

Programmierkurs

Steffen Müthing

Interdisciplinary Center for Scientific Computing, Heidelberg University

November 16, 2018

Standardbibliothek

Datenstrukturen

Algorithmen

Variablen und Referenzen

Aufrufkonventionen

Credential Caching in git

Es gibt zwei Möglichkeiten, das ständige Eingeben von Passwörtern in git zu umgehen:

- ▶ Erstellen eines SSH-Schlüssels, Hochladen des Schlüssels auf den Server (über das Webinterface) und aktivieren des SSH agents (siehe Tutorials im Internet, SSH Agent hängt von der Linux-Distribution ab)
- ▶ Aktivieren des **Credential Cache**:

```
git config --global credential.helper cache
```

Durch dieses Kommando wird git auf dem lokalen Rechner so konfiguriert, dass Benutzername und Passwort für 15 min gespeichert werden

C++-Standardbibliothek

- ▶ C++ enthält eine umfangreiche Standardbibliothek mit vielen Datenstrukturen und Algorithmen.
- ▶ Wenn möglich, ist es **immer** besser, Funktionen aus der Standardbibliothek zu nehmen als eigene.
 - ▶ Gut optimiert
 - ▶ Umfangreich getestet
- ▶ Bestandteile:
 - ▶ Datenstrukturen
 - ▶ Algorithmen
 - ▶ Mathematische Funktionen
 - ▶ Input / Output
- ▶ Gute Referenz auf <https://cppreference.com>.

Coding Style

- ▶ Bei der Verwendung der Standardbibliothek sollte das `std::` normalerweise explizit hingeschrieben werden.
- ▶ In besonderen Fällen ist es manchmal erforderlich, eine Funktion ohne `std::` aufzurufen. Diese sollte dann lokal importiert werden:

```
void foo()  
{  
    int a, b;  
    using std::swap;  
    swap(a,b);  
}
```

- ▶ `using namespace std;` hat in einem C++-Programm nichts zu suchen!
 - ▶ Schwierig zu sagen, woher eine Funktion kommt.
 - ▶ Kann zu sehr schwer zu debuggenden Fehlern führen.

Container (Datenstrukturen)

Container bzw. Datenstrukturen verwalten eine zusammenhängende Menge von Werten mit bestimmten Eigenschaften. C++ liefert einige praktische Container mit:

- ▶ `array` - für Listen von Werten, deren Anzahl zur Compile-Zeit bekannt ist
- ▶ `vector` - für Listen von Werten, deren Anzahl zur Compile-Zeit nicht bekannt ist
- ▶ `list` - für Listen von Werten, bei denen man oft in der Mitte Elemente hinzufügen oder entfernen muss
- ▶ `(unordered_)map` - für das Zuordnen von Werten zu Schlüsseln beliebigen Typs (z.B. für ein Wörterbuch). Es gibt eine Variante, die die Einträge nach den Schlüsseln sortiert, und eine, die das nicht tut.
- ▶ `pair` - Ein Paar von Werten, die unterschiedliche Typen haben können
- ▶ `tuple` - Eine Liste von Werten, die unterschiedliche Typen haben können

array

- ▶ Liste, deren Länge zur Compile-Zeit bekannt ist:

```
std::array<Datentyp,Länge>
```

- ▶ Einfachster Typ, der das C++-Container-Interface erfüllt
- ▶ Beispiel:

```
#include <array>
#include <iostream>

int main(int argc, char** argv)
{
    std::array<int,4> a = {{1,2,3,4}};
    std::cout << a.size() << std::endl; // 4
    a[2] = 4;
    std::cout << a[3];
}
```

- ▶ Manchmal sieht man auch C arrays (`int a[4];`), aber besser C++-Version verwenden.

Eigenschaften von Containern

- ▶ Container sind **Objekte** (mehr dazu später).
 - ▶ Objekte haben **member variables** und **member functions**, die zu dem jeweiligen Objekt gehören und darin gespeichert werden oder auf dieses wirken:

```
p.first = 3; // member variable  
a.size();  // member function
```

