dieser Operator kann immer durch eine `if`-Anweisung vermieden werden, kann aber in vereinzelten Situationen klarer sein. Da er anderenfalls eine der zahlreichen Möglichkeiten in C/C++ ist, seine Programme für andere erfolgreich unlesbar zu gestalten, sollte man ihn sparsam einsetzen.

2.4.3 (do) while-Schleifen

Schleifen, die mit `while` gebildet werden, werden so oft durchlaufen, bis die Bedingung im Bedingungsteil der Anweisung nicht mehr erfüllt ist. Sollte diese Bedingung bereits falsch sein, wenn das Programm auf die `while`-Schleife trifft, so wird der Schleifenkörper nicht durchlaufen.

```
char c=0;                       // 'loop initialization'
while( c != 'y' && c != 'n' )   // first time c==0, so we enter the loop
  std::cin >> c;                // can also be a block in { }
```

Das obige Beispiel erfordert die Initialisierung der Zeichenvariablen mit einem Wert, so daß die Schleife auf jeden Fall durchlaufen wird. In solchen Fällen ist häufig die Verwendung einer nicht abweisenden Schleife mit `do ... while(...)` intuitiver:

```
char c;
do {
  std::cin >> c;
} while( c != 'y' && c != 'n' );
```

2.4.4 for-Schleifen

Wie sich oben bereits andeutete, enthalten Schleifen neben der Abbruchbedingung häufig noch einen Schleifeninitialisierungsteil und möglicherweise Anweisungen, die den Wahrheitsgehalt der Schleifenbedingung beeinflussen und jedesmal während eines Durchlaufes ausgeführt werden.

Bei der Arbeit mit Feldern benötigt man häufig Schleifen, die die Feldelemente einzeln ansprechen. Solch eine Schleife, programmiert mit `while`, könnte etwa so aussehen:

```
const int Size = 20;
// ..
int a[Size];
// ..
int i=0;                // loop initialization
while ( i < Size ) {    // loop condition, rejecting if false
  a[i] = i;
  i++;                  // loop control, performed after each pass
}
```

Zur Abkürzung läßt sich in diesem Falle `for(;;)` verwenden. Die folgende Konstruktion mit `for` ist der obigen (bis auf den Sichtbarkeitsbereich der Variablen `i`) vollständig äquivalent, aber übersichtlicher und kürzer.

```
int a[20];
// ..
for( int i=0; i < 20; i++ )
  a[i] = i;            // work on elements 0 .. 19
```

Diese Formulierung besitzt zwei Vorteile. Erstens ist die Schleifenkontrolle in den drei durch Semikolon getrennten Teilen der `for`-Anweisung gut sichtbar zusammengefaßt und nicht über den Schleifenkörper verstreut. Zweitens ist der Verwendbarkeitsbereich der Variablen `i` auf die Schleifenkontrollstruktur und den Schleifenkörper beschränkt.

2.4.5 break und continue

Aus einer einzigen nicht verschachtelten Ebene einer mittels `for` oder `while` gebildeten Schleife kann man durch die Benutzung von `break` herausspringen. Die Schleifenbearbeitung wird abgebrochen und das Programm direkt nach der Schleife fortgesetzt, ohne noch weitere Kontrollanweisungen oder Schleifentests zu beachten. Das `break` kann also z.B. dazu dienen, Schleifen frühzeitig abzubrechen oder “Endlosschleifen” zu verlassen.

```
// copy input to output
while( true ) {
  char c;
  cin >> c;
  if ( cin.fail() ) break;   // end of input or something wrong
  cout << c;
}
```

Die `continue` Anweisung bewirkt, daß die Bearbeitung des Schleifenkörpers an dieser Stelle abgebrochen und dann je nach Art der Schleife bei der Kontrollanweisung (`for(;;)`), bzw. dem Schleifentest (`(do{}) while()`) wieder aufgenommen wird.

```
#include <cmath>              // for sqrt() function

double a[20];
// ...
double sum_sqrt=0.;
for( int i=0; i < 20; i++ ){
  if ( a[i] < 0. ) continue;   // work on next element
  sum_sqrt += sqrt( a[i] );
}
```

Es sei nochmals betont, daß sich `continue` und `break` nur auf die innerste Schleifenebene beziehen und nicht dazu benutzt werden können, tiefer verschachtelte Schleifen zu verlassen.

2.4.6 goto-Anweisung, Labels

Zu dem Zweck, tiefer verschachtelte Schleifen zu verlassen, kann die `goto` Anweisung benutzt werden.

```
void f(){
  int a[20][30];

  for( int i=0; i<20; i++ )
    for( int j=0; j <30; j++ )
       if ( a[i][j] == 42. )  // found the answer
         goto end_of_loop;    // multi level break
end_of_loop:                  // label, we jump here from inside the loop
```

Der Label (Name) `end_of_loop` kennzeichnet das Sprungziel des `goto`. Der Name des Sprungzieles kann innerhalb des Funktionskörpers frei gewählt werden und wird durch einen Doppelpunkt abgeschlossen. Das Problematische an `goto`-Anweisungen ist, daß man einem Label nicht ansehen kann, *von wo aus* sie mit `goto` angesprungen werden. Außerdem erzwingt häufig die Verwendung eines `goto` die Verwendung mindestens eines weiteren. Im obigen Beispiel müßte vermutlich nach der Schleife eine Behandlung des "nicht-gefunden" Falles anschließen und man müßte dann das Ansprungziel des `goto` mit einem weiteren `goto` umgehen. Besser setzt man oben eine Variable `found` auf einen bestimmten Wert und benutzt diesen dann in weiteren Verzweigungsanweisungen.

Mit `goto` dürfen lokale Blöcke/Klammerebenen verlassen, aber nicht betreten werden. Man darf beispielsweise nicht vom `if`-Teil einer Entschiedung in den `else`-Teil springen, oder von außen in einen Block/Schleifenkörper hinein. Ebenso darf der Kontext einer Funktion nicht verlassen werden.

Programmierweisheit besagt: `goto`'s können immer vermieden werden (im Prinzip durch Ersetzen des `goto` durch die Anweisungen nach dem Sprungziel), sind allerdings in seltenen Einzelfällen bei der Strukturierung hilfreich.

2.4.7 switch-Anweisung

Ein gezähmtes `goto` ist die switch-Anweisung, die je nach Wert einer ganzzahligen Variablen verschiedene Sprungziele einer direkt anschließend mit `case`-Anweisungen gebildeten Liste auswählt und die Verarbeitung dann an dieser Stelle fortsetzt. Dieses Konstrukt ist häufig anzutreffen, wenn eine Variable dazu benutzt wird, eine Auswahl aus einer vorgegebenen Anzahl möglicher Alternativen festzulegen.

Im Prinzip können `switch`-Anweisungen immer durch `if...else if`-Konstrukte ersetzt werden, sind aber häufig übersichtlicher.

```
char c;
cin >> c;
switch( c ) {                  // begin case block
case 'a': cout << "hello world!";
          break;
case 'b': cout << "HELLO ";    // fallthrough intended, comment missing break
case 'c': out << "WORLD!";
          break;
default:  cout << "unknown input";
          break;               // not necessary, but good style
}                              // end case block
```

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: