Einführung in die Programmierung mit C++

Programmierung = Modifikation von Daten mittels Operationen

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen

Inhalt

Inhalt

Daten:

- Basis
  - ohne Typ (z.B. **void** f())
  - ganze Zahlen (z.B. **int** i)
  - Gleitkommazahlen (z.B. **double** d)
  - darstellbare Zeichen (z.B. **char** c)
  - Wahrheitswert (z.B. **bool** b)
- Zeiger (z.B. **int** *ip)
- Referenzen (z.B. **int** &ir)
- Felder (z.B. **int** i[42])
- Standardbibliothek (z.B. std::vector<**int**>)
- nutzerdefiniert (z.B. **class** K)
- Konstanten

Operationen: Operatoren, Funktionen

Inhalt

Quelltext von Programm wird als Textdatei (z.B. `f.cpp`) auf der Festplatte gespeichert.

Übersetzung mithilfe des Compilers resultiert in einem ausführbaren Programm in Form einer Binärdatei auf der Festplatte (z.B. `a.out`).

Nach Start des Programms wird der Inhalt der Binärdatei in das Codesegment des Hauptspeicher geladen. Zum Ausführen werden die einzelnen Anweisungen in die Befehlsregister der CPU (Central Processing Unit) geladen.

Daten werden im Datensegment des Hauptspeicher an einer eindeutigen Adresse abgelegt und zur Weiterverarbeitung (Arithmetik, Ausgabe, Umwandlung, ...) in die Datenregister der CPU geladen.

Wir werden den Hauptspeicher (leicht vereinfacht) als Feld von Bytes (jeweils 8 Bit, die jeweils die Werte 0 und 1 annehmen können) betrachten.

→ Tafel: Bild Computerarchitektur

- Abfrage von Adressen von Variablen mittels Adressoperator & (im C++ Programm sowie im Debugger) ermöglicht Inspektion des Hauptspeichers

  → Live: Speicherlayout für Tour_MF001.cpp

- Darstellung im Hexadezimalformat (Basis 16) markiert durch Präfix 0x, z.B. $0x1a3c_{16} \hat{=} 6716_{10}$, da

$$12 \cdot 16^0 + 3 \cdot 16^1 + 10 \cdot 16^2 + 1 \cdot 16^3 = 6716;$$

- Die Länge (in Bits) einer "Speicherzelle" bestimmt die Größe des adressierbaren Hauptspeichers. Dessen Maximum beträgt auf
  - 32-Bit-Computern $2^{32}B = 2^{32-10}kB = 2^{32-20}mB = 2^{32-30}gB = 4gB$ ($\hat{=}$20 *EUR*)
  - 64-Bit-Computern $2^{64}B = 2^{64-30}gB = 2^{64-40}tB = 2^{64-50}pB = 2^{64-60}eB = 16eB$ ($\approx$20 Milliarden *EUR*...)

  Preise schwanken.

Inhalt

Variablen

Deklaration und Gültigkeitsbereiche


```
1  int main() {
2    int i=42;
3    {
4      int j;
5      j=24;
6      i=j;
7      j=42;
8    }
9    i=j; // error
10   return 0;
11 }
```

→ gdb

- Deklaration = Zuweisung (Allozieren) von Speicher (z.B. 4 Byte für **int**; Variablen sind Aliasnamen für Speicherbereiche.
- Optimierungen seitens des Compilers können zu Elimination von Variablen (Deallozierung des entsprechenden Speichers) führen.
- Verwendung von Variablen nur innerhalb ihres Gültigkeitsbereichs
- Freigabe des Speichers bei Verlassen des Gültigkeitsbereichs; (z.B. in {...} eingeschlossen)
- einmalige Initialisierung bei Deklaration (z.B. i)
- potentiell wiederholte Zuweisung von Werten (z.B. j)

