Programmierkurs C/C++

"Montag"

Sandro Andreotti
andreott@inf.fu-berlin.de
WS 2012/13
(Slides: Andreas Döring)

Organisation

- Homepage:
  http://www.mi.fu-berlin.de/w/ABI/CppIntroWS12

Vormittags: Seminar
Raum HS 001 (Arnimallee 3)
Anwesenheitspflicht!

- Nachmittags: Übungen am Rechner
  Raum R 017 (Arnimallee 6)
  Abteilung A: 13.00 – 15.00 Uhr
  Abteilung B: 15.00 – 17.00 Uhr

Die Anwesenheitspflicht beim Seminar gilt für diejenigen Teilnehmer, die einen Schein erwerben wollen.

Aufgrund der beschränkten Anzahl der verfügbaren Rechnerplätze bitten wir die Teilnehmer, bei den Übungen Gruppen von jeweils 2 bis maximal 3 zu bilden, und gemeinsam die Aufgaben zu bearbeiten.

Scheinvergabe

- Benoteter ABV-Kurs:
  2 SWS = 2 LP (ECTS)
- Voraussetzungen:
  - Teilnahme am Seminar (Unterschrift!)
  - Vorführen einer „Prüfungsaufgabe" am Freitag in der Übung.
  - Kurztest am Freitag im Seminar.

An jedem Tag von Montag bis Donnerstag erhalten Sie eine „Prüfungsaufgabe", die Sie unbedingt bearbeiten sollten. Am Freitag müssen Sie eine davon am Rechner vorführen, welche entscheidet der Dozent. Die Vorführung kann in Gruppen von bis zu 3 Personen erfolgen; jeder, der einen Schein bekommen möchte, muss daran teilnehmen. Jeder Teilnehmer muss dazu befähigt sein, Fragen zum erstellten Programm zu beantworten. Besteht berechtigter Zweifel daran, dass einer der Teilnehmer nicht aktiv an der Programmierung der vorgeführten Aufgabe teilgenommen hat, kann der Teilnahmeschein verweigert werden.

Ziele

- Was wir lernen:
  - Kennen der wichtigsten Sprachelemente und einiger Bibliotheksfunktionen
  - Schreiben von einfachen Programmen
  - Überblick über den Sprachumfang von C++

einen Fortgeschrittenenkurs gibt es
im nächsten Semester.

Das C++-Symbol kennzeichnet diejenigen Abschnitte oder Spachfeatures, die es nur in C++, nicht aber in C gibt.

Literatur

Standardwerke:

- Kernighan, Ritchie, "Programmieren in C"
- Stroustrup, "Die C++ Programmiersprache"

Online Dokumentation:

- z.B. **msdn.microsoft.com**

(Siehe Veranstaltungshomepage)

Das C++-Symbol kennzeichnet diejenigen Abschnitte oder Spachfeatures, die es nur in C++, nicht aber in C gibt.

Literatur

Kernighan, Ritchie, "Programmieren in C" (1990), ISBN 3-446-15497-3, € 32,90
Stroustrup, "Die C++ Programmiersprache" (2000), ISBN 3-8273-1660-X, € 49,95

Lippman, Lajoie, "C++" (2002), ISBN 3-8266-1429-1, € 25,00

Josuttis, "The C++ Standard Library - A Tutorial and Reference" (2002), ISBN 0201379260, c.a. € 55,00

Meyers, "Effektiv C++ programmieren" (1997), ISBN 3-8273-1305-8, € 29,95
Meyers, "Mehr Effektiv C++ programmieren" (1997), ISBN 3-8273-1275-2, € 29,95
- Eine Art "Tipps & Tricks"-Fundgrube

Links finden sich auf der Veranstaltungshomepage.

Überblick

"Montag":

Einführung, einfache Typen, Anweisungen, Funktionen

"Dienstag":

Compiler-Workflow, Preprocessor, Pointer, Arrays

"Mittwoch":

Speichermodell, Standardbibliotheken, Fehlershow

"Donnerstag":

Structs, Klassen, Überladung

"Freitag":

Vererbung, virtuelle Funktionen, Operatorüberladung

Was ist C/C++?

