Einführung in die Programmierung mit C++

Funktionen

Uwe Naumann

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

RWTH Aachen

Inhalt

Speicherlayout (*stack*)

Variablen und Gültigkeitsbereiche

- Statische Variablen
- Vergleich von Statischen mit Lokalen und Globalen Variablen

Parameterübergabe

- Zeiger
- Felder
- Funktionen
- Kommandozeilenparameter

Inhalt

Speicherlayout (*stack*)

Variablen und Gültigkeitsbereiche
- Statische Variablen
- Vergleich von Statischen mit Lokalen und Globalen Variablen

Parameterübergabe
- Zeiger
- Felder
- Funktionen
- Kommandozeilenparameter

Funktionen

Speicherlayout (*stack*)

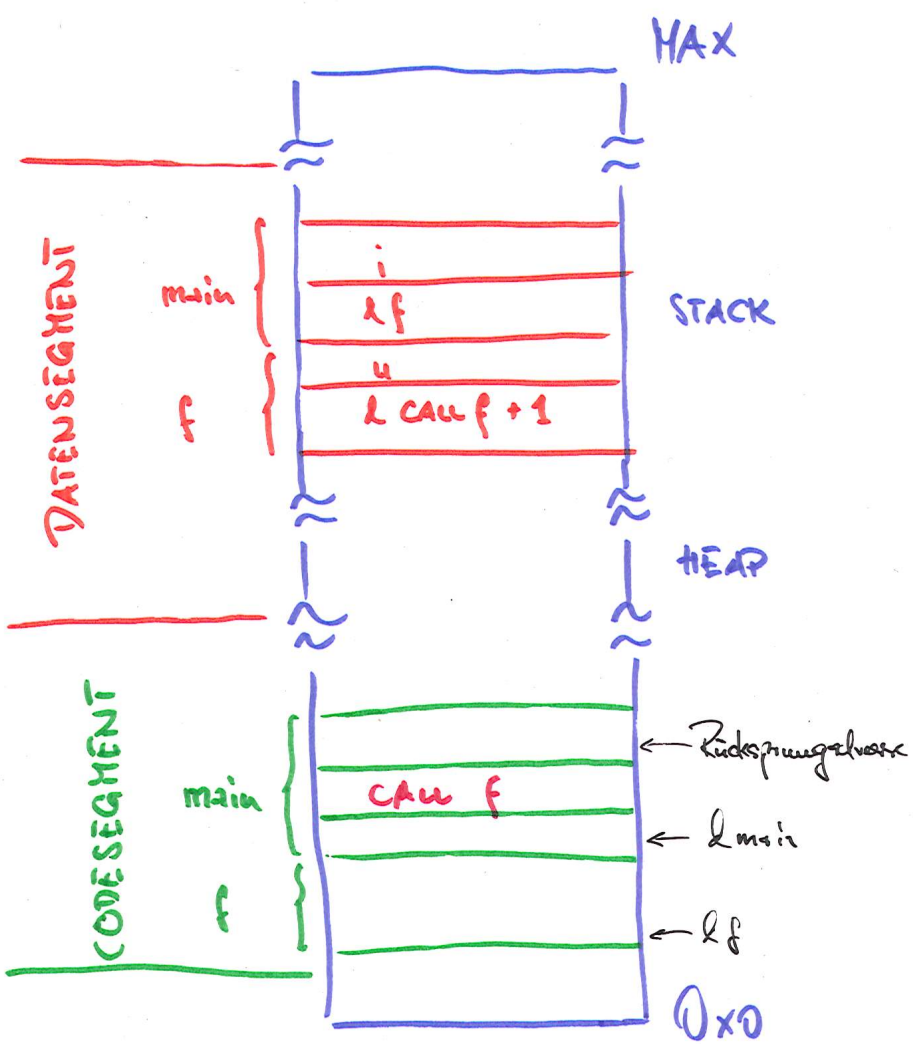

```
1  #include<iostream>
2
3  int h() {
4    int w=42; return w;
5  }
6
7  int g() {
8    int v=h(); return v;
9  }
10
11 int f() {
12   int u=g(); return u;
13 }
14
15 int main() {
16   int i=f();
17   std::cout << i << std::endl;
18   return 0;
19 }
```