bool ... Wahrheitswerte **true**, **false**

[**unsigned**] **char** ... darstellbare Zeichen (−> ASCII)
[**unsigned**] **short** ... ganze Zahlen (reduzierter Wertebereich; $[-2^{15}, 2^{15}-1] = [-32768, 32767]$)
[**unsigned**] **int** ... ganze Zahlen (Standardwertebereich; $[-2^{31}, 2^{31}-1] = [-2147483648, 2147483647]$)
[**unsigned**] **long** ... ganze Zahlen (erweiterter Wertebereich; $[-2^{63}, 2^{63}-1]$)
[**unsigned**] **long long** ... ganze Zahlen (maximaler Wertebereich; $\geq [-2^{63}, 2^{63}-1]$)
size_t ... **unsigned int** (zur Iteration über Elemente von C++ Variablen)

float ... Gleitkommazahlen (einfache Genauigkeit; 6 Dezimalstellen)
double ... Gleitkommazahlen (doppelte Genauigkeit, 15 Dezimalstellen)
long double ... Gleitkommazahlen (maximale Genauigkeit; $\geq$ 15 Dezimalstellen)

void ... undefinierter bzw. fehlender (bei Rückgabewerten von Funktionen) Typ

sizeof


```
1  #include<iostream>
2
3  int main() {
4    using namespace std;
5    int i=0,j=1;
6    cout << i << "!=" << j
7         << endl
8         << sizeof(i) << "==" << sizeof(int)
9         << endl;
10   return 0;
11 }
```

generiert Ausgabe

```
0!=1
4==4
```

Von der Basisadresse beginnend, belegen die verschiedenen Datentypen unterschiedliche Anzahlen von Byte, z.B.

```
sizeof(bool)==1
sizeof(char)==1
sizeof(int)==4
sizeof(unsigned int)==4
sizeof(float)==4
sizeof(double)==8
```

Anwendung auf Variablen liefert deren Größe im (statischen) Hauptspeicher.

AUFWACHEN!

Akzeptiert der Compiler den folgenden Code?

```
1 #include<iostream>
2
3 int main() {
4   using namespace std;
5   cout << (sizeof(int)+sizeof(float))/sizeof(double) << endl;
6   return 0;
7 }
```

[Warum nicht?] Falls ja, was wird dann ausgegeben?

Wach?

Akzeptiert der Compiler den folgenden Code?

```
1  #include<iostream>
2
3  int main() {
4    using namespace std;
5    cout << (sizeof(int)+sizeof(float))/sizeof(double) << endl;
6    return 0;
7  }
```

Ja

Falls ja, was wird dann ausgegeben?

1

Inhalt

```
1 char c=8; to_bin(c);
2 int i=8; to_bin(i);
3 float f=8; to_bin(f);
```

```
1 00001000
2 00000000 00000000 00000000 00001000
3 01000001 00000000 00000000 00000000
```

Daten werden als Sequenzen von Byte codiert. Ein Byte enthält acht binäre Wahrheitswerte (Bit $\in \{0, 1\}$). Die Art der Codierung und die Anzahl der belegten Byte hängt vom jeweiligen Datentyp ab.

Mithilfe der durch uns geschriebenen Funktion to_bin() können Variablen in ihre Binärdarstellung konvertiert werden.

Eine prominente Fehlerquelle in C++ stellt die implizite Typumwandlung dar, z.B. bei der Zuweisung des Inhalts einer **double** Variable zu einer **float** Variablen. Der Effekt (hier Genauigkeitsverlust) kann mithilfe der Funktion to_bin() detailliert nachvollzogen werden.

→ Live: Verwendung von to_bin()

Binärcodierung von Daten

to_bin()


```
1  #include "to_bin.hpp"
2
3  int main() {
4    int i=6; to_bin(i);
5    short s=6; to_bin(s);
6    long l=6; to_bin(l);
7    char c='6'; to_bin(c);
8    return 0;
9  }
```

generiert die folgende Ausgabe:

```
1  00000000 00000000 00000000 00000110
2  00000000 00000110
3  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000110
4  00110110
```

Es können Aliasnamen für Datentypen mittels

```
typedef Datentyp Alias;
```

bzw.

```
using Alias=Datentyp;
```

generiert werden, z.B.

```
typedef double T;
using S=int;
```

Wir werden diese Funktionalität vor allem später zur Abkürzung komplexer (langer) Datentypnamen verwenden, z.B.

```
using T=std::list<std::vector<unsigned int>>::reverse_iterator;
```

```
1  int k=42;
2
3  int main() {
4    int i=0;
5    {
6      int k=24;
7      i=k;
8      k=0;
9    }
10   i=k; // ok
11   k=24;
12   return 0;
13 }
```