Fragen:

- Seit wann gibt es C, seit wann C++?
- Hat sich die Sprache seitdem verändert?
- Was ist der Unterschied zwischen C und C++?

- C entstand in den 70ern, C++ Anfang der 80er
- C++ ist aus C entstanden und (weitgehend) abwärts-kompatibel.
- Es gibt inzwischen Standards für C und C++.
- Compiler erfüllen die Standards nicht zu 100%.

Für C existiert ein ANSI-C-Standard von 1988/89, dazu auch ein ISO/IEC Standard (9899).

Für C++ gibt es den ISO/IEC Standard (14882) „Programming languages – C++“ von 1998.

C wurde von Kernighan und Ritchie in den 70ern als Nachfolger für die Programmiersprache „B“ entwickelt.

Anfang der 80er erweiterte Stroustrup die Sprache um Klassen gab ihr den Namen „C++“.

Da C/C++ nicht auf einen Schlag entwickelt wurde, sondern das Produkt vieler Autoren und einer langjährigen Entwicklung ist, gibt es eine Vielzahl von Sprachkonstrukten, von denen inzwischen jedoch einige „aus der Mode“ gekommen sind. Außerdem hat die Sprache stellenweise mit ihrer Abwärtskompatibilität zu kämpfen.

Warum C/C++?

Warum überhaupt C/C++?

- C/C++ ist schnell.
- C/C++ ist weit verbreitet.
- C/C++ verfügt über viele Sprachkonstrukte.
- C/C++ erlaubt „maschinennahes" Programmieren.
- C/C++ macht Spaß!

Trotz aller gegenteiligen Meldungen sind Programme in C/C++ in den meisten Fällen immer noch deutlich schneller als z.B. vergleichbare Programme in Java.

Aber Vorsicht:

-Schlecht programmiertes C++ kann langsamer sein als gut programmiertes Java.

-Wenn es nicht auf Geschwindigkeit oder Maschinennähe ankommt, ist eine andere Programmiersprache vermutlich die bessere Wahl.

Die Maschinennähe ist ein Vor- und Nachteil zugleich: Ein Vorteil, weil darin eines der Geheimnisse für die Effizienz von C/C++ liegt; aber auch ein Nachteil, weil C/C ++ deswegen schwieriger zu handhaben ist.

"Hallo Berlin!"

Ein einfaches C++-Programm:

```
#include <cstdio>

int main()
{
   printf("Hallo Berlin!");
   return 0;
}
```

Speichern als `test.cpp` ab und compiliere mit:

```
g++ test.cpp -o test
```

Dann Programm ausführen mit:

```
./test
```

g++ ist der GNU C Compiler, der unter UNIX/LINUX-System weit verbreitet ist.

#include ist ein ein Preprocessor-Befehl. Damit wird der „Header" einer Standardbibliothek in den Programmtext eingebunden. Man hätte auch #include ″stdio.h″ schreiben können, aber es ist üblich, die Namen von Standardbibliotheken in spitzen Klammern anzugeben.

main ist der Name der Funktion, die als erstes im Programm aufgerufen wird. Die Abarbeitung von C/C++-Programmen beginnt (im Wesentlichen) immer bei der Funktion **main**.

int ist der Rückgabewert der Funktion **main**, im Beispiel wird 0 zurückgeliefert. Dies ist ein Error-Code (0 = „alles klar").

printf wird in **cstdio** definiert und gibt einen Text auf der Konsole aus.

Wichtig: Bei C++-Programmen sollte die Datei auf .cpp oder .C (großes C) enden, bei C-Programmen auf .c;

Beobachtungen

- Imperative Programmiersprache.
- Quelltext wird compiliert.
- Quelltext ist case-sensitiv.
- Anweisungen enden mit einem ; -Zeichen.
- Das Programm beginnt bei `main()`

„Imperative Programmiersprache“: Es werden Instruktionen nacheinander abgearbeitet. Der Kontrollfluss wird durch Anweisungen gesteuert.

Ein C/C++-Programm ist zunächst ein Text, der dann durch ein spezielles Programm, den „Compiler“ in ein lauffähiges Programm übersetzt wird.

