Programmierkurs
Vorlesung 2

Dr. Ole Klein

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg

13. November 2020

C++-Hintergrund

C++-Grundlagen

- Grobstruktur
- Funktionen
- Statements
- Variablen
- Expressions (Ausdrücke)
- Ein- und Ausgabe
- Kontrollfluss

Was ist C++?

C++ ist eine Programmiersprache, die sowohl für systemnahe als auch für Anwendungsprogrammierung entwickelt wurde.

- Unterstützung verschiedener Programmier-Paradigmen: imperativ, objekt-orientiert, generisch
- Fokus auf Effizienz, Performance und Flexibilität
- Einsatzgebiete von embedded Controllern bis zu Supercomputern
- Erlaubt direkte Verwaltung von Hardware-Ressourcen
- "Zero-cost abstractions"
- "Pay only for what you use"
- Offener Standard mit mehreren Implementierungen
- Wichtigste Compiler:
  - Open Source: GCC und Clang (LLVM)
  - Kommerziell: Microsoft und Intel

Geschichte

1979	Entwicklung “C with Classes” durch Bjarne Stroustrup
1985	Erster kommerzieller C++-Compiler
1989	C++ 2.0
1998	Standardisierung als ISO/IEC 14882:1998 (C++98)
2011	Nächste große Version mit neuen Funktionen (C++11)
2014	C++14 mit vielen kleinen Fehlerkorrekturen und praktischen Features
2017	C++17, aktuelle Version von C++
2020	C++20, kommende Version, wird erst in den nächsten Jahren Verwendung finden

C++-Hintergrund

C++-Grundlagen

- Grobstruktur
- Funktionen
- Statements
- Variablen
- Expressions (Ausdrücke)
- Ein- und Ausgabe
- Kontrollfluss

Ein erstes C++-Programm

```cpp
/* make parts of the standard library available */
#include <iostream>
#include <string>

// custom function that takes a string as an argument
void print(std::string msg)
{
  // write to stdout
  std::cout << msg << std::endl;
}

// main function is called at program start
int main(int argc, char** argv)
{
  // variable declaration and initialization
  std::string greeting = "Hello, world!";
  print(greeting);
  return 0;
}
```

Zentrale Bestandteile eines C++-Programms

- Include-Direktiven, um Bibliotheken zu verwenden
  - Stehen am Anfang des Programms
  - Werden in Zukunft erwähnt, wenn erforderlich
  - Nur eine Include-Direktive pro Zeile
- Eigene Funktionen
  - Ähnlich wie mathematische Funktionen mit Parametern und Rückgabewert
  - Jedes Programm muss die Funktion

```
int main(int argc, char** argv)
{
  ...
}
```

enthalten. Diese wird vom Betriebssystem beim Programmstart ausgeführt.

Kommentare

- Kommentare dürfen überall stehen und erklären das Programm für andere Progammierer
- Kommentare mit `//` gehen bis zum Ende der Zeile:

```
int i = 42; // the answer
int x = 0;
```

- Mehrzeilige Kommentare werden durch `/*` begonnen und durch `*/` beendet:

```
/* This comment spans
   multiple lines */
int x = 0;
```

Hinweis

Kommentare erscheinen oft als sinnlose Zusatzarbeit, aber nach einer Woche können Sie sehr dabei helfen, das Programm selber zu verstehen und dem Tutor zu erklären!

Funktionen

- Während der Ausführung eines C++-Programm werden Funktionen aufgerufen, beginnend mit der speziellen Funktion `main(int argc, char** argv)`
- Funktionen können andere Funktionen aufrufen
- Funktionsdefinitionen bestehen aus einer Funktionssignatur und einem Funktionsrumpf (body)

```
return-type functionName(arg-type argName, ...) // signature
{
  // function body
}
```

- Die Funktionssignatur legt fest, wie die Funktion heißt und welche Argumente benötigt werden
- Eine Funktion hat immer einen *return type* (genaueres dazu später). Falls die Funktion nichts zurückgeben soll, verwendet man als return type `void`
- Der Funktionsrumpf beschreibt, was die Funktion macht und wird durch geschweifte Klammern eingefasst

Statements

```
int i = 0;
i = i + someFunction();
anotherFunction();
return i;
i = 2; // never executed
```

- Eine C++-Funktion besteht aus einer Reihe von Statements, die nacheinander ausgeführt werden
- Statements werden durch ein Semikolon voneinander getrennt
- Das spezielle Statement `return something;` verlässt sofort die aktuelle Funktion und gibt `something` als Rückgabewert der Funktion zurück
  - Bei `void`-Funktionen kann man den Rückgabewert oder das gesamte `return` statement weglassen

