

Skript zur Vorlesung

Objektorientierte Programmierung in C++

Dr. Klaus Ahrens
Institut für Informatik
Humboldt-Universität zu Berlin
2001 / 2002



Inhalt

1. Die Sprache im Überblick	5
1.1. Ursprung und Autor	5
1.2. Chronologie der Sprache	5
1.3. Der „Stammbaum“ von C++	7
1.4. Compiler	9
1.5. Standardisierung	9
1.6. Literatur	9
2. Ein erstes C++ -Programm	10
2.1. keywords	12
2.2. Lexik	12
2.3. Ein-/Ausgabe (als blackbox)	17
3. Vom Quelltext zum Programm	18
3.1. {{{edit}, compile}, link}, debug}, run	18
3.2. separate Übersetzung	20
4. Elementares C++	22
4.1. Präprozessor	22
4.2. <i>Built-in</i> Typen	24
4.3. einfache Typkonstrukte	27
4.4. Ausdrücke	36

4.5. Funktionen	38
4.6. Anweisungen	42
4.7. Migration von C nach C++	45
5. Klassen und Objekte	46
5.1. Grundprinzipien der OOP	46
5.2. Objekte im Speicher	49
5.3. Konstruktoren und Destruktoren I	52
5.4. Zugriffsschutz I	57
5.5. Lokalität	60
5.6. Zeiger auf Member	62
6. Vererbung	64
6.1. Redefinitionen	66
6.2. Konstruktoren und Destruktoren II	68
6.3. Zugriffsschutz II	71
7. Polymorphie und Virtualität	74
7.1. Die IST EIN - Relation	74
7.2. virtuelle Funktionen	75
7.3. abstrakte Basisklassen	83
7.4. Wiederverwendbarkeit	84
7.5. Objektidentität	85

8. Überladung und Konversion	90
8.1. Funktionen	90
8.2. Operatorüberladung	92
8.3. Typumwandlungen und Casts	106
8.4. Ein-/Ausgabe	109
9. Mehrfachvererbung	109
9.1. virtuelle und nicht virtuelle Basisklassen	109
9.2. Mehrdeutigkeiten	111
9.3. Konstruktoren und Destruktoren III	114
9.4. Zugriffsschutz III	117
10. Templates	118
10.1. generische Klassen	118
10.2. parametrisierte Klassen	120
10.3. Funktionstemplates	122
11. Standard Template Library (STL)	124
11.1. STL - Vektoren	134
11.2. STL - Deques	136
11.3. STL - Listen	137
11.4. STL - Mengen und - Multimengen	139
11.5. STL - Maps und - Multimaps	141

12. Exceptions	148
12.1. Programmausnahmen	148
12.2. hierarchische Behandlung	150
12.3. Programmierstil	152
13. Namespaces	155
13.1. Motivation	155
13.2. namespace std	159
13.3. Namespaces in g++	159
14. RTTI und neue Castoperatoren	161
14.1. Typidentifikation	161
14.2. Typumwandlung	164

1. Die Sprache im Überblick

1.1. Ursprung und Autor

- Bjarne Stroustrup Ph.D. Arbeit 1978/79 an der Universität Cambridge: „Alternative Organisationsmöglichkeiten der Systemsoftware in verteilten Systemen“
- erste Implementation in Simula auf IBM360 (Simula67, NCC Oslo)
- Stroustrup: „Die Entwicklung des Simulators war das reinste Vergnügen, da Simula nahezu ideal für diesen Zweck erschien. Besonders beeindruckt wurde ich durch die Art, in der die Konzepte der Sprache mich beim Überdenken der Probleme meiner Anwendung unterstützten. Das Konzept der Klassen gestattete mir, die Konzepte meiner Anwendung direkt einzelnen Sprachkonstrukten zuzuordnen. So erhielt ich Programmcode, der in seiner Lesbarkeit allen Programmen anderer Sprachen überlegen war, die ich bisher gesehen hatte.“
- Simula - Compiler damals mit extrem schlechten Laufzeiteigenschaften
- S.: „Um das Projekt nicht gänzlich abzubrechen - und Cambridge ohne Ph.D. zu verlassen -, schrieb ich den Simulator ein zweites Mal in BCPL Die Erfahrungen, die ich während des Entwickelns und der Fehlersuche in BCPL sammelte, waren grauenerregend.“
- erste Ideen zu C++ im Kontext von Untersuchungen Lastverteilung in UNIX-Netzen bei den Bell Labs Murray Hill, New Jersey: Stroustrup: „Ende 1979 hatte ich einen lauffähigen Präprozessor mit dem Namen *Cpre* geschrieben, der C um Simula-ähnliche Klassen erweiterte.“ -> *C with classes*

1.2. Chronologie der Sprache

1979	Mai	Start der Arbeiten an C with classes
	Oktober	1. Einsatz einer Implementierung von C with classes
1980	April	1. interne Freigabe der Bell Labs zu C with classes
1982	Januar	1. öffentlich Freigabe von C with classes
1983	August	1. Einsatz einer C++ -Implementierung
	Dezember	1. Nennung von C++
1984	Januar	1. C++-Handbuch
1985	Februar	1. freigegebene C++ Version
	Oktober	<i>Cfront</i> Version 1.0 (erste kommerzielle Version) <i>The C++ Programming Language</i>
1986	September	1. OOPSLA-Konferenz
	November	1. kommerzielle <i>Cfront</i> -Portierung auf einem PC (Glockenspiel <i>Cfront</i> 1.1)
1987	Februar	<i>Cfront</i> Version 1.2, noch als Quelle zur Crosscompilation
	November	verfügbar
	Dezember	1. USENIX-Konferenz zu C++ 1. C++-Version von GNU

Objektorientierte Programmierung in C++

1988	Januar	1. C++ Version von Oregon Software
	Juni	1. C++ Version von Zortech
	Oktober	1. USENIX-Arbeitsgruppe für C++-Implementierer
1989	Juni	Cfront Version 2.0
	Dezember	1. organisatorisches Treffen des ANSI X3J16
1990	März	1. Fachtreffen des ANSI X3J16
	Mai	1. C++ Version von Borland (TC1.0)
	Mai	<i>The Annotated C++ Reference Manual</i> (ARM)
	Juli	Annahme von Templates
	November	Annahme von Exception Handling
1991	Juni	<i>The C++ Programming Language</i> 2. Auflage
	Juni	1. ISO WG21-Tagung
	Oktober	Cfront Version 3.0 (mit Templates)
1992	Februar	1. C++ Version von DEC (incl. Templates, Exc. Handl.)
	März	Handl.)
	Mai	1. C++ Version von Microsoft
		1. C++ Version von IBM (incl. Templates, Exc. Handl.)
1993	März	Annahme von RTTI
	Juli	Annahme von namespaces
1994	September	Termin für Entwurf des ANSI/ISO Standards
1995	April	Draft Standard X3J16/95-0087, WG21/N0687
1998	Juni	Standard offiziell verabschiedet ISO/IEC 14882

Quelle: B. Stroustrup: „*Design und Entwicklung von C++*“, Addison Wesley 1994

einer davon:

C++ is like teenage sex

- It is on everyone's mind all the time.
- Everyone talks about it all the time.
- Everyone thinks everyone else is doing it.
- Almost no one is really doing it.
- The few who are doing it are:
 - Doing it poorly.
 - Sure it will be better next time
 - Not practicing it safely.



Groffti found in a toilet stall at the Faculty of Computer Science, Technion--IIT, Haifa, Israel on 8 November 1993.

Objektorientierte Programmierung in C++

1.4. Compiler

hier im Hause verfügbar:

- GNU C++ g++ [alle WS-Pools: SUN, DEC, IBM] (Version 2.95, +T, +- E)
Linux (>=2.95), DOS-Portierung DJGPP
- SPARCWORKS Professional C++ 3.01 (SunOS, cfront, +T, -E)
- SPARCWORKS Professional C++ 4.01 (Solaris, +T, +E)
- MS Visual C++ 6.0

1.5. Standardisierung

- gemeinsamer ANSI/ISO -Standard de facto im November 1997 beschlossen !
- legt Sprachsyntax/-semantik und Mindestumfang von Standardbibliotheken fest
- kompletter Sprachumfang wird bislang nur von wenigen Compilern unterstützt

1.6. Literatur

C++

Stroustrup, B.: „The C++ Programming Language“ (3rd edition !) Addison Wesley, 1997.
(auch in deutsch)

Stroustrup, B.: „The Design and Evolution of C++“ [D&E], Addison Wesley, 1994. (auch in deutsch)

Stroustrup, B., Ellis, M.: „The Annotated C++ Reference Manual“ [ARM], Addison Wesley, 1990 und weitere.

Lippman, S.: „C++. Einführung und Leitfaden“, Addison Wesley, 1990.

Hansen, T.: „The C++ Answer Book“, Addison Wesley, 1990.

OOP

Booch, G.: „Object Oriented Design with Applications“ 2nd ed. Benjamin Cummings, 1993.

Cox, B.: „Object-Oriented Programming: An Evolutionary Approach“, Addison Wesley 1986.

Meyer, B.: „Object-Oriented Software Construction“, Prentice Hall, 1988.

Ahrens, K.; Fischer, J.: „Objektorientierte Programmierung“, Verlag Technik, 1992.

Ahrens, K.; Fischer J.: „Objektorientierte Prozesssimulation in C++“ Addison Wesley 1996

Web-Sources

http://www.yahoo.com/Computers_and_Internet/Software/Programming_Tools/Object_Oriented_Programming/



2. Ein erstes C++ -Programm

- Objekte eines Gegenstandsbereiches werden identifiziert
- gemeinsame Eigenschaften / gemeinsames Verhalten werden in Form von Klassenvereinbarungen festgehalten:
- **KLASSE** aller Menschen
(u.a.) folgende Eigenschaften: Name, Geschlecht, verheiratet sein, Ehepartner, ...
(u.a.) folgendes Verhalten: Geburt, Tod, Heirat,

```
#include <iostream>
#include <cstring>

using namespace std;

const class People* nobody=0;

enum Sex {male, female};

class People
{
    const char *name;
    const Sex sex;
    People* in_marriage_to;

    bool married() const { return in_marriage_to!=nobody; }

    static void m_error (const char* e)
    {
        cout << "*** marriage error: " << e << endl;
    }

    const char* wife_or_husband() const
    {
        return sex==male ? "wife" : "husband";
    }

    People* partner() const
    {
        if (married())
            return in_marriage_to;
        else return const_cast<People*>(nobody);
    }

public:
    // enter the scene
    People(const char* n, Sex s)
        :name(new char[strlen(n)+1]),
        sex(s), in_marriage_to(const_cast<People*>(nobody))
    {
        strcpy(const_cast<char*>(name), n);
        cout << "Hi, this is " << name << endl;
    }

    // leave the scene
    ~People()
    {
        cout << "Bye, bye says " << name << endl;
        delete[] const_cast<char*>(name);
    }
}
```

Objektorientierte Programmierung in C++

```
void hello() const
{
    cout << "Hello, I am " << name;
    if (married())
        cout << " and this is my "
            << wife_or_husband() << ' '
            << partner()->name;
    cout << endl;
}

void marries (People& to_who)
{
    if (sex==to_who.sex)
    {
        m_error ("bad sex"); return;
    }
    if (married() || to_who.married())
    {
        m_error ("bigamy"); return;
    }
    in_marriage_to = &to_who;
    to_who.in_marriage_to=this;
}

};

People Fred ("Fred", male);
People Wilma ("Wilma", female);
People Barney ("Barney", male);
People Betty ("Betty", female);

int main()
{
    Fred .marries (Wilma);
    Wilma .marries (Barney);
    Barney.marries(Fred);
    Betty .marries(Barney);

    Fred .hello();
    Barney.hello();
    Wilma .hello();
    Betty .hello();

    return 0; // not necessary (VC++ needs it !)
}
```

alle Beispiele unter:

[http://www/~ahrens ...](http://www/~ahrens...) bzw.

<ftp://ftp/pub/local/vorlesung/c++>



- Anliegen des Beispiels: „bekannter Gegenstandsbereich“; Möglichkeit, Konzepte und Notation zumindest zu „erahnen“, Vorstellung einer Vielzahl von Ausdrucksmitteln der Sprache

- Ausgaben des Programms:

```
Hi, this is Fred
Hi, this is Wilma
Hi, this is Barney
Hi, this is Betty
***marriage error: bigamy
***marriage error: bad sex
Hello, I am Fred and this is my wife Wilma
Hello, I am Barney and this is my wife Betty
Hello, I am Wilma and this is my husband Fred
Hello, I am Betty and this is my husband Barney
Bye, bye says Betty
Bye, bye says Barney
Bye, bye says Wilma
Bye, bye says Fred
```

man lese auch <ftp://ftp.research.att.com/dist/c++std/WP/CD2>
C++ draft standard als pdf/ps!!!



2.1. keywords

asm(C)	do(C)	if(C)	return(C)	typedef(C)
auto(C)	double(C)	inline	short(C)	typeid(N)
bool(N)	dynamic_cast(N)	int(C)	signed(C)	typename(N)
break(C)	else(C)	long(C)	sizeof(C)	union(C)
case(C)	enum(C)	mutable(N)	static(C)	unsigned(C)
catch(E)	explicit(N)	namespace(N)	static_cast(N)	using(N)
char(C)	export(N)	new	struct(C)	virtual
class	extern(C)	operator	switch(C)	void(C)
const(C)	false(N)	private	template(T)	volatile(C)
const_cast(N)	float(C)	protected	this	wchar_t(N)
continue(C)	for(C)	public	throw(E)	while(C)
default(C)	friend	register(C)	true(N)	
delete	goto(C)	reinterpret_cast(N)	try(E)	

- 63 !!! reservierte Bezeichner
- einige intuitiv klar: Typnamen `int`, `long`, `bool`, `char`, `float`, `double` wie in anderen Sprachen; ebenso Ablaufsteuerung: `do`, `if`, `else`, `for`, `(goto)`, `return`, `while`
- Kategorie (C): Keywords von (ANSI-) C
- Kategorie (N): Neuerungen nach 1993 noch nicht von allen Compilern unterstützt
- Kategorie (T): Templates
- Kategorie (E): Exception Handling
- hier im Kurs zunächst Beschränkung auf den „Rest“ (incl. (C))

einige Bezeichner, die man nicht verwenden sollte, weil sie als alternative Notation für Operatoren zugelassen sind:

bitand	and	bitor	or	xor
compl	and_eq	or_eq	xor_eq	not
not_eq				

2.2. Lexik

- ~Morpheme, ~Token: die „Wörter“ der Sprache, es gibt 5 Arten:
- *Identifier* (Bezeichner) = nutzerdefinierte Namen für Variablen, Typen, ...

ident::=letter { letter | digit } .
letter::= „_“ | „a“ | ... | „z“ | „A“ | „Z“ .
digit::= „0“ | ... | „9“ .

Groß-/Kleinschreibung wird unterschieden, empf. Stil: Klassen, Typen groß; Variablen klein; `_` zur Strukturierung langer Namen, nicht am Anfang



- *Keywords* s.o., müssen in Kombination mit Bezeichnern/Keywords durch sog. „whitespaces“ separiert sein ! `const<int`

Objektorientierte Programmierung in C++

- *Literale*

A) Integer-Literale: ganze Zahlen, mit/ohne Vorzeichen in drei Zahlenformaten (dezimal, hexadezimal, oktal) mit optionalen Suffixes, Repräsentation maschinenabhängig (16, 32, 64 Bit)



[+][-]nnn...nnn - (n=0..9) dezimal
Onnn...nnn - (n=0..7) oktal
0xnnn...nnn - (n=0..9,a..f,A..F) hexadezimal
0Xnnn...nnn - dito

Integer-Literale sind zunächst vom Typ `int`, können jedoch durch direkt nachgestelltes (ohne whitespace)

u | U - zu unsigned, bzw.
l | L - zu long

werden.

B) Character-Literale: Zeichen in einfache Apostrophe eingeschlossen, die für sich selbst stehen, bzw. in oktaler/hexadezimaler Codierung, ein Byte, Vorzeichenausdehnung maschinenabhängig



'a' - das Zeichen a
'0' - das Zeichen 0 (nicht der numerische Wert 0, das ist `'\0'`)
'\nnn' - (n=0..7) das Zeichen mit dem oktalen Code nnn
'\xnn' - (n=hexdigit)hexadezimalennn
'\?' '\?' '\?' '\?' '\a' '\b' '\f' '\n' '\r' '\t' '\v' - sog. escape-Zeichen häufig benutzter, nicht-druckbarer Zeichen bzw. Maskierung:

new-line	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
alert	\a

internationale Zeichensätze (UNICODE): **wchar_t**, **L'ab'**

C) Floating-Literale: reellwertige Konstanten verschiedener Genauigkeit, mit/ohne Vorzeichen, mit/ohne Dezimalpunkt, mit/ohne Exponent mit/ohne Vorzeichen, ganzer Anteil oder Mantisse muss vorhanden sein, Dezimalpunkt oder Exponent muss vorhanden sein, Typ ist `double`, außer wenn **f|F** (`float`) bzw. **l|L** (`long double`) nachgestellt wird, Repräsentation aller floating-point Zahlen maschinenabhängig



Beispiele:

Objektorientierte Programmierung in C++

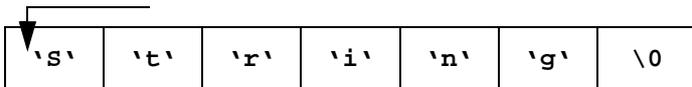
1.5 -2 5e2 +123.0001 -9.99E-12 ok
 4 e2 -23e .e1 falsch

D) String-Literale: Zeichenketten in Doppelapostrophe (") eingeschlossen aus Zeichen, die für sich selbst stehen; escape-Zeichen erlaubt. Typ ist `const char[]`, Überlappungen impl.-abhängig, Verhalten bei Modifikationen undefiniert, Darstellung stets mit abschließender `\0` im Speicher



“das ist ‘ne Zeichenkette mit Zeilenende\n“
 “eine, die \“ und Umlaute enthält“

“String“



internationale Zeichensätze (UNICODE): L“abcd“

E) Boolean-Literale: noch nicht in allen Compilern implementiert, `int` -Typen sind logisch auswertbar !

true
false

verschiedene ‚work arounds‘ nicht 100% identisch:

```
#define false 0
#define true 1
```

oder

```
enum bool {false, true};
```



- *Operatoren*: arithmetische, logische, bitweise Verknüpfungen, Typumwandlung, dynamische Speicherverwaltung

Operator	Standardsemantik	Benutzung	Assoz.
::	Bereichsauflösung	K::E	links
::	global	::N	links
->	Elementauswahl (Zeiger)	Z -> E	links
.	Elementauswahl	L.E	links
[]	Feldindizierung	Z [A]	links
()	Funktionsaufruf	A (AL)	links
()	Konstruktion	T (AL)	links
sizeof	Objektgröße	sizeof A	rechts
sizeof	Typgröße	sizeof (T)	rechts
++	prefix inkrement	++L	rechts
++	postfix inkrement	L++	rechts
--	prefix dekrement	--L	rechts

Priorität abnehmend ----->

Objektorientierte Programmierung in C++

Operator	Standardsemantik	Benutzung	Assoz.
--	postfix dekrement	L--	rechts
~	Komplement	~A	rechts
!	Negation	!A	rechts
-	unäres Minus	-A	rechts
+	unäres Plus	+A	rechts
&	Adressoperator	&L	rechts
*	Dereferenzierung	*A	rechts
new	Objekte anlegen	new T	rechts
delete	Objekte freigeben	delete Z	rechts
new []	Objektvektoren anlegen	new T[n]	rechts
delete[]	Objektvektoren freigeben	delete [] Z	rechts
()	Typkonvertierung	(T) A, T(A)	rechts
const_cast<>	Typkonvertierung	const_cast<T> A	rechts
static_cast<>	Typkonvertierung	static_cast<T>A	rechts
dynamic_cast<>	Typkonvertierung	dynamic_cast<T> A	rechts
reinterpret_cast<>	Typkonvertierung	reinterpret_cast<T> A	rechts
*	Multiplikation	A * A	links
/	Division	A / A	links
%	Modulo	A % A	links
+	Addition	A + A	links
-	Subtraktion	A - A	links
<<	Linksshift	L << A	links
>>	Rechtsshift	L >> A	links
<	kleiner	A < A	links
<=	kleiner gleich	A <= A	links
>	größer	A > A	links
>=	größer gleich	A >= A	links
==	gleich	A == A	links
!=	ungleich	A != A	links
&	bitweises UND	A & A	links
^	bitw. excl. ODER	A ^ A	links
	bitweises ODER	A A	links
&&	logisches UND	A && A	links
	logisches ODER	A A	links
?:	Bedingung	A ? A : A	rechts
=	Zuweisung	L = A	rechts
*=	Multipl. Zuweisung	L *= A	rechts
/=	Division Zuweisung	L /= A	rechts
%=	Modulo Zuweisung	L %= A	rechts
+=	Addition Zuweisung	L += A	rechts
-=	Subtrakt. Zuweisung	L -= A	rechts
<<=	Leftshift Zuweisung	L <<= A	rechts

Priorität abnehmend ----->

Objektorientierte Programmierung in C++

Operator	Standardsemantik	Benutzung	Assoz.
>>=	Rightshift Zuweisung	L >>= A	rechts
&=	bitw. UND Zuweisung	L &= A	rechts
=	bitw. ODER Zuweisung	L = A	rechts
^=	excl. ODER Zuweisung	L ^= A	rechts
,	Sequenz	A ,A	links

Legende: (L)value,(Z)eiger,(K)lasse,(T)yp, (E)lement,(N)ame,(A)usdruck, (A)usdrucks(L)iste

Achtung Falle: `int f (int*=0)` und `template1<template2<int>>`

- *Satzzeichen* (Punctuators):

! % ^ & * () - + = { } | ~
[] \ ; ' : " < > ? , . /

spezielle Präprozessortokens

##

- *Whitespaces*: tragen keine eigene Bedeutung, dienen zur freien Formatierung und Kommentierung von C++ (bzw. zur Trennung von Token, die sonst zusammengefasst werden s.o.)

Leerzeichen, Zeilenenden, horizontale/vertikale Tabulatoren, Seitenenden (form feed) und Kommentare

2 Arten von Kommentaren:

⇒ sog. Blockkommentar (aus C): in `/*` und `*/` eingeschlossener Text, kann über mehrere Zeilen gehen, keine Verschachtelung; häufig für Quelltextidentifikation (Autor, Datum, Änderungshistorie) oder Strukturierung, aber auch zum ‚Ausblenden‘ von Programmteilen (dürfen keine Blockkommentare enthalten !)



```
/*-----
File: xyz.cc
Autor: Mr. X
Last Modifikation: 26.10.95
....
-----*/

/*-----*/
```

⇒ sog. Zeilenkommentar: auf `//` folgender Text bis zum Zeilenende

```
while(1) { .... } // hier passiert dies und das
```

2.3. Ein-/Ausgabe (als blackbox)

- durch Einschluss der Datei `iostream.h` (alt) bzw. `iostream` (neu) per

```
#include <iostream.h>           resp. #include <iostream>
```

wird in einem Programm der C++ -eigene I/O-Mechanismus verfügbar gemacht

- danach sind 3 sog. Streamobjekte benutzbar, die mit den Standardfiledeskriptoren verknüpft sind:

```
std::cout steht für (ggf. umgelenktes) stdout
std::cerr steht für (ggf. umgelenktes) stderr
std::cin steht für (ggf. umgelenktes) stdin
```

- in einen Ausgabestrom kann man mit dem Operator '`<<`' Objekte verschiedener Typen typgerecht ausgeben (zunächst für *built-in* Typen wie erwartet), besser aber i.a. aufwendiger als `printf(...)` a la C
- sog. Modifikatoren, die das Format der Ausgaben steuern, können ebenfalls ein-/ausgegeben werden:

```
std::endl Zeilenende ausgeben
std::flush Ausgabepuffer ausschreiben
std::dec folgende (int-)Zahl im Dezimalformat
std::oct folgende (int-)Zahl im Oktalformat
std::hex folgende (int-)Zahl im Hexadezimalformat ....
```

- Beispiel:

```
....
std::cout<<"Die Summe von "<<i<<" und "<<j<<" ist "<<i+j<<std::endl;
```

```
....
```

- folgt auf eine Ausgabe eine Eingabe, wird automatisch
- `flush` ausgelöst:

```
#include <iostream>
```

```
using namespace std;
```

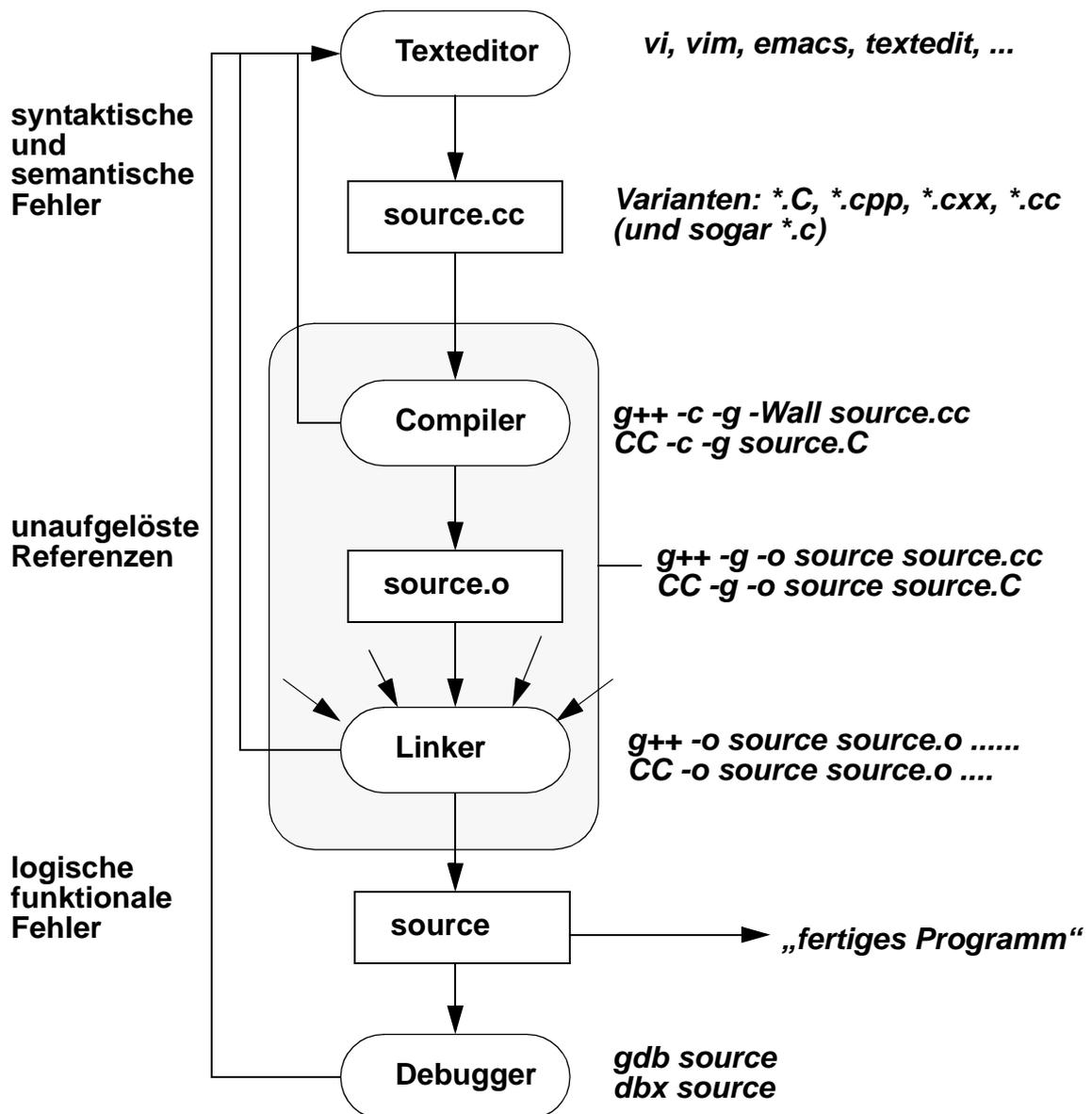
```
int main()
{
    int i;
    cout<<"Bitte geben Sie eine Zahl ein: ";
    cin>>i; // wartet auf Eingabe einer Zahl + <Return>
    cout<<"es war die "<<i<<endl;
}
```

3. Vom Quelltext zum Programm

3.1. {{{edit}, compile}, link}, debug}, run

- Programmierung in jeder Sprache ist iterativer Prozess,
- bei objektorientierten Sprachen ist es üblich, dass der eigentlichen Programmierarbeit eine objektorientierte Analyse-/Designphase vorausgeht, OMT, Booch-Methode ...
- hier: der reine Programmierzyklus im Vordergrund

UNIX-Umgebung (eine Quelle):



Objektorientierte Programmierung in C++

```
spurtefix ahrens 83 ( samples/io ) > g++ -o io -g io.cc
4.240u 2.370s 0:29.52 22.3% 0+661k 288+162io 726pf+0w
spurtefix ahrens 84 ( samples/io ) > ls -l io
-rwxr-xr-x  1 ahrens  simulant  425984 Oct 26 15:58 io*
spurtefix ahrens 85 ( samples/io ) > strip io; ls -l io
-rwxr-xr-x  1 ahrens  simulant  114688 Oct 26 16:00 io*
spurtefix ahrens 86 ( samples/io ) >
spurtefix ahrens 86 ( samples/io ) > CC -o io -g io.C
3.100u 1.440s 0:19.76 22.9% 0+468k 83+90io 320pf+0w
spurtefix ahrens 87 ( samples/io ) > ls -l io
-rwxr-xr-x  1 ahrens  simulant  180224 Oct 26 16:00 io*
spurtefix ahrens 88 ( samples/io ) > strip io; ls -l io
-rwxr-xr-x  1 ahrens  simulant  139264 Oct 26 16:04 io*
spurtefix ahrens 95 ( samples/io ) > ./io
Bitte geben Sie eine Zahl ein: 123456789
es war die 123456789
spurtefix ahrens 96 ( samples/io ) > ./io
Bitte geben Sie eine Zahl ein: 12345678987654
es war die 1942978950
spurtefix ahrens 97 ( samples/io ) > which g++
/usr/local/bin/g++
spurtefix ahrens 98 ( samples/io ) > which CC
/vol/spurtefix-vol3/lang/CC
spurtefix ahrens 101 ( samples/io ) > pwd
/tmp_mnt/vol/harvey-vol3/local/vorlesung/c++/samples/io
spurtefix ahrens 153 ( samples/io ) > g++ -v -o io io.cc
gcc -v -o io io.cc -lg++ -lstdc++
Reading specs from /usr/local/lib/gcc-lib/sparc/2.7.0/specs
gcc version 2.7.0
/usr/local/lib/gcc-lib/sparc/2.7.0/cpp -lang-c++ -v -undef -
D__GNUG__=2 -D__GNUG__=2 -D__cplusplus -D__GNUG_MINOR__=7 -
Dsparc -Dsun -Dunix -D__GCC_NEW_VARARGS__ -D__sparc__ -D__sun__
-D__unix__ -D__GCC_NEW_VARARGS__ -D__sparc -D__sun -D__unix -
Asystem(unix) -Asystem(bsd) -Acpu(sparc) -Amachine(sparc) io.cc
/usr/tmp/cca06624.ii
GNU CPP version 2.7.0 (sparc)
#include "... " search starts here:
#include <...> search starts here:
/usr/local/lib/gcc-lib/sparc/2.7.0/g++-include
/usr/local/include
/usr/local/sparc/include
/usr/local/lib/gcc-lib/sparc/2.7.0/include
/usr/include
End of search list.
/usr/local/lib/gcc-lib/sparc/2.7.0/cclplus /usr/tmp/
cca06624.ii -quiet -dumpbase io.cc -version -o /usr/tmp/
cca06624.s
GNU C++ version 2.7.0 (sparc) compiled by CC.
as -o /usr/tmp/cca066241.o /usr/tmp/cca06624.s
/usr/local/lib/gcc-lib/sparc/2.7.0/ld -e start -dc -dp -o io /
```