C/C++-Programme zu compilieren dauert oft ziemlich lange – an JIT („just in time“) wie bei Java ist nicht zu denken. Das hat u.A. seinen Grund darin, dass C/C++-Compiler oft sehr gute Optimierer anwenden, um den produzierten Code möglichst schnell zu machen.

Programmbeispiel 2

```
#include <cstdio>

int main()
{
   int dm;
   float euro;
   printf("Bitte ganze DM eingeben: ");
   scanf("%i", & dm);

   euro = dm / 1.95583;
   printf("Das sind %f Euro", euro);

   return 0;
}
```

Mit **int dm;** wird eine Variable mit Namen **dm** vom Typ **int** definiert. Der Typ einer Variablen gibt an, was in ihr gespeichert werden kann. In diesem Fall (**int**) handelt es sich um ganze Zahlen.

Mit **scanf** lassen sich Zahlen oder Strings von der Konsole einlesen. Der Wert wird in diesem Fall in die Variable **dm** geschrieben. (Man beachte das **&**-Zeichen vor **dm**; der genaue Grund dafür wird später erklärt.)

Das Ergebnis der Umrechnung ist eine Fließkommazahl (**float**), die in der Variablen **euro** gespeichert wird. Man beachte, dass hier offenbar eine Typkonversion von **int** zu **float** stattfindet (darauf werden wir noch näher eingehen).

Der letzte **printf**-Befehl gibt diesen Wert aus. Mit **%f** im Ausgabestring wird spezifiziert, dass an dieser Stelle eine Fließkommazahl ausgegeben werden soll. Die Variable **euro** wird **printf** als zweites Argument übergeben.

Variablen

Variablen definieren:

```
int j;
float x, y;
```

Optional sind auch Initialisierungen möglich:

```
int i = 32;
float z = 2.781;
char c = 'x';
```

Die Zuweisung von Initialisierungswerten ist nur bei Variablen*definitionen*, nicht bei bloßen Deklarationen erlaubt. Der Unterschied zwischen Deklaration und Definition wird erst in der nächsten Sitzung erläutert.

Wichtig: In C müssen alle lokalen Variablen (d.h. Variablen, die innerhalb einer Funktion definiert werden) am Anfang der Funktion definiert werden. In C++ ist es möglich, auch später innerhalb der Funktion Variablen zu definieren.

"Eingebaute" Typen

Jede Variable kann nur Werte eines bestimmten Typs speichern.

char:	Zahlen von 0 bis 255
short:	Zahlen von -32768 bis 32767
unsigned short:	Zahlen von 0 bis 65535
int:	Zahlen von -2^{31} bis $2^{31}-1$
unsigned int:	Zahlen von 0 bis $2^{32}-1$
bool:	0 (false) oder 1 (true)

Bei 64-bit-Maschinen kann **int** auch Zahlen von -2^{63} bis $2^{63}-1$ und **unsigned int** Zahlen von 0 bis $2^{64}-1$. Dies ist eine Compilereinstellung. Man sollte sich also nicht darauf verlassen, dass **int** genau 4 Byte groß ist.

Man kann statt **int** auch **long int** und statt **unsigned int** auch **unsigned long int** verwenden. Das sind zwar formal andere Typen, doch gewöhnlich passt jeweils das gleiche hinein, jedenfalls bei 32-bit-Maschinen. Näheres regelt der Compilerhersteller.

Es gibt auch **signed char** und **unsigned char**, beide Typen sind von **char** verschieden.

Man kann statt **short** auch **signed short** und statt **int** auch **signed int** schreiben.

Außerdem kann man auch **short int** statt **short**, **unsigend short int** statt **unsigned short**, **long** statt **long int** und **unsigned long** statt **unsigned long int** schreiben. Alles klar?

Den Typen **bool** gibt es nur in C++. Dafür kennt C inzwischen den Typ **_Bool**, doch es wäre zu empfehlen, statt dessen lieber **char** oder **int** zum Speichern eines boolschen Wertes zu verwenden. In C kann man allerdings den Header **<stdbool.h>** einbinden; dadurch werden die Symbole **bool**, **true** und **false** definiert. In C++ gehören diese Symbole direkt zum Sprachumfang, und stehen

weitere "eingebaute" Typen

Typen zur Speicherung von Kommazahlen:

`float`:	Fließkommazahl mit einfacher Genauigkeit
`double`:	Fließkommazahl mit doppelter Genauigkeit

Ob sich **float** und **double** überhaupt voneinander unterscheiden, hängt von der verwendeten Maschine bzw. von dem verwendeten Compiler ab. Einzige Regel: **double** darf nicht „ungenauer" als **float** sein. Typisch sind: **float** = 4-byte Fließkommazahl, **double** = 8-byte Fließkommazahl.

