Programmiertechnik II

Ralf Herbrich

C++ Einführung

C++

- Entwickelt in den 1980ern von Bjarne Stroustrup, AT&T Bell Labs
  - Ursprünglich „C with Classes“
  - cfront: Übersetzer von C++ nach C
- **Ziel**: Objektorientierte Programmierung in C
  - Klassen, Methoden, Mehrfachvererbung
  - später auch Templates, Ausnahmebehandlung, Namespaces
- **Standardisierung**:
  - 1998: ISO/IEC 14882:1998 (C++98)
  - 2011: ISO/IEC 14882:2011 (C++11)
  - 2017: ISO/IEC 14882:2017 (C++17)

Bjarne Stroustrup (1950 –)

Überblick

1. Unterschiede zu C
   - Kommentare, Initialisierungen & Speicherverwaltung
   - Primitive Datentypen
   - Zeiger & Referenzen
   - `const` Operator
   - Funktionen & (Operator)-Overloading
   - Templates
2. Abstrakte Datentypen & Klassen in C++
   - Vererbung
   - Spezielle Konstruktoren
3. Standard Template Library

Überblick

1. **Unterschiede zu C**
   - Kommentare, Initialisierungen & Speicherverwaltung
   - Primitive Datentypen
   - Zeiger & Referenzen
   - `const` Operator
   - Funktionen & (Operator)-Overloading
   - Templates
2. Abstrakte Datentypen & Klassen in C++
   - Vererbung
   - Spezielle Konstruktoren
3. Standard Template Library

C++ Kommentare, Initialisierungen, Speicherverwaltung

- **Kommentare**: Es gibt zwei Arten, Kommentare ins Programm einzufügen
  - `/*...*/` Kommentare, die sich über mehrere Zeilen erstrecken (wie C)
  - `//...` Kommentare, so dass der Compiler den Rest der Zeile ignoriert
- **Initialisierungen** bei Deklaration haben eine zusätzliche Syntax, die verlustvolle Typkonvertierungen erkennen und explizite Zuweisung verhindern: `T variable { value }` anstatt `T variable = { value }` oder `T variable = value`
  - **Beispiel**:
    - `int i = 7.2;  // warning is generated but compiles`
    - `int i { 7.2 }; // compile time type error in C++`
- **(Dynamische) Speicherverwaltung** ist Teil der Sprache mit den beiden Schlüsselworten `new` und `delete` (und `new[]` und `delete[]` für Felder)
  - **Beispiel**:
    - `int* arr = new int[10];`
    - `delete[] arr;`

C++ Primitive Datentypen

- **Primitive Datentypen sind ähnlich zu C mit einer Ausnahme!**
  - `bool`: 1 Byte (mögliche Werte: `true` und `false`)
  - `char`: 1 Byte
  - `int`: 4 Bytes (mit möglichen Qualifizierern `short`, `long` und `long long`)
  - `float`: 4 Bytes
  - `double`: 8 Bytes (mit möglichem Qualifizierer `long`)
- **Automatische Typinferenz (C++11)**
  - C++ ist eine statisch getypte Sprache (d.h. Typ aller Variablen **muss** zur Compilezeit ermittelt werden können)
  - Wenn der Typ automatisch bestimmt werden kann vom Compiler, dann immer das `auto` Schlüsselwort benutzen.
  - **Beispiel**:
    - `auto x = 3 + 4;`
    - `for(auto i = 0; i < 100; i++) { … }`

types.cpp

Zeiger und Referenzen

- **Zeiger *(pointer)* funktionieren in C++ genauso wie in C**
  - `T*`: Zeiger auf ein Objekt des Typs `T`
  - Kann beliebig zur Laufzeit verändert werden (Zeigerarithmetik und `&`-Operator)
  - `==` Operator vergleicht Speicheradressen
  - Inhalt des Speichers kann mit `*`-Operator gelesen (*rvalue*) und geschrieben (*lvalue*) werden
  - Muss nicht initialisiert werden
  - Belegt immer Speicherplatz, die der Größe des Adressraums entspricht (z.B. 32 bit)