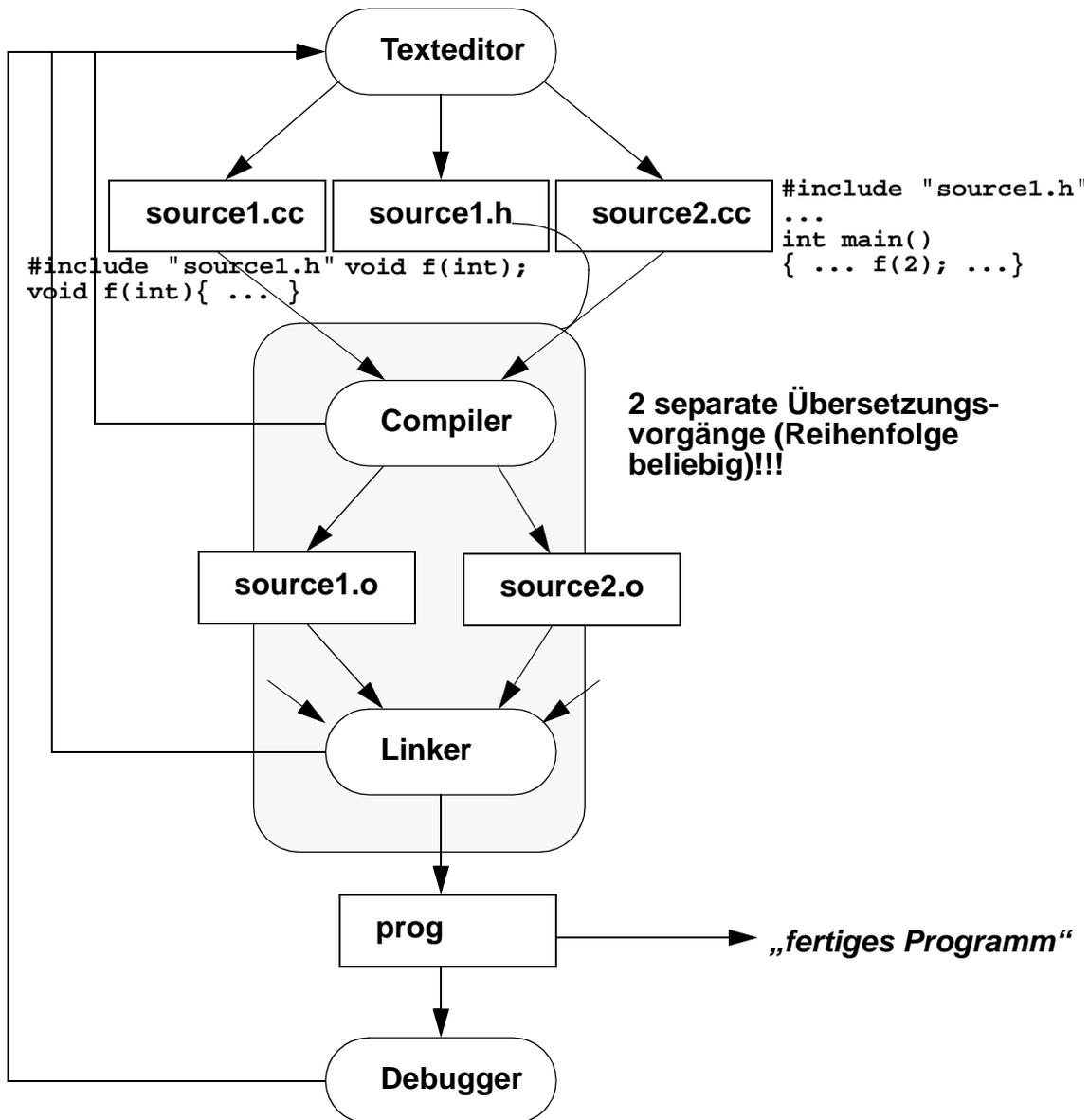
Objektorientierte Programmierung in C++

```
lib/crt0.o -L/usr/local/lib/gcc-lib/sparc/2.7.0 -L/usr/local/
lib /usr/tmp/cca066241.o -lg++ -lstdc++ -lgcc -lc -lgcc
3.420u 1.670s 0:11.76 43.2% 0+614k 9+161io 10pf+0w
spurtefix ahrens 154 ( samples/io ) > CC -v -o io io.C
mkdir /tmp/CC.06639.1.d
Creating /tmp/CC.06639.1.d/option
/vol/spurtefix-vol3/lang/SC2.0.1patch/acpp -C -B -
D__cplusplus=1 -Dunix -Dsun -Xa -D_NO_LONGLONG=1 -Dsparc -I/
vol/spurtefix-vol3/lang/SC2.0.1patch/include/CC_413 -I/vol/
spurtefix-vol3/lang/SC2.0.1patch/include/cc_413 io.C >/tmp/
acpp.06639.0.i
/vol/spurtefix-vol3/lang/SC2.0.1patch/cfront +fio.C +T/tmp/
CC.06639.1.d/dirinst +s +a1 +t/tmp/CC.06639.1.d/ptcf </tmp/
acpp.06639.0.i >/tmp/cfront.06639.1.c
rm /tmp/acpp.06639.0.i
grep '^[tf]' /tmp/CC.06639.1.d/ptcf >/dev/null 2>&1
/vol/spurtefix-vol3/lang/SC2.0.1patch/acomp -Qy -GG -cg87 -Xa -
Dunix -Dsun -Dsparc -xC -I/vol/spurtefix-vol3/lang/
SC2.0.1patch/include/cc_413 -i /tmp/cfront.06639.1.c -o /tmp/
acomp.06639.2.s
rm /tmp/cfront.06639.1.c
/vol/spurtefix-vol3/lang/SC2.0.1patch/as -o io.o -Q -cg87 /tmp/
acomp.06639.2.s
rm /tmp/acomp.06639.2.s
### CC: Note: LD_LIBRARY_PATH = /usr/local/X11/lib
### CC: Note: LD_OPTIONS = (null)
/bin/ld -dc -dp -e start -X -o io /vol/spurtefix-vol3/lang/
SC2.0.1patch/crt0.o /vol/spurtefix-vol3/lang/SC2.0.1patch/cg87/
_crt1.o /vol/spurtefix-vol3/lang/SC2.0.1patch/values-Xa.o -L/
vol/spurtefix-vol3/lang/SC2.0.1patch/cg87 -L/vol/spurtefix-
vol3/lang/SC2.0.1patch -u _fix_libc_ io.o -lm -lansi -lC -lc
>&/tmp/ld.06639.4.err
/vol/spurtefix-vol3/lang/SC2.0.1patch/c++filt </tmp/
ld.06639.4.err
rm /tmp/ld.06639.4.err
/vol/spurtefix-vol3/lang/SC2.0.1patch/patch io
rm io.o
rm -rf /tmp/CC.06639.1.d
```

3.2. separate Übersetzung

- „richtige“ Programme bestehen aus mehr als einem Quelltext: Handhabbarkeit, Überschaubarkeit, Teamwork, Rekompilationsaufwand
- erfordert Schnittstellenbeschreibungen: in C++ (C) nicht per IMPORT/EXPORT-Mechanismus, sondern durch sog. Headerfiles

UNIX-Umgebung (zwei (n) Quellen):



es lohnt sich bereits hier, die Abhängigkeiten in einem ‚Makefile‘ festzuhalten:

```
prog: source1.o source2.o
    g++ -o prog source1.o source2.o
source1/2.o:source1.h source1/2.cc
    g++ -c source1/2.cc
```

(zwei Regeln !)

Programmierungsumgebungen

- Unterstützung der Programmerstellung, Fehlersuche und Wartung über einfache “Kommandozeilen”-Werkzeuge hinaus
- schließen oft C++ syntaxorientierte Editoren ein, automatischer Aufruf notwendiger Übersetzungsschritte mit und ohne makefile-Unterstützung

Objektorientierte Programmierung in C++

- Beispiele: SPARCWorks C++, HP Softbench, ... SNIFF+ (Unix); Borland C++, Visual C++ (MS) (DOS, Windows)
- SNIFF+3.x: auf Basis der C++-Klassenbibliothek ET++ entwickelte Programmierumgebung, enthält keinen eigenen Compiler, jedoch einen Parser, der mit der Syntax von C++ vertraut ist und auch fehlerhafte Quelltexte analysieren kann, enthält neben Editor auch Crossreferenzer, sog. Retriever, generische Anbindung an Versionskontrolltools (RCS, SCCS, ...) und weitere Komponenten
- man braucht:

```
setenv SNIFF_DIR /vol/knecht-vol2/SNIFF+
setenv PATH $SNIFF_DIR/bin:${PATH}
setenv LM_LICENSE_FILE /vol/knecht-vol2/license.dat
```

- eine knappe Einführung in die Arbeit mit SNIFF ist verfügbar unter:
ftp://ftp/pub/local/vorlesung/c++/sniffing



4. Elementares C++

4.1. Präprozessor

- erster Schritt jeder C++ -Übersetzung noch vor eigentlichem ersten Pass des Compilers --> reine Textersetzung, (oft) unabhängig von C++-Syntax und -Semantik (u.U. unerwartete Effekte bei der anschließenden Übersetzung) s.u.
- /lib/cpp oder /usr/local/bin/cpp ...
- moderne C++ -Kompiler benutzen Präprozessoren, die C++ 'kennen'
- cpp wird vom Compiler implizit gerufen
- 1. und hauptsächliche Anwendung in C++: Einschluss von Headerfiles:

```
#include <iostream>
// quelltextmäßiger Einschluss der Datei iostream aus einem
// dem Compiler bekannten Systemverzeichnis (wegen <...>)
// ggf. gleichwertige Aktion (Repository o.ä.)
// z.B. /usr/local/lib/g++-include beim g++
```

```
#include "myheader.h"
// quelltextmäßiger Einschluss der Datei myheader.h aus dem
// aktuellen Verzeichnis oder einem, dem Compiler mittels
// Option -IPfad benannten (wegen "...")
```

- cpp erzeugt im Outputfile sog. line-Informationen: Zuordnung zu Originalquelltextzeilen nach Einschluss --> Fehlermitteilungen des Compilers beziehen sich auf den ursprünglichen Quelltext (#n bei g++, #line n)
- Viele Compiler kann man mit der Option **-E** veranlassen, lediglich cpp aufzurufen, Ausgabe ist dann **stdout**
- 2. Kategorie von Präprozessoranweisungen: sog. Makrodefinitionen **#define**, sollte man in C++ nicht verwenden, um Konstanten bzw. sog. Makrofunktionen zu erzeugen, Benutzung reduziert sich im wesentlichen auf bedingte Übersetzung (s.u.) Style: Makronamen in Großbuchstaben

Objektorientierte Programmierung in C++

```
#define MAX 100
// besser ist:
const int MAX=100;

#define INC(x) (++(x))
// besser ist:
inline int& inc(int x){return ++x;}
//Vorsicht, Falle bei C-Preprozessoren
#define m(x)    irgendwas_mit(x)    // macht dies und das
....
y = m(z) + 1;
  |
  v
y = irgendwas_mit(z) // macht dies und das + 1;
```

- 3. Kategorie von Präprozessoranweisungen: sog. bedingte Übersetzungsanweisungen **#ifdef**, **#ifndef**, **#if**, **#else**, **#elif**, **#endif**, steuern den Ein- oder Ausschluss von Programmteilen bei der Übersetzung

```
#ifdef VARIANTE1
.....
#else
#if defined (VARIANTE2 ||VARIANTE3)
.....
#endif
#endif
```

- Steuerung der Auswahl entweder im Quelltext (vor **#if**)

```
#define VARIANTE2
```

oder bei der Übersetzung

```
CC -DVARIANTE2 -c xyz.C
```

- es gibt eine Reihe vordefinierter Makros zur Unterscheidung verschiedener Compiler: g++ definiert implizit **__GNUG__**, Borland C++ definiert implizit **__BCPLUSPLUS__**, Sun CC definiert implizit **__SUNPRO_CC**
- bei jeder C++ -Übersetzung ist implizit **__cplusplus** definiert, dadurch lassen sich z.B. (System-) Headerfiles für gemeinsame Benutzung im Kontext C und C++ konditionalisieren

4.2. *Built-in* Typen

- **int**

ganze Zahlen mit Vorzeichen in Wortbreite des Prozessors, Varianten: **unsigned int** - kein Vorzeichen, **long int** - u.U. größerer Bereich, **short int** u.U. kleinerer Bereich (in Zusammensetzungen kann man **int** weglassen)
übliche Arithmetik (+, -, *, / {ganzahlige Division}, % {Divisionsrest}), Überläufe ungeprüft !

bitweise Operationen:

| bitweise ODER: `12 | 7 == 15`

^ bitweise excl. ODER: `12 ^ 7 == 11`

& bitweise UND: `12 & 7 == 4`

<< bitweise Left-Shift: ($\cdot 2^n$) "vorn fallen Bits raus"

>> bitweise Right-Shift: ($/ 2^n$) "vorn mit Vorzeichen auffüllen"!

für Ausdrücke der Form **x = x op n**, (op: *, /, %, +, -, <<, >>, &, |, ^)
gibt es die kompaktere Notation **x op= n**

Vergleiche (<, >, <=, >=, == !=) liefern 0 oder 1 (als **bool** oder **int** benutzbar)
z.B.

`... while (1) ...`

- **char**

mit ASCII-Zeichensatz identifizierter Teilbereich von **int** (ein Byte)

- **bool**

zwei Werte: **true** und **false**

logische Operatoren:

|| logisches ODER: `12 || 7 == true (1)`

&& logisches UND: `12 && 7 == true (1)`

! logische Negation: `!12 == false (0)`

- **float, double, long double**

reelle Werte unterschiedlicher Genauigkeit, übliche Arithmetik (+, -, *, /), Überläufe werden u.U. als Fehler behandelt (nicht bei IEEE -konformer Arithmetik)

- zwischen den *built-in* Typen existiert eine Reihe von impliziten Typkonvertierungen: Umwandlung von **char**, **short int** nach **int** (werterhaltend); Umwandlung von **int** nach **double/float** und umgekehrt (Truncation)

folgende arithmetische Umwandlungen werden automatisch vorgenommen:

- ist einer der Operanden **long double**, wird auch der andere in **long double** umgewandelt, ansonsten

- ist einer der Operanden **double**, wird auch der andere in **double** umgewandelt, ansonsten

Objektorientierte Programmierung in C++

- ist einer der Operanden `float`, wird auch der andere in `float` umgewandelt, ansonsten
 - ist einer der Operanden `unsigned long`, wird auch der andere in `unsigned long` umgewandelt, ansonsten
 - ist einer der Operanden `long`, wird auch der andere in `long` umgewandelt, ansonsten
 - ist einer der Operanden `unsigned`, wird auch der andere in `unsigned` umgewandelt, ansonsten sind beide `int`
- Variablen(Objekt-)definitionen: `T v;` ist eine Anweisung und führt die Variable (das Objekt) `v` vom Typ `T` in das Programm ein, mehrere Variablen (Objekte) gleichen Types können auch per
`T v1, v2, v3;`
definiert werden, Variablen (Objekte) können mit ihrer Definition auch mit initialen Werten belegt werden
`int eins=1, zwei=2;`
 - Man unterscheidet (Objekt-) *Deklaration* und (Objekt-) *Definition*:

Deklarationen charakterisieren Objekte, die (u.U.) an anderer Stelle angelegt sind Definitionen legen die Objekte im Speicher an, jede Definition ist auch eine Deklaration, in einem Programm (bestehend aus mehreren Quelltexten) muss es für jedes Objekt (in einem Gültigkeitsbereich s.u.) genau eine Definition geben, es kann beliebig viele Deklarationen geben:

```
// source1.cc:           // source2.cc:
int i;                  extern int i; // dasselbe i

(g++ -o prog source1.cc source2.cc)
```

der Kontext einer Definition entscheidet über die 'Lebensdauer' und den Gültigkeitsbereich von Objekten, man unterscheidet:

- *globale Objekte*: Definition außerhalb eines Blockes (`{ ...}`), werden vom Compiler in einem globalen Datensegment angelegt und existieren während der gesamten Programmlaufzeit
 - *lokale Objekte*: Definition im Kontext eines Blockes, werden erst zur Laufzeit beim Betreten des entsprechenden Blockes angelegt (auf dem sog. Laufzeit-Stack) existieren nur bis zur Beendigung der Abarbeitung des Blockes
- Blöcke können geschachtelt sein, jede Verwendung eines Objektnames in einem Programm bezieht sich auf eine Definition eines Objektes mit diesem Namen im kleinsten umgebenden Block, dieser ist der *Gültigkeitsbereich* des Objektes Redefinitionen von Objektnamen in lokalen Blöcken überdecken die Sichtbarkeit gleichnamiger Objekte aus umfassenden Blöcken, auf globale Objekte kann aus jedem Block mit dem *scope-resolution* Operator `::` zugegriffen werden:

Objektorientierte Programmierung in C++

```
#include <iostream>

int i=1;

void p(int i) { cout << i; } // Parameter verhalten sich wie
// lokale Objekte, die beim Aufruf der Funktion mit dem
// aktuellen Parameterwert initialisiert werden !

int main()
{
    p(i);
    int i=2;
    {
        p(i);
        int i=3;
        p(i);
        p(::i); // scope resolution !
    }
    p(i);
}
// Ausgabe: 12312
```

bei der Deklaration kann einer Variablen ein sog. *storage-class-specifier* zugeordnet werden:

```
storage-class-specifier:
    "auto"          |
    "register"      |
    "static"        |
    "extern"        |
    "mutable"      .
```

auto - sog. *automatic* Variablen nur innerhalb eines Blockes erlaubt und dann auch *default*-Einstellung (kann also weggelassen werden)

register - nur für int- (kompatible) lokale Variablen in Funktionen, sog. Compiler-Hint: Hinweis, diese Variable wenn möglich in ein Register zu legen (schnellerer Zugriff z.B. bei Schleifenzählern innerer Zyklen)

static - außerhalb jeder Funktion: die Variable ist global und extern sichtbar; innerhalb einer Funktion dito mit der Konsequenz, dass static Variablen nur in der entsprechenden Funktion sichtbar sind, den Funktionsaufruf aber „überdauern“ (sich global verhalten)

extern - Bezugnahme auf ein globales Objekt in einem anderen Modul

mutable - nur für Memberdaten von Klassen (☞ 5.1.)

- **void**

der „leere“ Typ, hat keine Werte, darf nur als Funktionsrückgabety (→ kein Wert wird übergeben), als (einziger) Parametertyp (→ keine Argumente) oder als Zeigertyp **void*** (generischer Zeiger auf beliebige Objekte) verwendet werden

4.3. einfache Typkonstrukte

- aus den *built-in* Typen (und nutzerdefinierten Typen) lassen sich durch einfache Strukturierungsmittel neue (nutzerdefinierte) Typen erzeugen: *Aufzählungstypen*, *Feldtypen*, *Zeigertypen*, *Referenztypen*, *Konstantentypen*, *Funktionszeigertypen*, *Memberzeigertypen* (♦ 5.6.), *Strukturtypen*, *Uniontypen* und *Klassentypen* (♦ 5.1.)
- *Aufzählungstypen*:
Typdefinition durch Angabe der unterschiedlichen symbolischen Werte, Typnamen können wie Built-in Typen verwendet werden:

```
enum Colour {red, green, blue};
enum io_state { goodbit=0, eofbit=1, failbit=2, badbit=4,
               hardfail=0200}; // aus iostream
....
Colour col=blue;
io_state last_io_result;
```

- *Feldtypen (Arrays)*:
Variablenvereinbarung der Form $\tau \ \mathbf{f} \ [\mathbf{n}]$; legt einen Vektor von τ -Objekten der Länge n an (n muss ein *konstanter Ausdruck*¹ sein), \mathbf{f} wird mit einem Verweis (Zeiger) auf den Beginn des Vektors identifiziert, die einzelnen Elemente erreicht man durch Indizierung $\mathbf{f}[0]$, ... $\mathbf{f}[\mathbf{n}-1]$ (Indizes fangen immer bei 0 an!), es gibt keine Tests auf Zugriffe außerhalb des Vektors !!! mit den Mitteln von C++ lassen sich jedoch leicht und elegant Vektor-Klassen mit entsprechenden Tests implementieren. Mehrdimensionale Felder $\tau \ \mathbf{f} \ [\mathbf{m}][\mathbf{n}]$; -> $\mathbf{m} \ \tau[\mathbf{n}]$ - Vektoren Die (erste) Dimension kann auch weggelassen werden. Dann handelt es sich um eine Variable die auf einen Vektor des Grundtyps τ ($\tau[\mathbf{n}]$) zeigen kann (die entsprechende Speicherfläche muss dann anderweitig bereitgestellt werden [Initialisierung oder Parameterübergabe!!])

```
// z.B.
char string[]="Das ist ein String";
int matrix[][3]={{1,2,3}, {4,5,6}, {7,8,9}, {10,11,12}};
```

Anders als in Sprachen der „Pascal-Familie“ werden Typen in C++ oft nicht benannt (insbesondere bei Feld-, Zeiger-, Referenz-, Konstanten- und Funktionstypen), sondern bei der Vereinbarung von Variablen explizit konstruiert. Es gibt jedoch die Möglichkeit, auch derartige Typen mit expliziten Namen zu versehen:

```
int field [100];
// entspricht:
typedef int Fieldtype [100]; // mit typedef wird ein Typname
                             // vereinbart
Fieldtype field;
```

1. der Wert eines *konstanten Ausdrucks* muss zur Übersetzungszeit (statisch) bestimmbar sein !
(**const** -Eigenschaft s.u. reicht dazu i.allg. nicht aus)



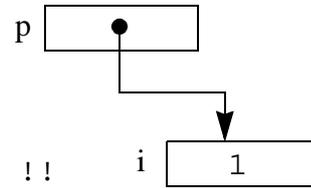
Objektorientierte Programmierung in C++

- *Zeigertypen:*

Variablenvereinbarung der Form $\tau *p$; definiert einen Zeiger, der Objekte des Typs τ verweisen kann, der Inhalt einer Zeigervariablen ist die Adresse des Objektes, der Inhalt von p aus obiger Definition ist zunächst undefiniert, bestenfalls 0, falls p global ist, Vorsicht bei nicht initialisierten Zeigern !!! Zugriff auf das referenzierte Objekt durch *Dereferenzierung* des Zeigers $*p$



```
// z.B.
int *p;
int i=1;
....
p= & i;      // der Adressoperator
(*p)++      // erhoeht den Wert in i !!
```



Mit Zeigern $\tau *p, *p1, *p2$; sind folgende arithmetischen Operationen erlaubt: (Achtung: $\tau * p1, p2, p3$ vereinbart einen Zeiger und zwei int 's)



$p + n$ (n mit int kompatibel): liefert den um n T-Objekte verschobenen Zeiger
 $p - n == p + (-n)$

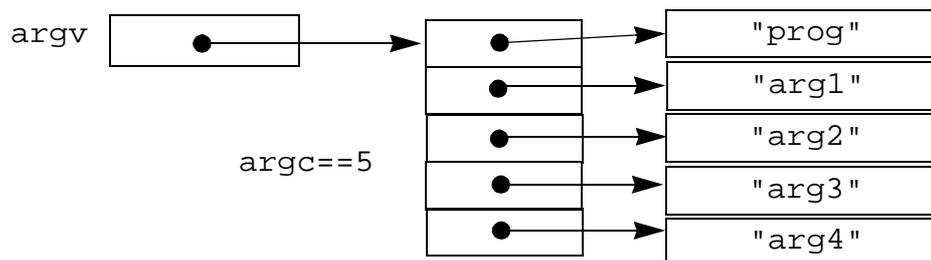
$p1 - p2$ liefert Anzahl der T-Objekte zwischen den Zeigern
 -> Arithmetik modulo($\text{sizeof}(\tau)$)

Feldvariablen sind nichts anders als konstante Zeiger:

$f[i]$ wird vom Compiler umgesetzt in $*(f + i)$, auch Zeiger können indiziert werden, allerdings ist einem Zeiger nicht anzusehen, ob er auf ein oder n Elemente des Grundtyps zeigt! ($f [i] = *(f + i) = *(i + f) = i [f] !!!$)

// 1. Beispiel: Parameterübergabe an jedes Hauptprogramm aus
 // der Aufrufumgebung (shell):

```
$ prog arg1 arg2 arg3 arg4
```



```
int main (int argc, char **argv) // oder auch ... char *argv[]
{
    argv[0] - der Programmname als String
    argv[i] - das i-te Programmargument als String
}
```

Objektorientierte Programmierung in C++

```
// 2. Beispiel: strcpy
void strcpy (const char* source, char* dest)
{ while (*dest++=*source++); }
```

- Zeiger bilden die Grundlage für *dynamische Objekte*: Neben lokalen Objekten (nur während des Aufrufs einer Funktion existent) und globalen Objekten (während der gesamten Programmlaufzeit existent) gibt es eine dritte Kategorie von Objekten deren Entstehung und Vernichtung explizit durch Operatoren gesteuert wird:

```
T *p = new T;
// reserviere dynamisch Speicher für ein T-Objekt
// und lasse p auf dieses verweisen
```

derartig erzeugte Objekte sind gleichberechtigt mit allen anderen (außer, das sie keinen Namen haben, sondern nur über Zeiger/Referenzen angesprochen werden können) T darf kein Referenztyp (s.u.) sein, man kann jedoch auf dynamische Objekte Referenzen setzen

```
int &ri= *new int; // ri ist jetzt ein Alias für das
                // dynamisch erzeugte Objekt
```

dynamische Objekte existieren solange, bis

```
delete p;
```

ausgeführt wird, dann wird der Speicherbereich zur Wiederverwendung freigegeben, `delete` ist nur auf Zeigern definiert, die mittels `new` initialisiert wurden



```
int i;
int *p=&i; delete p; // Fehler
p=new int [10];
p++; delete [p]; // Fehler
```

`new` und `delete` basieren auf `malloc` / `free`, in C++ sollte man nur `new` / `delete` verwenden !

wird die einzige Referenz auf ein dynamisches Objekt überschrieben, gibt es keine Möglichkeit, die entsprechende Speicherfläche später zurückzugeben: *memory leak* Memory leaks sind „tödlich“ in „long-running-systems“



```
int *p= new int; p=new int;
// das erste int-Objekt ist "verloren"
```

es existieren Varianten von `new` / `delete`, die zugleich einen ganzen Vektor von T-Objekten allokiert / deallokiert:

```
double *v=new double [100];
....
delete[] v; // dem Zeiger ist nicht "anzusehen", ob er
            // auf ein oder n Objekte verweist
```

Objektorientierte Programmierung in C++

falls nicht genügend dynamischer Speicherplatz verfügbar ist, erzeugt `new` eine sog. exception `bad_alloc` Wert 0, es gibt auch eine Variante von `new` mit sog. *placement syntax*, die bei Speicherknappheit 0 als Ergebnis liefert:

```
new (nothrow) XXX;
```

```
delete p; mit p==0 ist zulässig (mehr zu new / delete ♦ 8.2.)
```

- *Referenztypen:*

Variablenvereinbarungen der Form `T &r = tObject;` (mit einem deklarierten `T` `tObject;`) definieren eine Referenz, die einen Aliasnamen für ein anderes Objekt repräsentiert, Referenzen müssen (anders als Zeiger) stets initialisiert werden (ggf. beim Aufruf einer Funktion für Parameter von einem Referenztyp), Bezugnamen auf das Objekt über `tObject` und `r` sind gleichberechtigt.

Motivation (♦ auch 5.3. Copy-Konstruktoren): Beim Aufruf einer Funktion mit einem Parameter des Typs `T` wird eine Kopie des aktuellen Parameters an den formalen Parameter übertragen (*call by value*):

```
void swap (int i, int j) {
    int tmp=i;
    i=j;
    j=tmp;
}
```

hat nicht den beabsichtigten Effekt, weil nur lokale Kopien vertauscht werden, die ursprünglich übergebenen Objekte bleiben unverändert; man müsste stattdessen die Objekte indirekt über ihre Adressen der Funktion zur Verfügung stellen, dann beziehen sich die Kopien der Adressen auf die Originalobjekte:

```
void swap (int *i, int *j) {
    int tmp=*i;
    *i=*j;
    *j=tmp;
}
```

die Verwendung von Zeigern ist unelegant, man muss insbesondere beim Aufruf Adressen übergeben: `swap (&a, &b);` Referenztypen ermöglichen einfachere Umsetzung:

```
void swap (int &i, int &j) {
    int tmp=i;
    i=j;
    j=tmp;
}
```

Aufruf: `swap (a, b);` für `a` und `b` werden dynamisch die Aliasnamen `i` und `j` eingeführt, -> *call by reference*

Achtung: Funktionen mit Referenzparametern operieren immer auf Originalobjekten, u.U. soll aber garantiert werden, dass diese unverändert bleiben, dann kann man solche als `const T&` deklarieren !

Achtung: Auch Rückgabetypen von Funktionen können Referenztypen sein, dann muss sichergestellt werden, dass ein gültiges Objekt zurückgegeben wird!

Objektorientierte Programmierung in C++

```
// Danger:
#include <iostream>
using namespace std;

int & retref() {
    int i = 0; // lokal, nur waehrend des Aufrufs gueltig
    int &ri=i; // lokale Referenz
    return ri;
}

void any(){
    int i[]={12,13,14,15};
}

int main(){
    int &null=retref();
    cout<<null<<endl;
    null++;
    any();
    cout<<null<<endl;
}
```

```
$ g++ -Wall -o x refret.cc
refret.cc: In function `void any()`:
refret.cc:10: warning: unused variable `int i[4]`
3.450u 2.160s 0:26.39 21.2% 0+656k 262+162io 697pf+0w
$ ./x
0
15
$ CC -o x refret.cc
0.920u 1.130s 0:14.83 13.8% 0+392k 50+70io 209pf+0w
$ ./x
0
14
```

leider keine Warnung !!!