... und was ist mit Buchstaben?

Buchstaben *sind* ganze Zahlen:

`'A' == 65` ⇦ einfache Anführungszeichen für einzelne Zeichen: ASCII-Code

Auch Wahrheitswerte sind Zahlen

```
true == 1
false == 0
```

(und alle anderen Zahlen ≠0)

Da einzelne Zeichen auch Zahlen sind, kann man sie auch wie Zahlen behandeln:

```
'A' + 2 == 'C'
'F' < 'T'
```

Felder (arrays)

Felder = Mehrere Variablen, die im Speicher direkt hintereinander liegen.

```
int X[20];
```
⇦ Ein Feld von 20 `int`-Variablen

```
int X[3]={2,3,5};
```
⇦ Auch Felder kann man initialisieren.

```
int X[]={1,2,3,4};
```
⇦ Bei initialisierten Feldern kann man sich die Dimension sparen.

Strings sind Felder

In C sind Stings einfach Arrays von `char`:

```
char X[] = "hallo";
```

ist das gleiche wie

```
char X[] = {104, 97, 108, 108, 111, 0};
```

Man beachte die Null am Ende!

Mit der 0 wird das Ende des Strings angezeigt. Also Vorsicht: "a" braucht 2 bytes Speicher, während 'a' nur ein byte Speicher braucht.

Diese Form der Strings stammen noch aus den frühsten Zeiten von C, inzwischen existiert im Standard eine Stringklasse basic_string, die häufig viel komfortabler zu verwenden ist als einfache C-Strings.

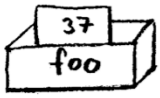
Zeiger (*pointers*)

Zeiger sind Variablen, deren Inhalt auf andere Variablen verweist.

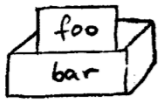
```
int * X;
float * Y;
```

⇦ Zeiger auf eine int-Variable

⇦ Zeiger auf eine float-Variable



int foo = 37;



int * bar = & foo;

Mit Pointern werden wir uns im nächsten Kapitel noch eingehender beschäftigen.

kleine Lesehilfe

Faustregel:

»Variablendeklarationen liest man von innen nach außen.«

Bsp:

```
char * X1 [3];          ⇦ array von pointern
char (* X2) [3];        ⇦ pointer auf array
int X3 [10][3];         ⇦ array von arrays
int * const X4;         ⇦ const pointer auf int
int const * X5;         ⇦ pointer auf const int
```

const bedeutet, dass die Variable nicht verändert werden kann. Wird das dennoch versucht, so mahnt der Compiler dies als Fehler an.

typedef

Mit `typedef` kann man Typbeschreibungen Namen geben:

```
typedef int * IntPtr;
IntPtr x;
```

⇐`x` ist nun vom Typ `int *`

```
typedef char (* T)[10];
T y;
```

Mit **typedef** werden jedoch keine neuen Typen definiert: Im Beispiel oben sind **IntPtr** und **int *** der selbe Typ, nur anders geschrieben.

Aufzählungstypen (Enums)

Definiere neuen Typ durch Aufzählung der Werte, die eine Variable dieses Typs einnehmen kann:

```
enum Farbe
{
 blau, rot, gelb, orange, gruen
};

Farbe x = gelb;
printf("%i", x);
```

Man beachte das ; hinter der schließenden Klammer }.

Die Werte des enums (blau, rot, gelb, orange, gruen) entsprechen Zahlen.

Mit der letzten Zeile wird der Zahlenwert fuer gelb (2) ausgegeben.

In C++ wird somit durch enum nicht nur die Namen der einzelnen Werte („blau“, „rot“, ...) definiert, sondern stets auch ein Typ („Farbe“).