→ gdb

- globale Variablen sind im gesamten Programm verwendbar
- Freigabe des Speichers erst bei Beenden des Programms
- einmalige Initialisierung bei Deklaration bzw. potentiell wiederholte Zuweisung von Werten
- Dominanz lokaler Variablen gleichen Namens (z.B. k)

Namensbereiche

Definition und Verwendung


```
1  #include<iostream>
2
3  namespace the_answer {
4    int k=42;
5  };
6
7  int main() {
8    using namespace the_answer;
9    {
10     using namespace std;
11     int k=24;
12     cout << k << "!=" << the_answer::k
13          << endl;
14   }
15   std::cout << k // =the_answer::k
16             << std::endl;
17   return 0;
18 }
```

- Namensbereich der Standardbibliothek std
- eigene Namensbereiche (z.B. the_answer
- Zugriff auf Elemente eines Namensbereichs (z.B. std) mittels Präfix-Notation (z.B. std::)
- Präfix überflüssig bei Verwendung des Namensbereichs im (lokalen) Gültigkeitsbereich (z.B. **using namespace** std;)

Konstanten

Definition


```
1  const int i=42;
2
3  int main() {
4    const int j=24;
5    int k;
6    i=j; // error
7    i=24; // error
8    k=i+j; // ok
9    return 0;
10 }
```

- Deklaration mittels **const**
- nicht notwendigerweise Assoziation von Hauptspeicher (Compiler entscheidet)
- einmalige Initialisierung bei Deklaration
- keine Zuweisung von Werten

Wir werden noch weitere Situationen kennenlernen, in denen das Schlüsselwort **const** verwendet werden kann und damit auch verwendet werden sollte. Es erlaubt dem Compiler, besseren (effizienteren) (Assembler-)Code zu erzeugen.

Inhalt

C++ stellt zahlreiche Operatoren zur Verfügung, z.B.

- Zuweisungsoperator (=)
- arithmetische Operatoren (+,−,*,/,%)
- relationale Operatoren (==,!=,>,<,>=,<=)
- logische Operatoren (!,&&,||)
- In(De)kremtierungsoperatoren (++,−− als Prä-(Post-)fix)
- Bitweise Operatoren (&,|,^,~,<<,>>)
- arithmetische Zuweisungsoperatoren
  (+=, −=, *=, /=, %=, >>=, <<=, &=, ^=, |=)
- bedingte Zuweisung (i=j==3?4:2;)

Die Mehrzahl dieser kann in ihrer Semantik modifiziert werden → Überladung.

Operationen

++,−−

- Inkrementierung als
  - Präfixoperator ++i: Zum Wert von i wird 1 addiert und dieser dann im aktuellen Ausdruck verwendet, z.B.

```
int i=42, j; j=++i; // i==43 && j==43
```

  - Postfixoperator i++: Im aktuellen Ausdruck wird der Wert von i verwendet und anschließend zu diesem 1 addiert, z.B.

```
int i=42, j; j=i++; // i==43 && j==42
```

- Dekrementierung als
  - Präfixoperator −−i: Vom Wert von i wird 1 subtrahiert und dieser dann im aktuellen Ausdruck verwendet, z.B.

```
int i=42, j; j=--i; // i==41 && j==41
```

  - Postfixoperator i−−: Im aktuellen Ausdruck wird der Wert von i verwendet und anschließend von diesem 1 subtrahiert, z.B.

```
int i=42, j; j=i--; // i==41 && j==42
```

Die Standardbibliothek stellt zahlreiche Funktionen zur Verfügung, z.B. die folgenden arthmetischen Funktionen:

- trigonometrische Funktionen (z.B. sin, cos)
- hyperbolische Funktionen (z.B. sinh, tanh)
- exponentielle und logarithmische Funktionen (z.B. exp, log)
- Potenzfunktionen (z.B. pow, sqrt)
- Rundungsfunktionen (z.B. ceil, floor)
- weitere Funktionen (z.B. fmax, fabs)