Referenzen sind „verkaptete Zeiger“, d.h. solche, die auf Quelltextebene nicht in Erscheinung treten (vgl. **VAR** Argumente in Pascal/Modula)

- *Konstantentypen:*

mit dem *type specifier* **const** kann man Typen konstruieren, deren Objekte unveränderlich sind, der Compiler garantiert, dass an **const** Objekten keine Veränderungen vorgenommen werden

const trägt zu Erhöhung der Sicherheit von Programmen bei, man sollte **const** benutzen, wo immer es sinnvoll erscheint

Konstante Objekte müssen bei Ihrer Definition initialisiert werden (eine spätere Zuweisung ist nicht möglich)



Objektorientierte Programmierung in C++

```
const double pi = 3.1415926;
pi = 5; // error: assignment to const type ::pi
```

C++ Konstanten lösen `#define`-Konstanten (fast) vollständig ab :

```
const int max=100; // statt #define MAX 100
```

Globale Konstanten haben kein external linkage (anders als in ANSI C), müssen also über Modulgrenzen `extern` bereitgestellt werden

```
// Modul1:
extern const double e = 2.7182818;

// Modul2:
extern const double e;
```

Bei Zeigern kann die Konstantheit des Zeigers und des referenzierten Objektes separat ausgezeichnet werden

```
const char *c1="XXX";// Zeiger auf einen konstanten String
char *const c2="YYY";// konstanter Zeiger auf einen String
const char *const c3 ="ZZZ";// konstanter Zeiger auf einen
//konstanten String
```

```
const int n = 1234;
const int & ri1 = n; //Referenz auf eine int-Konstante
int & const ri2 =n; // refs sind von Natur aus const !!
```

Konstante Objekte werden u.U. vom Compiler in einem *read-only segment* (wie der Programmcode) plaziert

konstante Objekte können auch über Zeiger nicht verändert werden, weil die Adresse einer `const T` Variablen vom Typ `const T*` ist

Bei Zeigern auf konstante Objekte erstreckt sich der Schreibschutz auf alle Feld-elemente die über diesen Zeiger erreichbar sind

```
c1=new char[20]; // ok c1 ist nicht konstant
c2=new char[20]; // Fehler: c2 ist konstant

c1[1]='A'; // Fehler: c1 zeigt auf konstanten String
c2[1]='A'; // ok
c1=c2; // ok
c2=c1; // Fehler: c2 ist konstant
```

Parameter von Funktionen können als `const` spezifiziert werden, dies ergibt zugleich:

- a) die Möglichkeit, auch ein konstantes Objekt anstelle des Parameters an die Funktion zu übergeben und

Objektorientierte Programmierung in C++

b) die Zusicherung, dass der Körper der Funktion dieses Objekt unverändert belässt

häufige Verwendung um trotz *call by reference* die Unveränderlichkeit von Parameterobjekten zu garantieren

```
void func (const very_large_object & r) {...}
// braucht nicht kopiert werden und ist trotzdem read-only
```



die Übergabe eines konstanten Objektes an einen nichtkonstanten Referenzparameter ist nicht erlaubt:

```
#include <iostream>

void foo(int& i){i=2;}
void bar(const int& r){}

int main ()
{
    const int i=1;
    // foo(i); nicht erlaubt !
    // foo(1); nicht erlaubt !
    bar(i);
    bar(1);

    const double& d = 2;
    // temporary created !
}
```

auch die Übergabe (der Adresse) eines konstanten Objektes an einen nichtkonstanten Zeigerparameter ist nicht erlaubt:

```
void f(int* i){*i=2;}
....
const int i=1;
f(&i);
CC: error: bad argument 1 type for f(): const int * ( int * expected)
```

```
void f(const int* i){*i=2;}
CC: error: assignment to const type
```

weitere Anwendungen von `const` ↪ 5.1. (*const member functions*)

- *Funktionszeigertypen:*
a la „Prozedurtypen“ aus Pascal-ähnlichen Sprachen, Werte der Typs sind Funktionen mit einheitlicher Signatur (Rückgabetyt und Parametertypen), Definition von Variablen meist ins Typkonstrukt „eingeschachtelt“,

```
double Sin (double); // die Signatur einer Sinusfkt.
double (*math_fkt) (double); // ein Funktionszeiger, passt auf
// alle Funktionen double -> double
```

Objektorientierte Programmierung in C++

```
math_fkt=Sin;           // oder auch &Sin
x>(*math_fkt)(0.5);     // Aufruf
x=math_fkt (0.5);      // ebenfalls moeglich !!!
```

„infix“-Notation u.U. schwer zu lesen, ggf. Einführung geeigneter **typedef**'s

```
// z.B. UNIX-Standardfunktion signal:
// Einrichtung eines Signalhandlers
void (*signal (int signo, void (*func)(int)))(int); // ??????
// dasselbe wie:
typedef void Sigfunc (int);
Sigfunc *signal (int, Sigfunc *);
```

- *Strukturtypen:*
a la „RECORD -Typen“ aus Pascal-ähnlichen Sprachen: Komponenten verschiedener Typen werden zu einem neuen umfassenden Typ kombiniert, in C++ hauptsächlich wegen der angestrebten Abwärtskompatibilität zu C enthalten, (♦ 5.1. objektorientierte Aufwertung von Strukturen), hier zunächst Strukturen a la C:

```
struct Person {
    char *name;
    int age;
    double salary;
    long phone_no;
} p;
```

p

name	→
age		
salary		
phone_no		

Person ist ab hier ein „first class“ Typname, anders als in C dort wird erst die Rekombination **struct Person** zum Typnamen! daher findet man dort haeufig:

```
struct _Dummy_name_ { ..... };
typedef struct _Dummy_name_ Typname;
Typname var;
// auch in C++ auch moeglich, aber ueberfluessig !
```

Initialisierung von Strukturvariablen ist möglich (aber eher unüblich in C++, ♦ 5.3.):

```
Person willibald = {"Willibald Wusel", 44, 3333.00, 12345678 };
```

Zugriff auf Komponenten mittels **strukturvariable.komponentenname**

```
void raise_salary (Person &p, int percentage)
{
    p.salary*=1 + percentage/100.0;
}
```

Die Größe eines Objektes eines Strukturtyps (**sizeof(Person)**) hängt sowohl von den Größen der Komponententypen, als auch von Ausrichtungsvorgaben der Prozessorarchitektur (*alignment*) ab !



Objektorientierte Programmierung in C++

Strukturen können auch benutzt werden, um sog. *Bit-Felder* zu vereinbaren, dazu kann auf Komponenten ganzzahliger Typen eine Längenangabe erfolgen, diese legt fest, wieviele Bits zur Repräsentation dieser Komponenten zu verwenden sind (extrem hardwareabhängig), Zeiger auf Bit-Felder gibt es nicht:



```
struct Bits {  
    int b1: 24;  
    int b2: 16;  
    int b3: 24;  
};
```

Werden Strukturobjekte über Zeiger referenziert, so gibt es eine handliche und anschauliche Notation (->) für den Zugriff auf Komponenten:

```
Person *anybody = &willibald;  
  
cout << "Sein Name ist " << anybody->name << endl;  
    // anstelle von: (*anybody).name
```

- *Uniontypen:*
syntaktisch wie Strukturtypen aufgebaut, die Komponenten werden jedoch nicht „nacheinander“ im Speicher plaziert, sondern „übereinander“, man sieht ein und dasselbe „Stück Speicher“ unterschiedlich an, damit lassen sich z.B. variable Teile von Strukturen a la RECORD... CASE ... aufbauen, in C++ gibt es für derartige Situationen weit bessere Ausdrucksmittel (♦ 6.1.), es bleiben seltene Verwendungen in hardwarenahen Bereichen

BEISPIEL: die Darstellung von double-Zahlen schliesst nach IEEE-Standard auch den Wert `NaN` (not a number) ein, dieser steht u.a. für $+\infty$, $-\infty$ und kann z.B. auf SUN Sparc bei reeller Arithmetik durchaus entstehen:

```
double d=2.0;  
while (1)  
{  
    cout<<d<<endl;  
    d*=d;  
}
```



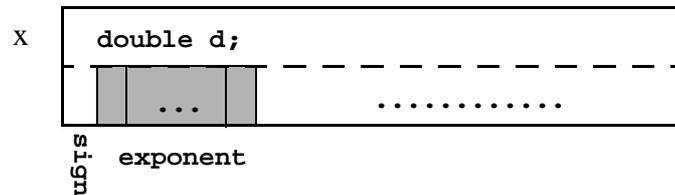
```
2  
4  
16  
256  
65536  
4.29497e+09  
1.84467e+19  
3.40282e+38  
1.15792e+77  
1.34078e+154  
Infinity  
Infinity  
.....
```

Gesucht ist eine Testfunktion, die entscheidet, ob eine `double` Variable `NaN` ist (bei `NaN` sind alle Exponentenbits = 1, Bit 0: Vorzeichen, Bits 1..11 Exponent)

Folgende Lösung benutzt `union`'s:

Objektorientierte Programmierung in C++

```
union HACK { double d; struct { unsigned :1, e:11; } s; };  
  
int NaN(double x) { HACK h; h.d=x;  
                    return h.s.e == 0x7ff;  
}
```



4.4. Ausdrücke

- hier keine vollständige Wiedergabe der Syntaxkonstrukte zur Bildung von Ausdrücken (ca. 3 Seiten C++ -Grammatik) sondern Beschränkung auf bislang unerwähnte Konzepte
- alle Operatoranwendungen bilden Ausdrücke (Vorrangregeln ♦ 2.2.), einstellige Operatoren (*, &, +, -, !, ~) -> *unäre Ausdrücke*; zweistellige Operatoren (fast alle anderen) -> *binäre Ausdrücke*
- die Reihenfolge der Auswertung von Operandenausdrücken ist **nicht definiert** !

```
int i=2;  
cout << i-- * i++; // g++: 2 , CC: 4
```

- einzige Ausnahme: logische Ausdrücke werden von links nach rechts berechnet: log. ODER (||) liefert 1 (**true**) sobald der erste Teilausdruck 1 (**true**) liefert, log. UND (&&) liefert 0 (**false**) sobald der erste Teilausdruck 0 (**false**) liefert danach bricht die weitere Berechnung von Teilausdrücken ab

```
Person *p;  
if (p && p->age > 18) ...  
// falls p==0, ist p->age nicht definiert
```

- Ausdrücke bilden Anweisungen :

expression-statement ::= [expression] ";"

- jeder Ausdruck liefert einen Wert (ggf. den leeren Wert bei Funktionen mit Rückgabotyp **void**), ein nicht-leerer Wert kann, muss aber nicht weiterverwendet werden

```
int f(int);  
....  
f(3); // Ergebnis wird ignoriert  
int k=f(4); // Ergebnis wird weiterverwendet
```

- auch alle Formen der Zuweisung (=, **op=**) sind Ausdrücke, der Wert einer Zuweisung ist der zugewiesene Wert -> Zuweisung = (nicht **op=**) ist kaskadierbar:

```
a=b=c=1 ----> a=(b=(c=1))
```

Objektorientierte Programmierung in C++

- ein weiterer spezieller zweistelliger Operator ist der *Kommaoperator*: mehrere Ausdrücke bilden durch “,” getrennt einen neuen Ausdruck, der Wert des Gesamtausdrucks ist der Wert des letzten Ausdrucks (nicht zu verwechseln mit dem Komma als Trennzeichen für Funktionsargumente!)

```
f((i=11, i*i)) ---> f(121)
f(i=11, i*i ) ---> f(11, 121)
```

- einziger „dreistelliger“ Ausdruck: *conditional expression* kompakte Notation von alternativen Werten:

```
logical-expression “?” expression1 “:” expression2
```

expression1 und 2 müssen gleiche (bzw. kompatible) Typen haben

```
char* name;
if (name) cout<<name; else cout<<"nobody";
cout << ( name ? name : "nobody" );
        // Klammern muessen hier sein
```

- einziger „n-stelliger“ Ausdruck: *Funktionsaufruf*, Operanden sind Funktionsname und Argumentausdrücke

```
func ( exp1, exp2, . . . . , expn)
```

- Objekte können mittels expliziter Typumwandlung

“(“ **Typkonstrukt** “)“ **Objekt** oder

Typname “(“ Objekt “)“¹

in einen anderen Typ umgewandelt werden, sofern dabei nicht eine *built-in* Typumwandlung (*conversion*) aktiviert wird, liegen alle auftretenden Effekte in der Verantwortung des Nutzers, Anwendung zumeist, um „verlorengegangene“ Typinformation bei Zeigern zu rekonstruieren:



```
Person *p=new Person; ...
typedef Person* PersonPointer;

void f(void *any)
{
    // wenn klar ist, das any ein Person* -Zeiger ist,
    // so ist sinnvoll:
    ((Person*)any)->salary = 4000.0;
    // Person*(any) geht nicht: kein Typname
    // aber: PersonPointer(any)->salary=4000.0
}
```

1. erhält später ♦ 5.3. und ♦ 8.4. eine kanonische Erweiterung im Kontext von Klassen

Objektorientierte Programmierung in C++

```
f(p); // ok: Person* -> void* -> Person*
f(1); // warning: passing `int' to argument
      // 1 of `f(void *)' lacks a cast
// zur Laufzeit: Bus error (core dumped)
```

die Typprüfung durch den Compiler wird bei Typcasts explizit außer Kraft gesetzt der Compiler ist hier „machtlos“, kann bestenfalls warnen: 1 kann durchaus ein sinnvoller Wert des Typs `void*` sein.

B.Stroustrup (D&E): „Typumwandlungen gehören zu den fehlerträchtigsten Konzepten, die C++ unterstützt. Zudem sind sie aus syntaktischer Sicht auch die hässlichsten. ... keine für Systemprogrammierung geeignete Sprache [kommt] ohne das Mittel der Typumwandlung [aus] ... Ziel kann es daher nur sein, den Einsatzbereich von Typumwandlungen zu minimieren und diese so gutartig wie möglich zu gestalten.“ Ergebnis dieser Bemühungen ist die Einführung neuer Cast-Operatoren im Sprachstandard (`dynamic_cast`, `static_cast`, `reinterpret_cast` und `const_cast`) ↗12.3.

4.5. Funktionen

- in allen bisherigen Beispielen wurde ein intuitives Verständnis des Konzeptes der Funktionen vorausgesetzt: Funktionen sind die Grundkonstrukte der algorithmischen Strukturierung in C++, sie erlauben die Aggregation von parametrisierten Anweisungsfolgen, Funktionen sind durch ihre *Signatur* charakterisiert: Eine Deklaration der Form (auch *Prototyp* der Funktion `f` genannt)

```
return_type f (parameter_type_1 p_1, ..., parameter_type_n p_n);
```

beschreibt im mathematischen Sinne die Abbildung

$f: \text{parameter_type_1} \times \dots \times \text{parameter_type_n} \rightarrow \text{return_type}$

Bei jedem Aufruf einer Funktion (*expression*) prüft der Compiler die korrekte Verwendung, indem die Typen der aktuellen Parameter mit den Typen des Funktionsprototyps auf Kompatibilität (Gleichheit bzw. Überführbarkeit) untersucht werden -> jede benutzte Funktion muss zuvor deklariert sein ! dies räumt eine der „meistbenutzten“ Fehlerquellen von (Kernighan/Ritchie [K&R]) C aus (s. folgende Gegenüberstellung)

- eine Funktionsdefinition definiert neben der Signatur auch den Körper der Funktion:

```
return_type f (parameter_type_1 p_1, ..., parameter_type_n p_n)
{
  .... benutze p_1 .... p_n
  .... errechne ein return_type ret ....
  return ret;
}
```

Objektorientierte Programmierung in C++

(K&R) C:

```
main ()
{
    printf ("%f %s\n", sin(1),
           " = sin(1)");
}
```

```
$ cc sin.c -lm
```

```
$ ./a.out
```

```
Segmentation fault (core dumped)
```

```
Δ: #include <math.h> vor main:
```

```
$ cc sin.c -lm
```

```
$ ./a.out
```

```
0.000000 = sin(1)
```

```
Δ: ... , sin(1.0), ...:
```

```
$ cc sin.c -lm
```

```
$ ./a.out
```

```
0.841471 = sin(1)
```

C++ (auch ANSI C):

```
#include <iostream>
```

```
main ()
```

```
{
```

```
    std::cout<<sin(1)
```

```
    <<" = sin(1)"<<std::endl;
```

```
}
```

```
$ CC sin.cc -lm
```

```
"sin.cc", line 5: error:
```

```
undefined function sin called
```

```
Δ: #include <cmath> vor main:
```

```
$ CC sin.cc -lm
```

```
$ ./a.out
```

```
0.841471 = sin(1)
```

[es hilft hier nicht:

double sin(double); weil sin
sog. C-linkage hat ↗4.7.]

Parameternamen sind optional, natürlich will man i.allg. im Körper einer Funktion auf die Parameter zugreifen: dann müssen sie auch benannt sein; in Prototypen ist es üblich, keine Parameternamen zu verwenden, man kann jedoch Namen angeben:

```
void stringcopy (char*, char*); // von wo nach wo ???
```

```
void stringcopy (char* source, char* dest); //aha
```

- bei Funktionen, die einen (non-void) Wert zurückgeben, sollte dies auch auf allen möglichen Ausführungspfaden erfolgen, ansonsten *undefined behaviour* !! Compiler warnen i.allg. nur triviale Fälle !!

```
int f(int i) { if (i) return i; }
```

warning: if with return but no else with return

wird kein Rückgabotyp angegeben, war dieser früher in C++ (leider) implizit **int**

- Parameterübergabe beim Aufruf *call by value* (Kopie!), bei Referenztypen *call by reference*, Übergabe der Funktionsresultates ebenfalls als Kopie, nicht bei Referenztypen
- Funktionsdefinitionen können nicht geschachtelt sein (a la Pascal), sich aber gegenseitig (auch rekursiv) aufrufen, Funktionen operieren auf dem sog. *Stack*, auf dem beim Aufruf Kopien der Argumente (bzw. Zeiger auf diese) angelegt werden, auf dem zur Laufzeit Platz für lokale Objekte (außer für **static** und **register** Objekte) reserviert wird und der beim Verlassen der Funktion wieder zurückgesetzt wird. Stack und Heap wachsen in vielen Implementationen „gegeneinander“, Stacküberläufe werden unterschiedlich behandelt: typisch DOS: feste Größe (Stack + Heap = 64Kb) ggf. Abbruch mit Fehlermeldung **stack overflow**,

Objektorientierte Programmierung in C++

typisch UNIX: Ausdehnung bis zu einer Maximalgröße in Abh. von Systemressourcen (z.B. > 8 Mb) ggf. `segmentation fault (core dumped)`

- (globale) Funktionen haben standardmäßig Speicherklasse `extern`, werden sie jedoch als `static` deklariert, sind sie nur innerhalb des Quelltextes sichtbar (File-lokale Hilfsfunktionen)
- geht einer Funktionsdefinition das Schlüsselwort `inline` voraus, bemüht sich der Compiler um eine sog. *inline substitution*, d.h. es erfolgt kein eigentlicher Aufruf, sondern eine typgerechte textliche Einsetzung des Funktionskörpers bei Aufrechterhaltung der Aufrufsemantik (nicht so trivial wie Präprozessor-Makro-Ersetzung)! „kleine“ Funktionen sind gute Kandidaten für inline-Substitution (☞ 5.1.)

der Programmcode wird potentiell schneller aber größer, `inline` ist ein Compiler-Hinweis, der nicht befolgt werden muss (zu große/komplexe Funktionen, rekursive Aufrufe, ...) falls nicht `inline` substituiert wird, erhält man nicht bei allen Compilern eine Warnung!

Während der Testphase ist es sinnvoll, `inline` „auszuschalten“
(`g++: -fno-inline, cc: +d`) um mit einem Debugger den Aufruf von Funktionen verfolgen zu können,



...samples/inline

Definitionen von `inline` Funktionen gehören in Headerfiles, da der Compiler bei der Übersetzung den gesamten Körper der Funktion „sehen“ muss, um die Ersetzung vorzunehmen,

mittels bedingter Übersetzung kann man auch als Nutzer steuern, ob für eine Funktion eine (potentielle) `inline` -Variante oder eine Aufruf- („outline“) -Variante zu benutzen ist:

```
//-----  
// Header-File "bib.h"  
// int f(int) kann inline oder "outline" benutzt werden  
#ifndef _BIB_H  
#define _BIB_H  
#ifdef USE_INLINE_F  
inline int f (int i) {return i*i;} // Implementation  
#else  
int f (int); // Nur der Prototyp  
#endif  
#endif  
//-----  
// Implementation "bib.c"  
#undef USE_INLINE_F  
#include "bib.h"  
// nur der Prototyp bekannt  
int f (int i) {return i*i;}  
//-----
```

Objektorientierte Programmierung in C++

```
//-----  
// Applikation mit inline f "iapp.c"  
#define USE_INLINE_F  
#include "bib.h"  
void main() {int k = f(8); }  
// CC -o app iapp.c  
//-----  
// Applikation mit outline f "oapp.c"  
#undef USE_INLINE_F  
#include "bib.h"  
void main() {int k = f(8); }  
// CC -o app oapp.c bib.c  
//-----
```

besser als

```
#define MAX(i,j) ( (i)>(j) ? (i) : (j) )
```

ist

```
inline int max (int i, int j) { return i>j ? i : j; }  
//  
int m, n;  
m=0; n=1;  
cout << MAX (m++, n++); // -> 2 ????  
m=0; n=1;  
cout << max (m++, n++); // -> 1 !!!!
```

- in einer Funktionsdeklaration kann man Parameter mit sog. *default arguments* (Standardargumente) versehen, alle auf einen Parameter mit Standardwert folgenden Parameter müssen ebenfalls Standardwerte erhalten

```
int a (int, int=0, char* =0); //ok  
int b (int=0, int=0, char*=0);  
//Fehler: *= ist ein Operatorsymbol  
int c (int=0, int, char* = 0); // Fehler: Reihenfolge  
int d (int=0, int=0, char* =0);  
// ok, auch als d() aufrufbar
```

dann sind auch Aufrufe dieser Funktion mit weniger Parametern, als die Signatur vorgibt, möglich; fehlende Argumente werden vom Compiler durch die Voreinstellungen ersetzt -> Defaultargumente gehören in Headerfiles

man kann die „häufigste“ Verwendung einer Funktion kürzer notieren

```
int atoi (const char* string, int base = 10);  
// ascii to int on radix base  
  
atoi ("110"); // --> atoi("110", 10) --> 110  
atoi ("110", 2); // --> atoi("110", 2) --> 6
```



Objektorientierte Programmierung in C++

```
atoi ("110", 16)); // --> atoi("110", 16) --> 272
atoi (110); // Fehler: falscher Typ
```

Vorsicht bei gleichzeitiger Verwendung von Funktionsüberladung (⇒8.1.) und Standardargumenten !

```
int f(int);
int f(int, int=0);
f (1); // mehrdeutig: f(1) oder f(1,0)
```



- die strenge Kontrolle des Compilers der Parameter von Funktionen ist manchmal zu restriktiv, u.U. braucht man variable (in Länge und Typen) Parameterlisten, z.B. die C-Funktion `printf (stdio.h)` „lebt“ davon:

```
printf("ein String");
printf("%i + %i = %i\n", m, n, m+n);
```

Prototyp von `printf` aus (einem für C++ geeigneten) `stdio.h` ist:

```
extern "C" int printf (const char *, ...);
```

extern "C" ⇒4.7.

die Zeichenfolge `...` (Auslassung, *ellipsis*) kennzeichnet einen variablen Abschnitt einer Parameterliste, der vom Compiler nicht geprüft wird/werden kann Auslassung muss natürlich am Ende einer Parameterliste stehen, Zugang zu den beim Aufruf der Funktion übergebenen Parametern über spezielle Mechanismen möglich (⇒`stdargs.h` und `varargs.h`)

4.6. Anweisungen

- Grammatik:

statement:

```
labeled_statement |
expression_statement | ✓
compound_statement |
selection_statement |
iterator_statement |
jump_statement |
declaration_statement | ✓
try_block.
```

- labeled statement: vor jeder Anweisung kann ein Label (Marke, Identifier) stehen, solche Marken können bei der Abarbeitung des Programmes gezielt "angesprungen" werden (`goto` s.u.), `goto` sollte sehr sparsam (am besten gar nicht) verwendet werden (man kommt auch völlig ohne `goto` aus), Labels werden manchmal im Sinne von Kommentaren benutzt
- compound statement: Block, in `{...}` eingeschlossene Folge von Anweisungen, z.B. Funktionskörper, aber auch lokal geschachtelt, jeder Block eröffnet einen neuen Gültigkeitsbereich für Bezeichner

Objektorientierte Programmierung in C++

- selection statement: binäre Variante: if-Anweisung

```
if ( condition ) statement1
if ( condition ) statement1 else statement2
```

nur eine Anweisung erlaubt, mehrere müssen zu einem Block (compound statement) zusammengefasst werden, kein ‚then‘,
condition wird als log. Wert ausgewertet: # 0: statement1, !=0: leer / statement2,

Achtung bei `if (x=y) ...`

```
if ( i==1 && j==2 ) i=3; j=4; // j=4 unbedingt
```

```
if ( x ) if ( y ) A; else B;
```



Anweisung `statement1/2` darf keine Deklaration sein ! Innerhalb des Ausdrucks können Variablen vereinbart werden, deren Gültigkeitsbereich endet nach der if-Anweisung (Achtung: neu standardisiertes Sprachfeature, wird noch nicht von allen Compilern unterstützt !!!) gilt bei allen Anweisungen, die eine *condition* enthalten (if, switch, while, for)



Mehrweg-Variante: switch-Anweisung

```
switch ( condition ) statement
```

`statement` kann beliebig sein, häufigste Form ist jedoch ein Block in dem dann Alternativen des Wertes von `expression` ausgewertet werden können:

```
switch(i)
{
  case 0:      statements0; break;
  case 1:      statements1; break;
  case 2:      statements2; break;
  case 3:
  case 4:      statements3and4; break;
  case 5:      statements5only;
  case 6:      statements5and6; break;
  default:    statements; /* mit oder ohne */ break;
}
```

der Wert des Ausdrucks (i) wird zur Laufzeit berechnet und der Reihe nach mit den *case labels* verglichen, bei Übereinstimmung wird die Abarbeitung dort fortgesetzt, der Rest wird sequentiell durchlaufen bis ein `break` folgt, oder die `switch`-Anweisung beendet ist --> echte Alternativen stets mit `break` abschließen, sonst *fall through*

so lassen sich Spezialfälle vor allgemeinen Fällen behandeln, „passt“ kein Wert, wird der `default`-Zweig (sollte der letzte sein) abgearbeitet, fehlt dieser ist `switch` u.U. die leere Anweisung

Objektorientierte Programmierung in C++

keine Prüfung auf Vollständigkeit der Fallunterscheidung !

bei Schachtelung von `switch`-Anweisungen beziehen sich alle cases auf das kleinste umfassende `switch`

Deklarationen (mit Initialisierungen) dürfen nicht „übersprungen“ werden !

```
switch ( i ) {
    int v1 = 2; // error: jump past initialized var.
    case 1:
        int v2 = 3;
        // ...
    case 2:
        if (v2==7) // error: jump past initialized var.
            // ...
}
```

- iteration statement: while, do, for

iteration_statement:

```
while ( condition ) statement |
do statement while ( expression ) |
for ( for_init_statement [ condition ] ; [ expression ] ) st.ment.
```

for_init_statement:

```
expression_statement |
simple_declaration.
```

`while` wie üblich: * condition auswerten, Wert #0 statement ausführen weiter bei *

`do` anders als bei „repeat“ wird Bedingung als „Fortsetzungsbedingung“ interpretiert: * statement ausführen, condition auswerten, Wert # 0, weiter bei *

`for` kompakte „Laufanweisung“, `for_init_statement`: Initialisierung, `condition`: Fortsetzungsbedingung, `expression`: Iterationsschritt

```
for (int i=0; i < 100; i++) ....
for (p=list.first; p!=0; p=p->next) ....
for (;;) .... //forever, auch while (1) ....
```

- Jumps: `goto`, `break`, `continue`, `return`

jump_statement:

```
break ; |
continue ; |
return [ expression ] ; |
goto identifier ; .
```

`break`; (erlaubt in `switch` und allen Iterationen) beendet umfassende Anweisung

Objektorientierte Programmierung in C++

`continue`; (erlaubt in allen Iterationen) beendet den aktuellen Durchlauf und setzt Iteration fort. s.u. `continue` entspricht `goto contin`;

```
while ( foo ) {      do {                for ( .. ; ..; .. ){
    // .....          // .....          // .....
contin:             contin:             contin:
}                   } while (bar)      }
```

`return [expr] ;` (nur in Funktionsblöcken) beendet aktuellen Funktionsaufruf, ggf. mit Übergabe des Wertes von `expr` an die aufrufende Umgebung
`goto label`; Fortsetzung der Abarbeitung bei `label`; `label` muss im aktuellen Block definiert sein ! nur im Ausnahmefall verwenden i.allg. schlechter Stil !!!



- Exceptions: `try`, `catch`, `throw` ➔ 11.

4.7. Migration von C nach C++

- C++ soll abwärtskompatibel zu C sein: (fast) alle (ANSI-)C-Programme passieren auch einen C++ - Compiler mit identischer Semantik (bis auf wenige Ausnahmen)
- es muss auch möglich sein Programme aus C und C++ -Komponenten zu erstellen, insbesondere C-Funktionen aus C++ (C-Standardbibliotheken) aufzurufen und umgekehrt !
- C++ verwendet jedoch den sog. *name mangling* Mechanismus, um Namen von Funktionen als Eintrittspunkte bereitzustellen (*name mangling* Schema ist [noch] nicht standardisiert !!) :

```
// ff.cc:
void f() { ... }
void f(int) { ... }
```

```
-----
gcc -S ff.cc
-----
```

```
// ff.s
....
_f__Fv:
....
_f__Fi:
....
```

für C-Funktionen muss es möglich sein, dieses *name mangling* „abzuschalten“, aus
`int printf(const char* , ...);`

würde sonst `_printf__FPCce` in der C-Bibliothek steht jedoch nur `_printf` als Eintrittspunkt zu Verfügung !!

Lösung: sog. extern linkage Deklarationen:

```
extern "C" {int printf(const char* , ...); }
```

5. Klassen und Objekte

5.1. Grundprinzipien der OOP

Wichtigste Neuerung von C++ gegenüber C ist die Einführung des sog. Klassentyp-Konzeptes, um dieses ranken sich alle weiteren Prinzipien der OOP: Vererbung, Polymorphie, Zugriffsschutz, ...

- eine Klasse ist ein syntaktisches Konstrukt, welches eine Zusammenfassung von Eigenschaften (Attributen) von und Operationen (Funktionen) über einem abstrakten Datentyp beschreibt ! Klassen sind kanonische Erweiterungen von `struct`'s
- eine Klassendeklaration ist die vollständige Spezifikation aller Schnittstellen der Klasse, Variablen von einem Klassentyp heißen OBJEKTE (INSTANZEN) dieser Klasse
- daraus erwächst ein (gegenüber traditionellen Programmiersprachen) völlig neuartiges Typstrukturierungskonzept: neben der Kombination von Typen mittels elementarer Konstrukte (Zeiger, Felder, Referenzen, Aggregate ...) ist nunmehr auch die Definition neuer Typen mit adäquaten Operationen möglich:

```
/* C- Variante: Trennung von Daten */
typedef struct _Stack { int *data; int top, max } Stack;
/* und Operationen, z.B. */
int Stack_push (Stack* where, int what) { .... }
```

```
/* Benutzung: */
Stack s1; /* Initialisierung ??? */
```

```
Stack_init(&s1, 100);
Stack_push(&s1, 1);
```

```
// C++ -Variante:
```

```
//---- Stack -----
class Stack
{
protected:
    int *data;
    int top, max;
public:
    Stack(int dim=100);
    ~Stack();
    void push (int i);
    int pop();
    int full();
    int empty();
} // beliebter Fehler: fehlendes Semikolon
```



Daten

Operationen



Objektorientierte Programmierung in C++

```
// Benutzung, folgt der Syntax für struct-Zugriffe:

Stack s1; // schon initialisiert !!!
Stack *p; // nur ein Zeiger, bestenfalls == 0

s1.push(1); // Lesart: Sende an das Objekt s1 die Nachricht
           // push mit dem Parameter 1

p = new Stack; // hier entsteht das Objekt und wird
              // zugleich initialisiert

p->push(2);
delete p;
```

- die Operationen können auch durch Operatoren beschrieben werden -> kompakte Notation von arithmetischen Operationen auf nutzerdefinierten Datenobjekten:

```
class Complex {
    double real, imag;
public:
    Complex (double re=0, double im=0):real(re),imag(im){}
    friend Complex operator+ (Complex& z1, Complex& z2);
    friend ostream& operator<<(ostream& o, Complex& z);
// syntaktische Feinheiten später
};

Complex x(2,3); // 2+3*i
Complex y(4,5); // 4+5*i

cout << x+y; // !!! "6+8*i"
```

- Klassen sind vollwertige Typen, d.h. incl. aller weiteren Strukturierungsmittel und Konsistenzprüfungen durch den Compiler
- alle Datenkomponenten einer Klasse heißen MEMBERDATEN
- alle Funktionskomponenten einer Klasse heißen MEMBERFUNKTIONEN
- Namen von Mitgliedern haben *class scope*, Namen von Memberdaten einer Klasse müssen eindeutig sein, ein Name kann nicht zugleich ein Memberdatum und eine Memberfunktion benennen, Memberfunktionen gleichen Namens sind (im Sinne der Überladung (↔8.) erlaubt
- Memberfunktionen können nur für ein Objekt der entsprechenden Klasse aufgerufen werden !

```
pop(); // Fehler, sofern es nicht ein globales 'pop' gibt
```

Objektorientierte Programmierung in C++

```
Complex z;  
z.push(3); // Fehler: Complex hat keine Operation 'push'
```

- Memberdaten können von beliebigem Typ (auch Klassentyp -> umfassende Klasse wird dann auch als Containerklasse bezeichnet) sein
- alle Member unterliegen speziellen nutzerdefinierten Zugriffsrechten, die einen konsistenten Zustand der Objekte sichern (⇨5.4.)

```
Stack s1; s1.push(1); s1.push(2); s1.push(3); // ok  
s1.top = 0; // nicht erlaubt, weil protected
```

- Memberfunktionen können im Klassenkörper
~ nur als Prototyp vereinbart sein (z.B. alle **stack** - Memberfunktionen), dann muss die Implementation außerhalb des Klassenkörpers (u.U. in einem anderen Modul) bereitgestellt werden, dabei ist der Bezug zu der Klasse mit dem *scope resolution* Operator herzustellen

```
void Stack::push(int i) // auch inline moeglich  
{  
    if (!full()) data[top++]=i;  
    // Aufruf von full(), Zugriff auf data und top fuer  
    // das gleiche Objekt, fuer welches push aufgerufen  
    // wurde  
}
```

~ oder vollständig implementiert werden (z.B. **complex::complex(...)**), dann wird die Memberfunktion (sofern möglich) automatisch zu eine **inline**-Funktion, kurze Memberfunktion sind gute Kandidaten für **inline** (Klassenvereinbarungen und alle inline-Funktionen gehören in Headerfiles !!!)