Vorsicht: In C++ definiert enum keinen neuen Typen. Dort müsste man folgendes schreiben:

```
typedef enum FarbeEnum
{
     blau, rot, gelb, orange, gruen
} Farbe;
```

Enum-Werte sind Zahlen

Die Werte eines Enums werden bei der Definition zu Synonymen von (ganzen) Zahlen:

```
enum Farbe { blau, rot, gelb };
```

Dann sind: `blau == 0, rot == 1, gelb == 2.`

Die Werte kann man auch explizit angeben:

```
enum Lieblingszahlen
{
  Antwort = 42,
  Teufel = 666
};
```

Auch Mehrfachbelegungen der Werte sind erlaubt.

Bsp:

```
enum allesNull { ich = 0, du = 0, er = 0, sie = 0};
```

enums lassen sich prima für die Definition von Konstanten missbrauchen. Allerdings sind nur ganze Zahlen (auch negative) als Wert erlaubt.

Operatoren

Es gibt in C/C++ die üblichen Operatoren.

Besonderheiten:

- Test auf Gleichheit ist == (und nicht etwa =)
- Test auf Ungleichheit ist !=
- & und | stehen für bitweise UND bzw. ODER,
  && und || logisches UND bzw. ODER sind
- ~ steht für bitweises NOT,
  ! steht für logisches NOT

Vorsicht: Die Zeichen * und [ ], die bei der Deklaration/Definition von Variablen benutzt werden, finden ebenfalls als Operatoren Verwendung, bedeuten dann aber natürlich etwas anderes.

Argumentauswertung bei && und ||

Bei den Operatoren && (logisches UND)
und || (logisches ODER) gelten folgende Regeln:

(Ausdruck1 && Ausdruck2)

werte **Ausdruck2** *nur* aus, wenn
Ausdruck1 == true

(Ausdruck1 || Ausdruck2)

werte **Ausdruck2** *nur* aus, wenn
Ausdruck2 == false

Anmerkung für Experten: Es ist nicht möglich, dieses Verhalten nachzuahmen, wenn man selbst operatoren dieser Art implementiert. Um die Erwartungshaltung der Benutzer nicht zu enttäuschen, sollte man sich also genau überlegen, ob man diese Operatoren überhaupt implementiert.

...noch mehr Operatoren

Increment und Decrement:

```
++x;      ⇔  x = x + 1;
--x;      ⇔  x = x - 1;
```

Zuweisungsoperatoren:

z.B:

```
x += 20;  ⇔ x = x + 20;
x -= 12;  ⇔   x = x - 12;
x *= y;   ⇔   x = x * y;
x /= 2;   ⇔   x = x / 12;
x <<= 1;  ⇔   x = x << 1;
...
```

Beispiele:

```
int x, y;              // definiert zwei Variablen x und y
x = 12;                // weist x eine 12 zu
x += 4;                // jetzt ist x == 16
y = ++x;               // jetzt ist y und x sind == 17
y = x>>2;              // jetzt ist y == 4
x = y | 1;             // jetzt ist x == 5 (binäres oder)
y = y && 15;  // jetzt ist x == 1 (logisches und, 1 ist true)
```

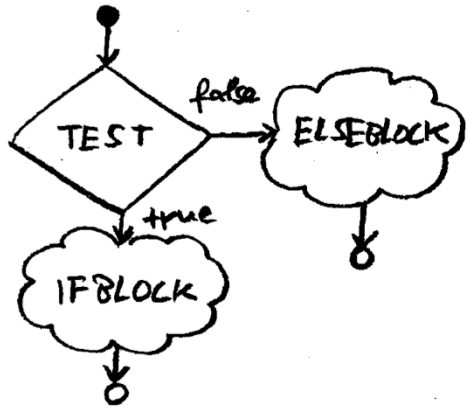
Kontrollstrukturen

Es gibt (u.A.) folgende Kontrollstrukturen:

- if und else
- while und do
- for
- switch-case

if und else

```
if (TEST)
{
  IFBLOCK
}
else
{
  ELSEBLOCK
}
```



Wenn im IFBLOCK nur eine einzelne Anweisung steht, kann man die geschweiften Klammern { } auch weglassen. Das gleiche gilt auch für den ELSEBLOCK und alle anderen Anweisungsblöcke.

