

Übungspaket 6

Polymorphie und Standardwerte in cpp

Übungsziele:

1. Definition Parametern mit Standardwerten
2. Definition und Implementierung von Methoden mit unterschiedlichen Signaturen

Literatur:

C++-Skript¹, Kapitel 22 und 23

Semester:

Wintersemester 2020/21

Betreuer:

Theo und Ralf

Synopsis:

Objektorientierte Programmiersprachen wie C++ bieten die Möglichkeit, Methoden (Funktionen) gleichen Namens aber unterschiedlichen Parametertypen zu definieren. Desweiteren ist es in derartigen Programmiersprachen oft möglich, Standardwerte für ausgewählte Parameter zu spezifizieren. Aufgabe des vorliegenden Übungspaketes ist es, dies anhand kleiner Beispiele zu üben.

¹www.amd.e-technik.uni-rostock.de/ma/rs/lv/hoqt/script.pdf

Teil I: Stoffwiederholung

Aufgabe 1: Signaturen

Was versteht man unter der Signatur einer Funktion bzw. Methode? Erläutere dies kurz anhand eines kleinen Beispiels.

Beantworte die folgenden Fragen zum Thema Signaturen, wobei der Begriff Funktion immer für Funktionen und Methoden steht.

Ist der Rückgabotyp einer Funktion Teil ihrer Signatur?

Gehört das Schlüsselwort `const` zur Signatur einer Funktion?

Können strukturierte Datentypen Teil der Signatur sein?

Welche Signatur hat die Struktur `struct{ int i; char c; }`?

Was ist bei der Definition von Methoden und Funktionen hinsichtlich ihrer Signatur zu beachten, wenn sie identische Namen haben?

Aufgabe 2: Standardwerte

Beschreibe mit eigenen Worten, was mit unter Standardwerten (für Parameter) versteht.

Wo können Standardwerte definiert werden?

1. 2.

Welche beiden Dinge sind bei der Definition von Standardwerten zu beachten?

1.

2.

Teil II: Quiz

Aufgabe 1: Signaturen

In dieser Aufgabe geht es darum, die Signaturen einer Funktion zu bestimmen. Neben den Basistypen haben wir noch die folgenden beiden Definitionen

```
1 typedef struct {
2     int i;
3     double d;
4 } T1;

1 typedef struct {
2     char c;
3     int *ip;
4 } T2;
```

Wie lauten die Signaturen der beiden Datentypen T1 und T2?

T1: T2:

Vervollständige die folgende Tabelle.

Funktion	Signatur	Anmerkung
f(char c1, char c2)
f(int i, int j, int k)
f(double, int, * double)
f(int, T1)
f(T2 * p)
f(int, const int, const double)
f(const int, const T1 *p)

Aufgabe 2: Polymorphie

Nehmen wir an, wir hätten die folgende Klassendefinition:

```
1 void m( int i ) { printf( "int: i=%d\n", i ); }
2 void m( double d ) { printf( "double: d=%3.1f\n", d ); }
```

Vervollständige die folgende Tabelle, wobei das Objekt c von der Klasse C wie folgt definiert ist: C c.

Aufruf: c.m(1) Ausgabe:

Aufruf: c.m(3.0) Ausgabe:

Aufgabe 3: Standardwerte

Das folgende Programm dient als Grundlage für unser nächstes Quiz

```
1  #include <stdio.h>
2
3  class C {
4      public:
5          void m( int i, int j, int k = 1, int l = 4711 );
6      };
7
8  void C::m( int i = 0, int j = 13, int k, int l )
9      {
10         printf( "i=%4d j=%4d k=%4d l=%4d\n", i, j, k, l );
11     }
12
13 int main( int argc, char **argv )
14     {
15         C c;
16         c.m( 0, 0, 0, 0 );
17         c.m( 1, 2, 3, 4 );
18         c.m( 2, 2, 2 );
19         c.m( 3, 4 );
20         c.m( 4 );
21         c.m();
22         return 0;
23     }
```

In diesem Beispielprogramm hat die Klasse `C` lediglich eine Methode namens `m` (Zeile 5). Die Aufgabe dieser Methode ist es, die Werte seiner Parameter `i`, `j`, `k` und `l` auszugeben. Die zugehörigen Standardwerte werden sowohl in Zeile 5 als auch Zeile 8 festgelegt. In der `main()`-Funktion legen wir ein Objekt dieser Klasse an und erzeugen zeilenweise entsprechende Methodenaufrufe.