- Memberfunktionen können **uneingeschränkt** direkt auf Member (-daten und -funktionen) der gleichen Klasse zugreifen, dies sind dann Zugriffe über das Objekt, an das sich der ursprüngliche Aufruf richtete !) vgl. C-Lösung:

```
int Stack_push (Stack* where, int what)  
{  
    if (!Stack_full(where)) where->data[where->top++]=what;  
}
```

- es sind auch sog. unvollständige Klassendeklarationen erlaubt, von einer solchen Klasse können jedoch bis zu ihrer vollständigen Deklaration lediglich Zeiger & Referenzen benutzt werden:

```
class B;  
class A {B * my_B; ....}; // oder ... class B* my_B;  
class B {A * my_A; ....};
```

Objektorientierte Programmierung in C++

- strukturell identische Klassen mit verschiedenen Namen bilden verschiedene Typen (es gibt jedoch die Möglichkeit, nutzerdefiniert Kompatibilität herbeizuführen):

```
class X { public: int i; } x;  
class Y { public: int i; } y;  
x=y; y=x; // beides falsch !!!
```

- Klassen können auch lokale (Klassen-)Deklarationen enthalten (☞5.5.)

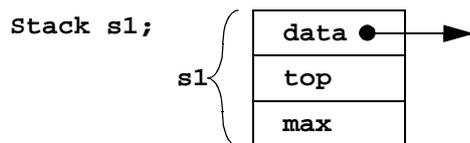
5.2. Objekte im Speicher

- ein Objekt einer Klasse wird bei seiner Entstehung im Speicher als lineare (ggf. nach *alignment* ausgerichtete) Anordnung aller Memberdaten der Klasse (bis hierher ohne jeglichen overhead) realisiert ---> Passfähigkeit zu C-structs, erst bei Klassen mit virtuellen Funktionen ☞7.2. bzw. Mehrfachvererbung ☞9.1. geht diese verloren

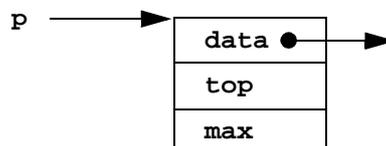
- Objekte können

~ global (sie werden vor dem eigentlichen Programmstart angelegt),
~ funktionslokal (sie werden beim Aufruf der entsprechenden Funktion angelegt)
oder
~ dynamisch (sie werden beim Aufruf von new 'class' angelegt)
sein.

- Die Art der Entstehung eines Objektes hat keinen Einfluss auf sein Speicherabbild !



Stack p = new Stack;



- Memberfunktionen werden pro Klasse nur einmal (als übersetzter Funktionscode) angelegt, diese erhalten neben den expliziten Parametern aus der Memberfunktionsdeklaration einen zusätzlichen, versteckten Parameter, an den beim Aufruf der Memberfunktion die Adresse des Objektes übergeben wird, für das die Memberfunktion gerufen wird (implizit entspricht dies der C-Lösung)

Objektorientierte Programmierung in C++

```
// z.B.
s1.push(1); // ----> _push__5StackFi(&s1, 1);
p->push(2); // ----> _push__5StackFi(p, 2);
                name mangling !!!
```

- der implizite Parameter ist innerhalb einer Memberfunktion unter dem reservierten Bezeichner **this** zugänglich, **this** liefert eine Zeiger auf das Objekt, für das die Memberfunktion aufgerufen wurde (in Memberfunktionen einer Klasse X ist **this** also vom Typ **x * const**)
- alle impliziten Bezugnahmen auf das **this**-Objekt könnte man redundant auch explizit vornehmen, dies ist eher unüblich, es gibt jedoch Situationen, wo man den **this**-Zeiger explizit braucht

```
void Stack::push(int i)
{
    if (!this->full())           // redundant
        this->data[this->top++] = i; // redundant
}

class Int {
    int rep;
public:
    Int& operator++() { ++rep; return *this; }
                        // unumgaenglich
};
```

- Memberfunktionen können als **const** deklariert werden, dies legt fest, wie das über **this** referenzierte Objekt zu behandeln ist, d.h. in einer **const** Memberfunktion ist **this** vom Typ **const x * const** (sonst nur **x* const**)

```
class Complex { // eine andere Variante
    double real, imag;
public:
    Complex(double x=0, double y=0):real(x), imag(y) {}
    double abs() const
    {return sqrt(real*real+imag*imag);}
    void scale() // kann nicht const sein !!!
    {   double oldabs=abs();
        real/=oldabs; imag/=oldabs;
    }
};
```

```
double r;
const Complex i (0, 1);
Complex z (1, 2);

r = i.abs() // ok r==1.0
r = z.abs() // ok r==√5
```

Objektorientierte Programmierung in C++

```
i.scale(); //Fehler: non-const func. called for const obj.
           // auch wenn sich der Wert gar nicht aendert !
z.scale(); // ok
```

- Memberdaten/-funktionen können auch als **static** (keine anderen Speicherklassen möglich) deklariert werden. **static** Memberdaten werden nicht pro Objekt, sondern einmal pro Klasse angelegt. **static** Memberdaten (manchmal auch als *class members* bezeichnet) werden mit ihrem Erscheinen im Klassenkörper lediglich deklariert, die Definition (incl. der Bereitstellung des Speicherplatzes für diese Daten) muss außerhalb der Klasse erfolgen.
- **static** Memberdaten dienen gewissermaßen als geteilter Datenpool für alle Objekte einer Klasse, alle Zugriffe über ein Objekt referenzieren das gleiche Datum, **static** Memberdaten existieren unabhängig von der Existenz von Objekten, daher gibt es eine weitere (klassenbezogene) Form des Zugriffs

```
class A {
    static int count;
public:
    static int c(){ return count; }
    static const double A_specific_const;
    A() {count++;}
    ~A(){count--;}
} a1, a2, a3;

int A::count = 0; // hier erst definiert
const double A::A_specific_const = 0.1234; // dito

void f() {
    double x = A::A_specific_const; // class access
    A::A_specific_const = 1.23; // Fehler: const !
    cout << "Es gibt jetzt " << a1.c() << " A-Objekte\n";
           // a1.count ist private
} // auch a2.c() oder a3.c() oder A::c() moeglich
```

- **static** Memberfunktionen sind solche, die nur mit **static** Memberdaten arbeiten, sie sind ebenfalls unabhängig von der Existenz von Objekten der entsprechenden Klasse aufrufbar (**A::c()**) und verfügen demzufolge **nicht** über den impliziten **this**-Zeiger, **static** Memberfunktionen können auch über Objekte aufgerufen werden, dabei wird jedoch die Objektidentität (die Adresse des Objektes) ignoriert und lediglich die Klassenzugehörigkeit des Objektes ausgewertet
- auch konstante **static** Memberdaten konnte man bislang nicht im Klassenkörper initialisieren, es gibt jedoch (nur für int-Konstanten) einen häufig benutzten *work around* , **mit nunmehr verfügbarem c++Standard erlaubt! nach wie vor müssen die static member außerhalb der Klasse deklariert werden !**

Objektorientierte Programmierung in C++

```
class X {
public:
    static const int eins=1; // sollte nun
    static const int zwei=2; // erlaubt
    static const int drei=3; // sein
}; // error: initializer for member eins, zwei, drei
class X {
public:
    enum { eins=1, zwei=2, drei=3 };
};
```



- die Identität eines Objektes ist durch seine Adresse im Speicher bestimmt, d.h. zwei verschiedene Objekte (auch Instanzen der leeren Klasse `class Empty{};`) besitzen verschiedene Adressen !
- Klassen (ohne virtuelle Funktionen) selbst besitzen zur Laufzeit keine Repräsentation, sie sind statische Beschreibungen des Aufbaus all ihrer Objekte, die der Compiler zur Übersetzungszeit als „Bauanleitungen“ für Objekte benutzt
- Bezeichner von Klassen sollten den Typcharakter von Klassen widerspiegeln, übliche Konvention: Klassennamen mit großen Anfangsbuchstaben, Objekte durchgängig klein



5.3. Konstruktoren und Destruktoren I

- kein Objekt sollte in einem uninitialisierten (inkonsistenten) Zustand in das Programmgeschehen eingreifen, ein Objekt, welches vernichtet wird (z.B. Verlassen eines lokalen Blockes, `delete`) sollte zuvor alle angeforderten Ressourcen (Speiche, Filedeskriptoren ...) zurückgeben
- Memberfunktionen wie etwa `init();` und `end();` könnten diese Aktionen beschreiben, ihr Aufruf ist jedoch nach wie vor von der „Disziplin“ des Benutzers abhängig
- C++ stellt stattdessen das Konzept von Konstruktoren und Destruktoren bereit, die implizit (& zwangsweise) aufgerufen werden, wenn Objekte entstehen/vergehen
- ein Konstruktor ist eine Memberfunktion einer Klasse mit dem gleichen Name wie die Klasse und ohne Rückgabotyp (auch nicht `void`), Konstruktoren können Parameter haben, die in die Initialisierung eines Objektes einfließen, mehrere Konstruktoren sind (im Sinne der Überladung) in einer Klasse möglich

```
Stack::Stack(int dim)
{
    max=dim; top=0;
    data=new int[max];
}
```

Objektorientierte Programmierung in C++

- Konstruktoren können in Erweiterung der Syntax für Funktionen eine sog. *initializer list* enthalten, das ist eine auf die Parameterliste und einen Doppelpunkt folgende, mit Komma separierte Liste von Initialisierungen der Form

```
memberdatum(initialer_wert)
// bessere Variante:
Stack::Stack(int dim)
           : top(0), max(dim), data(new int[dim])
{ /* leer ! */ }
```

- Initialisierung ist **keine** Zuweisung

```
class X {
    const int c;
public:
    X(): c(1) {} // ok, aber
// X() {c=1;} // falsch
};
```

```
#include <iostream>
class A {
public:
    A(int i){ std::cout << "A("<<i<<")\n"; }
};

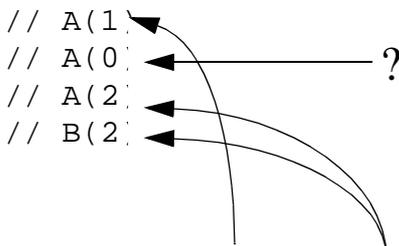
class B {
    A myA;
public:
    B (int n) { myA=n; std::cout << "B("<<i<<")\n"; }
    // error: member B::myA needs initializer
    // (no default constructor for class A )
    // mit A(int i=0) folgende Ausschriften:

    // A(1)
    // A(0)
    // A(2)
    // B(2)

};

int main() { A a(1); B b(2); }

// richtig waere:
// .... B (int n): myA(n) { .... }
// A(1)
// A(2)
// B(2)
```



Objektorientierte Programmierung in C++

- Konstruktorparameter sind beim Anlegen von Objekten (geeignet) anzugeben

```
B b1(1);    B b2 = B(2);    B b3 = 3;    B b4=B(4);  
B *pb = new B (5);
```

- Konstruktoren für globale Objekte laufen vor dem eigentlichen Hauptprogramm
- Konstruktoren für lokale Objekte laufen beim Betreten des entsprechenden Blocks
- Konstruktoren für dynamische Objekte laufen bei `new` (nach Bereitstellung des Objektspeicherplatzes)

- Klassen ohne (nutzerdefinierten) Konstruktor erhalten implizit einen sog. parameterlosen *default constructor* `x::x(){}` , sobald ein nutzerdefinierter Konstruktor vorliegt gibt es keinen impliziten *default constructor* mehr

Referenz !!!!
↓

- jede Klasse enthält implizit den sog. *copy constructor* `x::x(const x&)` mit der Semantik „memberweise Kopie“, dieser kann auch explizit mit anderer Semantik implementiert werden

```
class A { public: int i,j; }; // A(),A(const A&) implizit  
  
A a1; a1.i=2; a1.j=3;  
A a2 (a1); // oder auch A a2=a1; bzw. A a2=A(a1);  
// --> a2.i==2 && a2.j==3
```

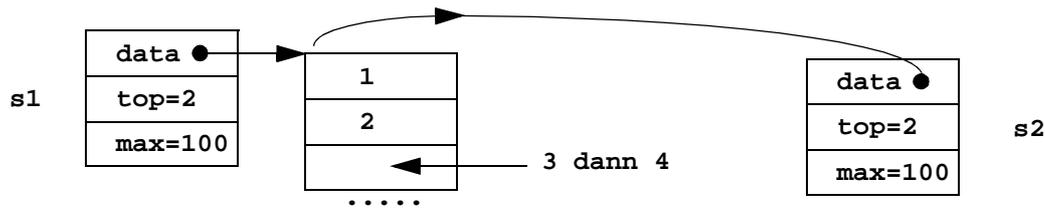
- der *copy constructor* sollte immer dann nutzerdefiniert implementiert werden, wenn die obige Standardsemantik nicht ausreicht, weil „nicht alle, den Zustand eines Objektes einer Klasse beschreibenden Informationen direkt im Objekt enthalten sind“, z.B. schon im Beispiel der `stack`-Klasse

```
#include <iostream>  
#include "Stack.h"  
  
int main()  
{  
    Stack s1;  
  
    s1.push(1);  
    s1.push(2);  
  
    Stack s2(s1); // separates Objekt als Kopie  
  
    s2.push(3);  
    s1.push(4);  
  
    std::cout<<s2.pop(); //sollte wohl 3 sein ist aber 4  
}
```



...samples/stack

Objektorientierte Programmierung in C++



```
// erforderlich ist ein spezieller Copy-Konstruktor
// fuer Stack:
// Stack(const Stack&); im Klassenkoerper
// und Implementation z.B.
```

```
Stack::Stack(const Stack & other) :max(other.max),
                                   top(other.top), data(new int[other.max])
{
    for (int i=0; i<top; i++)
        data[i]=other.data[i];
}
```

- man spricht auch (in Anlehnung an die SmallTalk-Terminologie) von *shallow copy* und *deep copy*, der implizite Copy-Konstruktor macht immer eine *shallow copy*, manchmal ist jedoch eine *deep copy* erforderlich
- Konstruktoren und Destruktoren sollten stets nur klassenbezogene Aktionen (also keine globalen Effekte) auslösen
- ein Destruktor ist eine Memberfunktion einer Klasse mit dem Namen wie die Klasse mit einem vorangestellten ~ und ohne Rückgabotyp (auch nicht `void`), Destruktoren können keine Parameter haben, daher ist pro Klasse nur eine Destruktor erlaubt (keine Überladung möglich); Destruktoren sollten Ressourcen, die von Konstruktoren angefordert wurden, wieder freigeben

```
Stack::~~Stack()
{
    delete [] data;
}
```

- Destruktoren für globale Objekte laufen nach dem eigentlichen Hauptprogramm
- Destruktoren für lokale Objekte laufen beim Verlassen des entsprechenden Blocks
- Destruktoren für dynamische Objekte laufen bei `delete` (vor Freigabe des Objektspeicherplatzes)

```
// ein Beispiel zu Konstruktor-/Destruktoraufrufen
// Ausgaben via cout sind globale Effekte, die i.allg.
// nicht von Konstruktor/Destruktoren verursacht werden
// sollten, hier in rein "kognitiver Absicht" benutzt:
```


Objektorientierte Programmierung in C++

```
X::~X() for loc at 0xeffffa80
X::~X() for x1 at 0xeffffa88
X::~X() for x1 at 0x210b8
X::~X() for x1 at 0x210bc
*/
```

5.4. Zugriffsschutz I

Memberdaten, die den konsistenten Zustand eines Objektes repräsentieren, sollten vor absichtlichen/versehentlichen Zugriffen/Änderungen von „außen“ geschützt werden. Entsprechend dem Konzept der abstrakten Datentypen soll oft nur eine funktionale Schnittstelle zu der ansonsten „unsichtbaren“ internen Darstellung eines Objektes bereitgestellt werden.

- C++ bietet die Möglichkeit, die Member einer Klasse in drei verschiedene Zugriffsebenen einzuordnen:

private: Zugriff nur in der Klasse selbst (und in sog. **friends** s.u.) möglich

protected: Zugriff in der Klasse selbst und in direkt abgeleiteten Klassen erlaubt (☞ 6.4.)

public: Zugriff uneingeschränkt möglich

- beliebig viele **private/protected/public** -Abschnitte können in einer Klasse in beliebiger Reihenfolge enthalten sein, sofern die Reihenfolge der Memberdaten im Objektlayout keine Rolle spielt, sollte man zur besseren Lesbarkeit einer Klassendeklaration die Member in max. 3 Abschnitten (erst **private**, dann **protected**, dann **public**) gruppieren
- **class** und **struct** sind gleichberechtigte Schlüsselworte für die Vereinbarung von Klassen, sie unterscheiden sich nur in der Voreinstellung des Zugriffsschutzes für die entsprechend definierten Klassen:

class beginnt implizit mit **private** Membern

struct beginnt implizit mit **public** Membern

```
class X { .... }; <=====> struct X { private: .... };
struct Y { .... }; <=====> class Y { public: .... };
```

```
// Beispiel Stack:
class Stack {
    int top, max, *data;
public: ....
};
```



Objektorientierte Programmierung in C++

```
// interne Repraesentation kann weder direkt gelesen
// noch veraendert werden:
Stack s;
s.push(1);
s.top = 0; // Fehler: cannot access prot. member top
```

- die Implementation von abstrakten Datentypen mit Hilfe von Klassen in C++ ist ein Kompromiss zugunsten einer effektiven Umsetzung: **private** Member müssen im Klassenkörper vereinbart werden (und sind somit jedem Benutzer im Quelltext visuell zugänglich, können aber i. allg. nicht verwendet werden), dafür hat der Compiler alle Informationen über Aufbau und funktionelle Schnittstelle einer Klasse mit allen sich daraus ergebenden Prüfmöglichkeiten
- der Zugriffsschutzmechanismus soll gegen versehentliche Zugriffe schützen, absichtliche Zugriffe sind unter Umgehung der Schutzbarrieren möglich
- die Einschränkung des Zugriffs wird durch Einschränkung der Sichtbarkeit von Membern geregelt, keine separaten Lese- bzw. Schreibrechte, *read only* Variablen können durch folgenden Mechanismus realisiert werden

```
class X {
    int i; // read only
public:
    int get_i_value() { return i; }
};
```

- sollen die Implementationsdetails einer Klasse nicht direkt aus dem Klassenkörper „ablesbar“ sein (z.B. um die Möglichkeit alternativer Implementationen offenzuhalten) wird zumeist nach folgendem *idiom* („Redewendung“) mit Repräsentantenklassen gearbeitet

```
class ADT {
    class RepClass *rep;
public:
    ADT (RepClass *r): rep(r) { .... }
    // funktionelle Schnittstelle
};
// verschiedene Implementationen sind möglich, indem
// Ableitungen RepClass uebergeben werden, zum Austausch
// der Implementation muss ADT nicht neu uebersetzt werden!
```

- der Zugriffsschutz in C++ besteht pro Klasse, nicht pro Objekt:

```
class A {
    S secret;
public:
    void spy_another_A (A other) { secret = other.secret;}
};
```



Objektorientierte Programmierung in C++

- zwischen „Alles“ (`public`) und „Nichts“ (`private`) bestehen zwar noch spezielle Rechte für (direkt) abgeleitete Klassen (`protected`), dennoch ergibt es oftmals die Notwendigkeit einer eingeschränkten Menge von Funktionen/Klassen den Zugriff zu `private` Mitgliedern einzuräumen, ohne dass eine Vererbungsrelation angemessen ist, für diesen Zweck kann eine Klasse solche Funktionen/Klassen zum `friend` erklären

```
class B { public: void f(class A* pa); };

class A {
    S secret;
public:
    friend void trusted_function (A& a) // inline !!
    { ..../* can access: */ a.secret .... }
    friend void B::f(A*);
};
```

- `friend`-Funktionen sind keine Memberfunktionen der Klasse, die die `friend`-Relation einräumt, macht man eine ganze Klasse zum `friend`, werden alle Memberfunktionen dieser zu `friends`
- Vorsicht bei unterschiedlichen Kontexten für `inline`- und „outline“-Funktionen

```
typedef char* T;
class S {
    typedef int T;
    friend void f1(T) { .... } // void f1(int);
    friend void f2(T); // void f2(int);
};

void f2(T) { .... } // void f2(char*); also kein friend !
```

- die `friend`-Relation ist nicht symmetrisch, nicht transitiv und nicht vererbbar

```
class ReallySecure {
    friend class TrustedUser;
    ....
};
class TrustedUser {
    // can access all secrets
};

// -----
class Spy: public TrustedUser {
    // if friend relation would be inherited: aha !
};
```

- die Position einer `friend`-Deklaration in einem Klassenkörper (in einem `private`, `protected` oder `public` Abschnitt) ist ohne Bedeutung, dennoch sollte man `friend`-Deklarationen in einem `public` Abschnitt unterbringen, weil dieser die



Objektorientierte Programmierung in C++

nutzerrelevante Schnittstelle der Klasse beschreibt (und dazu gehören auch `friends`)

- der Zugriffsschutz betrifft alle Member einer Klasse, auch Konstruktoren und Destruktoren, so kann man z.B. steuern, „wer“ Objekte einer Klasse anlegen bzw. vernichten kann



```
class R { // restricted use
    R(); // private !
    friend class R_user;
    R(const R&); // not implemented !
};
```

```
R r; // Fehler R::R() not accessible
```

```
class R_user {
    R my_R;
    // do anything with my_R, but copying
    // e.g. void f(R); not allowed
};
```

- um das Instanzieren einer Klasse gänzlich zu unterbinden (diese Klasse ist dann nur für Vererbung zu gebrauchen) gibt es ein anderes spezielles Ausdrucksmittel *abstract base classes* ↪7.3.

5.5. Lokalität

- Klassen bilden lokale Namensräume für ihre Member, dies gilt auch für Typnamen (Klassen und `typedefs`) die innerhalb einer Klasse definiert werden (ab Sprachversion 2.0 konsistent unterstützt), enthält eine Klassendefinition eine weitere lokale Klassendefinition so spricht man auch von *nested classes*
- *nested classes* haben keinen direkten Zugang zu den Members der umfassenden Klasse (jedoch zu static Members, sofern der Zugriff erlaubt ist)
- Benutzung von *nested classes* spielt eine eher untergeordnete Rolle
- lokale `typedefs` werden z.B. in der STL (*standard template library*) massiv benutzt
- Klassenlokale Typnamen unterliegen den gleichen Zugriffsrechten wie Member, einige Compiler (z.B. g++ incl. 2.7.x, SunCC 3.0.1) lassen fälschlicherweise auch Bezugnahmen auf private Typnamen außerhalb der Klasse zu
- static Member lokaler Klassen müssen wie gehabt global definiert werden, Memberfunktionen lokaler Klassen können auf globalem Niveau definiert werden, in beiden Fällen dient der *scope resolution* Operator in kaskadierter Form zur Konstruktion der richtigen Namen

Objektorientierte Programmierung in C++

- ein Typname darf **nicht** innerhalb einer Klasse lokal redefiniert werden, wenn er bereits vorher in der Klasse verwendet wurde
- die Definition einer lokalen Klasse führt nicht automatisch dazu, dass die umfassende Klasse auch ein Objekt der lokalen Klasse enthält !
- Klassen können auch lokal zu Funktionen definiert werden, dabei bestehen keine impliziter Zugang der Funktion zu den Mitgliedern der Klasse und umgekehrt der Memberfunktionen der Klasse zu `auto` Variablen der Funktion, funktionslokale Klassen können keine `static` Member enthalten, ihre Memberfunktionen müssen im Klassenkörper (`inline`) definiert werden

```
class Y{};

typedef char* T;

class X {
    T i;
    // typedef int T; Fehler: Redefining T after use in X::Y.
    typedef double S;
private:
    static int sx;
    // Y y; Fehler: Redefining Y after use in X::Y.

    class Z {
        int fz();
    public:
        static int sz;
        Z(){};
    } z;
public:
    int fx(){return sx - Z::sz;}
    class P
    { public: int p; P():p(1){} }; // X enthaelt kein P !
} x;

int X::sx=0; // obwohl private
int X::Z::sz=0; // obwohl private

int X::Z::fz(){return 0;}

// SunSoft CC 4.0.1:
X::Z xz; // Warnung: X::Z is not accessible from file level.
...samples/local
X::S i; // Warnung: X::S is not accessible from file level.

X::P xp;
```



```
int main()
{
    return xp.p + x.fx();
}
```

5.6. Zeiger auf Member

Das Zeigerkonzept aus C erhält im Kontext von Klassen eine kanonische Erweiterung, um spezielle Zeigerkonstrukte (Zeiger relativ zu einem Objekt der Klasse) auszudrücken

- auf Memberdaten eines konkreten Objektes können wie gehabt, „normale Zeiger“ gerichtet werden, um die Adresse eines Memberdatums zu bestimmen muss der Zugriff auf dieses erlaubt sein

```
class X {
    int priv;
public:
    int publ;
    X(int i=100):priv(i){}
    int* p() {return &priv;}
};
```

```
int main()
{
    X x;

    int* p;
    // p=&x.priv; main() cannot access X::priv: private member
    p=&x.publ; // ok

    *x.p()=1; // ok: X selbst gibt Zugriff via p frei
}
```

- darüber hinaus besteht in C++ die Möglichkeit, Zeiger auf Member relativ zum Objektanfang zu bilden, solche Zeiger repräsentieren keine Adressen, sondern lediglich (objektunabhängige) Offsets, die erst im Zusammenhang mit einem konkreten Objekt Speicheradressen konstruieren (. * und ->* sind 2 neue Operatoren)

```
class X {
public:
    int p1,p2,p3;
};
```

```
int main() {
    X x; X* pp=&x;
    int X::*xp; // *xp ist ein int in X
```



...samples/mempointer

Objektorientierte Programmierung in C++

```
    xp=&X::p2;          // ok
// xp=&x.p2; error: bad assignment type: int X::* = int *
// int *p;
// p=&X::p2; error: bad assignment type: int * = int X::*
p=&(x.*xp);           // ok, ohne Klammern falsch: (&x).*xp
pp->*xp = 1;
}
```

- auch auf (nicht **static**) Memberfunktionen kann man spezielle Zeiger (kein „normalen Funktionszeiger“) verweisen lassen, auch hier ist eine Aufruf der Memberfunktion über den Zeiger nur unter Einbeziehung eines konkreten Objektes möglich, **static** Funktionen verhalten sich wie globale Funktionen
- es ist nicht erlaubt, die Adresse von Konstruktoren/Destruktoren zu bilden
- sobald man die Adresse einer Memberfunktion bestimmt, muss diese durch den Compiler (auch) in einer **outline**-Version bereitgestellt werden

```
#include <iostream>
using namespace std;
```

```
class X {
public:
    void f1(){cout<<"X::f1()\n";}
    void f2(){cout<<"X::f2()\n";}
    static void f3(){cout<<"static X::f3()\n";}
    typedef void (X::*Action)();
    void repeat(Action a=&X::f1, int count=1);
//////// nicht mehr: a= X::f1 !!!
};
```

```
void X::repeat (Action a, int count)
{ while (count-->0) (this->*a)(); }
```

```
int main() {
    X x; X* pp=&x;
    void (X::*xfp)();
    // Zeiger auf Memberfkt. in X mit Signatur void->void
    xfp=&X::f1; // nicht mehr: xfp=X::f1
// xfp();
// object missing in call through pointer to memberfunction

    (x.*xfp)(); // X::f1()
    xfp=&X::f2; // nicht mehr: xfp=X::f1
    (pp->*xfp)(); // X::f2()
// xfp=X::f3;
// bad assignment type: void (X::*)(()) = void (*)()static
// aber:
    void (*fp)()=X::f3;
    fp(); // auch (*fp)();
    x.repeat(xfp, 2);
}
/*
```



...samples/mempointer

```
X::f1()
X::f2()
static X::f3()
X::f2()
X::f2()
*/
```

6. Vererbung

das Klassenkonzept bildet die Grundlage zur Definition abstrakter Datentypen, die als Einheit von Daten und Operationen einen Problembereich strukturieren, indem problemadäquate Typen eingeführt werden können

zwischen Objekten eines Typs sind Initialisierung und Zuweisung standardmäßig implementiert (und können davon abweichend redefiniert werden, *copy constructor* ↪5.3. bzw. *assignment operator* ↪8.2.)

ansonsten gibt es bislang kein Ausdrucksmittel, welches es gestattet, Ähnlichkeiten / Gemeinsamkeiten unterschiedlicher Typen auszudrücken,

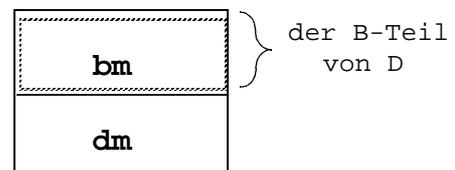
bei der allgemeinen Methodologie „Klassifizierung“ ist jedoch gerade die Bildung hierarchischer Strukturen von ausschlaggebender Bedeutung (vgl. Biologie), um einen Problembereich zu „ordnen“, wobei Eigenschaften allgemeinerer Klassen auf speziellere übertragen - „vererbt“ - werden

- C++ bietet ein syntaktisches Ausdrucksmittel für die Vererbung von Eigenschaften: die Definition eines Klassentyps ‚**Derivate**‘ kann sich auf einen anderen bereits definierten Klassentyp ‚**Base**‘ beziehen, z.B.

```
class Base {
public:
    int bm;
    int bf(int);
};

class Derivate: public Base {
public:
    int dm;
    void df(double);
};
```

Layout von Derivate-Objekten



wobei der neue Typ alle Eigenschaften aus der alten Typ übernimmt, **Base** ist (direkte) Basisklasse, **Derivate** die abgeleitete Klasse, wir werden diesen Sprachgebrauch gegenüber der in SmallTalk üblichen Terminologie **Base**=Superklasse, **Derivate**=Subklasse bevorzugen

Objektorientierte Programmierung in C++

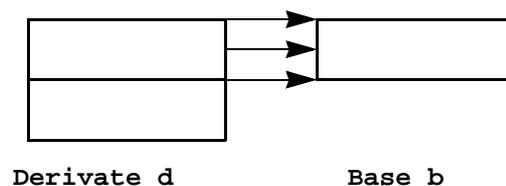
- in der Tat verhält sich die Klasse `Derivate` so, als wäre sie definiert worden per:

```
class Derivate {  
public:  
    int bm;  
    int dm;  
    int bf (int);  
    void df(double);  
};
```

- darüber hinaus sind automatisch mit der (`public`) Vererbung gewisse Beziehungen zwischen `Base` und `Derivate` definiert: ein `Derivate`-Objekt besitzt alle Eigenschaften (Memberdaten) eines `Base`-Objektes (nicht nur die `public` Member !), außerdem sind alle `public` Operationen auf `Base`-Objekten auch für `Derivate`-Objekte anwendbar, damit kann ein `Derivate`-Objekt anstelle eines `Base`-Objektes fungieren (nicht umgekehrt), jedes `Derivate`-Objekt **IST EIN** `Base`-Objekt

```
Derivate d, *dp=&d;  
Base b=*dp, *pb;    // (1)  
  
void f (Base fb);  
Base& g (Base *fp) {return *fp;}  
....  
pb=&d; // auch pb=new Derivate;  
b=d;    // (2)  
f(d);    // (2)  
b=g(&d);  
....
```

- Initialisierungen (1) und Zuweisungen (2) von `Derivate` nach `Base` (also auf Objektebene) sind implizit als *Projektion* definiert



bei Operationen mit Zeigern/Referenzen wird lediglich die Adresse eines `Derivate`-Objektes übergeben, am Zeiger „hängt“ ein „größeres“ Objekt, über einen `Base`-Zeiger sind natürlich nur die `Base`-Komponenten des `Derivate`-Objektes erreichbar, anders ausgedrückt: es gibt implizite Typumwandlungen von `Derivate*` nach `Base*` und von `Derivate&` nach `Base&`

- es werden stets alle Eigenschaften vererbt, eine Auswahl ist nicht möglich
- Vererbung ist **das Schlüsselkonzept** der OOP, aus softwaretechnologischer Sicht eröffnet es den Weg zu modularen, wiederverwendbaren und !!erweiterbaren!! Programmbausteinen

Objektorientierte Programmierung in C++

- es sind auch andere Formen der Vererbung (nicht `public`) möglich ↪6.3., dann bestehen allerdings nicht die angegebenen Beziehungen zwischen Basis und Ableitung, Vererbung ist nur für Klassentypen (keine built-in Typen und elementare Typkonstrukte) erlaubt ! (hybrider Charakter der Sprache C++)
- C++ bietet auch die Möglichkeit einen neuen Klassentyp als Ableitung einer ganzen Menge von Basistypen zu konstruieren, man spricht von *multiple inheritance* - Mehrfachvererbung ↪9.