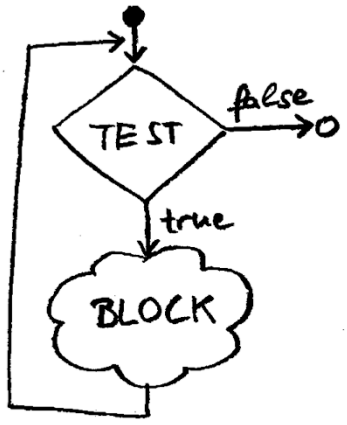
Wenn man mehrere **if**- und **else**-Anweisungen hintereinander hat, dann gilt: Das **else** bezieht sich stets auf das letzte (mögliche) **if**.

Folgende Konstruktion ist z.B. möglich:

```
if (x == 1) printf("eins");
else if (x == 2) printf("zwei");
else if (x == 3) printf("drei");
else printf("weder 1 noch 2 noch 3");
```

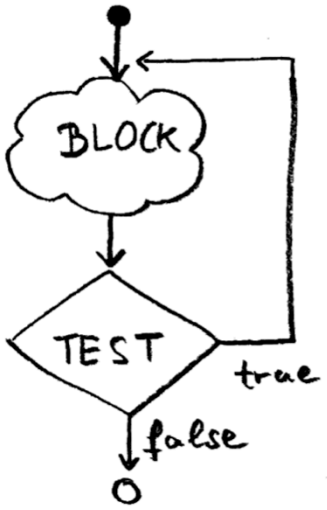
while

```
while (TEST)
{
  BLOCK
}
```



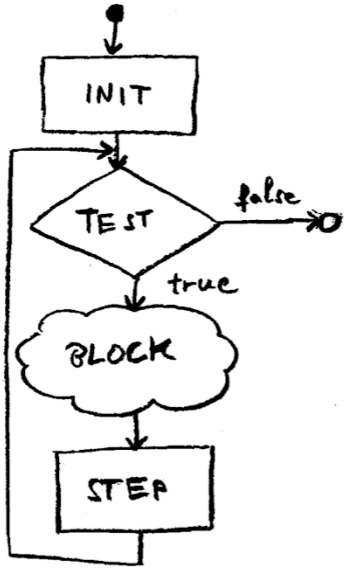
do-while

```
do
{
  BLOCK
} while (TEST);
```



for

```
for (INIT;TEST;STEP)
{
  BLOCK
}
```



switch-case

```
switch (X)
{
  case 1: case 2:
   //Code für X==1 und X==2
   break;
  case 3:
   //Code für X==3
   break;
  default:
   //Code für alle anderen Werte von X
}
```

Wenn man das break weglässt, dann läuft das Programm in den nächsten **case** mit hinein. Bsp:

```
switch (X)
{
      case 10:
            //Code für X==10
      case 13:
            //Code für X==10 und X==13
}
```

Funktionen

Definition:

```
int max (int x, int y)
{
   if (x > y)
   {
      return x;
   }
   return y;
}
```