- ▶ Member function werden auch **Methoden** genannt.
- ▶ Container sind **Templates**. Templates sind parametrisierte Typen, die als Templateparameter in spitzen Klammern andere Typen oder Konstanten übergeben bekommen.
 - ▶ Bei Containern wird hierüber z.B. festgelegt, welchen Typ von Werten sie speichern können.
 - ▶ Mehr zu Templates später.
- ▶ Der Zugriff auf Inhalte eines Containers erfolgt meistens mit eckigen Klammern:

```
my_array[3];  
my_map["key"];
```


vector

Dynamisch anpassbare Liste von Objekten:

```
#include <vector>
#include <iostream>

int main(int argc, char** argv)
{
    std::vector<int> a; // leere Liste
    std::vector<int> b = {{1,2,3,4}};
    std::cout << b.size() << std::endl; // 4
    b[2] = 4;
    a = b; // kopiert den Inhalt
    a.resize(100); // Grösse anpassen
    a.push_back(1); // Wert 1 hinten anfügen
    a.pop_back(); // Letztes Element entfernen
}
```

- ▶ Bei grossen Listen (> 100-1000) immer besser als `std::array`.

vector

Dynamisch anpassbare Liste von Objekten:

```
#include <vector>
#include <iostream>

int main(int argc, char** argv)
{
    std::vector<int> a; // leere Liste
    std::vector<int> b = {{1,2,3,4}};
    std::vector<int> c(20); // Liste mit 20 Einträgen
    std::cout << b.size() << std::endl; // 4
    b[2] = 4;
    a = b; // kopiert den Inhalt
    a.resize(100); // Grösse anpassen
    a.push_back(1); // Wert 1 hinten anfügen
    a.pop_back(); // Letztes Element entfernen
}
```

- ▶ Bei grossen Listen (> 100-1000) immer besser als `std::array`.

Iterieren über Container

```
std::vector<int> v = {{1,2,3,4}};  
for (int i = 0 ; i < v.size() ; ++i)  
    std::cout << v[i] << std::endl;
```

- ▶ Aufpassen am Ende!
- ▶ Index wird nicht auf Gültigkeit überprüft.
- ▶ Bei Fehlern: Ersetze `v[i]` durch `v.at(i)` oder kompiliere mit `-fsanitize=address`.

Iterieren über Container

```
std::vector<int> v = {{1,2,3,4}};  
for (int i = 0 ; i < v.size() ; ++i)  
    std::cout << v[i] << std::endl;
```

- ▶ Aufpassen am Ende!
- ▶ Index wird nicht auf Gültigkeit überprüft.
- ▶ Bei Fehlern: Ersetze `v[i]` durch `v.at(i)` oder kompiliere mit `-fsanitize=address`.

Vereinfachter Loop zum Iterieren über Standard-Container:

```
std::vector<int> v = {{1,2,3,4}};  
for (int entry : v)  
    std::cout << entry << std::endl;
```

- ▶ Funktionert für alle Standard-Container.
- ▶ Besser lesbar.
- ▶ Keine Gefahr, Fehler am Ende zu machen (`<` vs. `≤`).

pair und tuple

Kurze Listen, die Werte von unterschiedlichem Typ speichern können.

- ▶ pair speichert genau zwei Werte. Oft benutzt, um zwei Werte aus einer Funktion zurückzugeben:

```
#include <utility>

std::pair<std::string,int> nameAndGrade(int matrikelNr) {
    return std::make_pair("Max",2);
}

std::pair<std::string,int> r = nameAndGrade(42);
std::cout << "Name: " << r.first << std::endl
           << "Note: " << r.second << std::endl;
```

pair und tuple

Kurze Listen, die Werte von unterschiedlichem Typ speichern können.

- ▶ tuple speichert mehrere Werte. Etwas komplizierterer Zugriff

```
#include <tuple>

std::tuple<int,int,int> birthday(int matrikelNr) {
    return std::make_tuple(1,1,1990);
}

std::tuple<int,int,int> bday = birthday(42);
std::cout << "Tag: " << std::get<0>(bday) << std::endl
          << "Monat: " << std::get<1>(bday) << std::endl
          << "Jahr: " << std::get<2>(bday) << std::endl;
```

Sortieren

- ▶ C++ hat einen hochoptimierten, eingebauten Sortier-Algorithmus.
- ▶ Der Algorithmus basiert auf *Iteratoren*, was im Moment aber bis auf die Syntax zum Aufrufen egal ist:

```
#include <vector>
#include <algorithm>

int main(int argc, char** argv)
{
    std::vector<int> a = .....;
    // sortiert a nach aufsteigenden Zahlenwerten
    std::sort(a.begin(), a.end());
}
```

Variablen (Wiederholung)

- ▶ Variablen repräsentieren eine Stelle im Arbeitsspeicher, an der Daten eines bestimmten Typs gespeichert sind.
- ▶ Jede Variable hat einen Namen und einen Typ.
- ▶ Der Speicherbedarf einer Variablen
 - ▶ hängt von ihrem Typ ab.
 - ▶ kann mit dem Operator `sizeof(var)` oder `sizeof(type)` abgefragt werden.
- ▶ Die Stelle im Speicher, an der der Wert einer Variablen gespeichert ist, kann nicht verändert werden.

