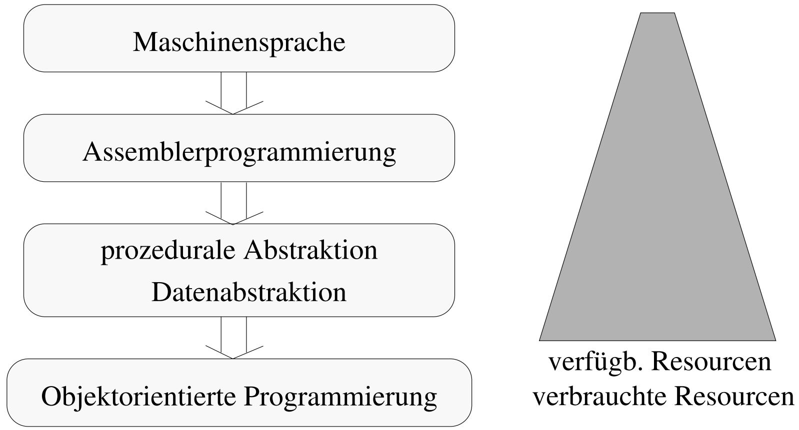
Gliederung des Seminars

- Überblick
- Unterschiede zwischen C und C++
- Begriffe der Objektorientierten Programmierung
  Klassen in C++
- Vererbung von Klassen
- Operator-Overloading
  Streams
- Templates
  objektorientierter Software-Entwurf mit C++

Entwicklung der Programmiermethodik



Historie von C++

- ❏ 1972: D. Ritchie entwirft C
- ❏ 1980: D. Ritchie entwirft „C with classes“
- ❏ 1983: Bjarne Stroustrup präsentiert C++ V1.0
- ❏ 83-88: C wird standardisiert
- ❏ 1989: Standardisierungskomitee für C++ (ANSI) V2.0
- ❏ 1991: C++ V3.0 mit templates
- ❏ 1993: C++ V3.1 mit namespaces, type identification

Unterschiede zwischen C und C++

- ❏ Kommentare
- ❏ Ein- und Ausgabe
- ❏ Inlining
- ❏ Speicherallokierung
- ❏ Funktionsüberladung
- ❏ Referenzen
- ❏ Defaultwerte
- ❏ Konstanten
- ❏ Verschiedenes

Kommentare

❏ normaler Kommentar wie bereits in C: /* ... */

❏ neuer Kommentar: //

- Bedeutung: alles bis zum *Zeilenende* wird ignoriert,
  kann irgendwo in der Zeile vorkommen

❏ Beispiel:

```
//   Das ist ein Beispielkommentar
int test_variable;          // Testvariable zu Demozwecken
```


Ein- und Ausgabe

❏ neue Konstrukte: *cin*, *cout*, *cerr*, *clog*, <<, >>

- globale Variablen *cin, cout*, *cerr*, *clog (iostream.h):*

  cin: Eingabestream
  cout, *cerr, (clog)*: Ausgabestream

- Operatoren << und >>

  <<: Ausgabeoperator
  >>: Eingabeoperator

```
int test;
cin >> test;
cout << 1234 << "\n";
```


Inlining

❏ Funktionsdefinition mit Schlüsselwort *inline* vorangestellt

```
inline int max (int a, int b)
{ return (a>b)?a:b; }
```

- Es wird keine eigentliche Funktion angelegt
- Body wird beim Aufruf in Quellcode eingebunden (vgl. *#define*)
- Funktion muß in „.h“-Datei implementiert werden

❏ Unterschied zwischen *#define* und *inline*:

- Typprüfung und Syntaxprüfung bei *#define* inaktiv
- wiederholte Auswertung von Ausdrücken unterbleibt bei *inline*

Zugriffsoperator ::

❏ im normalen (nicht Klassen) Kontext:

- Referenzierung auf globale Variablen

❏ Beispiel:

```
int Test;

void main ()
{
    int Test;

    Test = 5;          // Setzen der lokalen Variable
    ::Test = 10;       // Setzen der globalen Variable
}
```

Speicherallokierung

❑ neue Befehle zur Speicherallokation/-freigabe:

- new : Speicheranforderung
- delete : Speicherfreigabe

❑ Verwendung in Programmen:

```
void main ()
{
    int *ip = new int;
    int *ap = new int[7];
     ...
    delete ip;
    delete[] ap;                    // nicht delete ap !!
}
```


Speicherallokierung

❑ Freigabe in der Verantwortung des Programmierers

- keine Garbage-Collection
- Zeiger auch nach *delete* noch zugreifbar
  - => häufige Ursache für Programmfehler

❑ delete auf NULL ist erlaubt

❑ Beispiel:

```
int *x = 0;            // ok
delete x;              // ok
x = new int;           // ok
delete x;              // ok
delete x;              // falsch
```


Funktionsüberladung

❑ gleicher Funktionsname für mehrere Implementierungen

❑ Unterscheidungskriterien:

- Anzahl der Parameter
- Typen der Parameter
- Nicht: Returntyp der Funktion

❑ Beispiel:

```
void    Print ();              // ok
void    Print (int, char*);    // ok
int     Print (float);         // ok
int     Print ();              // nicht unterscheidbar
```


Referenzvariablen

❑ Adreßoperator & bei Variablendeklarationen

❑ Referenzvariablen

- keine eigenständigen Variablen
- Stellvertreter für andere Variablen
- müssen mit L-Value initialisiert werden

❑ Beispiel:

```
int x = 5;
int &rx = x;        // Initialisierung muß Variable sein
x = 6;              // x==6 und rx==6
rx++;               // x==7 und rx==7
```

Referenzvariablen

❑ Referenzparameter

- call-by-reference Semantik
- Keine expliziten Pointer mehr nötig
- Aufrufer macht ganz normalen Aufruf
- Nachteil: beim Aufruf nicht erkennbar

❑ Beispiel:

```
void increment (int& x) {
    x ++;
}
...
int n = 5;
increment(n);
```


Defaultparameter

❑ Parameter am Ende können ***default***-Werte bekommen

❑ Beispiel:

```
void Print (char* string, int nl = 1);

Print (“Test“, 0);
Print (“Test“);                    // Print (“Test“, 1);
Print ();                          // Fehler, da mind. char*
```

❑ nach einem default-Parameter müssen alle Parameter default sein

❑ Achtung: Doppeldeutigkeiten

```
void Print (char* string);
void Print (char* string, int nl = 1);
Print (“Test“);                    // ????????????
```


Konstanten

❑ Schlüsselwort **const** als Variablen-Modifizierer

❑ kennzeichnet ‘nur lesbare’ Variablen

❑ const-Variablen können nur bei Deklaration gesetzt werden

❑ Beispiel:

```
const int k = 42;
char* const str1 = “Teststring1“;
const char* str2 = “Teststring2“;
const char* const str3 = “Teststring3“;

k = 4;                             // Fehler: k ist read-only
str1 = “New Teststring“;           // Fehler: str1 ist read-only
*str1 = ’P’;                       // ok
str2 = “New Teststring“;           // ok
```


Weitere Neuerungen

❑ Ein Namensraum für type, struct, union, enum

- nach Deklaration können struct, union und enum weggel. werden

❑ anonyme Union

❑ beim Aufruf einer Funktion muß Signatur bekannt sein

❑ **int f ();** ist Definition einer Funktion f mit keinem Parameter

❑ alternative Typumwandlung:

- **(typ)var** ⇔ **typ(var)**

❑ Deklarationen immer möglich, nicht nur am Blockanfang

❑ strenge Typisierung