Diese können in ihrer Semantik modifiziert werden → Überladung.

```
1 float x=1.1,y=2.2,z=3.3;
2 z=(2.0*y*sin(x)+y/1e-3)*sqrt(z);
3 std::cout << z << std::endl;
```

generiert Ausgabe 4003.62... .

```
1 int i=1,j=2,k=3;
2 k=++i&j--;
3 std::cout << k << std::endl;
```

generiert Ausgabe 2.

```
1 bool a=false,b=true,c=true;
2 c=!a||b;
3 std::cout << c << std::endl;
```

generiert Ausgabe 1 ($\hat{=}$ true).

```
1 float x=1.1,y=2.2; int i=42;
2 y+=cos(x)+i;
3 i = x<y ? 1 : 2;
4 std::cout << i << std::endl;
```

generiert Ausgabe 1.

AUFWACHEN!

Was wird ausgegeben?

```
1  #include<iostream>
2  #include<cmath>
3
4  int main() {
5    int i=3; float f=cos(0);
6    std::cout << f*(++i)--;
7    int j=i;
8    std::cout << (f+exp(i-3)*5)-++j << std::endl;
9    return 0;
10 }
```

Wach?

Was wird ausgegeben?

```
1  #include<iostream>
2  #include<cmath>
3
4  int main() {
5    int i=3; float f=cos(0);
6    std::cout << f*(++i)--;
7    int j=i;
8    std::cout << (f+exp(i-3)*5)-++j << std::endl;
9    return 0;
10 }
```

42 (*What else ...?*)

Inhalt

C++ hat den Vor- bzw. Nachteil, dass in bestimmten Fällen eine Typumwandlung automatisch (implizit) durchgeführt wird, z.B. bei Zuweisung eines **int**-Wertes zu einer **float**-Variablen.

Bei Umwandlung genauerer Typen in weniger genaue Typen kommt es potentiell zu Informationsverlust (*narrowing*).

Bei der Initialisierung von Variablen kann bzw. sollte implizite Typumwandlung durch Verwendung der {...}-Syntax vermieden werden.

- implizite Typumwandlung
  - **int** i=0.1
  - **int** i(0.1)
- keine implizite Typumwandlung (Übersetzungsfehler)
  - **int** i={0.1}
  - **int** i{0.1}

Bei Deklaration von Variablen inklusive Initialisierung kann der Typ der Variablen durch den Compiler automatisch aus dem Typ des Initialisierungsausdrucks abgeleitet und unter Verwendung von <typeinfo> abgefragt werden, z.B. erzeugt

```
1 #include<iostream>
2 #include<typeinfo>
3
4 int main() {
5   auto x=3.14159265359; std::cout.precision(12);
6   std::cout << x << " " << typeid(x).name() << std::endl;
7   return 0;
8 }
```

die Ausgabe

```
3.14159265359 d
```

wobei `d` für **double** steht.

AUFWACHEN!

Was wird nach erfolgreicher Übersetzung des folgenden Codes ausgegeben?

```
1  #include<iostream>
2  #include<cmath>
3  #include<typeinfo>
4
5  int main() {
6    auto x=acos(0)*sizeof(short);
7    std::cout << x << std::endl;
8    std::cout << typeid(x).name() << std::endl;
9    return 0;
10 }
```

Bemerkung: Definitionsbereich von acos ist $[-1, 1]$. Wertebereich ist $[0, \pi]$.

Wach?

Was wird nach erfolgreicher Übersetzung des folgenden Codes ausgegeben?

```
1  #include<iostream>
2  #include<cmath>
3  #include<typeinfo>
4
5  int main() {
6    auto x=acos(0)*sizeof(short);
7    std::cout << x << std::endl;
8    std::cout << typeid(x).name() << std::endl;
9    return 0;
10 }
```

```
3.14159
d
```

