Praxis der Programmierung

Strings, Aufteilung der Quelltexte

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Klausurtermin

Änderung des Klausurtermins:

Klausur in der letzten Vorlesung:
Mittwoch, 16.07.2014, 12:30 bis 14:00 Uhr im Hörsaal 1

Strings

Arten von Zeichenketten

- **C**-Zeichenketten
  - `char`-Array (*also* Pointer auf `char`)
  - null-terminiert (letztes Element ist `'\0'`)
  - String-Konstanten der Form `"Hallo Text!"` werden als **C**-Strings angelegt
- Klasse `string` der Standardbibliothek
  - einfacherer Umgang mit Strings
    (keine Überläufe der Arraygröße, keine Nullterminierung)
  - Objekt fordert nötigen Speicherplatz selbst an
  - viele komfortable Methoden vordefiniert

```
#include <string>
using namespace std;

string str;   // Aufruf des Standardkonstruktors
```

Die Klasse `string`

- Konstruktoren
  - Standardkonstruktor (ermöglicht spätere Initialisierung)
  - Initialisierungskonstruktor mit einer String-Konstante als Parameter
  - `string(int n, char c)` erzeut String aus `n` Zeichen `c`

```
string name;
name = "Programmer";
string ort("Bithausen");
string trennlinie(60, '-');
```

- Kompatibilität zu **C**-Strings
  - **C**-String $\longrightarrow$ `string`: per Zuweisung oder Parameterübergabe
  - `string` $\longrightarrow$ **C**-String: Methode `c_str()`

    $\rightsquigarrow$ konstanter `char`-Pointer auf die Zeichenkette

    $\rightsquigarrow$ unveränderlich; mit `strncpy()` in ein `char`-Array überführen

Beispiel Konvertieren in einen C-String

```
#include <string>
#include <cstring>
using namespace std;

int main() {
   string ort("Potsdam");
   const char *cStringPointer = ort.c_str();

   char veraenderlich[60];
   strncpy(veraenderlich, cStringPointer, 60);
   veraenderlich[0] =  'p';   // ohne Probleme
   ort[1] = 'O';              // verdirbt cStringPointer
   cStringPointer[1] = 'u'    // verboten
}
```

Einige Methoden der Klasse `string` (1)

- `append(str)` fügt `str` hinten an
  Alternative: `s1 = s1 + s2;` `// auch  s1 += s2;  ist s1.append(s2);`
- `at(i)` liefert Zeichen an Position `i`
  Alternative: `str[i];` `// ist str.at(i);`
- `clear()` löscht alle Zeichen des Strings
- `insert(p, str)` fügt `str` an Position `p` ein
- `erase(p, n)` entfernt ab Position `p` `n` Zeichen
- `replace(p, n, str)` ersetzt ab Position `p` `n` Zeichen durch den String `str`

Einige Methoden der Klasse `string` (1)

- `length()` und `size()` liefern die Länge des Strings
- `substr(p,n)` liefert die Teilzeichenkette ab Position `p` der Länge `n`
- `compare(str)` vergleicht den String mit `str`
- `empty()` gibt `true` zurück, wenn der Sring leer ist

weitere Methoden auf

`http://www.cplusplus.com/reference/`

`http://en.cppreference.com/w/cpp`

$\rightsquigarrow$ umfassende Dokumentationen der Klassen der Standardbibliothek

Aufgabe 1

String-Iteratoren

- `string`-Methoden `begin()` und `end()` liefern je ein Objekt vom Typ `string::iterator`
- Iteratoren sind Pointer, die `string`-Objekte durchlaufen (Pointerarithmetik!)
- `begin()`-Pointer zeigt auf erstes Zeichen des Strings
- `end()`-Pointer zeigt hinter das letzte gültige Zeichen des Strings
- Vergleich mit dem `end()`-Pointer zeigt an, dass das Stringende erreicht wurde
- *"symmetrisch":* `string::reverse_iterator, rbegin(), rend()`
- Gültigkeit nach Veränderung des Strings nicht mehr garantiert
  $\rightsquigarrow$ immer `begin()` unmittelbar vor Gebrauch des Iterators!

String-Iteratoren – Beispiel

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string s = "Testlauf";
    string::iterator i;
    for (i = s.begin(); i != s.end(); i++)
        cout << *i;
    cout << endl;
}
```

String-Vergleiche

ohne Bibliotheksfunktionen möglich:

==	gleich
!=	ungleich
<	in lexikalischer Reihenfolge vorher
>	in lexikalischer Reihenfolge hinterher
<=	in lexikalischer Reihenfolge vorher oder gleich
>=	in lexikalischer Reihenfolge hinterher oder gleich

Umwandlung von oder in Zahlen

- Bibliothek `sstream` stellt Klassen `istringstram` und `ostringstream` bereit
- diese Klassen erlauben, Daten mit >> bzw. << in bzw. aus Strings umzuleiten
- `ostringstream` stellt Methode `str()` zur Umwandlung des Streams in ein Exemplar der Klasse `string` zur Verfügung

```
int zahl = 0;
string s = "123";
istringstream in(s);
in >> zahl;   // zahl = 123
```

```
zahl = 3778;
ostringstream os;
os << zahl;
s = os.str();   // s = "3778"
```

Aufteilung der Quelltexte

Leitgedanken bei der Quelltext-Aufteilung

- je Klassendefinition eine Datei
- Comiler-Aufruf zum Erzeugen der Objektdateien (`.o`) für jede beteiligte Datei einzeln möglich: `g++ -Wall -c datei.cpp`
- Linker wird aktiv bei Aufruf von g++ ohne Option oder mit Option `-o`
  - alle beteiligten Dateien als Argumente übergeben (`.cpp` oder `.o`, `.h`)
  - genau eine beteiligte Datei hat eine `main`-Funktion
  - es entsteht eine ausführbare Datei
- weitere Aufteilung einer Klasse in
  - eine Header-Datei `.h` mit der Definition der Klasse, in der Datenelemente und Methoden nur deklariert werden
  - eine `.cpp`-Datei mit den Implementierungen der Methoden

Aufgabe 2

Vermeiden von Doppel-Definitionen mit dem Präprozessor

```
#ifndef _DATE_h      // nur, falls die Variable _DATE_h nicht definiert ist
#define _DATE_h      // wird diese Konstante
class Date {         // und die Klasse Date definiert
   ...
};
#endif               // sonst wird alles bis hier uebersprungen
```

Systematisierende Übung

Aufgabe 3