Clicker

Bitte den Link

https://vc2.sonia.de/b/har-zq1-o0p-dhs

für WS 2024/25 nutzen.

	Termin			Vorlesung	Übungen und Feedback
Vorles	Woche			Freitag ; Block 1+2	
6	7	06. Nov	Mi	tiefe und flache K	
7	7	08. Nov	Fr	Operatoren; Templates	
	8	15. Nov	Fr	fällt aus	tiefe, flache Kopie, minimale Std-Schnittstelle; dyn Array Abgabe bis Fr 15.11
8	9	22. Nov	Fr	DLR-Besuch mit Aufgabe 04	Unterstützung/Vorabnahme der Übungen zusätzlicher Termin in diesemZeitraum; für
9	10	29. Nov	Fr	STL, Iteratoren; Algorithmus versus	
10	11	06. Dez	Fr	lineare und assoziative Container;	Unterstützung/Vorabnahme der Übungen; z.B 11.12 für Abgabe am 18.12 (Zusatztermin in diesem Zeitraum)
11	12	13. Dez	Fr	Verschiebeoperatoren Klasse unique_ptr,, shared_ptr, Lambda-Ausdrücke (Polymorphie)	finale Abgabe der Übungen; Di 18.12
12	13	20. Dez	Fr	Vorbereitung Klausur	

Freitag, 15.11 ist Abgabe der 2. und 3. Aufgabe, Unterstützung per BBB gewünscht?

Terminänderung
Klausur: Do, 16.1.25, 14-15:30 Uhr
Klausureinsicht: Fr. 24.1.2025 10:00- Uhr

Unterstützung per BBB? (Maximal 20 Minuten)

- Do 7.11 15-18 Uhr
- Und dann wieder ab 18.11

Rückblick

Wiederholung, d.h. Aufgaben aus dem einem Test für vorherige Semester
Referenz oder Zeiger oder Werte oder doch was anderes?

Klassen beginnen

- Am Beispiel von Vektor, Konstruktor, Destruktor
- Konstruktoren

Klasse string
Klasse vector<T>, Klasse array<T,size>

Klasse Vektor als Demo am Rechner von HHe.
Tiefe Kopie gegenüber flache Kopie: MacheNix, NeFaitRien

- +Z, -Z: Wann werden Objekte erzeugte?

Aufgabe_02_03; Zweite bewertete Abgabe

6. Vorlesung; Mi. 06.11.2024

Vorlesung

Wiederholung / Ankündigung (20.10.2023) tiefe und flache Kopie (20.10.2023)
Weitere Informationen zu Konstrukturen (Selbststudium als Clickeraufgaben) (20.10.2023)

Übungsaufgaben WS 2024/25; Sprechfunk-Annotation, So 15.11.2024, 23:59 Uhr über SVN zu

Exercise: Tiefe Kopie der Levenshtein-Klasse etc. (24.10.2024)

Übungsaufgaben

Clicker

Geben Sie den Code syntaktisch richtig an, um über die Variable pp1 die Variable i auf dem Wert 140 zu setzen.

```
typedef int*   IntPointer;
int i = 32;
IntPointer p1 = &i;
int* p2 = new int[2];
int** pp2 = new IntPointer[2];
int** pp1 = &p1;
(p2)[1] = 138;
pp2[1] = &(p2[1]); /*1*/
```

Welche der 4 folgenden Möglichkeiten folgt nach /* 1 */?

1. *pp1 = 140;
2. **pp1 = 140;
3. &pp1 = 140
4. pp1[0] = 140;

Clicker

Geben Sie den Code syntaktisch richtig an, um über die Variable pp1 die Variable i auf dem Wert 140 zu setzen.

```
typedef int*   IntPointer;
int i = 32;
IntPointer p1 = &i;
int* p2 = new int[2];
int** pp2 = new IntPointer[2];
int** pp1 = &p1;
(p2)[1] = 138;
pp2[1] = &(p2[1]); /*1*/
```

1. *pp1 = 140;
2. **pp1 = 140;
3. &pp1 = 140
4. pp1[0] = 140;

1. *pp1 = 140;
2. **pp1 = 140;
3. &pp1 = 140
4. pp1[0] = 140;

Aufgabe

```
void setVar(int* erg, int w) {
    int quad = w * w;
    ??? = quad; /*1*/
}
void funk3(void) {
    int i = 3;
    int res = 14;
    setVar(??? , i); /*2*/ /*res uebergeben */
    datei << "res= " << res;
}
```

Ausgabe soll 9 sein.
Was wird also jeweils für „???“ eingesetzt?

