Einführung in C/C++

Vortragender:
Tim Schürmann

Vortrag Projektgruppe 369
WS 00/01
Universität Dortmund

Inhalt

1. **Einleitung**
2. C
3. C++
4. Compiler und make

1. Einleitung

- C++ ist eine Erweiterung von C um objektorientierte Konzepte
  - Vorteil: „alte“ Software und Bibliotheken nutzbar
  - Nachteil: Man kann Objektorientierung umgehen
- C und C++ sind Standards

  Bücher:
  - „Programmieren in C“ Kernighan, Richie; Hanser
  - „Die C++ Programmiersprache“, Stroustroup; Addison-Wesley

Inhalt

1. Einleitung ✓
2. **C**
3. C++
4. Compiler und make

Einführung (1)

- Erfunden von Dennis Richie für UNIX
- Von den Sprachen BCPL und B inspiriert
- Standard nennt sich „ANSI-C“, urspüngliches C als „K&R C“ bezeichnet

Einführung (2)

- maschienennahe Sprache und doch „abstrahierend“, wurde als „Systemprogrammiersprache“ bezeichnet
- bedeutet auch: es gibt keine
  - eingebauten Eingabe / Ausgabefunktionen
  - keine garbage collection (Speicher muss selbst verwaltet werden!)
  - ursprünglich auch keine Bibliotheken (in ANSI-C vorhanden)
- Vorteil: C ist verdammt schnell und klein!

„Hello World“

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf(”Hello World\n”);
}
```

Kommentare

- Syntax:
  `/* dies ist ein Kommentar */`
- Kommentare dürfen nicht verschachtelt sein:
  `/* dies ist /* ein */ Kommentar */`

Variablen (1)

- Variablenname
  - besteht aus Buchstaben und Ziffern
  - Unterstrich „_“ zählt als Buchstabe
  - Wort muss mit Buchstabe beginnen
  - Groß- und Kleinbuchstaben werden unterschieden
  - mind. die ersten 31 Zeichen sind signifikant

Variablen (2)

- Elementare Datentypen:
  - `char` ein Byte, nimmt ***<u>ein Zeichen</u>***, repräsentiert durch z.B. ASCII-Wert auf; der Inhalt dieser Variable ist eine Zahl!
  - `int` ***<u>ganzzahliger Wert</u>*** in der „natürlichen Größe“ der Maschine
  - `float` ***<u>einfach genauer Gleitkommawert</u>***
  - `double` ***<u>doppelt genauer Gleitkommawert</u>***
- Die max. Größe der Werte ist Maschinenabhängig!
- Deklaration: **`int`** `zaehler;`

  `int` `z1, z2, z3;`

Variablen (3)

- Varianten:
  - Integer: `short` und `long`, Bsp:

```
short int sh;      /* abgekürzt: short sh;      */
long int counter; /* abgekürzt: long counter; */
```

  - Char und Integer: `unsigned` und `signed`, Bsp:

```
unsigned char buchstabe2;
```

  - Double: `long`, Bsp:

```
long double counter3;
```

Variablen (4)

- Erklärung:
  - `int` soll die „natürliche Größe“ einer Maschine sein, meistens belegt `short int` 16 Bits, `long int` 32 und `int` hat entweder 16 oder 32 Bits
  - `unsigned`-Werte sind immer positiv oder Null; wird z.B. `char` durch 8 Bit repräsentiert, können Variablen vom Typ `unsigned char` Werte von 0 bis 255 annehmen, bei `signed` von -127 bis 127; druckbare Zeichen sind immer positiv
  - in `<limits.h>` findet man, wie Werte auf der entsprechenden Maschine definiert sind

Variablen (5)

- Bsp: Minimal zugelassene Grenzwerte:

unsigned char	−127 bis 127
signed char	0 bis 255
short int	−32767 bis 32767
unsigned short int	0 bis 65535
int	−32767 bis 32767
unsigned int	0 bis 65535
long int	−2147483647 bis 2147483647
unsigned long int	0 bis 4294967295

Variablen (6)

- Zeichenzuweisung:

```
char tom;
tom= ‘d‘;  /* bei tom=‘0‘; hat tom den Wert 48 */
```

- Sonderzeichen (Ersatz für *ein* Zeichen):

```
tom= ‘\n‘;
```

- erlaubte Sonderzeichen:

```
\a Klingelzeichen       \\ Gegenschrägstrich
\b backspace            \? Fragezeichen
\f Seitenvorschub       \‘ Hochkomma
\n Zeilenvorschub       \" Anführungszeichen
\r Wagenrücklauf        \ooo Oktale Zahl
\t Tabulatorzeichen     \xhh hexadezimale Zahl
\v Vertikal-Tabulator   \Ziffer ASCII-Zeichen
```

Variablen (7)

- Konstante:

```
const double e=2.7128;
```

- In Funktionen kann damit sichergestellt werden, dass die Werte nicht verändert werden

```
int foo(const int myvalue);
```

Aufzählungen

- konstante Werte werden mit Namen verknüpft

```
enum boolean {true=0, false=1};
/* Kurzschreibweise (C nummeriert selbst): */
enum boolean {true, false};

boolean baldfruehling=false;
baldfruehling=0; /* entspricht =true */

enum month {jan=1, feb, mar, apr, may, jun, jul,
         aug, sep, oct, nov, dec};
         /* feb=2, mar=3, etc */
```

Arrays

```
char line[1000];        /* Zählung:0,1,...,999 */
line[3]='x';            /* Zugriff auf Elem. 4 */
int matrix[5][4];
matrix[2][4]=99;        /* Zugriff auf Eintrag 3,5 */
```

Größe des Arrays muss bei der Deklaration feststehen

Im Gegensatz zu C++ ist in ***C*** ein String immer ein **Array von chars**!

```
char msg[]="warning: ";
line="a1";
```

/* inhalt:

a	1	\0								

...*/

```
line="hello" "world";
```

/*inhalt:

h	e	l	l	o	w	o	r	l	d	\0			

...*/

Operatoren (1)

Vergleiche (Ergebnis in ***C***: 1 (Vergleich trifft zu) oder 0,
in ***C++***: true oder false):

`<  <=   =>  >`

logische Verknüpfungen (Ergebnis ist 1 oder 0 bzw. false oder true):

`==`	`!=`	(gleich, ungleich)
`||`		(logisches oder)
`&&`		(logisches und)

`||` und `&&` werden immer von links nach rechts ausgewertet, Bedeutung in if oder Schleifen:

`if( (i>1) || (i<7) )` ...

Operatoren (2)

Inkrement und Dekrement:

```
/* Anstelle von n=n+1 kann man schreiben: */
n++;/* n um genau 1 erhöhen */
n--;/* n um 1 subtrahieren */
```

Unterscheidung in Post- und Präfix, `n` sei 5:

```
x= n++;   /* x ist jetzt 5, n aber 6 */
x= ++n;   /* x ist 6, n ebenfalls */
```

Bei `n++;` wird die Variable *erst* verwendet und *dann* inkrementiert!

UNIVERSITÄT DORTMUND

Operatoren (3)

Bitoperatoren (selten benötigt):

```
& UND-Verknüpfung von Bits
| ODER-
^ XOR
<< Bit-Verschiebung nach links
>> Bit-Verschiebung nach rechts
~ Bit-Komplement
```

Achtung: `<<` und `>>` nicht mit den Operatoren aus C++ verwechseln!

Bsp: `n = n & 0177;` löscht in n alle Bits, außer den letzten sieben, dabei `n` bel. Variable

Typumwandlung und Abkürzungen

Typumwandlung erfolgt zum Teil automatisch:

```
float p;
int i=50;
p=i;  /* Umwandlung int in float */
```

oder explizit per Cast:

```
p = (float)i
```

sagt explizit: `i` ist jetzt eine Fließkommazahl

Abkürzungen für Ausdrücke:

```
i = i + 2;
i += 2;
```

Analog: `-=` , `*=`, etc.

If (1)

Variante 1:

```
if ( bedingung )
{
   befehle
}
else
{
  befehle
}
```

Variante 2:

```
if ( bedingung )
{
   befehle
}else if (bedingung)
{
  befehle
}else if (bedingung)
{...
```

In C: If wird ausgeführt, wenn `bedingung == 1` ist
In C++: If wird ausgeführt, wenn `bedingung != 0` ist

If (2)

Variante 3:

```
if ( bedingung ) befehl1;
else befehl2;
```

Variante 4 :

für

```
if(a>b) z=a;
else z=b;
```

lässt sich kurz schreiben:

```
z = (a>b) ? a : b;
```

while

```
int b=0;
int i=0;

while ( i<1 )
{
  b=i+5;
  i++;
}
```

```
int b=0;
int i=0;

do
{
  b=i+5;
  i++;
} while( i<1 );
```

In C++: while wird solange ausgeführt, bis `bedingung == false` ist

for

Initialisierung

Bedingung

Ausführung nach jedem Durchlauf