6.1. Redefinitionen

- jede Klasse definiert einen eigenen Namensraum, damit sind auch bei abgeleiteten Klassen Membernamen erlaubt, die bereits in einer Basisklasse verwendet wurden, ansonsten müsste ein Implementator einer abgeleiteten Klasse alle Namen aller Basisklassen kennen (und vermeiden)
- damit wird die Benutzung eines Namens zunächst immer im Kontext der Klasse des *statischen Typs* der Bezugnahme auf den Namen aufgelöst, erst wenn dort keine Definition des Namens existiert, kommen u.U. Definitionen aus Basisklassen in Betracht (Ausnahmen bilden Namen *virtueller Funktionen* ↪7.2., bei denen stets der dynamische Typ den Bezugsrahmen definiert)
- jede Benutzung eines Namens im Kontext einer Klasse `x` setzt einen Bezugspunkt in Form eines Objektes voraus, der Zugang zu diesem Objekt kann über einen Ausdruck des Typs `x`, `x*` oder `x&` beschrieben sein, den Bezugspunkt innerhalb des Klassenkörpers bzw. in Definitionen von Memberfunktionen außerhalb des Klassenkörpers bildet der `this`-Zeiger
- der *statische Typ* ist stets der Typ der Deklaration des Bezugspunktes zu einem Objekt (Objektname, -zeiger oder -referenz)
- der *dynamische Typ* ist der Typ des zur Laufzeit an diesem Bezugspunkt vorliegenden Objektes
- bei Bezugnahmen über Objekte gilt stets *statischer Typ = dynamischer Typ = Objekttyp*
- bei Bezugnahmen über Zeiger/Referenzen kann der *dynamische Typ* einer beliebige (public) Ableitung des *statischen Typs entsprechen*, dies gilt auch für den `this`-Zeiger in Memberfunktionen !!!

```
class X { .... } x, *px;  
class Y: public X { .... } y, *py=new Y;  
X& rx=y;  
px=&y;
```

Objektorientierte Programmierung in C++

	statischer Typ	dynamischer Typ
x	X	X
y	Y	Y
px	X*	Y*
py	Y*	Y*
rx	X&	Y&

- sofern ein (sichtbarer) Name einer Basisklasse durch eine Redefinition in einer abgeleiteten Klasse „überdeckt“ wird, kann auf ersteren weiterhin mittels expliziter *scope resolution* Bezug genommen werden
- wird ein Name einer Memberfunktion in einer abgeleiteten Klasse redefiniert, so werden die Implementationen der verschiedenen Klassenstufen **nicht** als überladene Funktionen ausgewertet, die Zuordnung eines Aufrufs zu einer konkreten Funktion wird also klassenlokal entschieden, dies verhindert, dass unbeabsichtigte Typumwandlungen bei der Zuordnung zu einer (dem Anwender u.U. gar nicht bekannten) Funktion gleichen Namens aus einer Basisklasse stattfinden.

```
#include <iostream>
using namespace std;

class B {
public:
    int bm;
    B (int i=0) : bm (i){}
    void bf(int i){cout<<"B::bf(int="<<i<<")\n";}
    void bf(char c){cout<<"B::bf(char="<<c<<")\n";}
    void out(){cout<<"B::bm="<<bm<<"\n";}
} b, *pb;

class D: public B {
public:
    int bm;
    int dm;
    D(int i=1, int j=2): bm(i), dm(j){}
    void bf(int i){cout<<"D::bf(int="<<i<<")\n";};
    void df(double x){cout<<"D::df(double="<<x<<")\n";}
    void out()
    {
        cout<<"B::bm="<<B::bm<<"\n";
        cout<<"D::bm="<<    bm<<"\n";
        cout<<"D::dm="<<    dm<<"\n";
    }
} d;

int main(){
    b.bf(1);
    b.bf('A');
```

Objektorientierte Programmierung in C++

```
d.bf(2);
d.bf('B');
d.df(3);

pb=&d;
pb->bf(4);
pb->bf('C');

b.out();
d.out();
}
```

```
/* Ausgaben des Programms:
B::bf(int=1)
B::bf(char=A)
D::bf(int=2)
* D::bf(int=66)
D::df(double=3)
B::bf(int=4)
B::bf(char=C)
* B::bm=0
B::bm=0
D::bm=1
D::dm=2
*/
```

!



- eine (identische) Redefinition der nicht überladenen Funktion in der abgeleiteten Klasse behebt das vermeintliche Problem, ist diese `inline`, gibt es keinerlei *overhead*

```
class D{      ....
    void bf(char c){B::bf(c);}
           ....
};
....
D::bf(char=B)
....
```

6.2. Konstruktoren und Destruktoren II

Die Regel, dass kein Objekt in einem uninitialisierten Zustand ins Programmgeschehen eingreifen soll, muss natürlich auch für Objekte abgeleiteter Klassen gelten. Bevor Initialisierungen auf der Stufe der abgeleiteten Klasse stattfinden wird deshalb in jedem Fall ein passender Konstruktor jeder Basisklasse aufgerufen. Symmetrisch dazu wird nach dem Aufruf des Destruktors der abgeleiteten Klasse stets der Destruktor jeder Basisklasse aufgerufen.

Objektorientierte Programmierung in C++

- ein Konstruktor einer abgeleiteten Klasse kann Konstruktoren direkter Basisklassen (MI) explizit in seiner Initialisiererliste aufrufen; fehlen diese, wird versucht den Basisklassenanteil des Objektes mit einem parameterlosen Konstruktor zu initialisieren; falls ein solcher nicht existiert, liegt ein statischer Fehler während der Übersetzung vor !
- Die Semantik von compilergenerierten Copy-Konstruktoren (memberweise Kopie, der Basisklassenanteil wird wie ein Member im resultierenden Objekt behandelt) stellt sicher, dass Basisklassenanteile im Objekt ggf. mit nutzerdefinierten Copy-Konstruktoren der Basisklasse kopiert werden
- für nutzerdefinierte Copy-Konstruktoren ist keinerlei Semantik vordefiniert, d.h. sofern der Basisklassenanteil per Copy-Konstruktor kopiert werden soll, so ist dieser ebenfalls explizit (d.h. in der Initialisiererliste) aufzurufen !!!
- Destruktoren haben keine Parameter, daher ist ihre Verkettung implizit möglich.



```
#include <iostream>
using namespace std;

class X {
public:
    X (int) {cout<<"X::X(int)\n";}
    X (const X&) {cout<<"X::X(const X&)\n";}
    ~X() {cout<<"~X::X()\n";}
};

class Y: public X {
public:
    Y(int i):X(i) {cout<<"Y::Y(int)\n";}
    Y(const Y& y):X(y) {cout<<"Y::Y(const Y&)\n";}
    // !!! sonst wuerde X() versucht werden
    ~Y() {cout<<"~Y::Y()\n";}
};

int main(){
    X x1(1);
    Y y1(1);
    X x2=x1;
    Y y2=y1;
}
```

X::X(int)
 X::X(int)
 Y::Y(int)
 X::X(const X&)
 X::X(const X&)
 Y::Y(const Y&)
 ~Y::Y()
 ~X::X()
 ~X::X()
 ~Y::Y()
 ~X::X()
 ~X::X()

- ein „reales“ Beispiel für Vererbung: Implementation einer Klasse `CountedStack` die eine „Statistik“ über alle jemals im Keller gespeicherten `int`-Werte führt (maximaler, minimaler Wert und Durchschnitt, Durchschnitt der aktuell im Stack enthaltenen Elemente):

Objektorientierte Programmierung in C++

Offenbar kann die reine Kellerverwaltung aus der Klasse `stack` geerbt werden, neu sind nur die Daten der „Statistik“ und eine modifizierte `push`-Funktionalität

```
//-*-Mode: C++;-*-
#ifndef _CountedStack_h_
#define _CountedStack_h_

#ifdef __GNUG__
# pragma interface
#endif

#include "Stack.h"

//---- CountedStack: some statistics on pushed values -----

class CountedStack : public Stack
{
    int min, max, n, sum;
public:
    CountedStack(int dim=100);
    void push (int i);      // redefined !
    int minimum();        // new
    int maximum();        // new
    double mean();        // new
    double actual_mean(); // new
};

#endif
-----
#ifdef __GNUG__
# pragma implementation
#endif

#include "CountedStack.h"
#include <stdlib.h>

//---- CountedStack -----

CountedStack::CountedStack(int dim):Stack(dim),n(0),sum(0) {}

void CountedStack::push(int i)
{
    if (!n++) {min=max=i;}
    else      {min=(i<min)?i:min; max=(i>max)?i:max;}
    sum+=i;
    Stack::push(i); // use base functionality
}

int CountedStack::minimum(){if (n) return min; else exit(-1);}
```



....samples/stack

Objektorientierte Programmierung in C++

```
int CountedStack::maximum(){if (n) return max; else exit(-2);}

double CountedStack::mean()
{if (n) return double(sum)/n; else exit(-3);}

double CountedStack::actual_mean()
{
    if (top) { int s=0;
                for (int i=0; i<top; i++) s+=data[i];
                return double(s)/top;
            } else exit(-4);
}
```

6.3. Zugriffsschutz II

bei einer Klasse, die durch Vererbung aus einer anderen Klasse hervorgegangen ist, stellt sich die Frage nach dem Zugriffsschutz zu den Membern sowohl hinsichtlich der eigenen (neuen) Member der abgeleiteten Klasse, als auch in Bezug auf die eingerbten Member, dabei ist noch zu unterscheiden zwischen der Benutzung der Member der Basisklasse innerhalb der abgeleiteten Klasse und „von außen“

für „neue“ Member, die eine abgeleitete Klasse einführt gelten die bereits bekannten Regeln des Zugriffsschutzes (**public**, **protected**, **private**-Sektionen)

ist **class B** eine direkte Ableitung von **class A**, so sind innerhalb von **B** alle Member von **A** benutzbar, die entweder **public** oder **protected** sind.

über die Weitergabe der Zugriffsrechte auf Member einer Basisklasse an der äußeren Schnittstelle einer Ableitung entscheidet die Art der Vererbung:

```
class B: public A { .... };
```

public Member von **A** sind auch **public** Member von **B**

protected Member von **A** sind auch **protected** Member von **B**,
mit anderen Worten, die Tatsache, dass **B** ein **A** enthält, ist öffentlich

```
class B: private A { .... };
```

public/protected Member von **A** sind **private** Member von **B**

mit anderen Worten, die Tatsache, dass **B** ein **A** enthält, ist nicht öffentlich

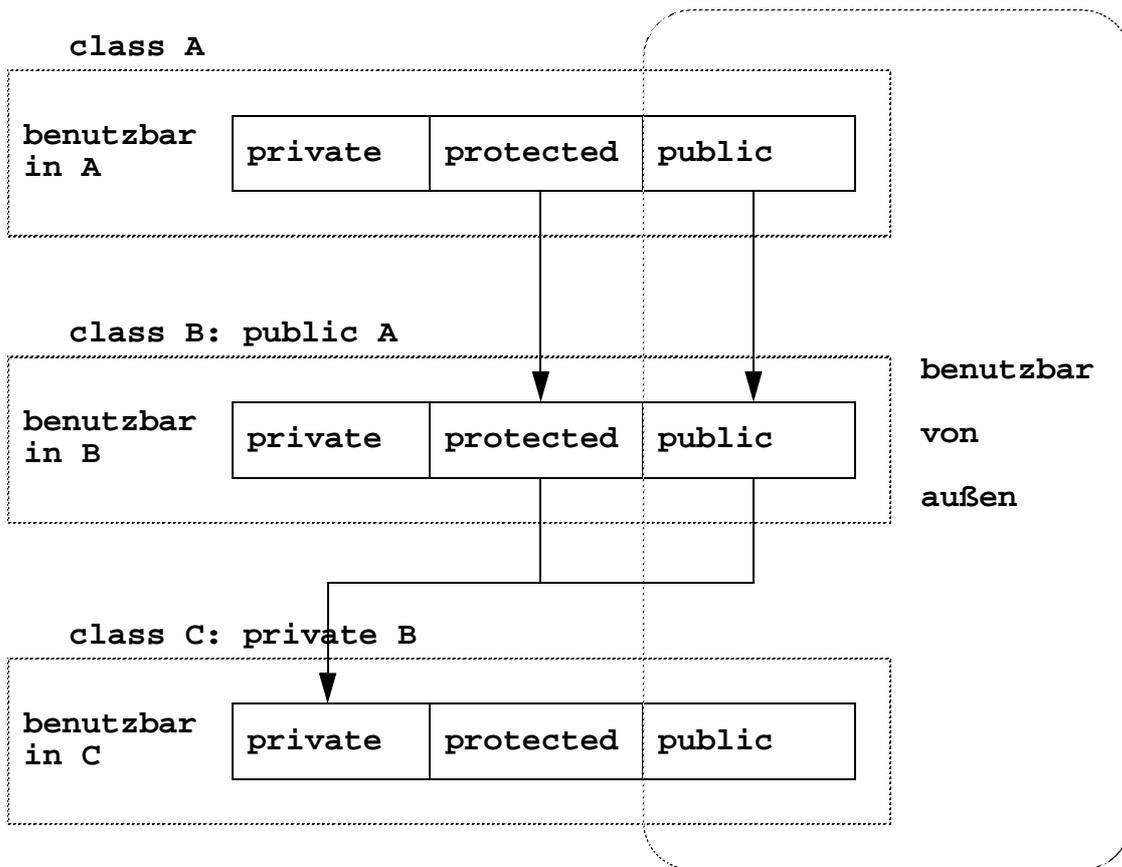
es gibt auch noch die dritte (seltener benutzte) Variante

```
class B: protected A { .... };
```

public/protected Member von **A** sind **protected** Member von **B**,

mit anderen Worten, die Tatsache, dass **B** ein **A** ist, ist nur für direkte Ableitungen von **B** bekannt

Objektorientierte Programmierung in C++



```
class A {
private:
    int a1;
protected:
    int a2;
public:
    int a3;
    int f() {return a1+a2+a3;}
}a;
```

```
class B: private A {
public:
    int f() {return a2+a3;}
}b;
```

```
class C: public A {
public:
    int f() {return a2+a3;}
}c;
```



.../samples/ppp

Objektorientierte Programmierung in C++

```
class D: protected A {
public:
    int f() {return a2+a3;}
}d;

class E: public D {
public:
    int f() {return a2+a3;}
}e;

int main()
{ // einzig erlaubte Zugriffe:
    a.a3=1;
    c.a3=1;
}
```

- eine Voreinstellung bzgl. der Art der Vererbung (**structs** erben immer **public**, **classes** erben immer **private**) wird nicht mehr unterstützt (z.T. noch als Warnung behandelt) !!
- die Weitergabe des Zugriffs auf alle Member einer Basisklasse (**public**) bzw. auf keines der Member (**private**) ist u.U. zu restriktiv, soll ein **public/protected** Member trotz **private** bzw. **protected** Vererbung in der abgeleiteten Klasse ebenfalls **public/protected** sein, so besteht die Möglichkeit dieses per sogenanntem *adjustment* in der abgeleiteten Klasse anzuzeigen, dazu ist das entsprechende Member einfach in einer entsprechenden Zugriffssektion der abgeleiteten Klasse neu aufzuführen (lediglich per Name mit *scope resolution*, ohne Typinformation!),
- ALT: dabei sind nur die ursprünglichen Rechte in der abgeleiteten Klasse rekonstruierbar, NEU: keine Einschränkung: alles was sichtbar ist kann *adjusted* werden 
- der Name eines solchen *adjusted* Member kann in der abgeleiteten Klasse nicht erneut verwendet werden,
- sofern es sich um eine überladene Funktion handelt, werden alle Implementationen erneut zugänglich, wenn diese in der Ableitung sichtbar, sonst liegt ein statischer Fehler vor

```
class A {
private:    int a1;
protected: int a2;
           void f(char){}
public:    void f(int){}
           int a3;
}a;

class B: private A {
public:
    using A::a2;    //ohne using: deprecated !!
    using A::f;    //ohne using: deprecated !!
}b; // b.a3 und b.f benutzbar
```

7. Polymorphie und Virtualität

7.1. Die IST EIN - Relation

Mit Hilfe des Vererbungsmechanismus ist es möglich, Eigenschaften vorhandener Klassen in abgeleiteten Klassen wiederzuverwenden, zu erweitern und zu modifizieren. Diese Art der Wiederverwendung ist jedoch auch mit traditionellen Kompositionsmechanismen (Verwendung von Datenstrukturen/Funktionen zur Implementation neuer) möglich.

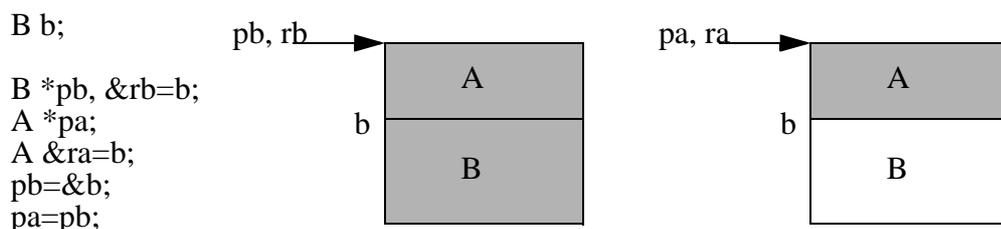
Ein entscheidende Vorzug der Vererbung besteht darin, dass es unter Umständen möglich ist, Objekte von (möglicherweise unterschiedlichen) abgeleiteten Klassen als Objekte einer gemeinsamen Basisklasse zu betrachten und zu benutzen. ==> Operationen, die auf der Basisklasse definiert sind, lassen sich so automatisch auf Objekte beliebiger Ableitungen anwenden, und zwar direkt und ohne zusätzlichen Aufwand (*wrapper* o.ä.). *Polymorphie* (Vielgestaltigkeit) ist der Schlüssel dazu.

- Besteht zwischen zwei Klassen **A** und **B** eine Vererbungslinie, bei der sämtliche Vererbungen `public` erfolgen (im einfachsten Fall also `class B: public A`)[und nur dann!!!], so **ist** jedes **B**-Objekt auch zugleich **ein** **A**-Objekt und kann demzufolge überall verwendet werden, wo **A**-Objekte erforderlich sind:

```
(1) void f(A a); .... B b; ... f(b);
(2) void f(A* pa); .... f (&b);      // Polymorphie
(3) void f(A& ra); .... f (b);      // Polymorphie
```

- Während bei (1) wegen der Parameterübergabe per *call by value* eine Kopie mittels Projektion angelegt wird (und demzufolge die Operation den Zustand des Originalobjektes unverändert lässt), wird bei (2) und (3) der direkte Zugang zum Originalobjekt bereitgestellt.
- Dies wird überhaupt erst möglich, indem in C++ die Regeln der Typkompatibilität zwischen Zeigern und Referenzen „aufgeweicht“ werden:

ein Zeiger/eine Referenz auf eine `public` Ableitung kann an einen Zeiger/eine Referenz auf eine Basisklasse zugewiesen werden (nicht umgekehrt !!!), ein solcher Zeiger/Referenz ist polymorph gebunden
aus Sicht des Objektlayout ist dies korrekt: weil jedes **B** wie ein **A** beginnt, wird über einen A-Zeiger/eine A- Referenz lediglich eine eingeschränkte Sicht auf ein Objekt freigegeben



Objektorientierte Programmierung in C++

- die bereits bekannte Möglichkeit, (`public`) Memberfunktionen der Basisklasse für Objekte der abgeleiteten Klasse aufzurufen

```
// z.B.  
CountedStack cs; ... cs.pop(); // Stack::pop !!!
```

erscheint in diesem Licht als Ausdruck der Polymorphie des `this`-Zeigers !

- Trotz der Verwandtschaft von Zeigern und Feldern ist bei der Benutzung von polymorphen Zeigern im Zusammenhang mit Objektfeldern Vorsicht geboten:

```
pa=new B[20]; // ok, pa zeigt polymorph auf das  
// erste B im Feld
```

```
pa[i] // ist sicher kein gueltiges B-Objekt !!!  
( pa[i]==(A*)((void*)p + sizeof(A)*i) )
```



7.2. Virtuelle Funktionen

Was geschieht beim Aufruf einer Memberfunktion, die sowohl in der Basisklasse, als auch in der abgeleiteten Klasse implementiert ist, wenn der Aufruf über einen polymorphen Zeiger/ eine polymorphe Referenz erfolgt ?

Beispielwelt: Geometrische Objekte in der Ebene

- Punkte: x, y Koordinaten, `class Point`
- abstrakte Figuren mit gewisser Grundfunktionalität `class Shape`:
 - ... haben einen „Ankerpunkt“
 - ... kann man zeichnen
 - ... kann man löschen
 - ... kann man bewegen: löschen, neuen Ankerpunkt setzen, zeichnen
- konkrete Figuren als Ableitung von `Shape`: `Circle`, `Box`, `Triangle`,

```
#include <iostream>  
  
class Point  
{  
protected:  
    int x, y;  
public:  
    Point (int px, int py): x(px), y(py) {};  
    int get_x() { return x; };  
    int get_y() { return y; };  
};  
  
const Point zero(0,0);
```

Objektorientierte Programmierung in C++

```
class Shape
{
protected:
    Point origin; // Ankerpunkt
public:
    Shape(const Point o=zero): origin(o) {};
    void move (const Point new_orig); // Bewegen
    void draw(); // Zeichnen
    void erase(); // Loeschen
};

void Shape::move (const Point new_orig)
{
    erase();
    origin=new_orig;
    draw();
}

void Shape::erase()
{
    std::cout << " cannot erase an abstract shape\n"; }

void Shape::draw()
{
    std::cout << " cannot draw an abstract shape\n"; }

class Box: public Shape // a Box is a Shape !!!
{
    Point dxdy; // Laenge & Breite
public:
    Box (const Point p1, const Point p2): Shape(p1),dxdy(p2) {}
    void erase() { std::cout << "erasing a box\n"; }
    void draw() { std::cout << "drawing a box\n"; }
};

class Circle: public Shape // a Circle is a Shape !!!
{
    int radius;
public:
    Circle (const Point m, int rad): Shape(m), radius(rad) {}
    void erase() { std::cout << "erasing a circle\n"; }
    void draw() { std::cout << "drawing a circle\n"; }
};

typedef int Color;
class Colored_Circle: public Circle
{
    Color col;
public:
    Colored_Circle (const Point m, int rad, Color c ):
        Circle(m, rad), col(c) {}
    void erase() {std::cout << "erasing a colored circle\n";}
    void draw() {std::cout << "drawing a colored circle\n";}
};
```



.../samples/shapes

Objektorientierte Programmierung in C++

```
void main()
{
    Box b (zero, Point (10, 10));
    Circle c (Point (10, 10), 20);
    Colored_Circle cc (Point (20, 20), 30, 1);

    b.draw();
    c.draw();
    cc.draw();

    b.move(Point(10, 10));
    c.move(zero);
    cc.move(Point(-10, -10));
}
/*
cout:
drawing a box
drawing a circle
drawing a colored circle
cannot erase an abstract shape
cannot draw an abstract shape
cannot erase an abstract shape
cannot draw an abstract shape
cannot erase an abstract shape
cannot draw an abstract shape
*/
```

- die Implementation der gemeinsamen Funktionalität von **move** auf der Ebene der Basisklasse **shape**, scheint nicht möglich zu sein: obwohl der Aufruf von **draw** und **erase** über einen polymorphen Zeiger (**this**) erfolgt, wird der Aufruf von **draw** und **erase** im aktuellen Kontext (**shape**) statisch gebunden

Auswege:

1. Redefinition von **move** im Kontext von **Box**, **Circle**, ...:

```
class Box: public Shape {
    ....
public:
    void move (const Point new_orig)
    {
        erase();
        origin=new_orig;
        draw();
    }
    ....
};
```

immer textlich gleich, aus Sicht wiederverwendbaren Codes keine akzeptable Lösung, zumal, wenn bereits **move** über einen **shape**-Zeiger gerufen wird

Objektorientierte Programmierung in C++

2. ein Mechanismus, der den Aufruf von `draw` und `erase` nicht zur Übersetzungszeit, sondern erst zur Laufzeit bindet, und zwar in Abhängigkeit von der dynamischen Qualifikation des Zeigers, über den der Aufruf erfolgt.

- Ein solcher Mechanismus wird in C++ standardmäßig mit den sog. *virtuellen Funktionen* bereitgestellt: der Aufruf einer virtuellen Funktion wird in C++ erst zur Laufzeit anhand der dynamischen Qualifikation des Zeigers/der Referenz gebunden.
- dies setzt voraus, dass zur Laufzeit noch Informationen über die möglichen „Kandidaten“ für den Aufruf bereitstehen, hier muss die Portabilität zu C aufgegeben werden
- dennoch ist die Implementation von virtuellen Funktionen nur mit einem geringen Overhead verbunden (Aufrufe von virtuellen Funktionen sind damit geringfügig aufwendiger als Aufrufe nicht-virtueller Funktionen)

```
// mit der minimalen Änderung
class Shape
{
protected:
    Point origin; // Ankerpunkt
public:
    Shape(const Point o=zero): origin(o) {};
    void move (const Point new_orig); // Bewegen
    virtual void draw(); // Zeichnen
    virtual void erase(); // Loeschen
};
// liefert main (s.o.) das erwuenschte:
/*
cout:
drawing a box
drawing a circle
drawing a colored circle
erasing a box
drawing a box
erasing a circle
drawing a circle
erasing a colored circle
drawing a colored circle
*/
```

- Grundfunktionalität alle Figuren (`move`) allgemein für beliebige Ableitungen in der Basisklasse definiert, Konkretisierung spezieller Funktionalität (`draw/erase`) virtuell mit geeigneten Reimplementationen in den Ableitungen.
- Die „Planung“ von austauschbarer Funktionalität muss in einer Basisklasse erfolgen
- eine Redefinition einer virtuellen Funktion liegt nur vor, wenn die Signatur exakt mit dem ursprünglichen Prototyp übereinstimmt



Objektorientierte Programmierung in C++

- eine Abschwächung dieser Regel ist in Sicht (DWP) und betrifft Rückgabetypen von virtuellen Funktionen, die polymorphe Zeigertypen sein können (derzeit noch nicht von allen Compilern unterstützt):

```
class X {
public:
    virtual X* clone () { return new X(*this); }
};

class Y: public X {
public:
    virtual Y* clone () { return new Y(*this); }
};

int main()
{
    X x, *px=x.clone();
    Y y, *py=y.clone();
}
```



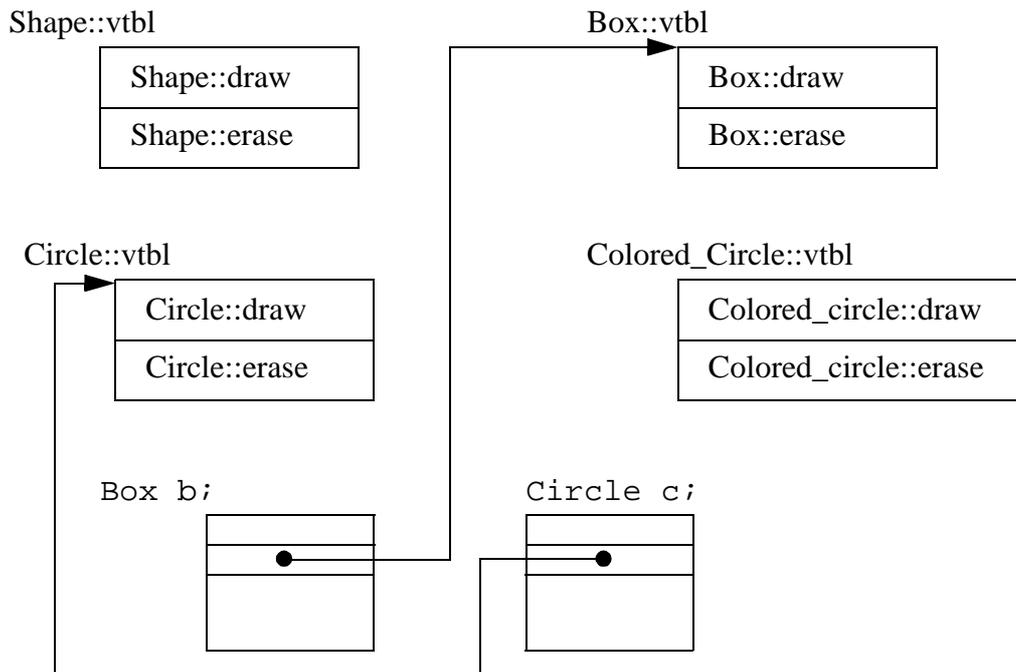
- **virtual <returntyp> fkt** oder **<returntyp> virtual fkt** sind synonym (bevorzugt 1. Variante)
- für die Benutzung virtueller Funktionen muss man ihr Implementationsschema zwar nicht zwingend kennen, es ist jedoch i.allg. hilfreich dies zu verstehen:
- Vom Compiler wird für jede Klasse, die virtuelle Funktionen besitzt (erbt) eine Tabelle mit den Eintrittspunkten dieser Funktionen berechnet, dabei wird sichergestellt, dass gleiche (redefinierte) Funktionen am gleichen Index in den VMT's (*virtual method table*) aller Klassen einer Vererbungslinie stehen, findet in einer Klasse keine Redefinition einer virtuellen Funktion statt, wird stattdessen der Eintrittspunkt der Funktionsimplementation in der nächstliegenden Basisklasse benutzt
- jedes Objekt einer Klasse mit virtuellen Funktionen erhält im Speicherlayout einen zusätzlichen (für den Benutzer unsichtbaren) Zeiger, der auf die „objektgemäße“ Tabelle verweist, dieser Zeiger hat im Objektlayout aller Klassen einer Vererbungslinie die gleiche Position !
- beim Aufruf einer virtuellen Funktion wird die aufzurufende Funktion zur Laufzeit durch eine zusätzliche Adress-Substitution (den VMT-Zeiger entlang) aus der richtigen VMT extrahiert.
- damit übersetzt der C++ Compiler Aufrufe virtueller Funktionen über polymorphe Zeiger/Referenzen sinngemäß als:

```
p->draw(); // ---> (p->vtbl[0]) (p)
```

- man spricht auch von *late binding*



Objektorientierte Programmierung in C++



- das Eintragen des richtigen Zeigers in ein Objekt ist eine implizite Aufgabe jedes Konstruktors, bei der Initialisierung eines Objektes ist seine Klassenzugehörigkeit bekannt
- Konstruktoren können daher nicht virtuell sein
- Destruktoren können (sollten) virtuell sein

```

class X {
public:
    ...
    ~X();
};
X* px = new Y;

delete px; // ruft falsch nur X::~~X() !!!
-----
class X {
public:
    ...
    virtual ~X();
};
X* px = new Y;

class Y: public X {
public:
    ...
    ~Y();
};

delete px; // ruft richtig Y::~~Y() !!!
    
```



- „einmal virtuell, immer virtuell“ (sofern die gleiche Funktion vorliegt), erneute **virtual** Deklaration eigentlich redundant, aber empfohlen



Objektorientierte Programmierung in C++

- oftmals benutzt eine Redefinition einer virtuellen Funktion die Funktionalität der unmittelbaren Basisklasse, dies hat dann mittels *scope resolution* zu erfolgen

```
class X {
public:
    virtual void do () { /* for X */...}
};

class Y: public X {
public:
    virtual void do () { X::do(); // first for X
                        /* additional for Y */ ....}
};
```

- was kein Compiler prüfen kann: jede Redefinition einer virtuellen Funktion innerhalb einer Klassenhierarchie soll im abstrakten Sinne die gleiche Operation realisieren



```
//-----
//aus: Robert B. Murray: C++ Strategies and Tactics, Add.Wesley
//-----
```

```
class Vehicle {
public:
    virtual void accelerate(double);
    double speed();
};

class Car : public Vehicle {
public:
    virtual void accelerate(double);
};

class Submarine : public Vehicle {
public:
    virtual void accelerate(double);
};

/*
conformance to the abstract model of the base class:

accelerate(x) => (speed      == speed      + x)
                  new         old

*/

void full_stop (Vehicle& v) {
    v.accelerate (-v.speed());
}
```

Objektorientierte Programmierung in C++

```
Submarine trident;
...full_stop(trident);

Car volkswagen;
...full_stop(volkswagen);

class Aircraft : public Vehicle {
public:
    virtual void accelerate(double);
};

Aircraft boeing_747;
...full_stop(boeing_747); // Hope we're on the ground

class Hot_rod : public Car {
public:
    virtual void accelerate(double);
};

void Hot_rod::accelerate(double delta) {
    // Hot_rods accelerate quickly!
    Car::accelerate(2*delta); // not conforming to the model !
}

Hot_rod ferrari; // moving at 100 km/h
...full_stop(ferrari); // 100 km/h in reverse !?
```

- über die Zugriffsrechte entscheidet bei einer virtuellen Funktion der Ort der ersten Einführung innerhalb einer Verebnungslinie, Redefinitionen sollten in der gleichen Zugriffssektion stehen
- virtuelle Funktionen können auch statisch gebunden werden:
 - wenn der Aufruf über ein Objekt erfolgt (dessen Klassenzugehörigkeit ja bereits zur Übersetzungszeit bekannt ist)
 - durch explizite *scope resolution*
 - innerhalb eines Konstruktors (um nicht vorzuschreiben, wann der Konstruktor seine implizite Aufgabe der vptr-Initialisierung vorzunehmen hat)
- eine Funktion kann sowohl als *inline* als auch virtuell spezifiziert sein, zur Laufzeit schließen sich natürlich *inline*-Aufruf und *late binding* aus.
- *virtual* und *static* schließen einander aus
- Destruktoren implizit virtuell anzulegen, bzw. generell Funktionen virtuell zu behandeln würde einerseits stets den (zwar minimalen) Laufzeitaufwand mit sich bringen und andererseits die Konsistenz zu C verletzen (structs mit virtuellen Funktionen hätten ein anderes Layout als die entsprechenden C-Structs, die nur aus den Daten bestehen), daher ist darauf verzichtet worden !