1. erg = quad; // setVar(&res, i);
2. *erg = quad; // setVar(&res, i);
3. *erg = quad; // setVar(res, i);
4. &erg = quad; // setVar(*res, i);
5. erg = quad; // setVar(res, i);

Aufgabe

```
void setVar(int* erg, int w) {
    int quad = w * w;
    ??? = quad; /*1*/
}
void funk3(void) {
    int i = 3;
    int res = 14;
    setVar(??? , i); /*2*/ /*res uebergeben*/
    datei << "res= " << res;
}
```

Ausgabe soll 9 sein.

```
void setVar(int* erg, int w) {
    int quad = w * w;
    *erg = quad; /*1*/
}
void funk3(void) {
    int i = 3;
    int res = 14;
    setVar(&res, i); /*2*/
    datei << "res= "<< res;
}
```

Überladen von Funktionen

```
int Add(int& a, int& b) {
    cout << "Add1 ";
    return a + b;
}
int Add(int a, int b, int c) {
    cout << "Add2 ";
    return a + b + c;
}
double Add(int& a, double b= 1.0) {
    cout << "Add3 ";
    return a + b;
}
```

```
int main() {
    int i = 10; double d = 4.4;
    cout << Add(i, d) << endl;
    cout << Add(i, i) << endl;
    cout << Add(i) << endl;
    cout << Add(i, i, i) << endl;
return 0;
}
```

```
1.    Add3 14.4, Add1 20,Add3 11, Add2 30
2.    Add1 14,   Add3 20,Add2 11, Add2 30
3.    Add1 14,   Add1 20,Add1 11, Add1 30
4.    Syntaxfehler
```

Überladen von Funktionen

```
int Add(int& a, int& b) {
    cout << "Add1 ";
    return a + b;
}
int Add(int a, int b, int c) {
    cout << "Add2 ";
    return a + b + c;
}
double Add(int& a, double b= 1.0) {
    cout << "Add3 ";
    return a + b;
}
```

```
int main() {
    int i = 10; double d = 4.4;
    cout << Add(i, d) << endl;
    cout << Add(i, i) << endl;
    cout << Add(i) << endl;
    cout << Add(i, i, i) << endl;
return 0;
}
```

```
1.    Add3 14.4, Add1 20,Add3 11, Add2 30
2.    Add1 14,   Add3 20,Add2 11, Add2 30
3.    Add1 14,   Add1 20,Add1 11, Add1 30
4.    Syntaxfehler
```

Überladen von Funktionen

```
int Add(int& a, int& b) {
    cout << "Add1 ";
    return a + b;
}
int Add(int a, int b, int c) {
    cout << "Add2 ";
    return a + b + c;
}
double Add(int& a, double b= 1.0) {
    cout << "Add3 ";
    return a + b;
}
```

```
int main() {
    int i = 10; double d = 4.4;
    cout << Add(i, d) << endl;
    cout << Add(i, i) << endl;
    cout << Add(i) << endl;
    cout << Add(i, i, i) << endl;
return 0;
}
```

```
1.    Add3 14.4, Add1 20,Add3 11, Add2 30
2.    Add1 14,   Add3 20,Add2 11, Add2 30
3.    Add1 14,   Add1 20,Add1 11, Add1 30
4.    Syntaxfehler
```

Überladen von Funktionen

```
int Add(int& a, int& b) {
    cout << "Add1 ";
    return a + b;
}
int Add(int a, int b, int c) {
    cout << "Add2 ";
    return a + b + c;
}
double Add(int& a, double b= 1.0) {
    cout << "Add3 ";
    return a + b;
}
```

```
int main() {
    int i = 10; double d = 4.4;
    cout << Add(i, d) << endl;
    cout << Add(i, i) << endl;
    cout << Add(i) << endl;
    cout << Add(i, i, i) << endl;
return 0;
}
```

```
1.    Add3 14.4, Add1 20,Add3 11, Add2 30
2.    Add1 14,   Add3 20,Add2 11, Add2 30
3.    Add1 14,   Add1 20,Add1 11, Add1 30
4.    Syntaxfehler
```