- Bei Aufruf einer Funktion wird deren statischer Speicherbereich auf dem *stack* angelegt. Dieser enthält alle *lokalen* Variablen sowie Adressen der aufgerufenen Funktionen (im Codesegment des Hauptspeichers) und entsprechende Rücksprungadressen (im Codesegment) innerhalb der aufrufenden Funktion.
- Bei Verlassen einer Funktion wird deren statischer Speicherbereich wieder freigegeben.

→ Tafel: Speicherlayout mit gdb

```
void f() {
    double u;
}

int main() {
    int i;
    f();
    return 0;
}
```



Inhalt

- Lokale und statische Variablen sind nach ihrer Deklaration nur innerhalb der jeweiligen Funktion verfügbar.
- Globale Variablen sind nach ihrer Deklaration innerhalb der gesamten restlichen Übersetzungseinheit (Quelldatei.cpp und evtl. *.hpp) verfügbar.
- Der Hauptspeicher für lokale Variablen wird bei Verlassen der Funktion wieder freigegeben.
- Der Speicher für statische Variablen bleibt während der Laufzeit des gesamten Programms alloziert und behält seinen Inhalt auch nach verlassen der Funktion.
- Der Speicher für globale Variablen bleibt während der Laufzeit des gesamten Programms alloziert.

```
1  #include<iostream>
2
3  void f() {
4    static int c=0;
5    std::cout << ++c << " ";
6  }
7
8  int main() {
9    f(); f(); f();
10   std::cout << std::endl;
11   return 0;
12 }
```

generiert Ausgabe

```
1 2 3
```

- Statische Variablen werden mittels static deklariert.
- Sie sind nach ihrer Deklaration nur innerhalb der jeweiligen Funktion verfügbar.
- Der durch sie belegte Hauptspeicher bleibt während der Laufzeit des Programms alloziert und behält seinen Inhalt auch nach Verlassen der Funktion.

Statische Variablen

Vergleich mit Lokalen und Globalen Variablen


```
1  #include<iostream>
2
3  int g=1;
4
5  void f() {
6    using namespace std;
7    static int s=0; int l=2;
8    if (!s) cout << &g << " " << &s << " " << &l << endl;
9    cout << ++g << ++s << ++l << endl;
10 }
11
12 int main() { f(); f(); f(); return 0; }
```

generiert z.B. die folgende Ausgabe:

```
0x556da9394010 0x556da9394158 0x7ffff7b40414
213
323
433
```

→ Tafel: Speicherlayout

Inhalt

- Werte
- konstante Werte
- Referenzen
- konstante Referenzen
- Rückgabewerte
- Zeiger
- Felder
  - 1D
  - *n*D
  - Standardbibliothek
- Funktionen
- an main (Kommandozeilenparameter)

Parameterübergabe

Werte (*by value*) vs. Referenzen (*by reference*)

- *call by value*: Alle Modifikationen geschehen auf lokaler Kopie. Der Wert des eigentlichen Arguments wird nicht verändert, z.B.

```
1 void f(double x) { x=0; }
2 ...
3     double v=1;
4     f(v); // v==1
```

- *call by reference*: Alle Modifikationen geschehen auf dem eigentlichem Argument, dessen Wert ausserhalb der Funktion dadurch potentiell verändert wird, z.B.

```
1 void f(double& x) { x=0; }
2 ...
3     double v=1;
4     f(v); // v==0
```

→ Tafel: Speicherlayout

Parameterübergabe

Konstante Werte und Referenzen. Rückgabewerte

- *call by const value:* Lokale Kopie ist *read-only*, z.B.

```
1 void foo(const double x) { /* x=0 nicht erlaubt */ }
```