Aufruf:

```
max (12, 1);
```

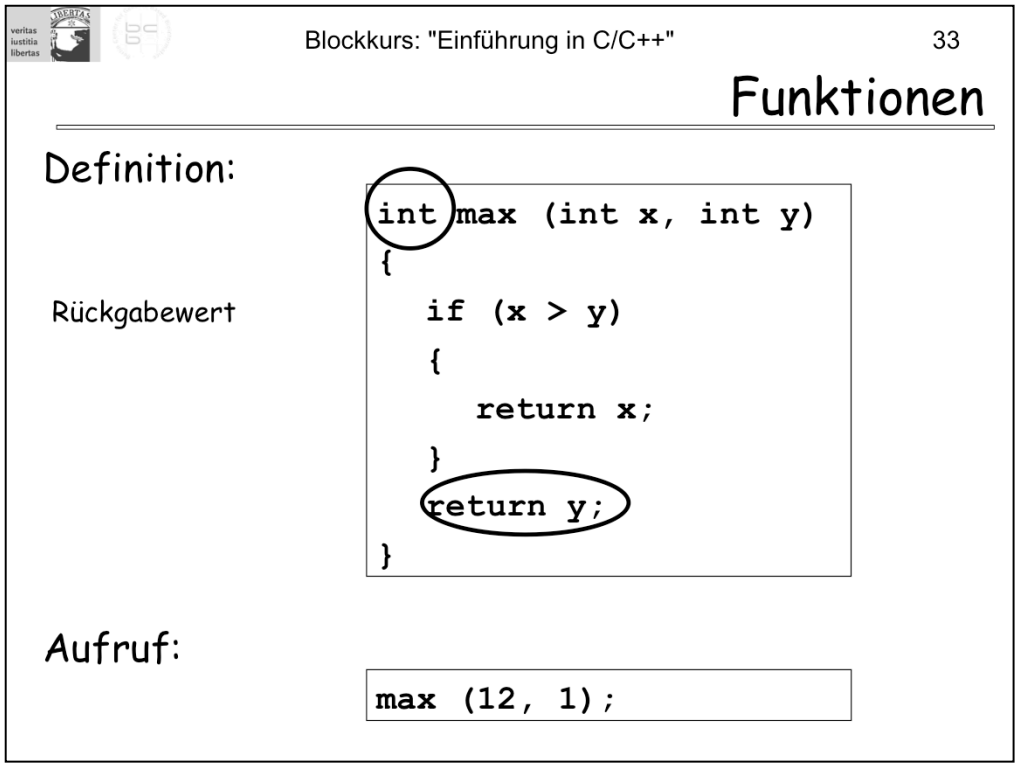
Wie oben bereits angemerkt, beginnt das Programm immer mit der Funktion **main()**.

Mit **return** werden Werte zurück gegeben.

Formale Parameter (im Beispiel **x** und **y**) sind lokale Variablen im {}-Block der Funktion.

C/C++- Funktionen sind re-entrant: Bei jedem Funktionsaufruf wird ein neuer Satz lokaler Variablen erzeugt.

Man beachte: In C/C++ gibt es keine lokalen Funktionen, d.h. Funktionen können nicht innerhalb von Funktionen definiert werden.



Wie oben bereits angemerkt, beginnt das Programm immer mit der Funktion **main()**.

Mit **return** werden Werte zurück gegeben.

Formale Parameter (im Beispiel **x** und **y**) sind lokale Variablen im {}-Block der Funktion.

C/C++- Funktionen sind re-entrant: Bei jedem Funktionsaufruf wird ein neuer Satz lokaler Variablen erzeugt.

Man beachte: In C/C++ gibt es keine lokalen Funktionen, d.h. Funktionen können nicht innerhalb von Funktionen definiert werden.

Funktionen

Definition:

```
int max (int x, int y)
{
   if (x > y)
   {
      return x;
   }
   return y;
}
```

Argumente

Aufruf:

```
max (12, 1);
```

Wie oben bereits angemerkt, beginnt das Programm immer mit der Funktion **main()**.

Mit **return** werden Werte zurück gegeben.

Formale Parameter (im Beispiel **x** und **y**) sind lokale Variablen im {}-Block der Funktion.

C/C++- Funktionen sind re-entrant: Bei jedem Funktionsaufruf wird ein neuer Satz lokaler Variablen erzeugt.

Man beachte: In C/C++ gibt es keine lokalen Funktionen, d.h. Funktionen können nicht innerhalb von Funktionen definiert werden.

...nochmal Funktionen

Funktionen ohne Rückgabewert:

```
void printplus1 (int x)
{
   printf("%i", x+1);
}
```

Argumente werden *by value* übergeben.

Funktionen ohne Rückgabewert kann man mit der Anweisung **return;** verlassen.

Funktionsargumente werden in C/C++ immer *by value* übergeben, d.h. die formalen Parameter werden mit Kopien der Aufrufsargumente gefüttert. Es gibt jedoch in C++ einen Referenztyp, mit dem sich Argumentübergaben *by reference* realisieren lassen. Außerdem kann man sich stets auch damit behelfen, dass man statt einer Variablen X einen *pointer* auf X übergibt.

Anmerkung: eine Variable kann nicht vom Typ **void** sein, aber Variablen vom Typ **void *** sind erlaubt.