Überladen von Funktionen

```
int Add(int& a, int& b) {
    cout << "Add1 ";
    return a + b;
}
int Add(int a, int b, int c) {
    cout << "Add2 ";
    return a + b + c;
}
double Add(int& a, double b= 1.0) {
    cout << "Add3 ";
    return a + b;
}
```

```
int main() {
    int i = 10; double d = 4.4;
    cout << Add(d, d) << endl;
    return 0;
}
```

```
1.    Add1 8.8
2.    Add2 13.2
3.    Add3 8.8
4.    Syntaxfehler
```

Überladen von Funktionen

```
int Add(int& a, int& b) {
    cout << "Add1 ";
    return a + b;
}
int Add(int a, int b, int c) {
    cout << "Add2 ";
    return a + b + c;
}
double Add(int& a, double b= 1.0) {
    cout << "Add3 ";
    return a + b;
}
```

```
int main() {
    int i = 10; double d = 4.4;
    cout << Add(d, d) << endl;
    return 0;
}
```

```
1.    Add1 8.8
2.    Add2 13.2
3.    Add3 8.8
4.    Syntaxfehler
```

```
: "Add": Durch keine der 3 Überladungen konnten alle Argumenttypen konvertiert werden.
nn "double Add(int &,double)" sein
r "int Add(int &,int &)"
ei Anpassung der Argumentliste "(double, double)"
```

Überladen von Funktionen

```
double Mult(int& a, int& b) {
   cout << "Mult1 ";
   return a * b;
}
int main() {
   cout << Mult(1, 2) << endl;
   return 0;
}
```

```
1.    Mult1 2
2.    Mult1 3
3.    Mult1 4
4.    Syntaxfehler
```

Überladen von Funktionen

```
double Mult(int& a, int& b) {
   cout << "Mult1 ";
   return a * b;
}
int main() {
   cout << Mult(1, 2) << endl;
   return 0;
}
```

```
1.    Mult1 2
2.    Mult1 3
3.    Mult1 4
4.    Syntaxfehler
```

int Mult(int &,int &)" :
Konvertierung von Argument 1 von
"int" in "int &" nicht möglich

Überladen von Funktionen

```
double Mult(int& a, int& b) {
   cout << "Mult1 ";
   return a * b;
}
double Mult(double a, double b) {
   cout << "Mult2 ";
   return a * b;
}
int main () {
   cout << Mult(1, 2) << endl;
   return 0;
}
```

```
1.    Mult1 2
2.    Mult2 2
3.    Mult1 2.1
4.    Syntaxfehler
```

Überladen von Funktionen

```
double Mult(int& a, int& b) {
   cout << "Mult1 ";
   return a * b;
}
double Mult(double a, double b) {
   cout << "Mult2 ";
   return a * b;
}
int main () {
   cout << Mult(1, 2) << endl;
   return 0;
}
```

```
1.    Mult1 2
2.    Mult2 2
3.    Mult1 2.1
4.    Syntaxfehler
```

Überladen von Funktionen

```
int Diff(int a, int b) {
    cout << "Diff1 ";
    return a - b;
}
int Diff(int a, int b, int c) {
    cout << "Diff2 ";
    return a - b - c;
}
double Diff(int a, double b = 1.0) {
    cout << "Diff3 ";
    return a - b;
}
```

```
int main () {
    int i=10; double d=4.4, e=5.5;
    cout << Diff(i, i) << endl;
    cout << Diff(e, d) << endl;
    return 0;
}
```

```
1.    Diff1 0 Diff3 0.6
2.    Syntaxfehler
3.    Diff1 0 Diff3 1.1
4.    Diff3 0 Diff3 1.1
5.    Diff1 0 Diff3 1
```