Variablen

- Variablen dienen dazu, Werte in Programmen zwischenzuspeichern
- Variablen in C++ haben immer einen festen Typ (ganze Zahl, Kommazahl, Text, ...)
- Variablennamen bestehen aus Groß- und Kleinbuchstaben, Ziffern und Unterstrichen, dürfen aber nicht mit Ziffern beginnen
- Groß- und Kleinschreibung wird unterschieden!
- Bevor eine Variable benutzt werden kann, muss sie deklariert werden
  - Normale Variablen werden durch ein Statement deklariert:

```
variable-type variableName = initial-value;
```

  - Funktionsparameter werden in der Funktionsdeklaration deklariert:

```
void func(var-type1 param1, var-type2 param2)
```

Wichtige Variablen-Typen

C++ kennt viele Typen von Variablen, hier einige wichtige (Zahlenreichweiten gültig auf 64-Bit Linux):

```
// 32-Bit Integer, ganze Zahlen aus [−2^31, 2^31 − 1]
int i = 1;
// 64-Bit Integer, ganze Zahlen aus [−2^63, 2^63 − 1]
long l = 1;
// 8-Bit Integer, ganze Zahlen aus [−2^7, 2^7 − 1]
char c = 1;
// Boolean (Wahrheitswert), true (=1) oder false (=0)
bool b = true;
// Text (Zeichenkette), benötigt #include <string>
std::string msg = "Hello";
// Fließkommazahl doppelter Genauigkeit
double d = 3.141;
// Fließkommazahl einfacher Genauigkeit
float f = 3.141;
```

Integer-Varianten für ausschließlich positive Zahlen durch vorangestelltes `unsigned` mit Wertebereich $[0, 2^{\text{bits}} - 1]$

Scopes und Variablen-Lebenszeit

- Ein block scope ist eine durch geschweifte Klammern eingeschlossene Gruppe von Statements
- Scopes können beliebig geschachtelt werden
- Variablen haben eine begrenzte Lebenszeit:
  - Die Lebenszeit einer Variablen beginnt am Punkt ihrer Deklaration
  - Die Lebenszeit einer Variablen endet, sobald das Programm das scope verlässt, in dem sie deklariert wurde

```
int cube(int x)
{
  // x existiert in der gesamten Funktion
  {
    int y = x * x; // y existiert ab hier
    x = x * y;
  } // y existiert ab hier nicht mehr
  return x;
} // x existiert ab hier nicht mehr
```

Scopes und Namenskollisionen

- Es ist nicht möglich, im selben scope zwei Variablen mit demselben Namen anzulegen

```
{
  int x = 2;
  int x = 3; // Compile-Fehler!
}
```

- Namen in einem inneren Scope überschreiben temporär Namen im äußeren Scope

```
int abs(int x) { ... }

{
  int x = -2;
  {
    double x = 3.3; int abs = -2;
    std::cout << x << std::endl; // gibt 3.3 aus
    x = abs(x); // Compile-Fehler, abs ist eine Variable!
  }
  x = abs(x); // danach: x == 2
}
```

Expressions (Ausdrücke)

Um Dinge in C++ zu berechnen, verwenden wir Expressions (Ausdrücke)

- Ausdrücke sind Kombinationen von Werten, Variablen, Funktionsaufrufen und mathematischen Operatoren, die ein Ergebnis produzieren, das einer Variablen zugewiesen werden kann:

```
i = 2;
j = i * j;
d = std::sqrt(2.0) + j;
```

- Beim Auswerten zusammengesetzter Ausdrücke wie `(a * b + c) * d` gelten Klammern und erweiterte Punkt-Vor-Strich-Regeln, die sogenannte operator precedence

Regelübersicht

`https://en.cppreference.com/w/cpp/language/operator_precedence`

Operatoren für Zahlen-Variablen

- Es gibt die üblichen binären Operatoren +,-,*,/
- a % b rechnet den Rest der Ganzzahldivision von a durch b aus:

```
13 % 5 // result: 3
```

- Division rundet bei ganzen Zahlen immer ab
- Bei einer Ganzzahl-Division durch 0 stürzt das Programm ab
- = weist seine rechte Seite der linken zu und hat den gleichen Wert wie die Zuweisung

```
a = b = 2 * 21; // both a and b have value 42
```

- Für häufig vorkommende Zuweisungen gibt es Abkürzungen:

```
a += b; // shortcut for a = a + b (also for -,*,/,%)
x = i++; // post-increment, shortcut for x = i; i = i + 1;
x = ++i; // pre-increment,  shortcut for i = i + 1; x = i;
```