- *call by const reference:* Es wird Adresse des eigentlichen Arguments übergeben. Letzteres ist *read-only*, z.B.

```
1 void foo(const double& x) { /* x=0 nicht erlaubt */ }
2 ...
3     double v=1;
4     foo(v); // v==1
```

- *copy on return:* Die Rückgabe eines Wertes durch die Funktion geschieht als Kopie einer lokalen Variable gleichen Typs, z.B.

```
1 double foo(const double& x) { double v=0; return v; }
2 ...
3   double w=1;
4   w=foo(3.14); // w==0
```

AUFWACHEN!

Was wird ausgegeben?

```
1  int foo(int& x) {
2     int y=x+x; x++; return y;
3  }
4
5  int bar(int x) {
6     int y=x+x; x++; return y;
7  }
```

1. ???

```
1  int x=2;
2  int y=bar(x); x=foo(y);
3  cout << "x=" << x << endl;
4  cout << "y=" << y << endl;
```

2. ???

```
1  int x=2;
2  int y=bar(foo(x));
3  cout << "x=" << x << endl;
4  cout << "y=" << y << endl;
```

Wach?

```
1 int x=2;
2 int y=bar(x); x=foo(y);
3 cout << "x=" << x << endl;
4 cout << "y=" << y << endl;
```

8
5

```
1 int x=2;
2 int y=bar(foo(x));
3 cout << "x=" << x << endl;
4 cout << "y=" << y << endl;
```

3
8

→ Tafel: Speicherzugriffe

Für Zeiger ergeben sich die folgenden 8 Varianten:

```
1  void v0(int* p) { ++p; *p=42; }
2
3  void v1(const int* p) { ++p; *p=42; }
4
5  void v2(int* const p) { ++p; *p=42; }
6
7  void v3(const int* const p) { ++p; *p=42; }
8
9  void r0(int*& p) { ++p; *p=42; }
10
11 void r1(const int*& p) { ++p; *p=42; }
12
13 void r2(int*& const p) { ++p; *p=42; }
14
15 void r3(const int*& const p) { ++p; *p=42; }
```

Deren Übersetzung resultiert in den folgenden selbsterklärenden Fehlermeldungen.

```
3:32: error: assignment of read-only location ‘* p’
    3 | void v1(const int* p) { ++p; *p=42; }
      |                               ~~^~~
5:27: error: increment of read-only parameter ‘p’
    5 | void v2(int* const p) { ++p; *p=42; }
      |                           ^
7:33: error: increment of read-only parameter ‘p’
    7 | void v3(const int* const p) { ++p; *p=42; }
      |                                 ^
7:38: error: assignment of read-only location ‘*(const int*)p’
    7 | void v3(const int* const p) { ++p; *p=42; }
      |                                    ~~^~~
      |
      ...
```

```
...
11:33: error: assignment of read-only location ‘* p’
   11 | void r1(const int*& p) { ++p; *p=42; }
      |                                 ~~^~~
13:28: error: increment of read-only reference ‘p’
   13 | void r2(int* const& p) { ++p; *p=42; }
      |                            ^
15:34: error: increment of read-only reference ‘p’
   15 | void r3(const int* const& p) { ++p; *p=42; }
      |                                    ^
15:39: error: assignment of read-only location ‘*(const int*)p’
   15 | void r3(const int* const& p) { ++p; *p=42; }
      |                                     ~~^~~
```

AUFWACHEN!

Welche Fehler werden beim Übersetzen des folgenden Programms erkannt?

```
1 void f(int const ** const * const p) {
2     p++;
3     *p++;
4     **p++;
5     ***p++;
6 }
```

Wach?

```
2:3: error: increment of read-only parameter ‘p’
    2 |   p++;
      |   ^
3:4: error: increment of read-only parameter ‘p’
    3 |   *p++;
      |    ^
4:5: error: increment of read-only parameter ‘p’
    4 |   **p++;
      |     ^
5:6: error: increment of read-only parameter ‘p’
    5 |   ***p++;
      |      ^
```

WACH BLEIBEN!