Objektorientierte Programmierung in C++

- definiert eine abgeleitete Klasse eine Funktion mit dem gleichen Namen, wie eine geerbte virtuelle Funktion, aber mit anderen Parametern, so überdeckt diese die ursprüngliche virtuelle Funktion und „unterbricht“ die Linie der Virtualität

```
#include <iostream>
class A {
public:
    virtual void f(){std::cout<<"A::f()\n";}
};
class B: public A {
public:
    void f(int=0){std::cout<<"B::f()\n";}
    // warning: B::f() hides virtual A::f()

};
class C: public B {
public:
    void f(){std::cout<<"C::f()\n";}
};
int main()
{
    B *pb;
    C c;
    pb=&c;

    pb->f();
}
```

Ausgabe: B::f()
(ohne "int=0": C::f())

7.3. abstrakte Basisklassen

- oftmals kann eine Basisklasse, die austauschbare Funktionalität einer ganzen darauf aufbauenden Klassenhierarchie mittels virtueller Funktionen definiert, noch keine eigene (echte) Funktionalität implementieren (s. **Shape::draw, Shape::erase: cannot draw/erase an abstract shape**)
- dies ist zumeist Ausdruck, der Tatsache, dass diese Klasse lediglich ein abstraktes Konzept repräsentiert, dessen konkrete Ausprägung erst in Ableitungen entsteht, von solchen Klassen sollte man keine Objekte anlegen können (z.B. ein privater Default-Konstruktor), Zeiger und Referenzen sind dagegen erlaubt (sie können polymorph auf Objekte von nicht-abstrakte Ableitungen gerichtet sein)
- C++ bietet für diese recht häufige Situation ein syntaktisches Ausdrucksmittel, die sog. *abstract base classes*: eine Klasse ist eine *abstract base class*, wenn sie mindestens eine *pure virtual function* enthält,

Objektorientierte Programmierung in C++

```
class AbstractShape { ....
public:
    virtual void draw() = 0;
    virtual void erase()= 0;
    ....
};

// no objects allowed:
AbstractShape aShape;
---> error:  declaration of object of abstract class X
pure virtual function(s) have not been defined

AbstractShape *any; // ok

any = new Circle (Point(0,0), 100);
```

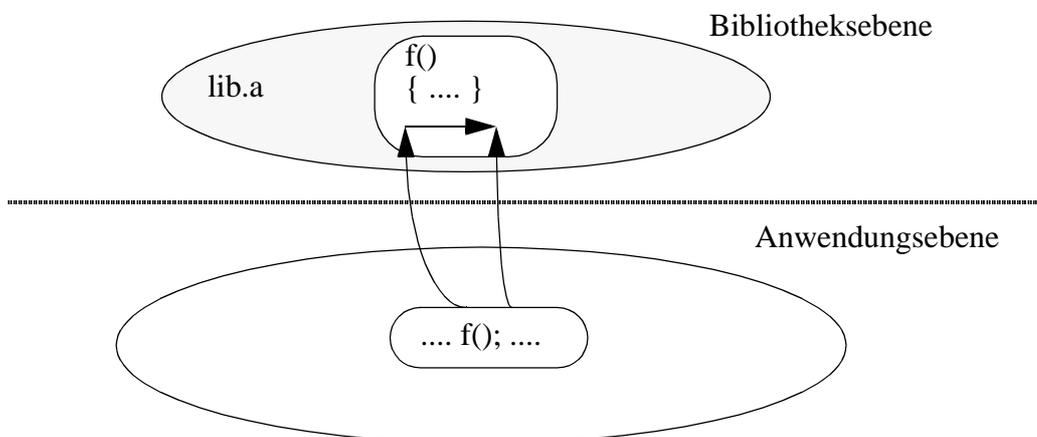
jede Klasse, die von einer *abstract base class* erbt und nicht für alle *pure virtual functions* Implementierungen bereitstellt, ist ebenfalls eine *abstract base class*

7.4. Wiederverwendbarkeit

virtuelle Funktionen ermöglichen die Zusammenfassung von Basisklassenfunktionalität in Bibliotheken, wobei auf der Anwendungsebene noch Funktionalität ausgetauscht/erweitert werden kann

- aus softwaretechnologischer Sicht begünstigt diese Möglichkeit die Schaffung von wiederverwendbaren Programmbausteinen
- traditioneller Ansatz zur Benutzung von fertigen Bibliotheken: entweder die gewünschte Funktionalität ist exakt in einer Bibliothek enthalten, oder man kann die Bibliothek nicht verwenden

traditioneller Ansatz

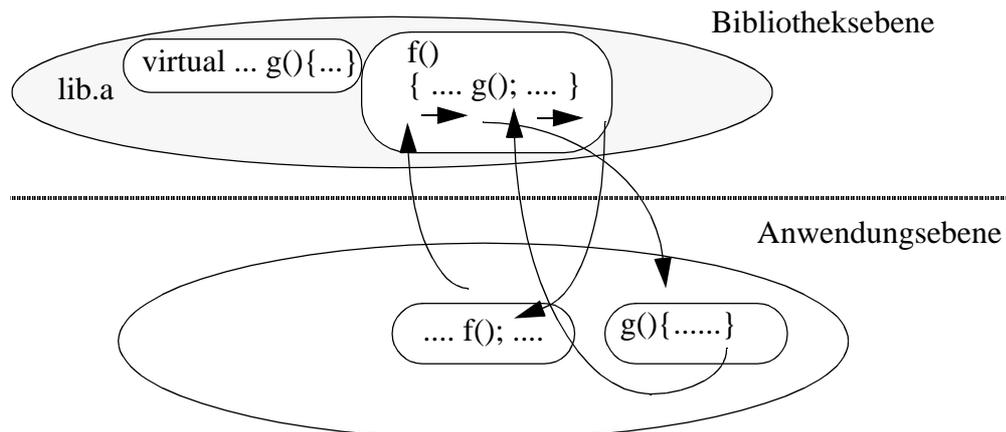


Objektorientierte Programmierung in C++

- virtuelle Funktionen erlauben es, in einem fertig übersetzten Modul noch Varianten oder Erweiterungen einzubringen ! dies kann natürlich nicht nachträglich geschehen: der Entwerfer einer Bibliotheksklasse muss die Möglichkeit zu späteren Modifikationen explizit durch Einführung entsprechender virtueller Funktionen vor-sehen, oft ist es daher üblich, in der Entwurfsphase einer neuen Bibliothek möglichst viel Funktionalität virtuell anzubieten, und dies erst später einzuschränken



objektorientierter Ansatz



7.5. Objektidentität

- der den virtuellen Funktionen gemäße Programmierstil zur Behandlung heterogener Objekte entspricht direkt dem Message-Paradigma von SmallTalk: der Aufruf einer Methode gleicht dem Senden einer Nachricht an ein Objekt, das Objekt selbst entscheidet, wie es auf diese Nachricht reagiert

```
// z.B.  
ShapeList l;  
l.add(Box(Point(0,0), Point(10,10)));  
l.add(Circle(Point(1,1), 20));  
.... // l ist heterogene Liste  
for (Shape* p=l.first(); p; p=l.next())  
    p->draw();
```

- anstelle von:

```
....  
enum Shapetypes {eBox, eCircle, /*1*/};  
for (Shape* p=l.first; p; p=l.next())  
{  
    if (p->type==eBox)        p->Box::draw();  
    if (p->type==eCircle)    p->Circle::draw();  
    /*2*/  
}
```

jede Einführung einer neuen Shape-Klasse würde hier wenigstens Quelltextänderungen bei /*1*/ und /*2*/ erforderlich machen !!!

Objektorientierte Programmierung in C++

- aus diesem Grund existierte ursprünglich in C++ (und noch heute bei manchen Implementationen) kein Mechanismus zur direkten Typabfrage, dies hat sich durch Einschluss von RTTI (*run time type identification*) (◆ 12.2.) in den Sprachstandard geändert
- RTTI kann auf der Basis von virtuellen Funktionen auch explizit implementiert werden, da virtuelle Funktionen bereits rudimentäre Typinformationen zur Laufzeit (die VMT-Zeiger in den Objekten) bereitstellen
- gebraucht wird ein solcher Mechanismus, wenn alle Objekte einer Klassenhierarchie eine gemeinsame Funktionalität aufweisen und Ableitungen einer gemeinsamen Basisklasse sind, die diese Funktionalität aber nicht bereitstellt

```
class Head {...}; // beliebige Listen über link-Objekte
class Link {...}; // Basis fuer ,listbare' Objekte

class Obst: public Link {
public: virtual void print()=0;
};

class Apfel: public Obst {
public:
    virtual void print(){cout<<"es ist ein Apfel\n";}
};
class Birne: public Obst {
public:
    virtual void print(){cout<<"es ist eine Birne\n";}
};
class Roter_apfel: public Apfel {
public:
    virtual void print(){cout<<"es ist ein roter Apfel\n";}
};

int main() {
    Head obst_schale;

    Apfel a;
    Birne b;
    Roter_apfel r;

    a.into(obst_schale);
    b.into(obst_schale);
    r.into(obst_schale);

    for (Link *p = obst_schale.first(); p!=0;
         p=p->suc())
        p->print();
    // error: no print in class Link
}
```

Objektorientierte Programmierung in C++

```
// Ausweg:  
... ((Obst*)p)->print();  
--> Problem, wenn sich auch nicht-Obst in obst-schale  
    befindet !!! (wahrscheinlich core dump)
```

- wünschenswert wäre also zumindest die Möglichkeit von „abgesicherten“ Typcasts (◆ 12.2. `dynamic_cast`)
- lässt sich auch explizit definieren, s. folgendes Beispiel: Pro Klasse wird (per Makro) ein static Member `int id`; eingeführt, die Adresse dieses id-Members ist pro Klasse eindeutig, ebenfalls per Makro eingeschleuste virtuelle (!!) Funktionen `type` und `subtype` entscheiden anhand eines Vergleichs mit dem Klassen-id bzw. rekursivem Vergleich mit Basisklassen-id's Typgleichheit, bzw. Unterklassenrelation, weitere Makros `IN` und `IS` machen diesen Test ‚handlich‘, ein Makro `IDENTIFY` sorgt für das Anlegen der static Member `id` pro Klasse, Zeigerzugriffe können leicht mit einem weiteren Makro `GUARD` geschützt werden und im Fehlerfall definiert weiterarbeiten/beenden

```
-----  
/* OBJECT.H */  
#ifndef OBJECT_H  
#define OBJECT_H  
  
#define IS(class) ->type()==&class::id  
#define IN(class) ->subtype(&class::id)  
#define BASE(base) public: virtual int* type(){return &id;} \  
                    static int id;\  
                    virtual int subtype(int *s)\  
                    {return (s==&id) || base::subtype(s);}  
  
#define IDENTIFY(class) int class::id=0;  
#define GUARD(class, pointer) ((p IN (class))?(class*)p:\  
    (class*)guard_error(#pointer, #class, __FILE__, __LINE__))  
  
struct Nothing { virtual int subtype(int*){return 0;}};  
  
class Object: public Nothing {  
    BASE(Nothing);  
};  
  
void* guard_error (const char*, const char*,  
                  const char*, const int);  
  
#endif  
-----
```

Objektorientierte Programmierung in C++

```
-----
/* OBJECT.C */
#include "object.h"
#include <cstdlib>

IDENTIFY(Object);

void* guard_error(const char* pointer, const char* classname,
const char* file, const int line)
{
    printf("cannot cast '%s' to (%s*) in %s at line %d\n",
           pointer, classname, file, line);
    exit(-999);
    return 0;
}
-----
```

- alle Klassen müssen die gemeinsame Basisklasse `object` besitzen, auf Object-Ableitungen kann man `IS` und `IN` anwenden

```
-----
/* LINK.H */
#include "object.h"

#ifndef LINK_H
#define LINK_H

class Linkage : public Object {
    BASE(Object);
protected:
    Linkage *l_pred, *l_suc;
public:
    Linkage();
    friend class Link;
    friend class Head;
};

class Link: public Linkage {
    BASE(Linkage);
public:
    Link();
    Link(class Head&);
    void into(class Head&);
    void out();
    Link* pred();
    Link* suc();
};
-----
```



...samples/rtti

Objektorientierte Programmierung in C++

```
class Head: public Linkage {
    BASE(Linkage);
    Link* first();
    Link* last();
public:
    friend Link;
    Head();
    int length();
};

#endif

-----
/* LINK.C */
Ubungsaufgabe, beginnt mit:

#include "link.h"

IDENTIFY(Linkage);
IDENTIFY(Link);
IDENTIFY(Head);
.....
-----jetzt das urspruengliche Beispiel-----
#include "object.h"
#include "link.h"
#include <iostream>

class Obst: public Link {
    BASE(Link);
public:
    virtual void print()=0;
};
class Apfel: public Obst {
    BASE(Obst);
public:
    virtual void print(){cout<<"es ist ein Apfel\n";};
};
class Birne: public Obst {
    BASE(Obst);
public:
    virtual void print(){cout<<"es ist eine Birne\n";};
};
class Roter_apfel: public Apfel {
    BASE(Apfel);
public:
    virtual void print(){cout<<"es ist ein roter Apfel\n";};
};
IDENTIFY(Obst);
IDENTIFY(Apfel);
IDENTIFY(Birne);
IDENTIFY(Roter_apfel);
```

```
int main()
{
    Head obst_schale;

    Apfel a;
    Birne b;
    Roter_apfel r;

    a.into(obst_schale);
    b.into(obst_schale);
    r.into(obst_schale);

    for (Link *p = obst_schale.first(); p!=0; p=p->suc())
        GUARD(Obst, p)->print();
}
```

- in vielen größeren C++ -Bibliotheken sind Typerkennungsmechanismen in analoger Weise, jedoch z.T. nicht miteinander kombinierbar implementiert, dies gab Anlass zur Aufnahmen von RTTI in den Sprachstandard (dort wird eine Art implizite Basisklasse aller Klassen mit virtuellen Funktionen von Compiler generiert)

8. Überladung und Konversion

8.1. Funktionen

- C++ erlaubt es, verschiedene Funktionen (auch Member-Funktionen) mit dem gleichen Bezeichner zu benennen

```
void write (int i) { //1
    std::cout << i;
}

void write (Complex c) { //2
    std::cout << c.real << "+i*" << c.imag;
} // friend von class Complex

....
write (1); // ---> 1
write (Complex(1,2)); // ---> 2
```

-

Voraussetzung dazu ist, dass sich die Funktionen in ihren Parameterlisten hinreichend unterscheiden, d.h. in Anzahl und Typen der Parameter:

char und **int** werden unterschieden !
signed / **unsigned** werden unterschieden !
const T und **T** werden unterschieden !
enum-Typen und **int** werden unterschieden !

Objektorientierte Programmierung in C++

Der Rückgabotyp reicht zur Unterscheidung verschiedener Funktionen nicht aus

```
int f();
char f();
// error: two different return value types for f(): int and char
```

kein Aufruf könnte eindeutig zugeordnet werden

- in gewissem Sinne ist die Überladung von Funktionen (u. Operatoren) auch eine Art *Polymorphie-Eigenschaft*: in Abhängigkeit von der konkreten Parametersituation in einem Aufruf wird entschieden, welche Funktionalität die angemessene ist. während Memberfunktionen auch dynamisch gebunden werden können (**virtual**), ist Überladung ein vollständig statisches Konzept, d.h. alle Zuordnungen werden zur Compilezeit hergestellt
- Überladung basiert auf dem sog. *name mangling*: Generierung von Funktionsnamen unter Einbeziehung der Parametertypen (in codierter Form)
- Sofern ein Aufruf nicht exakt typmäßig einer überladenen Funktion zugeordnet werden kann, bemüht sich der Compiler um Anpassung der Typen durch implizite Typumwandlungen, dabei kommen *built-in* Typumwandlungen (z. B. **int -> long**, **char -> int**, **int -> double**) und maximal eine nutzerdefinierte Typumwandlung (u. a. Konstruktoren [s.u.]) in Betracht es wird die Umwandlungslinie mit dem 'geringsten Aufwand' realisiert, sofern sie eindeutig ist, sonst liegt ein Fehler vor

```
int f(int i, char c){return 1;}
char f(char c, int i){return 'A';}

main()
{
    f(1, 2);
}

// error: ambiguous call: f ( int , int )
//           choice of f()s:
//           int f(int , char );
//           char f(char , int );
```

- Vorsicht bei Benutzung von default-Argumenten in überladenen Funktionen: sie können die Eindeutigkeit von Aufrufen zerstören

```
void f(int i);
int f (int i, int j=0);

// jeder Aufruf mit einem Argument ist mehrdeutig !!!
```

- Regeln für die Verwendung von Überladung (nicht durch Compiler überprüfbar):

Wird eine Funktion (ein Operator) überladen, dann sollte dies für jede denkbare (sinnvolle) Situation geschehen, um ungewollte Konvertierungen auszuschließen!

Ein Funktionsname (ein Operator) sollte nur durch solche Implementierungen überladen werden, die im abstrakten Sinne für die gleiche Operation auf verschiedenen Typen stehen !



8.2. Operatorüberladung

Durch die Verwendung von Operatoren erhalten Operationen eine kompakte und intuitiv verständliche Form, daher besteht in C++ die Möglichkeit auch Operatoren für neue Typen zu überladen

```
class Matrix {
    ....
};

Matrix m1, m2, m3;
....
m1=m2*m3; // jeder weiss, was gemeint ist !!
// besser als:
// m1=Matrix_mult (m2, m3); oder
// m1=m1.mult(m2); o.ä.
```

Operatoren besitzen allerdings vordefinierte Semantik, Signatur, Priorität und Assoziativität

- Die Semantik von Operatoren kann nutzerdefiniert überladen werden, nicht dagegen Signatur, Priorität und Assoziativität
- Es ist nicht möglich, neue Operatoren einzuführen (** %\$@#)
- Überladbar sind die folgenden Operatoren:

[]	()	->	++	--	&	*	+
-	~	!	/	%	<<	>>	<
>	<=	>=	==	!=	^		&&
	=	*=	/=	%=	+=	-=	<<=
>>=	&=	^=	=	,	new	delete	

- nicht überladbar sind dagegen

. .* :: ?:

die vordefinierte Semantik von Operatoren für *built in* -Typen muss erhalten bleiben:

Objektorientierte Programmierung in C++

```
// falsch:
// int operator+ (int i, int j) {return i-j;}
```

Ein Operator kann nur dann überladen werden, wenn in seiner Definition mindestens ein Parameter von einem Klassentyp ist (dies kann auch das implizite `this`-Argument sein) !

Operatoren können als Member einer Klasse oder als globale Funktionen implementiert sein, die dann meist **friend** einer Klasse sind.

	unäre Operatoren	binäre Operatoren
Member	<pre>class X { public: <typ> operator@ (); ...}; X x; @x; // Ergebnis: <typ> // ---> x.operator@();</pre>	<pre>class X { public: <typ1> operator@ (<typ2> arg); ...}; X x; <typ2> y x@y; // Ergebnis: <typ1> // ---> x.operator@(y);</pre>
Friend	<pre>class X { public: friend <typ> operator@ (X[&] p); ...}; X x; @x; // Ergebnis: <typ> // ---> operator@(x);</pre>	<pre>class X { //2 Varianten public: friend <typ1> operator@ (X[&] p, <typ2> arg); friend <typ1> operator\$ (<typ2> arg, X[&] p); ...}; X x; <typ2> y; x@y; // Ergebnis: <typ1> // ---> operator@(x, y); y\$x; // Ergebnis: <typ1> // ---> operator@(y, x);</pre>

jeweils nur eine Variante ist möglich (sonst Mehrdeutigkeit)

bei Memberfunktionen muss stets der erste Operand von einem Klassentyp sein !

bei der Realisierung als **friend** werden alle Operanden ggf. einer Typkonvertierung unterworfen, bei Member-Realisierung der erste Operand **nicht** !

dies kann u.U. zur Priorisierung einer der Varianten führen:

```
// BESSER IST FRIEND BEI:
class Complex {
  double real, imag;
public:
  Complex (double re=0, double im=0): real(re), imag(im){}
  Complex operator+(Complex& z)
  { return Complex(real+z.real, imag+z.imag); }
};
```

Objektorientierte Programmierung in C++

```
Complex z1, z2, z3;
...
z1=z2+z3;    // ok: z2.operator+(z3);
z2=z1+3;     // ok: z1.operator+(Complex(double(3)));
// aber:
// z2=3+z1;  // falsch !!! 3.operator+(z1) ???

// bessere Variante:
class Complex {
...
    friend Complex operator+(Complex& z1, Complex& z2)
    {    return Complex(z1.real+z2.real, z1.imag+z2.imag);}
};
...
z2=3+z1;     // ok: operator+(3, z1);

// BESSER IST MEMBER BEI:

class Direction { /* no details */ ...};

class Vector {
public:
    Vector (const Direction&, double magnitude = 0.0);
    friend Vector operator- (const Vector&);
};
// ---> Umwandlung Direction >>> Vector !!!

Direction d;
Vector v = -d;
// Vector::Vector(operator-(Vector::Vector(d,0.0))); !!!

// bessere Variante:
class Vector {
...
    Vector operator- () const;
};
....
Vector v = -d; // Compile time error !!!
```

generelle Empfehlungen (nach R.B. Murray: "C++ Strategies and Tactics"):

Operator	beste Variante
alle einstelligen	Member
= () [] ->	müssen Member sein
alle der Form @=	Member
alle anderen zweistelligen	Friend

Objektorientierte Programmierung in C++

```
#include <cstdlib>
#include <iostream>
using std::cout;

void vec_error(const char* msg) {
    cerr<<"error (vec): "<< msg <<"\n";
    exit(-1);
}

class Vector {
    int dim;
    double *data;
public:
    Vector (int size=1, int val=0);
    Vector (const Vector& v);
    ~Vector ();
    int size();
    double& operator[](int index);
    double operator[](int index) const;
    Vector& operator-();
    Vector operator-() const;
    Vector& operator()(int c); // const-Variante sinnlos
    Vector& operator()(Vector& v); // const-Variante sinnlos
    double operator*(Vector& v);
    Vector& operator= (const Vector& v);
    void print() const;
};

Vector::Vector (int size, int val) : dim (size) {
    cout<<"constructing a Vector at "<<this<<"\n";
    data = new double[size];
    for (int i=0; i<size; i++) data[i]=val;
}

Vector::Vector (const Vector& v) {
    cout<<"constructing (copying) a Vector at "<<this<<"\n";
    dim = v.dim;
    data = new double[dim];
    for (int i=0; i<dim; i++) data[i]=v.data[i];
}

Vector::~~Vector () {
    cout<<"destructing a Vector at "<<this<<"\n";
    delete [] data;
}

int Vector::size() { return dim;}

double& Vector::operator[](int index) {
    cout<<"operator[](int)\n";
```



...samples/vector

Objektorientierte Programmierung in C++

```
    if ((0 <= index) && (index < dim)) return data[index];
    else vec_error("bad index");
}

double Vector::operator[](int index) const {
    cout<<"operator[](int) const\n";
    if ((0 <= index) && (index < dim)) return data[index];
    else vec_error("bad index");
}

Vector& Vector::operator-() {
    cout<<"operator-()\n";
    for (int i=0; i<dim; i++) data[i]=-data[i];
    return *this;
}

Vector Vector::operator-() const {
    cout<<"operator-() const\n";
    Vector v(*this);
    for (int i=0; i<dim; i++) v.data[i]=-v.data[i];
    return v;
}

Vector& Vector::operator()(int c) {
    cout<<"operator()(int)\n";
    for (int i=0; i<dim; i++) data[i]=c;
    return *this;
}

Vector& Vector::operator()(Vector& v) {
    cout<<"operator()(Vector&)\n";
    if (dim==v.dim)
        for (int i=0; i<dim; i++) data[i]=v.data[i];
    else vec_error("bad copy");
    return *this;
}

double Vector::operator*(Vector& v) {
    // Skalarprodukt
    if(dim==v.dim) {
        double sum=0;
        for (int i=0; i<dim; i++) sum+=data[i]*v.data[i];
        return sum;
    }
    else {
        vec_error("bad dimension");
        return 0;
    }
}
```

Objektorientierte Programmierung in C++

```
Vector& Vector::operator= (const Vector& v) {
    int i;
    cout<<"assigning a Vector\n";
    if (&v==this) return *this; // !!!
    if (dim==v.dim)
        for (i=0; i<dim; i++) data[i]=v.data[i];
    else vec_error("bad assignment");
    return *this;
}

void Vector::print() const {
    cout << "[" << data[0];
    for (int i=1; i<dim; i++) cout << "," << data[i];
    cout << "]\n";
}

int main() {
// initialization
    Vector v1(10);           // constructing a Vector
    Vector v2=10;           // constructing a Vector
    Vector v3=v1;           // copying a Vector
    Vector v4(v3);          // copying a Vector

// assignments
    v1(10);                 // operator()(int)
    v2=10;                  // constructing & assigning a Vector
    v3=v1;                  // assigning a Vector
    v4(v3);                 // operator()(Vector&)

    v1.print();
    v2.print();
    v3.print();
    v4.print();

    for (int i=0; i< 10; i++) v1[i] = i;
    v4=v3=v2=v1;

    v1.print();
    v2.print();
    cout<<"v1*v2="<<v1*v2<<"\n";
    v3.print();
    v4.print();
    -v4;
    v4.print();

    const Vector v5(10,11);
    cout<<v5[4]<<endl;
    Vector v6(10);

    v6=-v5;
}
```

Objektorientierte Programmierung in C++

```
v5.print();
v6.print();

v1[10]=1;
// error (vec): bad index
}
// Ausgaben:
constructing a Vector at 0xeffff97c
constructing a Vector at 0xeffff974
constructing (copying) a Vector at 0xeffff96c
constructing (copying) a Vector at 0xeffff964
operator()(int)
constructing a Vector at 0xeffff95c
assigning a Vector
assigning a Vector
operator()(Vector&)
[10,10,10,10,10,10,10,10,10,10]
[0,0,0,0,0,0,0,0,0,0]
[10,10,10,10,10,10,10,10,10,10]
[10,10,10,10,10,10,10,10,10,10]
destructing a Vector at 0xeffff95c
operator[](int)
..... 8 x
operator[](int)
assigning a Vector
assigning a Vector
assigning a Vector
[0,1,2,3,4,5,6,7,8,9]
[0,1,2,3,4,5,6,7,8,9]
v1*v2=285
[0,1,2,3,4,5,6,7,8,9]
[0,1,2,3,4,5,6,7,8,9]
operator-()
[-0,-1,-2,-3,-4,-5,-6,-7,-8,-9]
constructing a Vector at 0xeffff950
operator[](int) const
11
constructing a Vector at 0xeffff948
operator-() const
constructing (copying) a Vector at 0xeffff940
assigning a Vector
[11,11,11,11,11,11,11,11,11,11]
[-11,-11,-11,-11,-11,-11,-11,-11,-11,-11]
operator[](int)

// stderr: error (vec): bad index
```

Objektorientierte Programmierung in C++

- Bis auf den Zuweisungsoperator werden alle Operatoren vererbt, d.h. die Operatoren können auch auf Objekte (`public`) abgeleiteter Klassen angewendet werden !
- Es gibt keinerlei Annahmen über die Semantik von überladenen Operatoren !!! (es gilt z.B. nicht: $\mathbf{x@=y} \lll \mathbf{x=x@y}$)
- Priorität & Assoziativität dagegen sind fest.
- Man sollte ein solches Operatorsymbol wählen, welches intuitiv und anschaulich die damit verknüpfte Operation beschreibt, und welches in den häufigsten Anwendungen ungeklammert benutzt werden kann.
- die kanonische Signatur des Zuweisungsoperators ist

```
<class_name>& operator= (const <class_name>&);
```

die komplette Semantik von "Zuweisung" ist nutzerdefiniert zu implementieren, incl. Zuweisung von enthaltenen Objekten bzw. Basisklassenbestandteilen

```
#include <iostream>
class A {
public:
    A& operator= (const A&) {
        std::cout<<"A::operator=(const A&)\n";
        return *this;
    }
};

class B: public A {
public:
    B& operator= (const B& b) {
        A* thisA = (A*) this; // Zuweisen des A-Anteils
        *thisA = b;
    // oder A::operator=(b); //egal, ob A::operator= definiert ist
        std::cout<<"B::operator=(const B&)\n";
        return *this;
    }
};

main()
{
    A a1, a2;
    B b1, b2;

    a1=a2;
    b1=b2;
}
```

Objektorientierte Programmierung in C++

- Operatoren können auch virtuell sein:

```
#include <iostream>

class X {
    int xi;
public:
    virtual int operator+(int i){return xi+i;}
    X(int i):xi(i){}
};

class Y: public X {
    int yi;
public:
    virtual int operator+(int i){return yi+i;}
    Y(int i):X(0), yi(i){}
};

int f(X& x) {return x+2; }
main(){
    X x(3);
    Y y(4);
    f(x); // 5
    f(y); // 6
}
```

- Die in `iostream` benutzten Operatoren `<<` und `>>` sind ebenfalls durch Operatorüberladung für die meisten *built in* -Typen definiert.