Vervollständige folgende Tabelle:

Zeile	Aufruf	Ausgabe
16	<code>c.m(0, 0, 0, 0)</code>
17	<code>c.m(1, 2, 3, 4)</code>
18	<code>c.m(2, 2, 2)</code>
19	<code>c.m(3, 4)</code>
20	<code>c.m(4)</code>
21	<code>c.m()</code>

Teil III: Fehlersuche

Aufgabe 1: Fehlerhafte Methoden und Parameter

DR. POLY von polytechnischen Hochschule in Lausanne ist begeistert von der Möglichkeit, mehrere Methoden mit dem selben Namen zu definieren und für ausgewählte Parameter Standardwerte bereitzustellen. Allerdings hat er noch nicht alle Details im Skript verstanden, sodass der Compiler seine ersten Versuche nicht so ganz mag. Sein aktuelles Programmzustand sieht wie folgt aus:

```
1  class C {
2      public:
3          void m_1( int i, double d );
4          void m_1( double d, int i );
5          void m_2( char c1 = 'a', char c2 = 'b' );
6          void m_3( int i = 4711, char c );
7          void m_4( int i );
8          void m_4( int i, int j = 0 );
9          void m_5( int i );
10         void m_5( const char c );
11         void m_6( double d );           3 mal die selbe Signatur!
12         void m_6( const double d );    Signatur nur abhängig von Datentypen
13         int  m_6( double d );          und deren Reihenfolge
14     };
15
16 int main( int argc, char ** aargv )
17 {
18     C c;
19     c.m_1( 3, 4 );                     Keine Eindeutige Wertezuweisung!
20     c.m_1( 3.0, 4.0 );
21     c.m_2( , 'z' );                   Fehlendes char!
22     c.m_3( 1000, 'z' );
23     return( 0 );
24 }
```

Moment ist DR. POLY noch im Modus Fehlerfinden. Daher geben alle Methoden nur ihren Namen, ihre Signatur und die Werte der aktuellen Parameter aus. Eine Beispielimplementierung könnte wie folgt aussehen:

```
1  int C::m_6( double d )
2  {
3      printf( "m_6 double %e\n", d );
4  }
```

Teil IV: Anwendungen

Aufgabe 1: Anwenden des Polymorphiekonzeptes

Im Rahmen von Übungspaket 5 haben wir ein Programm entwickelt, das Zeichen und Zeichenketten an einen bestehenden Puffer anhängt und diesen ggf. vergrößert. Als statische und stark abgespeckte Variante könnte ein Teil dieses Programms wie folgt aussehen:

```
1  #include <stdio.h>
2
3  #define BUF_SIZE          128
4  class BUF {
5      private:
6          char buf[ BUF_SIZE ];    // the actual buffer
7          int  len;                // number of used bytes
8      public:
9          char *get_buf();          // returning buf
10         int  add_c ( const char c ); // add one c
11         int  add_str( const char *s ); // add a string
12         int  reset ();            // reset: setting len = 0
13     };
14 int BUF::reset() { len = 0; return BUF_SIZE; }
15 char *BUF::get_buf() { return buf; }
16
17 int BUF::add_c( const char c )
18 {
19     if ( len < BUF_SIZE )          // space left ?
20         buf[ len++ ] = c;          // add the char
21     return BUF_SIZE - len;         // returning remaining size
22 }
23 int BUF::add_str( const char *s )
24 {
25     while( *s )                    // still some characters?
26         add_c( *s++ );              // add the next char
27     return BUF_SIZE - len;         // returning remaining size
28 }
29
30 int main( int argc, char **argv )
31 {
32     BUF buf; buf.reset();
33     buf.add_str( "Hi" ); buf.add_c( ' ' );
34     buf.add_str( "you!" ); buf.add_c( '\0' );
35     printf( "buf='%s'\n", buf.get_buf() );
36 }
```

Schau dir zunächst die Klassendefinition und die beiden Methoden `add_c()` und `add_str()` an und versuche zu verstehen, wie sie funktionieren. Obiges Hauptprogramm (`main()`) würde den folgenden Text ausgeben: `buf='Hi you!'`.

Änder die Klassendefinition so ab, dass die beiden Methoden `add_c()` und `add_str()` den selben Namen (aber unterschiedliche Signaturen) bekommen.