```
int b=0;

for ( int i=0;  i<100 ;  i++ )
{
  b=i+5;
  printf("wieder ein durchlauf");
}

for (;;)          /* ewige Schleife */
{
}
```

Ein Beispielprogramm

```
/* Umwandlung Fahrenheit in Celsius in 20er Schritten */
#include <stdio.h>
main()
{
    int fahr, ceslsius;
    int lower, upper;
    int step=20;
    lower = 9; upper=300;

    fahr=lower;
    while(fahr <= upper){
            celsius = 5 * (fahr-32) / 9;
            printf("%d\t%d\n", fahr, celsius);
            fahr += step;
    }
}
```

switch (1)

Problem: Verschachteltes if sehr häufig

```
if(c==1) befehl;
else if (c==2) befehl;
else if (c==3) befehl;
...
```

Lösung: switch-case:

```
switch ( expression ){
   case const-expr : statements
   case const-expr : statements
   default : statements
}
```

switch (2)

Beispiel für switch-case:

```
int c=5;
switch (c)
{
   case 0:
         c++;
         break;
   case 5: case 1:
         c=9;
   default :
         c--;
         break;
}
```

break und continue

Mit **`break`** kann eine Schleife sofort verlassen werden; **`break`** springt immer *eine Ebene* höher

Mit **`continue`** wird sofort der nächste Schleifendurchlauf eingeleitet

Bsp:

```
for (int z=0; z<6; z++)
{
  for (int i=1; i<5; i++){
          if (i!=4) break;
          else continue;
          /* Diese Zeile wird nie erreicht! */
  }
}
```

Funktionen (1)

```
#include <stdio.h>

int Funktionsname(int, double);

int main(){
    int test=2;
    Funktionsname(test , 4.598);
}

int Funktionsname(int par1, double par2)
{
    printf(„%d%f“, par1, par2);
    return 1;
}
```

Funktionen (2)

- Minimale Funktion:

```
dummy(){}
```

- Man kann explizit angeben, dass Funktion keinen Rückgabewert hat: Typ ist dann **`void`**
- Funktionen haben immer nur *einen* Rückgabewert
- Übergabe eines Arrays:

```
name(char s[])
```

- Sichtbarkeit von Variablen: „wie üblich“ (außerhalb deklarierte Variablen sind auch in den Funktionen sichtbar etc.)
- Funktionen dürfen nicht in anderen Funktionen definiert werden!

Funktionen (3)

- Die übergebenen Werte werden *immer kopiert:*

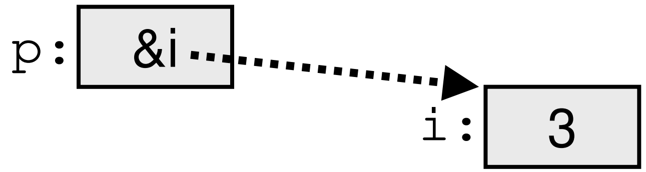
```
void swap(int x, int y) /* Falsch! */
{
    int temp=x;
    x=y;
    y=temp;
}

int a=7; int b=8;
swap(a,b);                /* a ist weiterhin=7, b=8 */
```

- Wie kann man nun Variablen durch eine Funktion ändern lassen?
- => Lösung: Referenzen und Zeiger

Zeiger (1)

- Ein ***Zeiger*** ist eine Variable, die als Wert die (virtuelle) ***Speicheradresse*** einer anderen Variable speichert!



- Beispiel:

```
int i=3;    /* Variable i wird angelegt */
int* p;     /* p ist ein Zeiger, der auf eine
               Integer-Variable zeigen kann */
p = &i;     /* &i liefert die Adresse von i */
int z= *p;  /* *p liefert den Inhalt der Adresse,
               auf die p zeigt, hier: 3; z hat somit
               den Wert 3 */
```

- Vorsicht beim Umgang mit den Zeichen `*` und `&` !

Zeiger (2)

- nun korrekte SWAP-Funktion:

```
void swap(int* px, int* py){
    int temp = *px;
    *px = *py;
    *py = temp;
}

swap(&a, &b);
```

- Zeigen auf Elemente in einem Array:

```
int z[10];
int* p;
p = &z[5];
p++;          /* jetzt zeigt p auf z[6] */
```

Zeiger (3)

- Bei Funktionen werden häufig Zeichenketten (Strings) in Form eines Zeigers übergeben, der auf das erste Element des Arrays zeigt:

```
printf(char* s); /* Vereinfachtes printf */
```

Verwendung am Bsp von printf:

```
printf("Hallo");
```

`printf` wird hier automatisch ein Zeiger auf den Anfang der konstanten Zeichenkette (im Speicher als Array abgelegt) übergeben.

Zeiger (4)

- Da Zeiger direkt auf den Adressen arbeiten, können *oft* ekeliege Abstürze entstehen.
- Vorteil von Zeigern: hohe Verarbeitungsgeschwindigkeit, deshalb gerade beim Sortieren von Zeichenketten eingesetzt (man sortiert eine Liste von Zeigern, das ist schneller als eine Liste von Arrays zu sortieren)
- Man kann sogar Zeiger auf Funktionen definieren (sehr ekelig), die:

  zugewiesen, in Arrays eingetragen, an Funktionen übergeben, als Resultatwert von Funktionen geliefert werden können, etc.

- Ebenfalls möglich: Zeiger auf Zeiger

Zeiger (5)

- Parameter der `main(int argc, char* argv[])` Funktion:
  - `argc` Anzahl der Argumente mit der das Programm aufgerufen wurde, der Programmname selbst ist das erste Argument!
  - `argv` Zeiger auf ein Array mit Zeigern auf Zeichenketten, die die einzelnen Argumente darstellen
- `NULL` ist ein Ersatz für die Zahl 0. `NULL` wird einem Zeiger zugewiesen um zu sagen, dass dieser Zeiger auf kein Objekt zeigt. Bsp:

```
int* z = NULL; /* z zeigt auf Speicherzelle „NULL“*/
```

In C++ sollte man dafür direkt die Zahl 0 verwenden.

Referenzen

- Eine ***Referenz*** ist ein ***Aliasname*** für eine Variable
- Beispiel:

```
int i=3;
int& r=i;  /* r ist ein Aliasname für die Variable
              i; ab sofort kann man beide Namen
              für dieselbe Variable verwenden */
```

- Anwendung in Funktionen (swap-Beispiel):

```
void swap(int& x, int& y){
   int temp=x;
   x=y;
   y=temp;
}
```

Strukturen (1)

- Eine Struktur ist eine Zusammenfassung von einer oder mehreren Variablen

```
struct point {
    int x;
    int y;
};   /* Semikolon nicht vergessen */
```

- struct definiert einen *Datentyp* und kann genauso verwendet werden:

```
point pt;
pt.x=1;
pt.y=2;
```

- geht auch: Definition und direkte Deklaration:

```
struct point { int x,y; } meinpunkt;
```

Strukturen (2)

- Zeiger auf Strukturen:

```
point i;
point* p = &i
p->x = 5; /*Zugriff auf x über den Zeiger*/
```

- Zeiger in Strukturen (Anwendung z.B. bei Listen):

```
struct point2{
    int x;
    int y;
    point2* next;
};
```

Zugriff auf das nächste Listenelement und dort auf x über Zeiger p:

```
p->next->x;
```

Dateikonzept (1)

- Aufteilung in mehrere Dateien:
  - Die Implementierung einer oder mehrerer Funktionen kommen in Dateien mit den Endungen .c (in genau einer Datei muss die Funktion main enthalten sein!)
  - Die Definition der Funktionen kommt in eine Definitionsdatei, das sog. „header-file“ (.h)
  - Man kann auch die Implementierung direkt in das header-file schreiben (unschön)
  - .h und .c Dateien werden nach belieben strategisch in Unterverzeichnissen verteilt
  - Vorteil: Benutzt eine andere Funktion eine spezielle Funktion, braucht sie nur das header-file einzubinden, den „Rest“ regelt der Compiler

Dateikonzept (2)

- Beispiel:

stack.h

```
void push(int);
int pop();
```

main.c

```
#include ”stack.h”

main()
{
        push(1);
        pop();
}
```

stack.c

```
#include ”stack.h”

void push(int){
...
}
int pop(){
...
}
```

Dateikonzept (3)

- Compiliervorgang:

  1. *Preprozessor* geht über die Dateien und nimmt Ersetzungen vor; fügt header-Dateien zusammen,

     d.h. insbesondere: alle header-Dateien, die von einer .c Datei eingebunden werden, werden vom Preprozessor in die .c-Datei <u>einkopiert!</u> (=> kann zu Zykeln führen)
  2. *Compiler* erstellt aus jeder .c Datei eine Objektdatei .o
  3. *Linker* fügt alle Objektdateien zu einer ausführbaren Datei zusammen und prüft, ob jede verwendete Funktion auch eine Implementierung hat

- .o-Dateien können zu einer Bibliothek (.a) zusammengefasst werden
  => spart Zeit beim compilieren

Preprozessor (1)

- nimmt im wesentlichen Ersetzungen vor
- zur Steuerung: Einfügen von speziellen Preprozessorbefehlen in den Code (Beginnen alle mit #):
  - Inhalt einer Datei wird vor der Übersetzung eingefügt:
    `#include` `<...>` oder **`#include "..."`**
  - Eine Zeichenkette wird durch andere Symbole ersetzt:
    `#define` `Zeichenkette Ersatztext`

    Bsp: `#define MAX_RANGE 14`

Preprozessor (2)

- Bedingte Übersetzung
  (stellt sicher, dass eine Datei nur einmal eingebunden wird):