Eine Erweiterung auf nutzerdefinierte Typen ist möglich:

- Member-Implementation scheidet aus: müsste in `ostream` stattfinden
- bleibt die `friend`-Variante, z.B.

```
class Complex {
    ...
public:
    friend ostream& operator<< (ostream& o, const Complex& z){
        return o << z.re << "+i*" << z.im ;
    }
};
```

- Der Operator `()` kann als (einziger) mehrstelliger Operator angesehen werden:

```
class X { public:
    <typ> operator()
        (<typ1> p1, <typ2> p2, ....., <typn> pn);
};

X x;
<typ1> o1;
```

Objektorientierte Programmierung in C++

```
<typ2> o2;  
....  
<typn> on;  
  
x(o1, o2, ....., on);
```

Syntaktisch verhält sich ein Objekt einer Klasse mit überladenem () wie eine Funktion ---> Idiom der *functional objects*

- Bei den einstelligen Operatoren ++ und -- ist eine Unterscheidung möglich:

```
#include <iostream>  
using namespace std;  
class X {  
    int i;  
public:  
    X(int p):i(p){}  
    X& operator++(){cout<<"prefix\n"; i++; return *this;}  
    X operator++(int){cout<<"postfix\n"; X x=*this; i++;  
                    return x; }  
    void operator=(int p){i=p;}  
    friend ostream& operator<< (ostream& o, const X& x) {  
        return o<<x.i<<endl;  
    }  
};  
  
main()                                1  
{                                     prefix  
    X x=1;                              2  
                                         2  
    cout << x;                          postfix  
    cout << ++x << x ;                   2  
    cout << x++ << x ;                   3  
}
```

- Auch die Operatoren **new** und **delete** können überladen werden:

Neben der Möglichkeit, **new** und **delete** pro Klasse mit eigener Semantik zu implementieren, kann zunächst die globale Bedeutung von **new** und **delete** überladen werden (dies betrifft dann alle dynamischen Speicheroperationen beliebiger Typen)

```
// :: new, delete Überladen:  
void* operator new (size_t [, weitere Parameter]);  
void operator delete (void* [, weitere Parameter]);
```

der erste Parameter von **new** wird implizit vom Compiler bereitgestellt:

Objektorientierte Programmierung in C++

```
new T; // ----> ::new(sizeof(T));
```

Beispiel: Initialisierung dynamischer Speicherbereiche mit 0; Trace-Verfolgung

```
void* operator new (size_t size, const char* info = 0)
{
// BEWARE OF:
//           cout<<"new called\n";
// because construction of static object cout
// could call ::new itself !!!!!!!!
    if (info)
        printf("%s\n", info); // noch besser write
    void* p=calloc(size, 1);
    if (!p) fprintf(stderr, "\nHeap Overflow\n");// dito
    return p;
}

void operator delete (void* p, const char* info = 0)
{
    if (info)
        printf("%s\n", info);
    if (p) free((char*)p);
}

....
X* p = new ("Allocating an X") X (ctorparml,...,ctorparmn);
delete ("Deleting an X") p;
```

- Die Überladung von **new** und **delete** in einer Klasse kommt nur bei der Dynamischen Allokierung/Deallokierung von Objekten des entsprechenden Typs zur Anwendung.

Kann z.B. davon Gebrauch machen, dass alle Objekte eines Typs gleich groß sind -
--> Objektspeicher-Recycling, oder den Aufwand zur Allokierung vieler kleiner
Objekte reduzieren, z.B.

```
#include <cstdlib>
#include <cstdio>

#define TRACE
#define size_t unsigned long int
const N=128;

struct mem{ mem* next;};

void* operator new (size_t size)
{
// BEWARE OF: cout<<"new called\n";
// because construction of static object cout
// calls new itself !!!!!!!!
```



samples/new

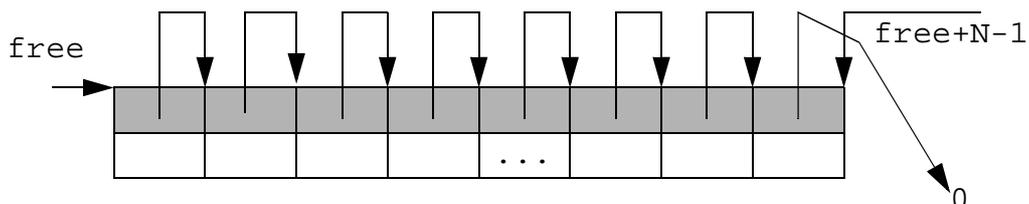
Objektorientierte Programmierung in C++

```
#ifdef TRACE
    printf("::new called, ordering %d bytes\n", size);
#endif
void* p=calloc(size, 1);
if (!p) fprintf(stderr, "\nHeap Overflow\n");
return p;
}

void operator delete (void* p)
{
#ifdef TRACE
    printf("::delete called\n");
#endif
    if (p) free((char*)p);
}

class small {
    static small* free;
    int i, j, k; // and possibly more
    static int count;

public:
    void* operator new (size_t s){
#ifdef TRACE
        printf("small::new called, ordering %d bytes\n", s);
#endif
        small* p;
        if (!free)
        {
            free=(small*)::new char [s*N];
// BEWARE OF:      .....= ::new small[N];
//                would call N 'small' constructors !!!
            for (p=free; p<free+N-1; p++)
                ((mem*)p)->next=(mem*)p+1;
            ((mem*)(free+N-1))->next=0;
        }
        p=free;
        free=(small*)((mem*)free)->next;
        return p;
    }
}
```



Objektorientierte Programmierung in C++

```
void* operator new[] (size_t s){
#ifdef TRACE
    printf("small::new[] called, ordering %d bytes\n", s);
#endif
    small* p;
    p=(small*)::new char [s];
// BEWARE OF:      .....= ::new small[N];
//                would call N 'small' constructors !!!
    return p;
}

void operator delete (void* p) // should be small*
{
#ifdef TRACE
    printf("small::delete called\n");
#endif
    if(p)
    {
        ((mem*)p)->next=(mem*)free;
        free=(small*)p;
    }
}

void operator delete[] (void* p) // should be small*
{
#ifdef TRACE
    printf("small::delete[] called\n");
#endif
    if(p)
        ::free(p);
}

small(){count++;}
~small(){--count;};

static void n()
    {printf("there are %d small-objects now\n", count);}
};

small* small::free=0;
int    small::count=0;

class a_little_bigger: public small { int l; };

main()
{
    printf("vvvvvvvvvvvvvvvvvvvvvvvvvvvv\n"
    small::n();
    small* s=new small;
    small::n();
```


Objektorientierte Programmierung in C++

```
inline void *operator new(size_t, void *place)
{ return place; }

inline void *operator new[](size_t, void *place)
{ return place; }

...
void* at = <irgendwoher>;

X *px = new (at) X;
```

In früheren Sprachversionen (<= C++ 1.2) gab es dafür die Möglichkeit, in einem Konstruktor eine Zuweisung an `this` vorzunehmen. Dies wird nicht mehr unterstützt.

8.3. Typumwandlungen und Casts

Konstruktoren die (ggf. durch Substitution von *default*-Argumenten) mit einem Argument aufrufbar sind, beschreiben nutzerdefinierte Typumwandlungen von Argumenttyp zum entsprechenden Klassentyp,

diese werden bei Bedarf implizit aufgerufen !!!

```
#include <iostream>

class X {
public:
    X (int, double=0);           // int --> X
    X (char*, int=1, int=2);    // char* --> X
    X (X&){};
    X operator+ (X){};
};

void f (X) { };
X g() { std::cout<<"X g()\n"; return 1; } // 1 --> X

class Y {
public:
    Y(X){ std::cout<<"Y(X)\n";}; // X --> Y
};

main(){
    X x1 = 1;                    // 1 --> X
    X x2 = "ein_X";              // "ein_X" --> X
    f(2);                        // 2 --> X
    x2 = x1 + 3;                 // 3 --> X
    Y y = 0; // Fehler: nicht int --> X --> Y
}
```

Objektorientierte Programmierung in C++

es kommt **maximal eine nutzerdefinierte** Typumwandlung zum Einsatz !!!

- mit dem (neuen) Schlüsselwort `explicit` kann man implizite Konstruktoraufrufe verhindern !

```
class X { ...
    explicit X (int, double=0);    // int --> X
    ...
};

f(2);        // Fehler: keine implizite Umwandlung int --> X
f(X(2));    // ok
```

- Ziel der Umwandlung per Konstruktor ist immer ein Klassentyp
- Neben Konstruktoren gibt es die sog. Konversionsfunktionen, bei diesen kann Ziel der Umwandlung ein beliebiger Typ sein, Quelle ist stets ein Klassentyp
- Sie werden als Memberfunktionen in Anlehnung an die Operatornotation definiert:

```
class X {
public:
    operator int() { return 1; };
};
```

- KEIN Rückgabotyp, KEINE Argumente
- Konversionsfunktionen sind normalerweise mit einem Informationsverlust verbunden !
- Auch hier muss jeder Aufruf eindeutig zuzuordnen sein:

```
class Y: public X {
public:
    operator char() { return 'A'; };
};

void main() {
    Y y;
    if (y) { }; // mehrdeutig: Y-->int und Y-->char
}
```

- Typumwandlung per Konstruktor und per Konversionsfunktion sind gleichberechtigt !

```
#include <iostream>
using namespace std;

class B {
public:
    operator int(){cout<<"operator int()\n"; return 1;}
};
```

Objektorientierte Programmierung in C++

```
class C {
public:
    C(B) {cout<<"C::C(B)\n";}
};

C operator +(C c1, C c2)
{cout<<"operator+ (C, C)\n"; return c1;}

C f( B a, B b) {
    return a+b;
}

main() {
    B b1, b2;
    f(b1, b2); // Fehler: Mehrdeutigkeit int+int oder C+C
}
```

- Ziel der Umwandlung kann ein beliebiger Typ (auch Zeigertyp) sein:

```
#include <iostream>
using namespace std;
class X {
public:
    virtual operator const char*() { return "X";}
};
class Y: public X {
    operator const char*() { return "Y";}
};
void main() {
    X* p = new Y;
    cout << *p << '\n'; // --> gibt aus: "Y"
}
```

- Nutzerdefinierte Typumwandlungen finden natürlich auch bei expliziten Typ-Casts statt
- Konversionsfunktionen lassen sich nicht per **explicit** "abschalten", sie sind eigens für die Typumwandlung gedacht und kommen bei Bedarf auch stets zum Einsatz, Ausweg:

```
class X{
public:
    // statt:
    // operator int();
    int to_int(); // muss explizit gerufen werden !
};
```

- Achtung: Typumwandlungsfunktionen agieren immer auf Objekten !! Im obigen Beispiel wären die beiden folgenden Konstruktionen korrekt, jedoch mit völlig verschiedener Semantik:

Objektorientierte Programmierung in C++

```
(char*) p ..... die Adresse des Objektes (als String !?)  
(char*)*p ..... der String "Y"
```

- manchmal kann auch `operator void*()` sinnvoll sein, vom Objekt bleibt dabei natürlich (fast) nichts übrig:

```
// aus iostream.h: hier noch in der alten Version  
// in iostream sinngemäß genauso...  
inline ios::operator void*()  
{ return fail()? 0 : this; } // very tricky  
  
// ios ist Basisklasse von ostream, ofstream  
// z.B.  
  
ofstream output ("file.txt");  
// wenn beim Einrichten des Files irgendein Problem auftrat  
// liefert fail() ---> true  
if (!output) cerr<< "cannot open output file\n";  
// ! (void*) output !!!!!
```

8.4. Ein-/Ausgabe

http://www.dinkumware.com/htm_cpl/index.html

<file:/vol/knecht-vol3/DOC5.0/lib/locale/C/html/stdlib/index.html>

9. Mehrfachvererbung

9.1. virtuelle und nicht virtuelle Basisklassen

bislang hat jede abgeleitete Klasse genau eine direkte Basisklasse

im Interesse der freien Kombinierbarkeit von Konzepten (Klassen) ist es jedoch auch sinnvoll, Eigenschaften von mehr als einer Basisklasse in eine neue Klasse aufzunehmen

- C++ unterstützt seit der Sprachversion 2.0 (Cfront 2.0) das Konzept der sog. Mehrfachvererbung (eigentlich müsste es "Mehrfacheinerbung" heißen) *multiple inheritance* : eine Klasse kann von beliebig vielen anderen Klassen direkt abgeleitet werden

```
class Combined: public Concept1,  
               public Concept2, private Concept3 {  
    ....  
};
```

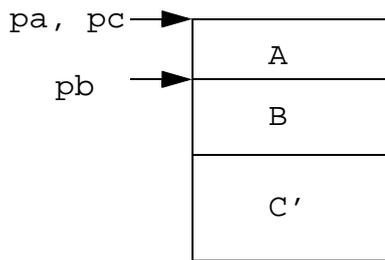
ein `Combined`-Objekt ist ein `Concept1`-Objekt und ein `Concept2`-Objekt und verfügt über die Eigenschaften eines `Concept3`-Objektes

Objektorientierte Programmierung in C++

insbesondere ist die Polymorphieeigenschaft zwischen **Combined** und **Concept1/2** gegeben, die Unterstützung dieser Eigenschaft erfordert jedoch zusätzliche Aktionen durch den Compiler:

während bei Einfachvererbung ein Objekt der abgeleiteten Klasse stets so beginnt, wie ein Objekt der Basisklasse, kann dies bei Mehrfachvererbung nicht mehr der Fall sein, Polymorphie ist also u.U. mit impliziten Adressumrechnungen verbunden

```
class C: public A, public B {...};
```



```
C c; A *pa; B *pb; C* pc;
pc=&c; pa=pc; pb=pc;
// nicht pa=pb=pc; !
pa==pb !!!
pb==pc !!!
```

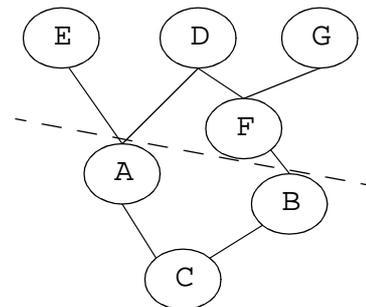
obwohl
(void*)pa !=(void*)pb
(ausser bei pa=0)

eine Klasse kann eine andere nicht direkt mehrfach erben:

```
// Fehler: class B: public A, public A {...};
```

jedoch kann bei der freien Kombination von Klassen nicht ausgeschlossen werden, dass diese eine gemeinsame Basisklasse besitzen:

```
class A: public D, public E {...};
class B: public F {...};
class F: public D, public G {...};
-----
class C: public A, public B {...};
```



wird eine Klasse mehrfach eingerebt, so kann man steuern, ob im resultierenden Objekt diese Klassenkomponente mehrfach oder einfach enthalten ist

dies hat auf der Ebene der direkten Ableitung von der mehrfachen Basisklasse zu erfolgen !! (d.h. bereits der Entwerfer der "Mittlerklasse" muss sich im Klaren darüber sein, wie die von ihm benutzten Basisklassen später in resultierenden Klassen behandelt werden sollen)

beide Verfahrensweisen sind sinnvoll, es hängt von der Intention des Implementators ab:

Objektorientierte Programmierung in C++

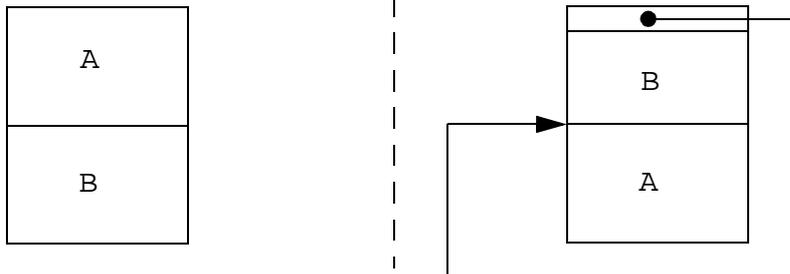
```
// class Listable { /* Listeneigenschaften */ };  
class A: public Listable { ... };  
// A's können in einer Liste erfasst werden  
class B: public Listable { ... };  
// B's können in einer Liste erfasst werden  
  
class C: public A, public B { ... };  
// C's können in zwei separaten Listen (als A und als B)  
// erfasst werden
```

```
-----  
  
class Person {...};  
class Angestellter: public virtual Person {...};  
class Student: public virtual Person {...};
```

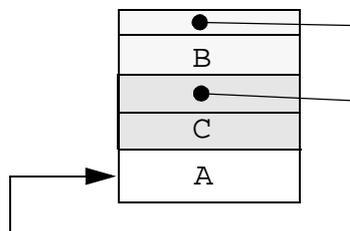
```
class Werkstudent: public Angestellter, public Student  
{...}; // ein und dieselbe Person !!!
```

virtuelle Basisklassen werden mit Hilfe eines veränderten Objektlayouts realisiert,
(nicht mehr zu C kompatibel !!!) z. B.:

```
class B: public A ... ; | class B: public virtual A ... ;
```



```
-----  
  
class C: public virtual A ... ;  
class D: public B, public C ...;
```



9.2. Mehrdeutigkeiten

Mehrdeutigkeiten entstehen bei der freien Kombination von Klassen mit Hilfe der Mehrfachvererbung relativ schnell

zu unterscheiden sind

Objektorientierte Programmierung in C++

1. Situationen die immer auf Mehrdeutigkeiten hinauslaufen (die man nicht umgehen kann) und die deshalb verboten sind und
2. Situationen in denen Mehrdeutigkeiten explizit aufgelöst werden können (und die deshalb erlaubt sind)

- Zur Kategorie 1 gehört der Versuch eine Klasse, die indirekt und nicht-virtuell eingeebt wird, auch noch direkt einzuerben (eine Klasse kann eine Basisklasse nur dann sowohl indirekt als auch direkt erben, wenn alle Vererbungen virtuell sind! Liegt die indirekte Vererbung bereits als nicht-virtuell vor, kann die Basisklasse nicht direkt eingeebt werden.)

```
class A {public: int i;};
class B: public A{};
class C: public A, public B {}; // ist fehlerhaft
// i ... welches i ? A::i ... welches A::i ?
```

- eine "Mittlerklasse" kann die Situation klären (in die Kategorie 2 überführen):

```
class A {public: int i;};
class B: public A{};
class D: public A{};
class C: public D, public B {}; // ist ok
// i mehrdeutig, aber B::i und D::i eindeutig
```

- Bei virtuellen Basisklassen kann es auf Grund der Maschenbildung sein, dass ein Name eines Member-Datums / einer Member-Funktion auf mehreren "Wegen" auflösbar ist, Eindeutigkeit liegt dann vor, wenn es (genau) einen kürzesten Weg gibt »*Dominanzregel*« (ansonsten muss ebenfalls qualifiziert werden):

```
#include <iostream>

class A {
public:
    void f(){std::cout<<"A::f()\n";}
};

class B: public virtual A {
public:
    void f(){std::cout<<"B::f()\n";}
};

class C: public virtual A{};

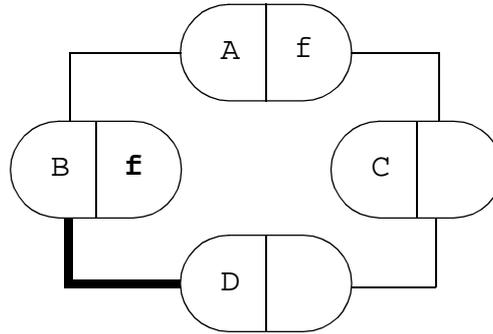
class D: public B, public C {};

main() {
    D d;
    d.f();
    d.A::f();
}
```

Objektorientierte Programmierung in C++

```
B *pb = &d;
pb->f();
C *pc = &d;
pc->f();
}
```

```
/*
B::f()
A::f()
B::f()
A::f()
*/
```



- Mehrfachvererbung und virtuelle Funktionen sind miteinander kombinierbar, im Falle von virtuellen Basisklassen stehen u.U. ebenfalls mehrere Wege der Auflösung zur Verfügung: falls dabei kein kürzester Weg existiert, ist eine Mehrdeutigkeit weder durch Aufruf über entsprechende Zeiger noch durch *scope resolution* - (statische Bindung!) lösbar, stattdessen muss dann in der am weitesten abgeleiteten Klasse eine Redefinition erfolgen:

```
#include <iostream>
```

```
class A {
public:
    virtual void f(){std::cout<<"A::f()\n";}
};
```

```
class B: public virtual A {
public:
    void f(){std::cout<<"B::f()\n";}
};
```

```
class C: public virtual A{
public:
    /*1*/ void f(){std::cout<<"C::f()\n";}
};
```

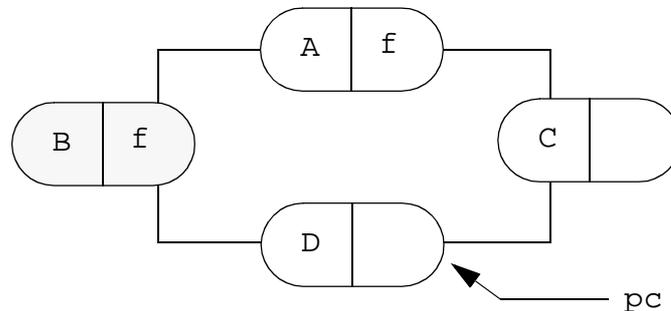
```
class D: public B, public C {
public:
    /*2*/ void f(){std::cout<<"C::f()\n";} // oder A::f oder B::f
};
```

```
main()
{
    D d;

    C *pc = &d;
    pc->f();
}
```

Objektorientierte Programmierung in C++

```
/*  
ohne 1 und 2: B::f dominant ---> B::f()  
mit 1 ohne 2: error: virtual base: ambiguous B::f() and C::f()  
mit 1 und 2: D::f eindeutig ---> C::f()  
*/
```



9.3. Konstruktoren und Destruktoren III

die Einführung virtueller Basisklassen hat Auswirkungen auf den Ablauf der Konstruktion und Destruktion von entsprechenden Objekten:

- virtuelle Basisklassenbestandteile einer Klasse werden vor allen anderen Basisklassenkomponenten konstruiert (und zwar nur einmalig) und nach allen anderen »destruiert«
- damit muss der Konstruktor für virtuelle Basisklassenkomponenten in der am weitesten abgeleiteten Klasse explizit aufgerufen werden (es sei denn es existiert ein parameterloser Konstruktor der virtuellen Basisklasse, dann wird dieser implizit gerufen)
- damit ist der vollständige Ablauf der Initialisierung von Objekten der folgende:
 1. besitzt die Klasse X (eine) virtuelle Basisklasse(n) und ist X der Ausgangspunkt des Initialisierungsvorgangs, so wird (werden) zunächst der(die) Konstruktor(en) der virtuellen Basisklasse(n). Die Reihenfolge wird (i.allg.) durch die textliche Reihenfolge der Klassen bestimmt.
 2. Alle Konstruktoren nicht-virtueller Basisklassen werden (ebenfalls in der textlichen Reihenfolge ihres Auftretens bei der Vererbung) aufgerufen. Bringen diese Basisklassen ihrerseits virtuelle Basisklassen ein, so wird deren erneute Konstruktion unterbunden, selbst, wenn im Konstruktorquelltext ein Aufruf der Konstruktors der virtuellen Basisklasse (in der Initialisiererliste) steht
 3. Die Konstruktoren von Member-Daten mit Klassentyp werden ebenfalls in der textlichen Reihenfolge der Member aufgerufen.
 4. Jetzt erst wird der Konstruktorkörper der abgeleiteten Klasse abgearbeitet.

Objektorientierte Programmierung in C++

```
#include <cstring>
#include <iostream>
using namespace std;

class Person {
protected:
    char* Name;
    char* Vorname;
    long Versicherungsnummer;
public:
    Person      (const char* vorname,
                 const char* name,
                 long vnummer):
        Versicherungsnummer(vnummer) {
        Name=new char[strlen(name)+1];
        strcpy(Name, name);
        Vorname=new char[strlen(vorname)+1];
        strcpy(Vorname, vorname);
    };
    virtual void print () { cout<<Vorname<<' '<<Name<<
                            " ist eine Person"<<endl;}
};

class Angestellter: public virtual Person {
protected:
    double Gehalt;
    // ....
public:
    Angestellter(const char* vorname,
                 const char* name,
                 long vnummer,
                 double gehalt):
        Person(vorname, name, vnummer),
        Gehalt(gehalt) {}
    virtual void print () { cout<<Vorname<<' '<<Name<<
                            " ist ein Angestellter"<<endl;}
    virtual double einkommen(){ return Gehalt; }
};

class Student: public virtual Person {
protected:
    double Stipendium;
    // ....
public:
    Student      (const char* vorname,
                 const char* name,
                 long vnummer,
                 double stip):
        Person(vorname, name, vnummer),
        Stipendium(stip) {}
};
```



...samples/multi

Objektorientierte Programmierung in C++

```
virtual void print () { cout<<Vorname<<' '<<Name<<
                        " ist ein Student"<<endl;}
virtual double einkommen(){ return Stipendium; }
};

class Werkstudent: public Student, public Angestellter {
public:
    Werkstudent      (const char* vorname,
                      const char* name,
                      long vnummer,
                      double stip,
                      double gehalt):
        Person(vorname, name, vnummer),
        Angestellter(vorname, name, vnummer, gehalt),
        Student(vorname, name, vnummer, stip) {}

    virtual void print () { cout<<Vorname<<' '<<Name<<
                            " ist ein Werkstudent"<<endl;}
    virtual double einkommen()
        { return Gehalt+Stipendium; }
};

void main() {
    Angestellter *pa;
    Student      *ps;
    Werkstudent  *pw;
    Person        *pp;
    Werkstudent martin ("Martin", "Schmidt",
                       123456789, 700, 900);

    pw=&martin;
    pa=pw;
    ps=pw;
    pp=pw;    // erlaubt, da virtuelle Basisklasse
    pw->print(); pa->print();
    ps->print(); pp->print();
    pw=(Werkstudent*)pa; // es ist ein Werkstudent !
    pw=(Werkstudent*)ps; // es ist ein Werkstudent !
    // pw=(Werkstudent*)pp; // nicht erlaubt
    // pa=(Angestellter*)pp; // nicht erlaubt
    // ----> dynamic_cast !!!
    pw->print();
}
/* 5x "Martin Schmidt ist ein Werkstudent" */
```

Typcasts von einer virtuellen Basisklasse zu einer Ableitung sind nicht erlaubt "*to avoid requiring an implementation to maintain pointers to enclosing objects*" (ARM p. 227)

9.4. Zugriffsschutz III

Auch bei Mehrfachvererbung wird die Frage nach Mehrdeutigkeiten unabhängig von Zugriffsrechten entschieden. Erst nachdem Eindeutigkeit gesichert ist, wird geprüft, ob der Zugriff überhaupt erlaubt ist! (vgl. Überladung & Zugriffsrechte)

```
class A {
private:
    void m();
};

class B {
public:
    void m();
};

class C: public A, public B {
void f() {
// m(); // Fehler: Mehrdeutigkeit
A::m(); // Fehler: kein Zugriff
B::m(); // ok
}
};
```

Wird eine virtuelle Basisklasse sowohl **private** als auch **public** vererbt, so dominiert **public** !! Bei nicht virtueller Vererbung gilt für jedes Auftreten einer Basisklasse das Zugriffsrecht entsprechend der direkten Vererbung

```
class A { public: int i; };
class B: private virtual A {};
class C: public virtual A {};
class D: public B, public C {
    void f() {
        i++; // erlaubt, da B: .... public A
    };
};

-----

class A { public: int i; };
class B: private A {};
class C: public A {};
class D: public B, public C {
    void f() {
//      i++; // Fehler: Mehrdeutigkeit
        C::i++; // ok
//      B::i++; // Fehler: kein Zugriff
    };
};
```

10. Templates

Vererbung erlaubt die Wiederverwendung von Programmcode, virtuelle Funktionen ermöglichen dabei den Austausch gewisser Funktionalität

häufig gibt es jedoch den Wunsch nach Wiederverwendung durch Parametrisierung von Klassen, z.B. `class Stack_of_int{...};` und `class Stack_of_double{...};` unterscheiden sich lediglich in der Benutzung von 'int' bzw. 'double' bei den Zugriffen auf die Elemente, alle Algorithmen sind identisch aufgebaut (agieren jedoch auf verschieden aufgebauten Objekten)

dies lässt sich nicht mittels Vererbung ausdrücken !!!