Konstante Variablen

- ▶ Variablen in C++ können mit dem Keyword `const` als konstant (unveränderlich) deklariert werden.
- ▶ Eine konstante Variable kann nicht mehr verändert werden, nachdem sie definiert wurde:

```
const double pi = 3.1415926535;  
pi = pi + 1; // compile error
```

- ▶ Konstante Variablen können helfen, Programmierfehler zu vermeiden.
- ▶ Unter bestimmten Umständen kann der Compiler schnelleren Code generieren, wenn Variablen konstant sind.

Referenzen

- ▶ Referenzen sind zusätzliche Namen für existierende Variablen.
- ▶ Der Typ einer Referenz ist der Typ der existierenden Variablen gefolgt von `&`. Der Typ einer Referenz auf `int` ist also `int&`.
- ▶ Eine Referenz wird **immer** in dem Moment initialisiert, in dem sie definiert wird:

```
int x = 4;  
int& x_ref = x;  
int& no_ref; // compile error!
```

- ▶ Eine Referenz zeigt immer auf die gleiche Variable.
- ▶ Die Referenz verhält sich genau so wie die Original-Variable.
- ▶ Änderungen an der Referenz verändern auch die Original-Variable und umgekehrt.

Referenzen: Beispiel

```
# include <iostream>

int main ()
{
    int a = 12;
    int & b = a; // definiert Referenz
    int & c = b; // Referenz auf Referenz
    float & d = a; // falscher Typ, Compile-Fehler
    int e = b;
    b = 2;
    c = a * b;
    std :: cout << a << std :: endl ; // 4
    std :: cout << e << std :: endl ; // 12
}
```

Iterieren über Container (II)

- ▶ Wenn man den Inhalt eines Containers beim Iterieren verändern will, muss man für die Variable einen Referenz-Typ verwenden:

```
#include <vector>
#include <iostream>

int main(int argc, char** argv)
{
    std::vector<int> v = {{1,2,3,4}};
    for (int& value : v)
        value *= 2;
}
```

Call by Value

- ▶ Wenn eine Funktion mit einem normalen Parameter aufgerufen wird, erstellt C++ eine Kopie des Parameterwerts, mit dem die Funktion dann arbeitet:

```
double square(double x);
```

- ▶ Diese Aufrufkonvention heißt *call by value*.
- ▶ Bisher haben wir in den Übungen nur call by value verwendet.
- ▶ Weil die Funktion eine Kopie der Original-Variablen bekommen hat, wirken sich Änderungen in der Funktion nicht auf die Original-Variable aus.

Call by Reference

- ▶ Wenn eine Funktion mit einem Referenz-Parameter aufgerufen wird, übergibt C++ eine Referenz auf die Original-Variable an die Funktion:

```
void sort(std::vector<int>& x);
```

- ▶ Diese Aufrufkonvention heißt *call by reference*.
- ▶ Funktionen mit dieser Aufrufkonvention können Werte außerhalb der Funktion verändern, ohne dass dies beim Aufruf direkt ersichtlich ist (sie haben *Seiteneffekte*).
- ▶ Nicht-triviale Datentypen wie `std::vector` sollten normalerweise per Referenz übergeben werden, weil das Kopieren teuer sein kann.

Dangling References

- ▶ Intern sind Referenzen eine spezielle Art von konstanter Variable, die den Speicherort der Original-Variablen enthält.
- ▶ Wenn die Original-Variable aufhört zu existieren, wird ihr Speicherort ungültig.
- ▶ Referenzen auf die nicht mehr existierende Variable greifen weiter auf den ungültigen Speicherort zu
⇒ Programm stürzt ab oder liefert falsches Ergebnis!
- ▶ Tipps für Referenzen:
 - ▶ Referenzen sind oft gut für Funktionsparameter (call by reference).
 - ▶ Niemals Referenzen als Rückgabewert von normalen Funktionen verwenden!