Überladen von Funktionen

```
int Diff(int a, int b) {
    cout << "Diff1 ";
    return a - b;
}
int Diff(int a, int b, int c) {
    cout << "Diff2 ";
    return a - b - c;
}
double Diff(int a, double b = 1.0) {
    cout << "Diff3 ";
    return a - b;
}
```

```
int main () {
    int i=10; double d=4.4, e=5.5;
    cout << Diff(i, i) << endl;
    cout << Diff(e, d) << endl;
    return 0;
}
```

```
1.    Diff1 0 Diff3 0.6
2.    Syntaxfehler
3.    Diff1 0 Diff3 1.1
4.    Diff3 0 Diff3 1.1
5.    Diff1 0 Diff3 1
```

Es gibt Warnung, dass bei Konvertierung von e nach double, wir einen möglichen Datenverlust haben, a ist dann 5 und nicht 5.5.

Wie sieht der Aufruf aus?

```
void fzeiger(Vorgang*  fp) { … }
void fref(Vorgang&  fr) { … }
void fwert(Vorgang   fw) { … }
```

```
Vorgang v;
```

```
fref(& v);  // 1
fref(   v);  // 2
fref(*  v);  // 3
   //4 keine Ahnung
```

Syntax / Semantik

Wert und Referenz (&) sind das gleiche (wenn es rein um die Syntax geht). Semantisch sind sie etwas völlig anderes

Ausgangstyp	Zieltyp	Operator/Zeichen
Wert/Referenz	Zeiger	&
Zeiger	Wert / Referenz	*
Sonst		Kein Operator, Passt schon

Zeiger und Referenz führen beide zum gleichen Assemblercode, aber mit unterschiedlicher Syntax im C++-Code. Deshalb sind beide im Sinne von Ausgangsparameter verwendbar. Das Original-Objekt wird manipuliert.

Ein Vorschlag für Namenskonventionen

1. Präfix:

Globale Variablen beginnen mit einem "g":

```
int g_zahl;
```

Funktionsargumente beginnen mit "a"

```
funk(double ad_durch,  const int ai_wert)
```

Member (Elemente) in Klassen mit "m"

```
class Mitarbeiter {...
double m_umsatz;
}
```

Lokale Variablen erhalten keinen ersten Präfix.

Ein Vorschlag für Namenskonventionen (2)

Der zweite Präfix gibt an, ob es sich um eine Referenz (r) oder um einen Zeiger (p) handelt.

```
int * p_anzahl;
funk( double & ard_durch, int * apn_zahlen)
```

Der dritte Präfix legt den Typ fest:

- i für int, s für short
- n für einen ganzzahligen Typ
- d für double, f für float
- c für eine Klasseninstanz
- t für einen Typ, der durch typedef definiert wurde
- ...

```
funk (Stack & arc_stack,  float *apf_summe)
int i_anzahl;
```