Beispiel: Funktionen

```
#include <stdio.h>

int fac(int i)
{
 if (i == 1) return 1;
 return i * fac(i-1);
}

int main()
{
 printf("%i\n", fac(10));
}
```

Default Argumente

Funktionsargumente können Defaults bekommen:

```
void print_person ( char * name,
          int alter = 18,
          char * initial = "N.N.")
{
  printf("%s (%i) %s", name, alter, initial);
}


print_person ( "Cindy", 20);
```

Ausgabe: **Cindy (20) N.N.**

In der Argumentliste darf nach einem mit Default versehenen Argument kein weiteres Argument kommen, das nicht mit Default belegt wäre. Anders ausgedrückt: Die Argumente mit Default sind immer die hintersten in der Liste der Funktionsargumente.

Bsp:

```
int f(int x, int y = 0, int z); //geht nicht!
```

Vorsicht: Überladung (gibt es nur bei C++) und Default-Argumente können sich gegenseitig behindern:

Bsp:

```
int f(int x) { ...}
int f(int x, int y= 0) {... }

f(10);  //Problem: Welche Funktion f soll er aufrufen?
        //Compiler gibt Fehlermeldung.
```

Quiz: Was gibt dieses Programm aus?

```
int i = 14;
for (i = 0; i < 10; ++i);
{
  printf("%i ", i);
}
```

Vorsicht Falle!

Antwort: Das Programm gibt „10“ aus. Grund: Die **for**-Anweisung endet mit einem Semikolon, darum tut sie nichts weiter als **i** bis 10 hochzuzählen. Im anschließenden { }-Block wird dann **i** einmal ausgegeben.

Aufgaben zum Montag:

1. Aufgabe:

Besorgen Sie sich das kleine C-Programm „sample.cpp“ von der Veranstaltungshomepage. Wenn Sie mit Visual Studio arbeiten, erzeugen Sie dazu ein neues, leeres „Win32 Console Project“ für C++. Bringen Sie das Programm zum laufen. Modifizieren Sie dann den Programmcode und spielen ein wenig mit den Möglichkeiten.

Bei Microsoft Visual Studio erzeugt man ein neues Projekt wie folgt:

-Unter „File“- „New“- „Project“: Bei den Visual C++-Projekten „Win32 Console Project“ auswählen.

-Namen und Pfad des neuen Projekts eingeben, dann „OK“ drücken.

-Bei „Application Settings“ den Punkt „Empty project“ anhaken, erst dann „Finish“ drücken.

-Neue Dateien im Project View hinzufügen.

Machen Sie sich auch mit den Befehlen **printf** und **scanf** vertraut, insbesondere mit der Notation des Formatstrings. Eine Dokumentation finden Sie online.

2. Aufgabe:

Programmieren Sie eine Art kleinen Taschenrechner mit verschiedenen Operationen. Benutzen Sie **printf** und **scanf** für die Ein- und Ausgabe.

3. Aufgabe:

Nehmen Sie an, Sie rufen eine Funktion mit zwei Argumenten auf, z.B. **f(x+1, y-z)**. Bevor dieser Aufruf stattfinden kann, müssen erst die Argumente ausgewertet werden. Die Frage ist: Wird zuerst „**x+1**“ berechnet, und dann „**y-z**“ oder umgekehrt? Versuchen Sie dies herauszufinden, indem Sie kluge Ausdrücke für die Argumente verwenden.

PRÜFUNGSAUFGABE:

Schreiben Sie ein Programm, das mit **scanf** eine Zahl X einliest und anschließend die X-te Fibonacci-Zahl fib(X) ausgibt. Dabei ist fib(i) wie folgt definiert:

fib(0) = fib(1) = 1

fib(i+2) = fib(i+1) + fib(i), für i aus {0, 1, 2, ...}

Achtung: Kluge Informatikstudenten sollten das Problem so lösen, dass das Programm nur konstant viel Speicher verwendet. Der kluge Bioinformatikstudent natürlich auch. Und wer zwar weder Informatik noch Bioinformatik studiert, aber trotzdem klug ist (soll ja vorkommen), der kann das ebenfalls mal versuchen.