- Es gibt auch Pre- und Post-decrement (--)

Vergleichsoperatoren

- Vergleichsoperatoren produzieren Wahrheitswerte:

```
4 > 3; // true
```

- Verfügbare Operatoren

```
a < b; // a strictly less than b
a > b; // a strictly greater than b
a <= b; // a less than or equal to b
a >= b; // a greater than or equal to b
a == b; // a equal to b (note the double =)!
a != b; // a not equal to b (note the double =)!
```

Kombination von Wahrheitswerten

Zur Kombination von Testergebnissen gibt es symbolische oder text-basierte Operatoren:

- Kombination mehrerer Tests mit und bzw. oder:

```
a == b ||  b == c; // a equal b or b equal c
a == b or  b == c; // a equal b or b equal c
a == b &&  b == c; // a equal b and b equal c
a == b and b == c; // a equal b and b equal c
```

- Invertierung eines Wahrheitswerts:

```
!true == false;
not true == false;
```

Texte / Strings

- Texte bzw. Zeichenketten werden in Variablen vom Typ `std::string` gespeichert und oft als string bezeichnet
- Feste Strings werden im Programm durch doppelte Anführungszeichen eingeschlossen

```
std::string msg = "Hello world!";
```

- Strings können mit + kombiniert werden

```
std::string hello = "Hello, ";
std::string world = "world";
std::string msg = hello + world;
```

- Strings können mit == und != verglichen werden

```
std::string a = "a";
a == "b"; // false
```

Warnung

Beim Vergleich oder Kombinieren von Strings muss links immer eine Variable stehen!

Ausgabe auf das Terminal

- Um mit dem Benutzer am Terminal zu kommunizieren, kann unser Programm auf die drei Streams stdin, stdout und stderr zugreifen (vgl. Shell)
- Zur Ausgabe verwenden wir `std::cout`. Alles, was wir ausgeben wollen, "schieben" wir mit << in die Standardausgabe

```
#include <iostream> // required for input / output
...
std::string user = "Joe";
std::cout << "Hello, " << user << std::endl;
```

- Um einen Zeilenumbruch zu erzeugen, gibt man `std::endl` (end line) aus

Eingabe vom Terminal

- Zum Einlesen von Benutzereingaben verwenden wir `std::cin`
- Hierfür muss die Variable vorher deklariert werden
- Eingaben "ziehen" wir mit `>>` aus der Standardeingabe

```
#include <iostream>
...
std::string user = "";
int answer = 0;
std::cout << "Enter your name: " << std::endl;
std::cin >> user;
std::cout << "Enter your answer: " << std::endl;
std::cin >> answer;
std::cout << "Hi " << user << "! Your answer was: "
          << answer << std::endl;
```

- Eingaben an der Konsole müssen mit Return abgeschlossen werden

Kontrollfluss

Die meisten Programme lassen sich nicht oder nur umständlich als einfache Folge von Statements aufschreiben, die in festgelegter Reihenfolge ausgeführt werden.
Beispiele

- Eine Funktion, die den Betrag einer Zahl zurückgibt
- Eine Funktion, die eine Division durch 0 abfängt und einen Fehler ausgibt
- Eine Funktion, die die Summe aller Zahlen von 1 bis N berechnet
- ...

Programmiersprachen enthalten spezielle Statements, die basierend auf dem Wert einer expression unterschiedliche Anweisungen ausführen

Verzweigungen / Branches

- Das `if`-Statement führt unterschiedlichen Code aus, je nachdem, ob ein Ausdruck wahr oder falsch ist

```
int abs(int x)
{
  if (x > 0)
  {
    return x;
  }
  else
  {
    return -x;
  }
}
```

- Der `else`-Teil der Anweisung ist optional:

```
if (weekday == "Friday")
{
  ipk_lecture();
}
```

Wiederholte Funktionsausführung

- Ein Programm muss oft den gleichen Code wiederholt ausführen, z.B. um

$$\sum_{i=1}^{n} i$$

  zu berechnen
- Zwei Programmieransätze:
  - Rekursion: Die Funktion ruft sich mit veränderten Argumenten selbst wieder auf
  - Iteration: Eine spezielle Anweisung führt einige Statements wiederholt aus