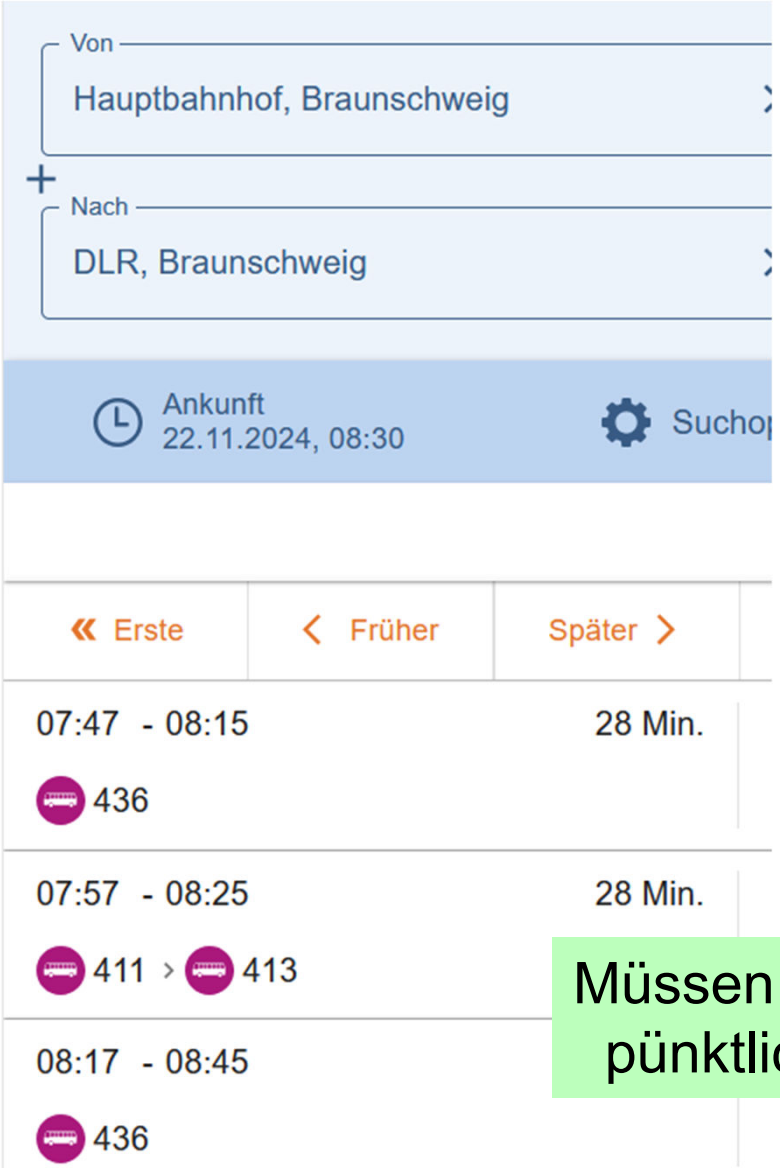
Besuch im DLR von Interesse?

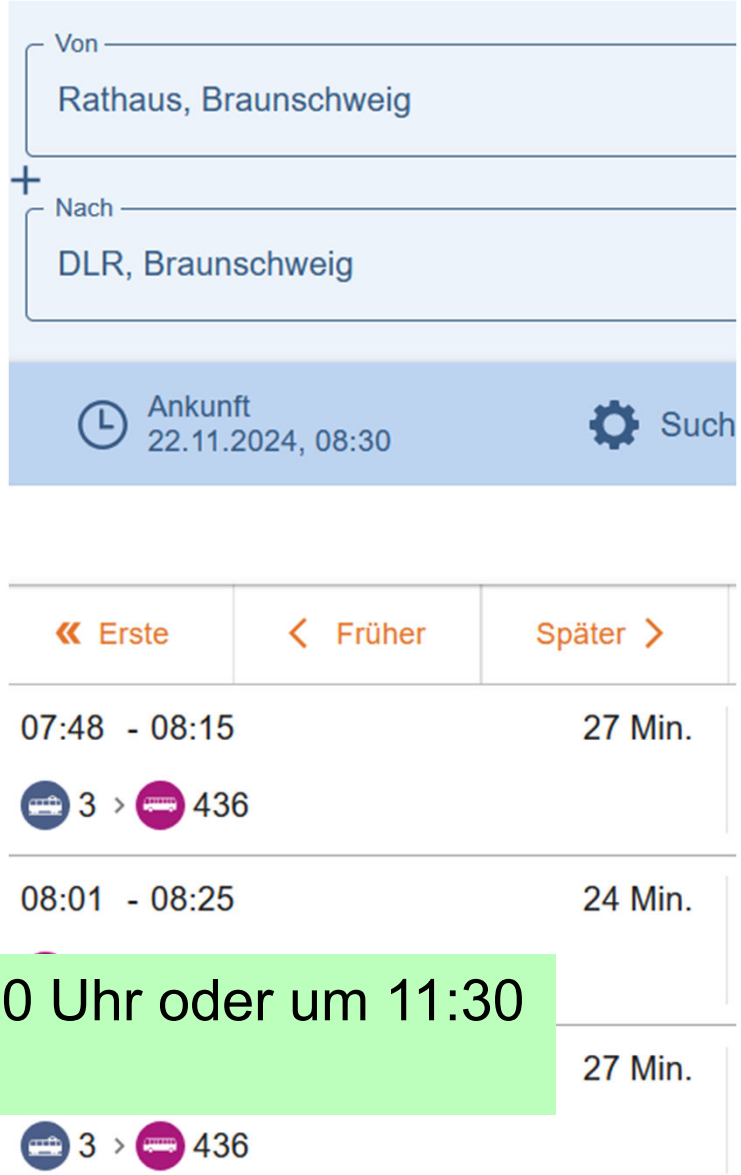
Termin Freitag, 22.11.2024 im Rahmen der Vorlesung.
Eintreffen zwischen 08:30 und 09:00
Ende 11 Uhr
Genauer Termin wird noch festgelegt, ggf. abhängig von ÖPNV

