Übungspaket 6

Polymorphie und Standardwerte in cpp

Übungsziele:

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

Literatur:

C++-Skript[1], Kapitel 22 und 23

Semester:

Wintersemester 2021/22

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.

[1] www.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? Erläuter dies kurz anhand eines kleines Beispiels.

Die Signatur einer Funktion ist die Folge der Typen ihrer Parameter. So hat die Funktion `f( int i, double j)` die Signatur `int` × `double`.

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

Ist der Rückgabetyp einer Funktion Teil ihrer Signatur?	Nein.
Gehört das Schlüsselwort `const` zur Signatur einer Funktion?	Nein.
Können strukturierte Datentypen Teil der Signatur sein?	Ja.
Welche Signatur hat die Struktur `struct{ int i; char c; }`?	`(int × char)`

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

Ihre Signaturen müßen eindeutig voneinander unterscheidbar sein.

Aufgabe 2: Standardwerte

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

In Programmiersprachen wie `C++` ist es möglich, Parametern Standardwerte zuzuordnen. Diese Standardwerte werden übernommen, wenn beim Methoden bzw. Funktionsaufruf kein Wert für diesen Parameter angegeben wird.

Wo können Standardwerte definiert werden?

1. Bei der Klassendefinition
2. Bei der Methodenimplementierung

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

1. Sofern angegeben, müssen sie in der Parameterliste rechtsbündig definiert werden.
2. Für jeden Parameter darf nur einmal ein Standardwert definiert werden.

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`: `int × double` `T1`: `char × int *`

Vervollständige die folgende Tabelle.

Funktion	Signatur	Anmerkung
`f( char c1, char c2 )`	`char × char`	
`f( int i, int j, int k )`	`int × int × int`	
`f( double, int *, double )`	`double × int * × double`	
`f( int, T1 )`	`int × (int, double)`	`int und T1`
`f( T2 * p )`	`* (char × * int)`	`Zeiger auf T2`
`f( int, const int, const double )`	`int × int × double`	`ohne const`
`f( const int, const T1 *p )`	`int × * (int × double)`	`ohne const`

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: int: i=1

Aufruf: `c.m( 3.0 )` Ausgabe: double: d=3.0

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 Standarwerte 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	`c.m( 0, 0, 0, 0 )`	`i=  0 j=  0 k=  0 l=  0`
17	`c.m( 1, 2, 3, 4 )`	`i=  1 j=  2 k=  3 l=  4`
18	`c.m( 2, 2, 2 )`	`i=  2 j=  2 k=  2 l=4711`
19	`c.m( 3, 4 )`	`i=  3 j=  4 k=  1 l=4711`
20	`c.m( 4 )`	`i=  4 j= 13 k=  1 l=4711`
21	`c.m()`	`i=  0 j= 13 k=  1 l=4711`

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 );
12              void m_6( const double d );
13              int  m_6( double d );
14       };
15
16 int main( int argc, char ** aargv )
17     {
18        C c;
19        c.m_1( 3, 4 );
20        c.m_1( 3.0, 4.0 );
21        c.m_2( , 'z' );
22        c.m_3( 1000, 'z' );
23        return( 0 );
24     }
```

Im 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      }
```

Zeile	Fehler	Erläuterung	Korrektur
6	Standardwerte	Standardwerte müssen rechtsbündig angegeben werden. Wenn wir einen Standardwert für den ersten Parameter haben, müssen wir auch einen für den zweiten Parameter definieren.	Wert für `c`
7/8	Signatur	Beiden Signaturen scheinen sich Allerdings wäre ein Aufruf `m_6( 1.2 )` nicht eindeutig. Der Compiler weiss nicht, welche der beiden Varianten er auswählen soll.	Signatur ändern
9/10		Diese beiden Varianten sind korrekt, da `int` etwas anderes ist als `char`.	
11/12	`double` vs. `const double`	Beide Signaturen sind identisch, da `const` nicht Teil der Signatur ist.	nur eine Variante
13	Rückgabetyp	Auch diese Version ist identisch zu denen aus den Zeilen 11 und 12, da der Rückgabetyp nicht Teil der Signatur ist.	Namensänderung
19	Parameter	Hier kann der Compiler nicht entscheide, welche der beiden Varianten (Zeilen 3 und 4) er auswählen soll. Beide Varianten haben je einen korrekten Parametertyp.	expliziter Cast
20	Parameter	Hier haben wir das selbe Problem wie in Zeile 19.	expliziter Cast
21	Standardwerte	Bei vorhandenen Standardwerten dürfen die Parameter nur von rechts weggelassen werden. Da der zweite Parameter angegeben ist, muss auch der erste einen Wert erhalten.	Parameter ergänzen

Das korrigierte Hauptprogramm `main()` könnte wie folgt aussehen:

```
1 int main( int argc, char ** aargv )
2     {
3         C c;
4         c.m_1( 3, (double) 4 );
5         c.m_1( 3.0, (int) 4.0 );
6         c.m_2( 'a', 'z' );
7         c.m_3( 1000, 'z' );
8         return( 0 );
9     }
```

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.

```
 1 #include <stdio.h>
 2
 3 #define BUF_SIZE        128
 4
 5 class BUF {
 6           private:
 7              char buf[ BUF_SIZE ];     // the  actual   buffer
 8              int  len;                 // number of used bytes
 9           public:
10              char *get_buf();                  // returning buf
11              int   add( const char  c );       // add   one   c
12              int   add( const char *s );       // add a  string
13              int   reset  ();        // reset: setting len = 0
14         };
15
16 int BUF::reset()     { len = 0; return BUF_SIZE; }
17 char *BUF::get_buf() { return buf; }
18
19 int BUF::add( const char c )
20     {
21        if ( len < BUF_SIZE )                  // space left ?
22           buf[ len++ ] = c;                   // add the char
23        return BUF_SIZE - len;     // returning remaining size
24     }
25
26 int BUF::add( const char *s )
27     {
28        while( *s )                 // still some  characters?
29           add( *s++ );                      // add the next char
30        return BUF_SIZE - len;     // returning remaining size
31     }
32
33 int main( int argc, char **argv )
34     {
35        BUF buf; buf.reset();
36        buf.add( "Hi"   ); buf.add( ' ' );
37        buf.add( "you!" ); buf.add( '\0' );
38        printf( "buf='%s'\n", buf.get_buf() );
39     }
```