... und hier?

```
1 void f(int const ** const * const p) {
2   p++;
3   (*p)++;
4   (*(*p))++;
5   (*(*(*p)))++;
6   *(*(*p))++;
7 }
```

Noch Wach?

```
zeigerAufwachen_2.cpp: In function ‘void f(const int** const*)’:
2:3: error: increment of read-only parameter ‘p’
    2 |   p++;
      |   ^
3:4: error: increment of read-only location
                                          ‘*(const int** const*)p’
    3 |   (*p)++;
      |   ~~~^~
5:4: error: increment of read-only location
                        ‘* *(const int**)(*(const int** const*)p)’
    5 |   (*(*(*p)))++;
      |   ~~~~~~~~~^~
```

Warum sind Zeilen 4 und 6 korrekt?

Für statische 1D-Felder (konstante Zeiger) ergeben sich theoretisch 10 Varianten, wobei die v0_* semantisch äquivalent sind.

```
1  void v0_1(size_t n, int p[2]) { ++p; p[0]=42; }
2
3  void v0_2(size_t n, int p[]) { ++p; p[0]=42; }
4
5  void v0_3(size_t n, int* p) { ++p; p[0]=42; }
6
7  void v1(size_t n, const int* p) { ++p; p[0]=42; }
8
9  void v2(size_t n, int* const p) { ++p; p[0]=42; }
10
11 void v3(size_t n, const int* const p) { ++p; p[0]=42; }
12
13 ... // r0 und r1 unzulässig, siehe unten
14
15 void r2(size_t n, int* const& p) { ++p; p[0]=42; }
16
17 void r3(size_t n, const int* const& p) { ++p; p[0]=42;
```

Der Aufruf aller Funktionen geschieht wie folgt.

```
1  const size_t n=2;
2  int p[n]={1,2};
3  v0_1(n,p); ...
```

Die folgenden beiden Varianten sind nicht zulässig. Übergabe *by reference* (nicht **const**) wird vom Compiler abgelehnt, da Zeigerkonstanten nicht modifizierbar sind.

```
1  void r0(size_t n, int*& p) { ++p; p[0]=42; }
2
3  void r1(size_t n, const int*& p) { ++p; p[0]=42; }
```

Bis auf v0_* resultieren alle weiteren Funktionen in selbsterklärenden Übersetzungsfehlern.

→ Live: Diskussion aller Übersetzungsfehler

Statische nD Felder werden vorzugsweise explizit übergeben. Die führende Dimension ist für den Compiler irrelevant. Die Größen aller weiteren Dimensionen wird zur Definition der Struktur des nD Feldes benötigt.

```
1  #include<iostream>
2
3  void f(const size_t m, const size_t n/*=3*/, const size_t p/*=2*/, int t[][3][2]) {
4    std::cout << t[m%2][n%2][p%2] << std::endl;
5  }
6
7
8  int main() {
9    const size_t m=2,n=3,p=2;
10   int t[m][n][p]={1,2,3,4,5,6,6,5,4,3,2,1};
11   f(m,n,p,t);
12   return 0;
13 }
```

Man verwendet besser std::array oder std::vector!

Felder der Standardbibliothek können wie jede andere Variable an Funktionen übergeben werden, vorzugsweise jedoch per Referenz bzw. per konstanter Referenz sofern möglich, z.B.

```
1  #include<iostream>
2  #include<vector>
3
4  using VT=std::vector<int>; using MT=std::vector<VT>; using TT=std::vector<MT>;
5
6  void f(const TT& t) {
7    std::cout << t[t.size()%2][t.at(0).size()%2][t.at(0).at(0).size()%2] << std::endl;
8  }
9
10
11 int main() {
12   size_t m=2,n=3,p=2;
13   TT t(m,MT(n,VT(p,42)));
14   f(t);
15   return 0;
16 }
```