```
#if !defined(STACK_H)
#define STACK_H

        /* Inhalt der Header-Datei stack.h */

#endif
```

Was wurde weggelassen

- GOTO und Sprungmarken
- „externe“ Variablen (umgehen Konzept der Kapselung in C++)
- Anforderungen von Speicher über malloc (funktioniert in C++ etwas anders)
- Zeiger auf Funktionen (wird und sollte äußerst selten eingesetzt werden)
- Unions (werden kaum eingesetzt)
- Beschreibung der Ein-/Ausgabefunktionen aus der „Standardbibliothek“ (funktioniert in C++ anders)
- typedef

Inhalt

1. Einleitung ✓
2. C ✓
3. **C++**
4. Compiler und make

Einführung

- Erweiterung der Sprache C um objektorientierte Elemente
- „Erfinder“ Bjarne Stroustrup
- Alle im Folgenden vorgestellten Elemente können verwendet werden, müssen aber nicht

„Hello World“

```
#include <iostream>

int main()
{
    std::cout << “Hello, World\n“;
}
```

Unterschiede zu C (1)

- In C++ wurde der Datentyp **`bool`** aufgenommen
  mögliche Werte: **`true`** und **`false`**

- `String:` In C++ wird die Klasse `string` verwendet, um Zeichenketten zu verwalten. `string` ist eine *Klasse* aus der Standard Template Library (STL) und *kein* eigener Typ

- Kommentare:

```
// Dies ist ein Kommentar; bis zum Ende der Zeile
```

Unterschiede zu C (2)

- Typkonvertierung:
  - verwandte Typen (z.B. in Klassenhierarchie):

```
int i=99;
double p = static_cast<double>(i);
// in C: double p=(double)i;
```

  - nicht verwandte Typen (z.B. int und einem Zeiger):

```
IO_device* d1=reinterpret_cast<IO_device*>(0xff00);
```

Unterschied: static_cast ist sicherer als reinterpret_cast

  - entfernen von `const`:

```
const_cast< T >(arg)
```

  - Typkonvertierung, die einen gültigen Zeiger liefert, wenn das Objekt vom erwarteten Typ ist (herausfinden von welchem Typ Objekt ist):

```
int* pb=dynamic_cast<int*>(pw)
```

Zeiger (1)

- dynamische Speicherverwaltung:
  - Mit **`new`** wird ein neues Objekt im Speicher erzeugt, mit **`delete`** wieder zerstört; es existiert *keine* Garbage-Collection => jedes angelegte Objekt muss auch wieder zerstört werden!
  - Bsp Struktur (für Klassen bzw. Objekte Vorgehen analog):

```
struct point {int x, int y};

point* punkt = new point; //new gibt Zeiger zurück
punkt->x=4;
delete punkt;
```

Nach dem Zerstören zeigt punkt weiter auf die Speicherzelle, wo zuvor das Objekt war => `punkt` sollte auf `NULL` bzw. `0` gesetzt werden:

```
punkt=0;
```

Zeiger (2)

- Das gilt auch für Arrays:

```
char* s = new char[100];
// und wieder löschen:
delete[] s;
```

- **void** kann auch als Basistyp für Zeiger auf Objekte unbekannten Typs verwendet werden, sollte aber vermieden werden:

```
void* pv;
```

Funktionen

- Möglich sind überladene Funktionen (unabh. von Objekten):

```
void ausgeben(int);
void ausgeben(const char*);
```

- Default-Argumente:

```
void ausgeben2(int value, int basis = 10);
```

Namensbereiche (1)

- Ein Namensbereich (namespace) ist ein Mechanismus zum Ausdrücken von logischen Gruppierungen
- Sinn der Namensbereiche:
  - Alle Namen, die nicht in Klassen, Strukturen oder Funktionen eingekapselt sind, werden im globalen Namensbereich verwendet
  - Dies kann zu Überschneidungen führen =>
  - Mit den Namensbereichen sollen Namenskonflikte im globalen Gültigkeitsbereich verhindert werden
- Alle Fkt. der Standardbibliothek sind im Namensbereich `std`

Namensbereiche (2)

- Bsp:

```
 // Definition:
namespace Math {
  double expr(bool) {...};
  double prim(bool get) {...};
  double term(bool get);
}
// Implementierung von term ausserhalb:
double Math::term(bool get) {...};
```

- Anwendung:

```
 double result = Math::prim(true);
```

Namensbereiche (3)

- Using namespace (macht alle Namen eines Namensbereiches im Folgenden verfügbar):

```
using namespace Math;
// ...
double result = prim(true);
```

Steht meistens am Anfang einer Datei:

```
#include <string>

using namespace std;

...
```

Klassen (1)

- Definition:

```
class Datum
{
        int t,m,j;
public:
        void init(int tt, int mm, int jj);

        void addiereJahr(int n);
        void addiereMonat(int n);
        void addiereTag(int n);
};  // Hier Semikolon nicht vergessen
```

Klassen (2)

- Definition der Elementfunktionen:

```
void Datum::init(int tt, int mm, int jj)
{
   t=tt;
   m=mm;
   j=jj;
}
...
```

- In der Regel findet man die Definition der Klassen (vorherige Folie) in den Header-Dateien, wärend die Implementierung der Funktionen in den gleichnamigen .cpp Dateien zu finden sind

Klassen (3)

- Anwendung:

```
// Instanz anlegen:
Datum heute;



// Aufruf öffentlicher Elementfunktionen:
heute.init(12, 1, 2000);
heute.addiereJahr(5);
```

Klassen (4)

- Zugriffsbeschränkungen:

```
class X{
private:
        // alle Namen können nur von Element-
        // funktionen der Klasse X benutzt werden
protected:
        // alle Namen können nur von Element-
        // funktionen der Klasse X und denen aus
        // ihr abgeleiteten Klassen benutzt werden
public:
        // alle öffentlichen Elemente
};
```

- Kein Bezeichner: alle Namen standardmäßig `private`

Klassen (5)

- Konstruktor:

```
class Datum{
        //...
public:
        Datum(int, int, int);
        Datum();    // Standardinitialisierung
};
```

- Anwendung:

```
Datum heute;        // Standardinitialisierung
Datum morgen(13,1,2000);
```

Klassen (6)

- Destruktor wird aufgerufen, wenn das Objekt zerstört wird; dient zum aufräumen:

```
class Datum
{
      int* p;
public:
      Datum() {p = new int;}
      ~Datum() {delete p;}
};
```

Klassen (7) - Beispiel (1)

```
class Datum {
        int tag, monat, jahr;
public:
        Datum(int, int, int);
        ~Datum();
        void setzeDatum(int, int, int);
        void holeDatum(int&, int&, int&);
};

Datum::Datum(int t, int m, int j) {
        tag=t; monat=m; jahr=j;
}

void Datum::setzeDatum(int t, int m, int j) {
        tag=t; monat=m; jahr=j;
}
// hole Datum kopiert die Werte
void Datum::holeDatum(int& t, int& m, int& j) {
        t=tag; m=monat; j=jahr;
}
```

Klassen (8) - Beispiel (2)

Datei: *datum.h*

```
#if defined(DATUM_H)
#define DATUM_H

class Datum {
        int tag, monat, jahr;
public:
        Datum(int, int, int);
        ~Datum();
        void setzeDatum(int, int, int);
        void holeDatum(int&, int&, int&);
};

#endif
```

Klassen (9) - Beispiel (3)

Datei: *datum.cpp*

```
#include "datum.h"

Datum::Datum(int t, int m, int j) {
        tag=t; monat=m; jahr=j;
}

void Datum::setzeDatum(int t, int m, int j) {
        tag=t; monat=m; jahr=j;
}
// hole Datum kopiert die Werte
void Datum::holeDatum(int& t, int& m, int& j) {
        t=tag; m=monat; j=jahr;
}
```

Klassen (10)

- Statische Elemente (nur *eine* Version in allen Objekten):

```
class Datum{
      static int ueberallgleich;
public:
      // ...
};
```

- Kopieren:

```
Datum morgen(13,1,2000);
Datum heute(morgen); // elementweises kopieren

// sonst: selbst Copy-Konstruktor definieren:
Datum::Datum(const Datum&);
```

Klassen (11)

- Selbstreferenz:
  - Jede Klasse besitzt eine Zeiger-Variable **`this`**, die auf das Objekt selbst zeigt
  - **`this`** ist keine gewöhnliche Variable: man kann weder die Adresse von **`this`** ermitteln, noch kann man **`this`** etwas zuweisen

Überladene Operatoren (1)

- In C++ kann man eigene Operatoren definieren; z.B. kann man sein eigenes + definieren
- Anschaulich: jeder Operator ist eine Funktion, die man überschreiben kann; Bsp:

```
class complex {
   double re, im;
public:
   complex(double r, double i){re=r;im=i;}
   complex operator +(complex);
   complex operator *(complex);
};
```

Überladene Operatoren (2)

- Implementierung:

```
complex complex::operator +(complex) {...}
```

- Anwendung:

```
complex a(1,3.1);
complex b(1.2,2);
complex c(0,0);

// Anstelle von a = b.addiere(c); schreibt man
a=b+c;
```

- Nicht überladen werden können: `::` `.` `.*` `?:` `#` `##`

Überladene Operatoren (3)

- Anfangsbeispiel (Hello World):

```
#include <iostream>

int main()
{
        std::cout << “Hello, World\n“;
}
```

Vordefiniertes Objekt

Namensbereich std

Überladener Operator

Abgeleitete Klassen (1)

- Manager wird von Angestellter abgeleitet:

```
class Angestellter{
    string vorname, nachname;
    //...
  public:
    void ausgeben();
};

class Manager : public Angestellter{
    short stufe;
    //...
  public:
    void ausgeben(); //überschreibt Angestellter::ausgeben
};
```

Abgeleitete Klassen (2)

- Sichtbarkeiten:

```
class Manager : public Angestellter{
   //...
};
```

	El. der Basisklasse	werden in abgel. Klasse
public	public protected	public protected
protected	public protected	protected protected
private	public proteceted	private private

Abgeleitete Klassen (3)

- Konstruktoren-Problem: Manager sieht `vorname` und `nachname` *nicht*, da beide `private` sind!:

```
class Angestellter{
    string vorname, nachname;
  public:
        Angestellter(string v, string n);
};
class Manager : public Angestellter{
    short stufe;
  public:
        Manager(string mv, string mn);
};
```

- Lösung: Bei der Implementierung der Funktion Konstruktor aufraufen:

```
Manager::Manager(mv,mn) : Angestellter(mv,mn)  { ... }
```

Abgeleitete Klassen (4)

- Mehrfacherbung:

```
class Teilzeit {...};
class Sekretaer : public Angestellter {...};
class Tsek : public Teilzeit, public Sekretaer {...};
class Berater : public Teilzeit, public Manager {...};
```

- Was passiert, wenn `Teilzeit` und `Sekretaer` zwei genau gleich benannte Funktionen definieren und `Tsek` diese Funktion aus `Sekretaer` aufrufen will?

```
void Tsek::aufrufendeFunktion()
{
        //...
        Sekretaer::funktion();
        //...
}
```

Abgeleitete Klassen (5)

- Wie kommt man an überladene Funktionen:

```
class Angestellter{
    string vorname;
public:
    void ausgeben();
};
class Manager : public Angestellter{
public:
    void ausgeben();
    void ausgebenAngestellter();
}

Manager::ausgebenAngestellter() {
    Angestellter::ausgeben();
}
```

Abgeleitete Klassen (6)

- Virtuelle Funktionen: Eine virtuelle Funktion ***kann*** von den abgeleiteten Klassen ***redefiniert*** werden (und nicht nur überschrieben):

```
class Angestellter {
   string vorname
public:
   virtual void ausgeben();
};
```

Abgeleitete Klassen (7)

- Abstrakte Klassen: Eine Klasse wird zu einer abstrakten Klasse, wenn sie mindestens eine ***rein virtuelle*** Funktion besitzt (=virtuelle Funktion, die mit 0 Initialisiert wird):

```
class Form{
   //...
public:
   virtual void zeichne() =0;
};
```

Diese Klasse kann selbst nicht mehr instanziiert werden, sie dient nur noch als *Schnittstelle und als Basisklasse* für andere Klassen:

```
class Kreis : public Form{
 public:
       void zeichne();        //... };
```

Ausnahmebehandlung (1)

- Syntax:

```
class teiledurchnullfehler {};

int dividiere(int a, int b)      // teilt a durch b
{
   if(b==0) throw teiledruchnullfehler;
   else return a/b;
}
```

  – Fehler sind Objekte, wobei für jeden Fehler ein Fehlerobjekt erzeugt wird (hier: `teiledurchnullfehler`)

Ausnahmebehandlung (2)

```
void f()
 {
    try{
        dividiere(3,0);
    }
    catch (teiledurchnullfehler){
            //Behandle teiledurchnullfehler und alle
            //davon abgeleitete Fehlerklassen
    }
    catch(...){   // Behandle jede Ausnahme, die ...
        throw;    // ... hier einfach weiter geworfen wird
    }
 }
```

Wird eine Exeption nicht gefangen, terminiert das gesamte Programm

Templates (1)

- Funktions-Templates: Bsp: Maximum
  - Funktion `max(x,y)` soll das Maximum von `x` und `y` zurückgeben, dabei sollen `x` und `y` beliebige Typen sein
  - Lösung: Funktions-Template:

```
template <class T> T max(T x, T y){
        return (x>y) ? x : y;
};
```

  - Anwendung:

```
int i;
int j=max(i,0);
```

Templates (2)

- Klassen-Template (auch generische oder parametrisierbare Klasse):
  - ist eine Art „Maske“ oder ***„Vorlage“ für Klassen*** und dient vorwiegend für „Container“, wie z.B. einer allgemeinen Klasse Stack
  - STL baut auf Templates auf
  - Wird genauso definiert und deklariert, wie normale Klassen, nur das ein oder mehrere Elemente nicht in der Klasse definiert werden

Templates (3)

- Beispiel: Definition:

```
template <class T> class Vector{
   T* data;
public:
   Vector (int n);
   ~Vector() {delete[] data;}
};

template <class T> Vector<T>::Vector(int n)
{
   data= new T[n];
};
...
```

Templates (4)

- Beispiel: Instanziierung:

```
void f()
{

   Vector<int> integerVector(10);

}
```

- Templates lassen sich u.a. ableiten und spezialisieren

STL (1)

- Die Standard Template Library enthält eine Sammlung von Templates für häufig verwendete Datenstrukturen und Algorithmen
- Bsp. für eine Liste, die in diesem Fall Integer-Werte aufnehmen soll:

```
#include <list>

main()
{
        list<int> integerListe;
}
```

STL (2)

- Die Klassen der STL verwenden sog. `Iteratoren` =Zeiger auf die Elemente des entsprechenden Containers
  - Iteratoren verhalten sich wie Zeiger, sind aber keine - sie können aber durch Zeiger implementiert sein
  - Abstraktion wurde gewählt, damit die Verwendung nach außen einheitlich ist
  - Bsp: durchlaufen der Liste und Ausgabe der Elemente:

```
list<int>::iterator aktuell=integerListe.begin();
while(aktuell!=integerListe.end())
{
        cout<<(*aktuell);
        aktuell++;
}
```

Was wurde weggelassen

- friends
- detaillierte Beschreibung der STL (würde den Rahmen sprengen)
- Beschreibung der Ein-/Ausgabe Klassen und Objekte aus der „Standardbibliothek“, insbesondere der Dateifunktionen
- kleinerer „Krimskrams“
- inline-Funktionen

Inhalt

1. Einleitung ✓
2. C ✓
3. C++ ✓
4. **Compiler und make**

Compiler und make (1)

- $\underbrace{\texttt{-o hello}}_{\text{Name des Programms}}\ \underbrace{\texttt{hello.cpp}}_{\text{Quelldatei}}$ `g++ -o hello hello.cpp`

Wird -o nicht angegeben erstellt Compiler die ausführbare Datei namens `a.out`

- `g++ -I/usr/openwin/include fred.c`

$\underbrace{\texttt{-I/usr/openwin/include}}_{\text{Sucht Header-Dateien in nicht standardisierten Ablagorten}}$

Compiler und make (2)

- Bibliotheken:

  `.a` statische Bibliotheken

  `.so` bzw. `.sa` dynamische Bibliotheken

- Bibliothek linken

```
g++ -o fred fred.c /usr/lib/libm.a
g++ -o fred fred.c -lm
g++ -o x11fred -L/usr/openwin/lib fred.c -lX11
```

Compiler und make (3)

- Make automatisiert den Compiliervorgang und sorgt dafür, dass von Änderungen betroffenen Dateien rekompiliert werden
- dazu: erstellen eines MAKEFILEs, Bsp:

```
# dies ist ein Kommentar
myapp: main.o 2.o
   gcc –o myapp main.o 2.o

main.o: main.c a.h
   gcc –c main.c

2.o: 2.c a.h b.h
   gcc –c 2.c
```

Compiler und make (4)

- Erweitertes Beispiel:

```
# Bei mehrere Programmen, beginne mit Pseudoziel all
all: myapp
# Verwende GNU-C Compiler gcc
CC=gcc
# Hier sind weitere Header-Dateien abgelegt
INCLUDE=/project/include
# Diese Flags werden jedem Compileraufruf mitgegeben
CFLAGS= -g -Wall -ansi

myapp: main.o 2.o
  $(CC) -o myapp main.o 2.o
main.o: main.c a.h
  $(CC) -I$(INCLUDE) $(CFLAGS) -c main.c
# ... und so weiter
```

Inhalt

1. Einleitung ✓
2. C ✓
3. C++ ✓
4. Compiler und make ✓


12
3