Rekursion

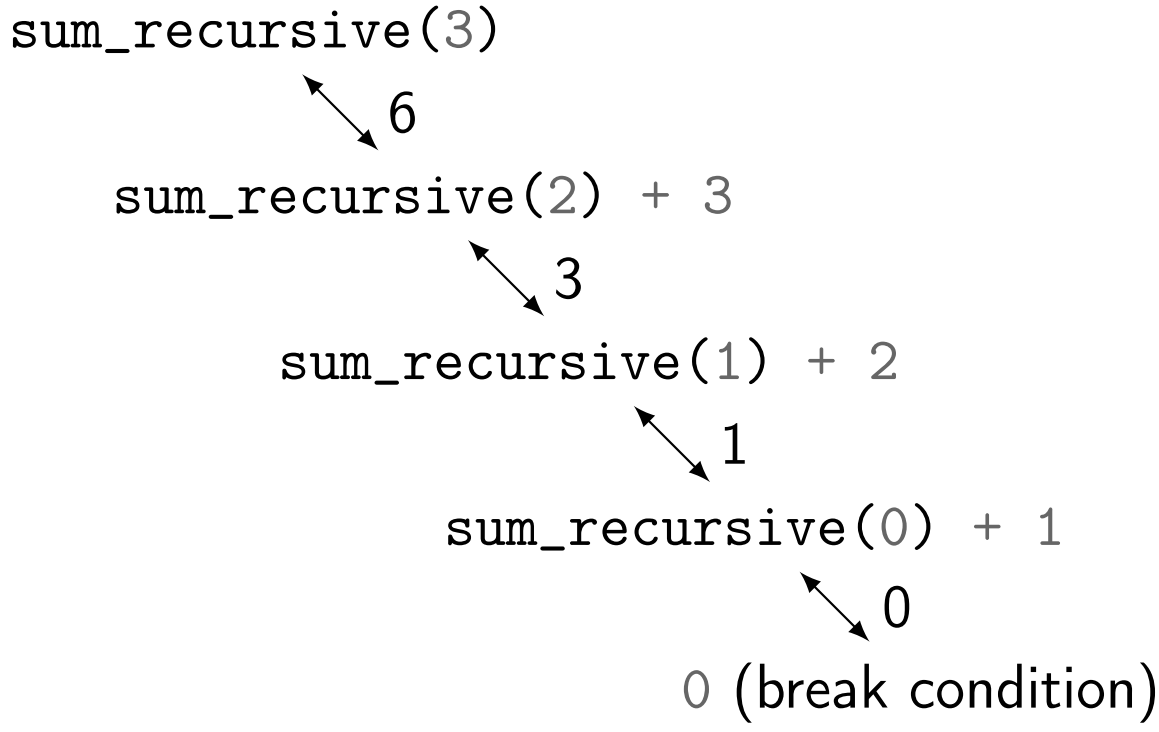
- Idee: Eine Funktion ruft sich mit veränderten Argumenten so oft selbst wieder auf, bis eine Abbruchbedingung erfüllt wird:

```
int sum_recursive(int n)
{
  if (n > 0)
  {
    return sum_recursive(n - 1) + n;
  }
  else
  {
    return 0;
  }
}
```

- Erfordert immer mindestens ein `if`-Statement, bei dem in genau einem Branch die Funktion erneut aufgerufen wird!
- Schlecht geeignet, wenn die Funktion nichts berechnet, sondern nur Seiteneffekte hat (Beispiel: die ersten N Zahlen auf das Terminal ausgeben)

Rekursion: Beispiel

- Berechnung von `sum_recursive(3)`
- Die Zahlen an den Pfeilen geben die Rückgabewerte der Funktionsaufrufe an



Iteration mit while-Schleife

- Eine `while`-Schleife führt den nachfolgenden Block von Statements immer wieder aus, so lange die Bedingung wahr ist

```
int sum_iterative(int n)
{
  int result = 0;
  int i = 0;
  while (i <= n)
  {
    result += i;
    ++i;
  }
  return result;
}
```

- Oft einfacher nachzuvollziehen
- Ist oft etwas expliziter und benötigt mehr Variablen

Iteration mit for-Schleife I

- Viele Schleifen werden wiederholt für verschiedene Werte einer Zähler-Variablen ausgeführt
- C++ hat eine spezielle `for`-Schleife für solche Fälle:

```
int sum_for(int n)
{
  int result = 0;
  for (int i = 0 ; i <= n ; ++i)
  {
    result += i;
  }
  return result;
}
```

- Sagt dem Leser, dass hier über eine Zählervariable iteriert wird
- Beschränkt die Lebenszeit der Zählervariablen auf die Schleife
- Etwas komplizierter als die `while`-Schleife

Iteration mit for-Schleife II

Jede `for`-Schleife kann in eine äquivalente `while`-Schleife umgewandelt werden

```
for (int i = 0 ; i <= n ; ++i)
{
  ...
}
```

wird zu

```
{
  int i = 0;
  while (i <= n)
  {
    ... ;
    ++i;
  }
}
```