- **Referenzen (*reference*) sind wie einmalig intialisierte Zeiger (C++)**
  - `T&`: Referenz auf ein Objekt des Typs `T`
  - Kann nicht zur Laufzeit verändert werden (keine Zeigerarithmetik und `&`-Operator)
  - `==` Operator vergleicht referenzierte Objekte
  - Inhalt des Speichers wird direkt mit dem Variablennamen gelesen (*rvalue*) und geschrieben (*lvalue*)
  - **Muss** initialisiert werden
  - Belegt nicht notwendigerweise Speicherplatz (Compiler darf Referenz durch die eigentliche referenzierte Variable ersetzen)

pointers.cpp

`const` Operator

- `const` spielt eine **zentrale Rolle** in C++ aus zwei Gründen:
  1. Der Compiler kann `const`-Ausdrücke optimieren (Register, Konstanten)!
  2. Korrektheit von Programmsemantik (*const correctness*)

- **Primitiver Datentyp**: `const` bezieht sich auf den Typ direkt dahinter.
  - `const` Variablen müssen initialisiert werden und können kein *lvalue* sein

```
const int x = 1;
x = 2;                          // would be a compile time error
const int y = x + 2;            // this is ok; it's a rvalue
```

  - `const` Argumente dürfen nicht in der Funktion verändert werden

```
int f(const int x) {
  x = 2;                        // would be a compile time error
}
```

`const` Operator: Zeiger

- **Zeiger**: `const` des Zeigers bezieht sich auf den Zeigertyp direkt davor
  - **Beispiel**:

```
int x { 42 }, y { 12 };
int* const p { &x };

p = &y;                 // this is a compile time error
```

  - `const` Zeiger dürfen ihre Adresse nicht ändern aber der Inhalt des Speichers, auf den der Zeiger zeigt, ist **nicht** `const`!
  - **Beispiel (*ctd*)**:

```
*p += 1;                // but this is ok!
```

- **Referenzen vs. `const` Zeiger: Unterschiede**
  1. Ein `const` Zeiger kann auf **NULL** zeigen; eine Referenz nicht.
  2. Ein `const` Zeiger belegt Speicher und hat seine eigene Adresse.
  3. Ansonsten sind die beiden Konzepte gleich!

const.cpp

Funktionen & (Operator)-Overloading

- **Funktionen** werden genauso deklariert wie in C
  - `<ret-type> <function-name>(<type1> <arg1>, <type2> <args2>, …) {…}`
- **Drei neue Erweiterungen**:
  1. Argumente können **Standardwerte** haben (mit = bei Deklaration angeben)

```
void my_print(char *s, int width=80) { … }
```

  2. Funktionen können **überladen**, wenn die Argumenttypen sich unterscheiden

```
int f(int x, int y) { return (x + y); }
int f(double x, double y) { return ((int) x * (int) y); }
f(2,3)          // this returns 5
f(2.0,3.0)      // this returns 6
```

  3. Alle C++ Operatoren (z.B. Addition, Multiplikation) können auch überladen werden

```
MyType operator+(const MyType& a, const MyType& b) { … }
```

overloading.cpp

Templates

- Oft wollen wir uns bei der Implementierung von Funktionen und Strukturen **noch nicht auf** Argument- oder *Member*-**Typen festlegen**.
  - Verkettete Liste (von beliebigen Wertetypen!)
  - Nullstellverfahren (alle Typen, die Addition und Multiplikation unterstützen)
- C++ bietet Templates an, mit denen man **Typvariablen** definieren kann
  - `template <typename T>`
- Anstelle des eigentlichen Typs wird dann die Typvariable `T` benutzt
  - **Beispiel**:

```
List<T>* add_element(const List<T>* head, const T value) { … }
```

- Bei jeder Benutzung der Funktion oder des Typs während der Compilierung generiert der Compiler dann **den speziellen Code** für den eigentlichen Typ

```
const List<int>* i_list = NULL;
i_list = add_element(i_list, 42);
```

template.cpp

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Viel Spaß bis zur nächsten Vorlesung!