Inhalt

Bei impliziter Typumwandlung kann es zu unerwünschtem Genauigkeitsverlust (*narrowing*) kommen, z.B. erzeugt

```
1  #include<iostream>
2  #include<cmath>
3
4  int main() {
5    double d=exp(1); float f=d;
6    std::cout.precision(15);
7    std::cout << d << " != ";
8    std::cout << f << std::endl;
9    return 0;
10 }
```

die Ausgabe

```
2.71828182845905 != 2.71828174591064
```

Bei expliziter statischer Typumwandlung wird die Zulässigkeit durch den Compiler überprüft. Der Genauigkeitsverlust ist gewünscht bzw. akzeptiert, z.B. erzeugt

```
1  #include<iostream>
2  #include<cmath>
3
4  int main() {
5    double d=exp(1); auto f=static_cast<float>(d);
6    std::cout.precision(15);
7    std::cout << d << " != ";
8    std::cout << f << std::endl;
9    return 0;
10 }
```

die Ausgabe

```
2.71828182845905 != 2.71828174591064
```

Inhalt

Nutzerdefinierte Funktionen

Deklaration und Definition


```
1  #include<iostream>
2
3
4  void g(unsigned long l, float f) {
5    std::cout << l << " "
6          << f << std::endl;
7  }
8
9  unsigned long f(int i);
10
11 int main() {
12   int i=1000000;
13   std::cout << i*i << std::endl;
14   g(f(i),42.42);
15   return 0;
16 }
17
18 unsigned long f(int i) {
19   return i*i;
20 }
```

- nutzerdefinierte Funktionen mit und ohne Rückgabewert
- kein, ein oder mehrere Parameter (auch verschiedenen Typs)
- Simultante Deklaration und Definition (z.B. g)
- (Vorwärts-)Deklaration und separate Definition (z.B. f)
- Schachtelung von Funktionsaufrufen
- Ziel: keine Duplizierung von Quellcode
- Beachte hier: numerische Effekte

Nutzerdefinierte Typgenerische Funktionen

Funktionstemplates


```
1  // naumann@stce.rwth-aachen.de
2  #include<iostream>
3  #include<typeinfo>
4
5  void f(int a) {
6    std::cout << typeid(a).name() << " "
         << a << std::endl;
7  }
8
9  template<typename T>
10 void g(T a) {
11   std::cout << typeid(a).name() << " "
         << a << std::endl;
12 }
13
14 int main() {
15   f(4);
16   f(4.2);
17   g(4);
18   g(4.2);
19   return 0;
20 }
```

- Bauanleitung für den Compiler
- Automatische Generierung der benötigten Varianten (hier für T=**int** und T=**double**)
- Ausgabe

```
i 4
i 4
i 4
d 4.2
```

Nutzerdefinierte Typgenerische Funktionen

Vorwärtsdeklaration und mehrere Templateparameter


```
1  // naumann@stce.rwth-aachen.de
2  #include<iostream>
3
4  void f(int a, float b) {
5    std::cout << a << b << std::endl;
6  }
7
8  template<typename T, typename S>
9  T g(T a, S b);
10
11 int main() {
12   f(4,4.2); f(4.2,4.2);
13   std::cout << g(4,4.2) << std::endl;
14   std::cout << g(4.2,4.2) << std::endl;
15   return 0;
16 }
17
18 template<typename T, typename S>
19 T g(T a, S b) {
20   std::cout << a << b << std::endl;
21   return ++a;
22 }
```

- Ausgabe

```
44.2
44.2
44.2
5
4.24.2
5.2
```

AUFWACHEN!

Was wird ausgegeben?

```
1  // naumann@stce.rwth-aachen.de
2  #include<iostream>
3  #include<cmath>
4
5  template<typename T, typename S>
6  void g(T a, S b);
7
8  int main() {
9    g(3,static_cast<short>(exp(0)));
10   return 0;
11 }
12
13 template<typename T, typename S>
14 void g(T a, S b) {
15   std::cout << a-b << a+b << std::endl;
16 }
```

Wach?

```
1  // naumann@stce.rwth-aachen.de
2  #include<iostream>
3  #include<cmath>
4
5  template<typename T, typename S>
6  void g(T a, S b);
7
8  int main() {
9    g(3,static_cast<short>(exp(0)));
10   return 0;
11 }
12
13 template<typename T, typename S>
14 void g(T a, S b) {
15   std::cout << a-b << a+b << std::endl;
16 }
```

Fast richtig ... 24

Zusammenfassung

Überblick

Computerarchitektur

Variablen

Binärcodierung

Operationen

Initialisierung

Typumwandlung

Nutzerdefinierte Funktionen