Interesse?
1. Werde teilnehmen
2. Nein

Bitte kurze E-Mail an mich, dass Sie teilnehmen wollen.
Personalausweis auf jeden Fall mitbringen.

Beginn 8:30 oder 9:00 Uhr?

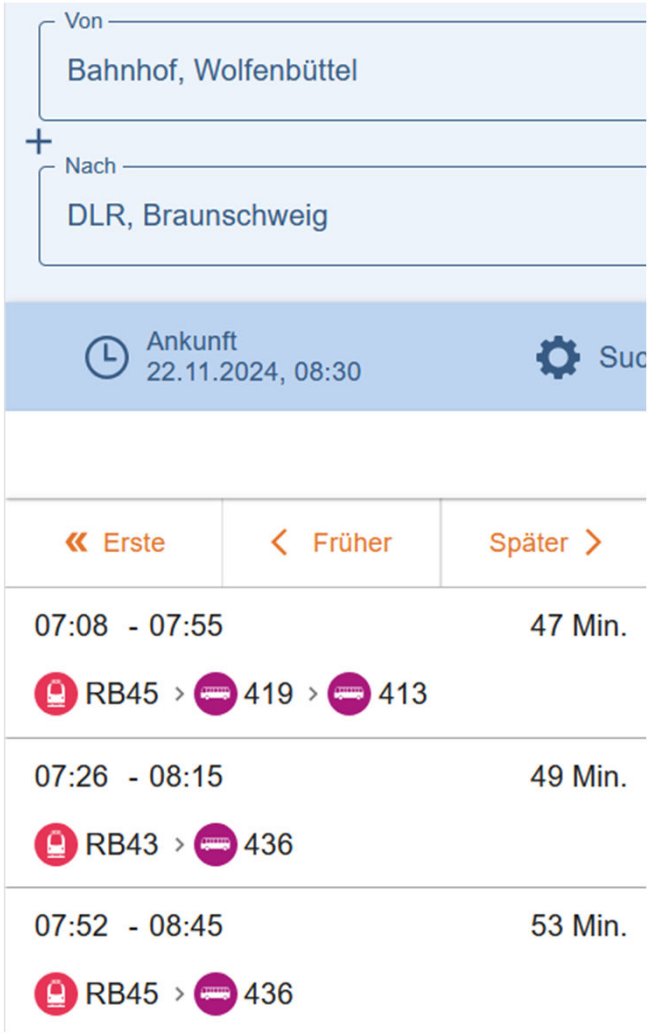


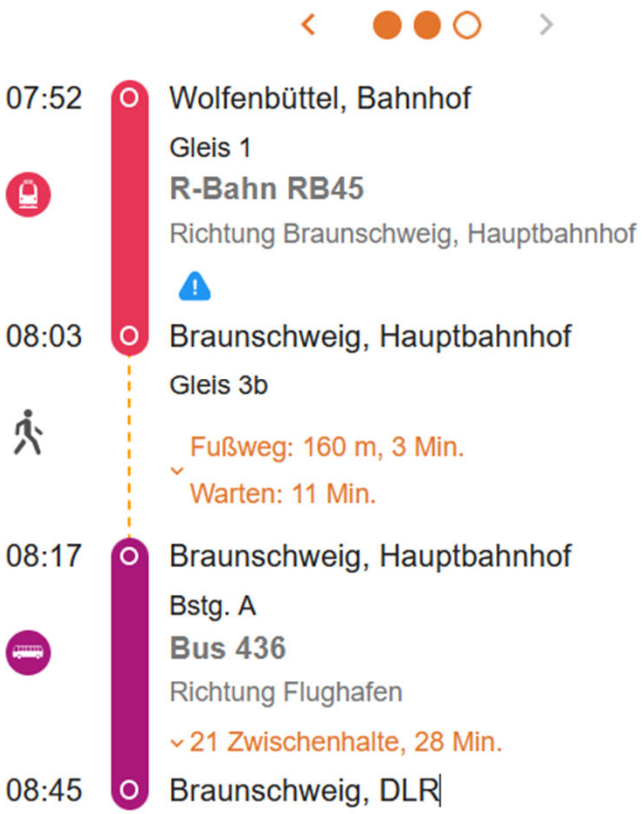


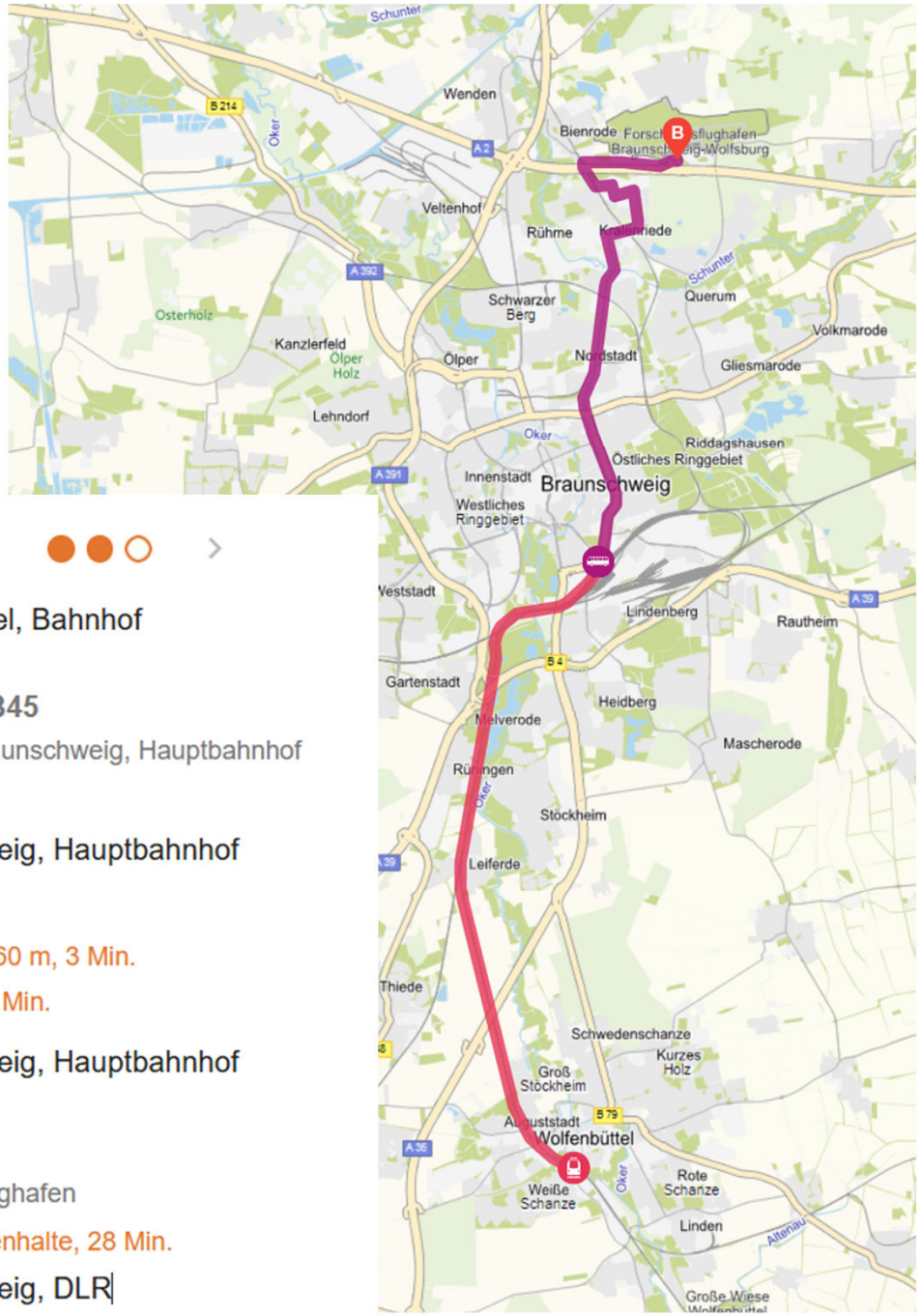
Müssen Sie, um 11:00 Uhr oder um 11:30 pünktlich weg?