Es können auch Zeiger auf Funktionen (Adressen im Codesegment) als Parameter an Funktionen übergeben werden, z.B.

```
1  #include<iostream>
2
3  int f1(int n) { return n%2; }
4
5  int f2(int n) { return n/2; }
6
7  int g(int (*h)(int), int n) {
8    return h(n);
9  }
10
11 int main() {
12   using namespace std;
13   cout << g(f1,5) << endl; // 1
14   cout << g(f2,5) << endl; // 2
15   return 0;
16 }
```

Dabei muss neben Zeigernamen (hier `h`) auch die Signatur (hier `(int)`) der zulässigen Argumentfunktionen angegeben werden.

Kommandozeilenargumente werden an main übergeben.

argc−1 Argumente werden als statisches 1D-Feld argv der Länge argc von 0-terminierten Feldern von **char**-Elementen (ASCII-Zeichenketten bzw. auch: *C-strings*) übergeben.

```
1 int main(int argc, char* argv[]) {
2   std::cout << argc << std::endl;
3   for (int i=0;i<argc;i++)
4     std::cout << argv[i] << std::endl;
5   return 0;
6 }
```

In argv[0] is der aktuelle Aufruf des ausführbaren Programms als Zeichenkette gespeichert.

AUFWACHEN!

Was geht hier schief?

```
1  #include<iostream>
2
3  int main() {
4      using namespace std;
5      char polizeiruf[3]="110";
6      cout << polizeiruf << endl;
7      return 0;
8  }
```

Wach?

```
1  #include<iostream>
2
3  int main() {
4      using namespace std;
5      char polizeiruf[3]="110";
6      cout << polizeiruf << endl;
7      return 0;
8  }
```

Übersetzungsfehler aufgrund eines “zu kurzen” Feldes polizeiruf. Für 0-Terminierung werden 4 **char** Einträge benötigt.

Die Standdardbibliothek stellt Konvertierungsroutinen zur Verfügung, die als Zeichenketten vorliegende ganze oder Gleitkommazahlen in Variablen vom Typ **int** bzw. **float** umwandeln.

```
1 #include<string>
2
3 int main(int c, char* v[]) {
4   int i=std::stoi(v[1]);
5   float f=std::stof(v[2]);
6   return 0;
7 }
```

Erwartet ein Programm Kommandozeilenargumente, so sollten potentielle Fehlerfälle behandelt werden, z.B.

```
1  #include<iostream>
2
3  int main(int c, char* v[]) {
4    if (c==2)
5      std::cout << v[1] << std::endl;
6    else {
7      std::cerr << "Falsche Parameterzahl! (1 erwartet)" << std::endl;
8      return -1;
9    }
10   return 0;
11 }
```

cerr bezeichnet den Standardausgabestrom für Fehlermeldungen (typischerweise auch der Bildschirm).

AUFWACHEN!

Was wird bei Aufruf des Programms mit den Kommandozeilenargumenten `2 2.1` ausgegeben?

```
1  #include<iostream>
2
3  int main(int c, char* v[]) {
4          std::cerr << std::stoi(v[1]) + static_cast<int>(std::stof(v[2]))
5                  << c-1 << std::endl;
6          return 0;
7  }
```

Wach?

Was wird bei Aufruf des Programms mit den Kommandozeilenargumenten `2 2.1` ausgegeben?

```
1 #include<iostream>
2
3 int main(int c, char* v[]) {
4         std::cerr << std::stoi(v[1]) + static_cast<int>(std::stof(v[2]))
5                   << c-1 << std::endl;
6         return 0;
7 }
```

42

Zusammenfassung

Speicherlayout (*stack*)

Variablen und Gültigkeitsbereiche

- Statische Variablen
- Vergleich von Statischen mit Lokalen und Globalen Variablen

Parameterübergabe

- Zeiger
- Felder
- Funktionen
- Kommandozeilenparameter