Die verschiedenen Programmierparadigmen von C++

Die verschiedenen Programmierparadigmen von C++







Vorlesungsplanung

- Wiederholung mit Referenz oder Zeiger oder Werte oder doch was anderes?
- FunktionLog, LogTrace
- MacheNix als Übung für Sie (Kopierkonstruktor und Zuweisungsoperator, selber „Hand anlegen“)

6. Vorlesung; Mi. 06.11.2024

Vorlesung

_Wiederholung / Ankündigung (20.10.2023) __tiefe und flache Kopie (20.10.2023)
Weitere Informationen zu Konstrukturen (Selbststudium als Clickeraufgaben) (20.10.2023)

Übungsaufgaben WS 2024/25; Sprechfunk-Annotation, So 15.11.2024, 23:59 Uhr über SVN zusamm

Exercise: Tiefe Kopie der Levenshtein-Klasse etc. (24.10.2024)

Übungsaufgaben

Infos zu cmake (05.10.2024)
DynVorgangsArray mit Probl~~emen VS 2022 bzw.~~ CMake (26.10.2024)
Vektor mit Kopier-Problemen VS 2022 bzw. CMake (26.10.2024)

Bitte Code schon mal herunterladen

Protokollierung von Funktionsaufrufen

Die LogTrace-Funktionalität

Weiteres Beispiel Motivation

```
void Func() {
   FunktionLog  var("Func");
   . . .
   Func2();
   . . .
}

void Func2() {
   FunktionLog  var("Func2");
   . . .
}
```

Es wird eine Funktionalität gewünscht, die am Anfang einer Funktion schreibt Funktion XXX betreten und beim Verlassen Funktion XXX verlassen.

Ausgabe sollte hier sein:

```
|>Func
    |> Func2
    <| Func2
<|Func
```

Weiteres Beispiel FunktionLog

```
class FunktionLog  {
      enum { MAX_LEN=50, BLANCS=3 };

      string FuncName;
      ...    // weiteres wie mi_aufrufEbene ...

public:
      // Gibt den Namen FuncName aus
      FunktionLog(string Name);

      // Gibt beim Verlassen der Funktion FuncName aus.
      ~FunktionLog();
};
```

Klasse FunktionLog, Teil 2

```
FunktionLog::FunktionLog(string Name) {
   FuncName=Name;

   for (int i=0; i < mi_aufrufEbene; ++i) {
      cout << setw(BLANCS) << " ";
   }
    ++mi_aufrufEbene;
   cout << "|> “  << FuncName;
   cout << endl;
}
```

```
FunktionLog::~FunktionLog() {
   --mi_aufrufEbene;
   for (int i=0; i < mi_aufrufEbene; ++i) {
      cout << setw(BLANCS) << " ";
   }
   cout << "<| “ << FuncName << endl;
}
```

FunktionLog, Anwendungsbeispiel

```
int main() {
  FunktionLog xxx("main");
  {
    FunktionLog xxx("Schleife 1");
    for (int i=0; i < 3; ++i) {
      FunktionLog xxx("i-Schleife ");
      for (int j=0; j < 2; ++j) {
        FunktionLog xxx("j-Schleife");
      }
    }
  }
}
```

Ausgabe:

```
|> main
  |> Schleife 1
    |> i-Schleife
      |> j-Schleife
      <| j-Schleife
      |> j-Schleife
      <| j-Schleife
    <| i-Schleife
    |> i-Schleife
      |> j-Schleife
      <| j-Schleife
....
<| i-Schleife
  <| Schleife 1
<| main
```