10.1. generische Klassen

erster und ursprünglich einziger Ausweg war die Benutzung des Präprozessors zur Erzeugung geeigneter Klassen aus Makros

```
// Beispiel: genStack.h
#include <generic.h>
/* u.a.:
    #define name2(X, Y) X ## Y
    #define declare(clas,t)      name2(clas,declare)(t)
    #define implement(clas,t)   name2(clas,implement)(t)
*/

#define Stack(TYP) name2(Stack_of_, TYP)

#define Stackdeclare(TYP) class Stack(TYP) \
{ \
    protected: \
        TYP *data; \
        int top, max; \
    public: \
        Stack(TYP)(int dim=100); \
        ~Stack(TYP)(); \
        void push (TYP i); \
        TYP pop(); \
} // ; bei Benutzung

#define Stackimplement(TYP) \
Stack(TYP)::Stack(TYP)(int dim): \
max(dim), top(0), data(new TYP[max]) { } \
void Stack(TYP)::push (TYP i) \
{ \
    data[top++]=i; \
} \

```


 .../samples/generics

Objektorientierte Programmierung in C++

```
Stack(TYP)::~Stack(TYP)() \
{ \
    delete [] data; \
} \
TYP Stack(TYP)::pop () \
{ \
    return data[--top]; \
}
```

```
-----

// Anwendung: gsapp.cc:
#include <iostream.h>
#include "genStack.h"

declare (Stack, double);
declare (Stack, int);

implement (Stack, double);
implement (Stack,int);
Stack_of_int s1;
Stack_of_double s2;

int main()
{
    s1.push(1);
    s2.push(2.0);
    cout << s1.pop() << endl << s2.pop() << endl;
}
```

```
/*
declare (Stack, int)
\
--> name2(Stack, declare)(int)
\
---> Stackdeclare(int)
\
--> class Stack(int) {...}
\
--> class Stack_of_int { ... }
*/
```

Vorteil: hohe Flexibilität

Nachteile: Unübersichtlich, Fehler sind schwer zu lokalisieren, keine Typprüfung

10.2. parametrisierte Klassen

- seit Sprachversion 3.0 gibt es die Möglichkeit, Klassen aus vorgefertigten Schablonen (*Templates*) unter Regie des Compilers zu erzeugen:--> typsicher, debuggungsfähig, effizientere Instanzierung z.B. mittels compilereigener 'Datenbasis' über bereits existierende und übersetzte Instanzierungen
- Parameter können sowohl Typen als auch Werte (eines festen Typs) sein
- eine Template-Klasse definiert ein allgemeines Muster für beliebig viele konkrete Klassen, aus dem Template selbst lässt sich kein Code generieren, dies geschieht erst bei der sog. Instanzierung der Template-Klasse
- es bleibt nach wie vor eine Operation über Quelltexten, eine echte Codeeinsparung ist nicht erreichbar (pro Instanzierung muss ein eigenes Objektfile erzeugt werden)

```
// Ein generisches Stack-Template:
template <class TYP, int DIM>
class Stack {
protected:
    TYP *data;
    int top, max;
public:
    Stack(int dim=DIM);
    ~Stack();
    void push (TYP i);
    TYP pop();
};

template <class TYP, int DIM>
Stack<TYP,DIM>::Stack(int dim):
max(dim), top(0), data(new TYP[max]) { }
template <class TYP, int DIM>
void Stack<TYP,DIM>::push (TYP i)
{
    data[top++]=i;
}

template <class TYP, int DIM>
Stack<TYP,DIM>::~~Stack ()
{
    delete [] data;
}

template <class TYP, int DIM>
TYP Stack<TYP,DIM>::pop ()
{
    return data[--top];
}
```



.../samples/generics

Objektorientierte Programmierung in C++

```
-----  
  
// Anwendung: tsapp.cc  
#include <iostream>  
using namespace std;  
  
#include "StackTmpl.h"  
  
#ifdef __GNUG__  
template class Stack<int, 10>;  
template class Stack<double, 20>;  
template class Stack<Stack<int, 10>, 10>;  
#endif  
  
Stack<int, 10> s1, s2;  
Stack<double, 20> s3;  
Stack<Stack<int, 10>, 10> ss;  
  
int main()  
{  
    s1.push(1);  
    s2.push(2);  
    ss.push(s1);  
    ss.push(s2);  
    s3.push(3.0);  
  
    cout << ss.pop().pop() << endl  
         << ss.pop().pop() << endl  
         << s3.pop() << endl;  
}
```

- die Instanziierung einer konkreten Klasse geschieht auf explizite Anforderung des Nutzers, eine solche kann jedoch weitere Instanzierungen nach sich ziehen, Instanziierung ist als aufwendiger Prozess bekannt, insbesondere muss verhindert werden, dass bei separaten Übersetzungsläufen mehrmalige Instanzierungen vorgenommen werden
- die Sequenz `<class xyz ...>` bedeutet nicht, dass nur mit Klassentypen instanziiert werden darf, beliebige Typen sind möglich, vielmehr kündigt `class` einen *'Meta-(Typ-)Parameter'* an
- Voreinstellungen für Template-Parameter sollen möglich sein, werden jedoch derzeit von keinem (hier verfügbaren) Compiler unterstützt, auch *member templates* nicht:

```
template <class TYP = int, int DIM = 20>  
class Stack ....
```

10.3. Funktionstemplates

im Kontext der Template-Klassen entstehen zugleich auch Schablonen für (Member-)Funktionen, auch für globale Funktionen ist es sinnvoll, aus Mustern konkrete Funktionen abzuleiten, C++ gestattet dies

```
template <class T>
void swap (T& a, T& b)
{
    T tmp = a;
    a=b;
    b= tmp;
}

// kann beliebige Objekte 'austauschen'
// Benutzung z.B.:

template <class T>
void simple_sort (T* vec, int size)
{
    for (int m=0; m<size-1; m++)
        for (int n=m+1; n<size; n++)
            if (vec[m]>vec[n]) swap(vec[m], vec[n]);
}

int v [100]; ....
simple_sort (v, 100);
```

- Instanzen von Funktionstemplates werden implizit erzeugt, sobald eine entsprechende Funktion "gebraucht" wird (auch bei g++)
- Instanzen können auch explizit angegeben werden `simple_sort<int> (...)`
- bei der Instanzierung benutzte Operatoren müssen für den entsprechenden Typ verfügbar sein

```
class X {
// operator < not defined
public:
    operator int () {return 1; }
};

X x [20];
simple_sort (x, 20);
/* Error: The operation "X > X" is illegal.
   Where: While instantiating simple_sort(X*, int).
   Where: Instantiated from non-template code.
*/
```

Objektorientierte Programmierung in C++

- die Zuordnung eines Funktionsaufrufes zu einer aus einem Funktionstemplate instanzierbaren Funktion erfolgt nur bei exakter Typübereinstimmung (keine Typumwandlungen werden vorgenommen)

```
template <class T>
inline const T& max (const T& a, const T& b)
{ return a>b ? a : b; }

int i=max (1, 2);           // ok: max (int, int)
double d=max (1.0, 2.0);   // ok: max (double, double)
int n=max(1, 2.0);         // Fehler
double x=max(1.0, 2);      // Fehler

int k=max(x[0], 2);        // sollte falsch sein
                           // CC merkt nichts, g++ schon
```

- Funktionstemplates beschreiben gewissermaßen unendlich oft überladene Funktionen
- ist für eine spezielle Parametersituation eine Funktion explizit definiert, so überdeckt diese eine Template-Instanzierung

```
inline const int& max (const int &a, const double& b)
{
    return max(a, int(b));
}
```

- das "Instanzierungsproblem" wird schnell "mehrdimensional":

```
// STL: pair.h
#ifndef PAIR_H
#define PAIR_H

#ifdef __GNUG__
#include <bool.h>
#endif
template <class T1, class T2>
struct pair {
    T1 first;
    T2 second;
    pair() {}
    pair(const T1& a, const T2& b) : first(a), second(b) {}
};

template <class T1, class T2>
inline bool operator==
(const pair<T1, T2>& x, const pair<T1, T2>& y) {
    return x.first == y.first && x.second == y.second;
}
```

Objektorientierte Programmierung in C++

```
template <class T1, class T2>
inline bool operator<
(const pair<T1, T2>& x, const pair<T1, T2>& y) {
    return x.first < y.first || (!(y.first < x.first) &&
        x.second < y.second);
}

template <class T1, class T2>
inline pair<T1, T2> make_pair(const T1& x, const T2& y) {
    return pair<T1, T2>(x, y);
}
#endif
```

11. Standard Template Library (STL)

- eine »unkonventionelle« C++ -Bibliothek, bestehend aus Headerfiles, die massiven Gebrauch von Templates machen (also keine fertig übersetzte *.a/lib/dll)!
- STL impliziert eine "Paradigmenänderung" im Umgang mit Headerfiles, inzwischen vom C++-Standard aufgegriffen:

```
// braucht man assoziative Felder so:
#include <map>

// derzeit noch Koexistenz mit dem *.h -Ansatz per
// /usr/local/lib/gcc-lib/sparc/2.7.2/g++-include/map:
// -*- C++ -*- forwarding header.
// This file is part of the GNU ANSI C++ Library.

#ifndef __MAP__
#define __MAP__
#include <map.h>
#endif
```

- Teil der C++ -Standardbibliothek:
die folgenden 51 Standard C++ headers (einschließliche der 18 zusätzlichen Standard C headers) machen eine sog. "hosted implementation of Standard C++" aus:

```
<algorithm>, <bitset>, <cassert>, <cctype>, <cerrno>, <cfloat>,
<ciso646>, <climits>, <locale>, <cmath>, <complex>, <csetjmp>,
<csignal>, <cstdarg>, <cstddef>, <cstdio>, <cstdlib>,
<cstring>, <ctime>, <wchar>, <wctype>, <deque>, <exception>,
<fstream>, <functional>, <iomanip>, <ios>, <iosfwd>, <iostream>,
<istream>, <iterator>, <limits>, <list>, <locale>,
<map>, <memory>, <new>, <numeric>, <ostream>, <queue>, <set>,
<sstream>, <stack>, <stdexcept>, <streambuf>, <string>, <strstream>,
<typeinfo>, <utility>, <valarray>, <vector>.
```

Objektorientierte Programmierung in C++

- ursprünglich unter HP-Copyright entwickelt, mittlerweile zum Standard erklärt und in ersten Implementationen verfügbar, freie Implementation (mit Erweiterungen) von SiliconGraphics (<http://www.sgi.com/Technology/STL/index.html>) (z.B. ObjectSpace (kommerziell): STL<ToolKit>, >=g++ 2.7.2 GNU publ. lic. [basiert auf SGI STL], Visual C++ 5.0 [P.J. Plaucher: Dinkumware])
http://www.dinkumware.com/htm_stl/index.html ist eine der besten Dokumentationen zur Verwendung der C++ -Standardbibliothek
- auf den ersten Blick im Widerspruch zu einer der Grundgedanken der objektorientierten Programmierung: Trennung von Datenstrukturen und Algorithmen
- stellt elementare Containerklassen für Listen, Vektoren, Mengen, Multimengen, Stacks, Assoziationen ... unter einem einheitlichen Blickwinkel bereit: über allen Typen sind sog. Iteratoren definiert, die sich verhalten, wie Zeiger in die entsprechenden Strukturen, kanonisch in Fortsetzung von C-Feldern und C-Zeigern) Zeiger eignen sich sogar als elementare Iteratoren
- der Vorteil dieser Herangehensweise ist die Möglichkeit der Implementation "generischer Algorithmen" --> Algorithmen (Suchen, Ordnen, Einfügen, Mischen ...) die unabhängig von den Containerklassen und ihren Basistypen implementiert werden können
- Beobachtung: Algorithmen sind abstrakt, in dem Sinne, dass sie unabhängig sind von den elementaren Typen sind, auf denen sie operieren:

```
// Beispiel: Lineares Suchen
int* find (int* first, int* last, int value) {
    while (first != last && *first != value)
        ++first;
    return first;
}
// geht genauso mit double*, XYZ*, ...

// Abstraktion vom elementaren Typ und Reduzierung auf
// Konzepte von "Anfang" "Ende", "Vergleich" ...
template <class InputIterator, class T>
InputIterator find
(InputIterator first, InputIterator last, const T& value) {
    while (first != last && *first != value)
        ++first;
    return first;
}
```

- ob eine Klasse den Anforderungen als konkreter Template-Parameter entspricht, entscheidet der Compiler bei der Instanzierung: alle benutzten Operationen müssen von der Klasse bereitgestellt werden, die Namen der formalen Template-Parameter (wie **InputIterator**) deuten zwar auf semantische Eigenschaften hin, es gibt jedoch kein syntaktisches Ausdrucksmittel für die abstrakte Darstellung dieser Eigenschaften !



Objektorientierte Programmierung in C++

- Anforderungen an C++ -Compiler:

1. Default-Template-Parameter:

```
template <class T, class container = vector<T> >
class Beispiel;
```

2. Schlüsselwort **typename** (und überhaupt lokale Typen):

```
template <class T> class Beispiel {
    typename T::TYP t; ...
    // Typ T hat einen lokalen Typ namens TYP
};
```

// typename auch als Synonym für class bei Template-
// Parametern verwendbar:

```
template <typename T> class Beispiel {
    typename T::TYP t; ...
};
```

3. Template-Memberfunktionen:

```
#include <iostream>
```

```
class X {
public:
    template <typename T>
    static void f(T t) {cout << t << endl;}
};
```

```
main()
{
    Ausgabe:
    X::f(1);           1
    X::f(1.0);        1
    X::f("1");        1
    X::f('1');        1
}
```

auch non-static, aber nicht virtuell !!!

wird insbesondere benutzt, um Typkompatibilität verwandter "Template-Typen" zu ermöglichen:

Objektorientierte Programmierung in C++

```
#include <stdio.h> // u.a. mit tendra übersetzt, eigentlich cstdio

template <typename T>
class X {
    T t;
public:
    X(T tp):t(tp){}
    template <typename T1> // ***
    X(const T1& src): t(src.val()){} // ***
    T val() const {return t;}
};

class Y {
public:
    int val() const {return 123;}
};

int main() {
    X<int> a=65;
    X<char> b='a';

    X<int> c = b; // nur mit ***
    X<char> d = a; // dito

    printf("%d\n%c\n%d\n%c\n", a.val(), b.val(), c.val(), d.val());

    Y y;
    X<int> e = y; // nur mit ***
    printf("%d\n", e.val());
}
```

Ausgaben:

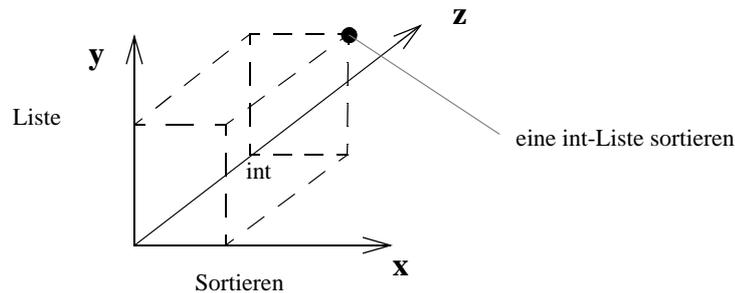
65
a
97
A
123

- Per Container gibt es lokale Typvereinbarungen, die die jeweiligen Spezifika des Containers "kennen", nach außen sich jedoch gleich als abstrakte Konzepte verhalten:

```
namespace std { // g++ kann namespace noch nicht richtig
                // sie werden mit einem Macro ausgeblendet
    template <class T> // (fast) aus <vector>
    class vector {
    public:
        typedef T value_type;
        typedef value_type* pointer;
        typedef const value_type* const_pointer;
        typedef value_type* iterator;
        typedef const value_type* const_iterator;
        typedef value_type& reference;
        typedef const value_type& const_reference;
        typedef size_t size_type;
        typedef ptrdiff_t difference_type;
        ....
    };
} // end namespace std
```

Objektorientierte Programmierung in C++

"Algorithmus **x** angewendet auf Containertyp **y** des Basistyps **z** "



Traditioneller Ansatz: m (Algorithmen) * n (Containertypen) * t (Basistypen)

Templates in ihrer primären Anwendung: m (Algorithmen) * n (template<t> Container)

Ansatz der STL: die Algorithmen agieren auf austauschbaren (instanzierbaren)
Zugriffsoperationen ---> **m (Algorithmen) + n (template<t> Container)**

- Iteratoren entkoppeln die Containertypen von den Algorithmen, die auf ihnen operieren
- ein Iteratorwert verkörpert eine Position, kann mittels **++** weitergesetzt werden
ACHTUNG: man benutze immer Prefix-In/Dekrement (keine temporäre Variable)

Iteratoren können auf Un-/Gleichheit geprüft werden

* liefert das referenzierte Element (wie bei Zeigern)

(Input-)Iteratorobjekte können kopiert werden !

- Beispiele zur Benutzung der STL:

```
////////////////////samples/stl/stl1.cc////////////////////  
#include <vector>  
#include <algorithm>  
#include <iostream>  
using namespace std;  
  
class IntOut {  
public:  
    void operator()(int v) {cout<<v<<' ';}  
};  
  
int main() {  
    vector<int> l; // ein leerer int-Vektor  
    int val;  
    cin >> val;  
    while (val>0) {  
        l.push_back(val); // hinten anhaengen  
    }  
}
```

Objektorientierte Programmierung in C++

```
        cin >> val;
    }
    // Vektor ausgeben:
    for_each(l.begin(), l.end(), IntOut()); cout<<endl;
        // generische Iteratoren
    // oder auch so:
    copy(l.begin(),l.end(),
        ostream_iterator<int>(cout," ") ); cout<<endl;
} // Vektoren sind unsortiert

/*
template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last,
                 Function f) {
    for ( ; first != last; ++first)
        f(*first);
    return f;
}
*/

//////////samples/stl/stl2.cc//////////
#include <set>
#include <algorithm>
#include <iostream>
using namespace std;
class IntOut {... wie oben ...};

int main() {
    set<int> l; // eine leerer int-Menge
    int val;

    cin >> val;
    while (val>0) {
        l.insert(val); // push_back gibts nicht
        cin >> val;
    }
    // Menge ausgeben:
    for_each(l.begin(), l.end(), IntOut()); cout<<endl;
} // Mengen sind sortiert und einhalten ein Elements
// hoechstens einmal

//////////samples/stl/stl3.cc//////////
#include <algorithm>
#include <iostream>

using namespace std;
class IntOut {...};

int main() {
    int s[10]; // ein C-array von ints
```

Objektorientierte Programmierung in C++

```
int val, i=0;
cin >> val;
while (val>0) {
    s[i++]=val;
    cin >> val;
}
// array ausgeben:
for_each(s, s+i, IntOut()); // Zeiger sind Iteratoren !
cout<<endl;
}
```

```
//////////samples/stl/stl4.cc//////////
```

```
#include <list>
#include <set>
#include <algorithm>
#include <iostream>
class IntOut {...};
```

```
int main() {
    set<int> s; // eine leere int-Menge
    int val;

    cin >> val;
    while (val>0) {
        s.insert(val); // einfuegen, keine kopien !
        cin >> val;
    }
    // Menge ausgeben:
    for_each(s.begin(), s.end(), IntOut()); cout<<endl;
    // Mengen koennen nicht (um-)sortiert werden !

    // list<int> l (s); // geht nicht direkt
    list<int> l(s.begin(), s.end());

    l.reverse(); // absteigend Sortieren

    // und erneut ausgeben:
    for_each(l.begin(), l.end(), IntOut()); cout<<endl;
}
```

```
// oder gleich so
```

```
//////////samples/stl/stl5.cc//////////
```

```
#include <set>
#include <algorithm>
#include <iostream>
class IntOut {...};
```

```
int main() {
    set<int, greater<int> > s; // eine leere int-Menge
    int val; // sortiert nach >
```

Objektorientierte Programmierung in C++

```
cin >> val;
while (val>0) {
    s.insert(val); // einfuegen, keine kopien !
    cin >> val;
}
// Menge ausgeben:
for_each(s.begin(), s.end(), IntOut()); cout<<endl;
}

//////////samples/stl/stl6.cc//////////

#include <list>
#include <algorithm>
#include <iostream>
using namespace std;
class IntOut {...};

int main() {
    list<int> l(100); // eine int-Liste mit 100 Nullen
    // Liste ausgeben:
    cout<<"neu erzeugt:"<<endl;
    for_each(l.begin(), l.end(), IntOut()); cout<<endl;
    generate_n(l.begin(), 100, rand);
    // Liste ausgeben:
    cout<<endl<<"zufaellig belegt:"<<endl;
    for_each(l.begin(), l.end(), IntOut()); cout<<endl;
    l.sort(); // Sortieren als Member nur bei list
    // sort(l.begin(), l.end()); // geht nicht: kein
    // RandomAccessIterator

    // und erneut ausgeben:
    cout<<endl<<"sortiert:"<<endl;
    for_each(l.begin(), l.end(), IntOut()); cout<<endl;
}
/*
template <
    class OutputIterator,
    class Size,
    class Generator
>
OutputIterator generate_n
(OutputIterator first, Size n, Generator gen) {
    for ( ; n > 0; --n, ++first)
        *first = gen();
    return first;
}
*/
```

- **find** (s.o.) ist bereits auf verschiedene Container anwendbar, lässt sich aber in seiner "Generizität" noch verbessern:

Objektorientierte Programmierung in C++

```

template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first,
                    InputIterator last, Predicate pred) {
    while (first != last && !pred(*first)) ++first;
    return first;
}
// pred ist Verallgemeinerung von ==( . , value)

```

- vordefinierte Funktionsobjekte

Aufruf	Operation
<code>negate<typ>()</code>	<code>- param</code>
<code>plus<typ>()</code>	<code>param1 + param2</code>
<code>minus<typ>()</code>	<code>param1 - param2</code>
<code>multiplies<typ>()</code> [neu]	<code>param1 * param2</code>
<code>times<typ>()</code> [alt]	<code>param1 * param2</code>
<code>divides<typ>()</code>	<code>param1 / param2</code>
<code>modulus<typ>()</code>	<code>param1 % param2</code>
<code>equal_to<typ>()</code>	<code>param1 == param2</code>
<code>not_equal_to<typ>()</code>	<code>param1 != param2</code>
<code>less<typ>()</code>	<code>param1 < param2</code>
<code>greater<typ>()</code>	<code>param1 > param2</code>
<code>less_equal<typ>()</code>	<code>param1 <= param2</code>
<code>greater_equal<typ>()</code>	<code>param1 >= param2</code>
<code>logical_not<typ>()</code>	<code>! param</code>
<code>logical_and<typ>()</code>	<code>param1 && param2</code>
<code>logical_or<typ>()</code>	<code>param1 param2</code>

- Funktionsadaptoren

Aufruf	Operation
<code>bind1st(op, wert)</code>	<code>op (wert, param)</code>
<code>bind2st(op, wert)</code>	<code>op (param, wert)</code>
<code>not1(op)</code>	<code>! op (param)</code>
<code>not2(op)</code>	<code>! op (param1, param2)</code>

Objektorientierte Programmierung in C++

- Ein Beispiel:

```

////////////////////////////////////sample/stl/odd.cc////////////////////////////////////
#include <list>
#include <iostream>
#include <algorithm>
#include <functional>

int main() {
    list<int> liste;
    for (int i=0; i<100; i++)
        liste.push_back(i);
    liste.remove_if(not1(bind2nd(modulus<int>(), 2)));
    copy (liste.begin(), liste.end(),
          ostream_iterator<int>(cout, "\n"));
}

```

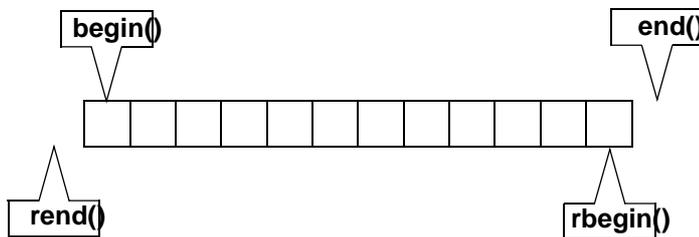
- gemeinsame (generische) Operationen aller Container-Klassen (*Josuttis: "Die C++-Standardbibliothek", Addison-Wesley 1994*)

Ausdruck	Bedeutung
Erzeugung/Zerstörung	
<code>ContType m</code>	erzeugt einen leeren Container ohne Elemente
<code>ContType m1(m2)</code>	erzeugt einen Container als Kopie eines anderen
<code>ContType m (anf, end)</code>	erzeugt einen Container und initialisiert ihn mit Kopien der Elemente aus dem Bereich [anf, end)
<code>m.~ContType()</code>	löscht alle Elemente und gibt deren Speicherplatz frei
Nichtverändernde	
<code>m.size()</code>	aktuelle Anzahl von Elementen
<code>m.empty()</code>	leer? (entspricht <code>m.size()==0</code> ABER SCHNELLER!)
<code>m.max_size()</code>	maximal mögliche Anzahl von Elementen
<code>m1 == m2</code> <code>m1 != m2</code> <code><, >, <=, >=</code>	Gleichheit (== auf alle Elemente angewendet) Ungleichheit (dito) lexikografische Ordnung
Zuweisende	
<code>m1 = m2</code>	Zuweisung
<code>m1.swap(m2)</code> <code>swap(m1, m2)</code>	Vertauschen aller Elemente !! immer schneller als <code>m1=m2</code>
Zugreifende/Iteratoren	
<code>m.begin()</code>	Iterator auf das erste Element
<code>m.end()</code>	Iterator hinter das letzte Element
<code>m.rbegin()</code>	Iterator auf das letzte Element

Objektorientierte Programmierung in C++

Ausdruck	Bedeutung
<code>m.rend()</code>	Iterator vor das erste Element
Einfügende/Löschende	
<code>m.insert(pos, elem)</code>	Kopie von elem bei pos einfügen (Rückgabewert und Bedeutung von pos hängen von Containertyp ab)
<code>m.erase (pos)</code>	löscht Element an der Position pos
<code>m.erase (anf, end)</code>	löscht alle Elemente aus dem Bereich [anf, end) liefert die Position des Folgeelements
<code>m.clear()</code>	leert den Container

- abstrakte Zeiger (Iteratoren)



- verschiedene Container besitzen (von hause aus) verschiedene Iteratoren mit unterschiedlicher Variabilität:

<code>vector</code>	<code>RandomAccessIterator</code>
<code>deque</code>	<code>RandomAccessIterator</code>
<code>list</code>	<code>BidirectionalIterator</code>
<code>set, multiset</code>	<code>BidirectionalIterator</code>
<code>map, multimap</code>	<code>BidirectionalIterator</code>

11.1. STL - Vektoren (`#include <vector>`)

dynamisches Array eines beliebigen Typs mit wahlfreiem Zugriff

Abstraktion von C-Feldern

definierte Reihenfolge der Elemente (unsortiert)

alle Algorithmen der STL sind anwendbar (RandomAccessIterator)

sehr gutes Zeitverhalten beim Löschen und Einfügen am Ende
ansonsten ist jedes Löschen/Einfügen mit dem Verschieben (Zuweisen) von Elementen verbunden ! (--> schlechtes Zeitverhalten)

Vektoren wachsen dynamisch:

<code>size()</code>	Anzahl der aktuell enthaltenen Elemente
<code>max_size()</code>	-"- der maximal möglichen Elemente (impl.abh.)
<code>capacity()</code>	-"- der noch (ohne) Reallokierung aufnehmbaren Elemente

Objektorientierte Programmierung in C++

Reallokierung = Copykonstruktor + Destruktor am alten Platz / Element !!!

(--> schlechtes Zeitverhalten)

`reserve(size_type)` Freihalten von Platz

- Spezielle `vector`-Operationen

Ausdruck	Bedeutung
Erzeugung/Zerstörung (alle generischen Operationen +)	
<code>vector<Elem> m (n)</code>	n Elem-Elemente per default-Konstruktor
<code>vector<Elem> m (n, elem)</code>	n Elem-Elemente als Kopie von elem
Nichtverändernde (alle generischen Operationen +)	
<code>m.capacity()</code>	freier Platz ohne Reallokierung
<code>m.reserve(size_type)</code>	Platz Freihalten
Zuweisende (alle generischen Operationen +)	
<code>m.assign(n)</code>	Zuweisung von n Elementen, die per default-Konstruktor (dc) erzeugt werden
<code>m.assign(n, elem)</code>	Zuweisung von n Elementen, die als Kopie von elem erzeugt werden
<code>m.assign(anf, end)</code>	Zuweisung von Kopien der Elemente aus dem Bereich [anf, end)
Zugreifende/Iteratoren (alle generischen Operationen +)	
<code>m.at(n)</code>	Element am Index n (mit Prüfung ob es existiert, ggf. <code>out_of_range</code> Exc. !)
<code>m[n]</code>	Element am Index n (ohne Prüfung ob es existiert !)
<code>f.front()</code>	das erste Element (ohne Prüfung ob es existiert!)
<code>f.back()</code>	das letzte Element (ohne Prüfung ob es existiert!)
Einfügende/Löschende (alle generischen Operationen +)	
<code>m.insert (pos)</code>	fügt dc Element an pos ein und liefert position des neuen Elements
<code>m.insert (pos,elem)</code>	fügt Element als Kopie von elem an pos ein und liefert position des neuen Elements
<code>m.insert (pos, anf, end)</code>	bei pos Kopien von [anf, end) return ??
<code>m.push_back (elem)</code>	Kopie von elem hinten anhängen
<code>m.pop_back ()</code>	löscht letztes Element (gibt es nicht zurück)

Objektorientierte Programmierung in C++

Ausdruck	Bedeutung
<code>m.resize(n)</code>	Größe auf n ändern und ggf. mit dc Elementen auffüllen
<code>m.resize (n, elem)</code>	Größe auf n ändern und ggf. mit Kopien von elem auffüllen

ACHTUNG: Einfügen/Löschen macht Verweise (Iteratoren) auf nachfolgende Elemente ungültig, falls der Platz beim Einfügen nicht reicht (implizites `resize()` werden alle Verweise ungültig !!!



- es gibt eine spezielle Implementation von dynamische Bitvektoren `vector<bool>` bei der boolesche Werte durch einzelne Bits repräsentiert werden, diese hat zusätzlich die Operationen

`m.flip()` alle Bits umklappen und
`m[n].flip()` dieses Bit umklappen

falls die Dimension eines Bitfeldes fest ist, benutzte man stattdessen `bitset`

11.2. STL - Deques ("Deck(s)") [double ended queue]

(`#include <deque>`)

nach zwei Seiten dynamisches Array eines beliebigen Typs mit wahlfreiem Zugriff

definierte Reihenfolge der Elemente (unsortiert)

alle Algorithmen der STL sind anwendbar (RandomAccessIterator)

sehr gutes Zeitverhalten beim Löschen und Einfügen am Anfang und am Ende
 ansonsten ist jedes Löschen/Einfügen mit dem Verschieben (Zuweisen) von Elementen verbunden ! (--> schlechtes Zeitverhalten)

Dequees werden i.allg. in mehreren Speicherblöcken (anders als Vektoren) automatisch verwaltet:

- > es gibt keine Vorreservierung, implizite Reallokierung
- > bei jedem Einfügen werden Verweise potentiell ungültig
- > wahlfreier Zugriff ist zwar möglich aber langsamer als bei Vektoren

- Spezielle Deque-Operationen

Ausdruck	Bedeutung
Erzeugung/Zerstörung (alle generischen Operationen +)	
<code>deque<Elem> m (n)</code>	n Elem-Elemente per default-Konstruktor
<code>deque<Elem> m (n, elem)</code>	n Elem-Elemente als Kopie von elem

Objektorientierte Programmierung in C++

Ausdruck	Bedeutung
Zuweisende (alle generischen Operationen +)	
m.assign(n)	Zuweisung von n Elementen, die per default-Konstruktor (dc) erzeugt werden
m.assign(n, elem)	Zuweisung von n Elementen, die als Kopie von elem erzeugt werden
m.assign(anf, end)	Zuweisung von Kopien der Elemente aus dem Bereich [anf, end)
Zugreifende/Iteratoren (alle generischen Operationen +)	
m.at(n)	Element am Index n (mit Prüfung ob es existiert, ggf. out_of_range Exc. !)
m[n]	Element am Index n (ohne Prüfung ob es existiert !)
f.front()	das erste Element (ohne Prüfung ob es existiert!)
f.back()	das letzte Element (ohne Prüfung ob es existiert!)
Einfügende/Löschende (alle generischen Operationen +)	
m.insert (pos)	fügt dc Element an pos ein und liefert position des neuen Elements
m.insert (pos,elem)	fügt Element als Kopie von elem an pos ein und liefert position des neuen Elements
m.insert (pos, anf, end)	bei pos Kopien von [anf, end) return ??
m.push_front (elem)	Kopie von elem vorn anhängen
m.push_back (elem)	Kopie von elem hinten anhängen
m.pop_front ()	löscht erstes Element (gibt es nicht zurück)
m.pop_back ()	löscht letztes Element (gibt es nicht zurück)
m.resize(n)	Größe auf n ändern und ggf. mit dc Elementen auffüllen
m.resize (n, elem)	Größe auf n ändern und ggf. mit Kopien von elem auffüllen

11.3. STL - Listen (#include <list>)

doppelt verkettete Liste eines beliebigen Typs ohne wahlfreien Zugriff (man muss sich "durchhangeln")
definierte Reihenfolge der Elemente (unsortiert)

Objektorientierte Programmierung in C++

nicht alle Algorithmen der STL sind anwendbar (BidirektionalIterator)
 gutes Zeitverhalten beim Löschen und Einfügen an beliebiger Stelle !
 Verweise auf Elemente bleiben dabei gültig !!

- Spezielle List-Operationen

Ausdruck	Bedeutung
Erzeugung/Zerstörung (alle generischen Operationen +)	
<code>list<Elem> m (n)</code>	n Elem-Elemente per default-Konstruktor
<code>list<Elem> m (n, elem)</code>	n Elem-Elemente als Kopie von elem
Zuweisende (alle generischen Operationen +)	
<code>m.assign(n)</code>	Zuweisung von n Elementen, die per default-Konstruktor (dc) erzeugt werden
<code>m.assign(n, elem)</code>	Zuweisung von n Elementen, die als Kopie von elem erzeugt werden
<code>m.assign(anf, end)</code>	Zuweisung von Kopien der Elemente aus dem Bereich [anf, end)
Zugreifende/Iteratoren (alle generischen Operationen +)	
<code>f.front()</code>	das erste Element (ohne Prüfung ob es existiert!)
<code>f.back()</code>	das letzte Element (ohne Prüfung ob es existiert!)
Einfügende/Löschende (alle generischen Operationen +)	
<code>m.insert (pos)</code>	fügt dc Element an pos ein und liefert position des neuen Elements
<code>m.insert (pos,elem)</code>	fügt Element als Kopie von elem an pos ein und liefert position des neuen Elements
<code>m.insert (pos, anf, end)</code>	bei pos Kopien von [anf, end) return ??
<code>m.push_front (elem)</code>	Kopie von elem vorn anhängen
<code>m.push_back (elem)</code>	Kopie von elem hinten anhängen
<code>m.pop_front ()</code>	löscht erstes Element (gibt es nicht zurück)
<code>m.pop_back ()</code>	löscht letztes Element (gibt es nicht zurück)
<code>m.remove (elem)</code>	löscht alle! Elemente mit dem Wert elem
<code>m.remove_if(op)</code>	löscht alle! Elemente für die op(elem) wahr ist
<code>m.resize(n)</code>	Größe auf n ändern und ggf. mit dc Elementen auffüllen
<code>m.resize (n, elem)</code>	Größe auf n ändern und ggf. mit Kopien von elem auffüllen

Objektorientierte Programmierung in C++

Ausdruck	Bedeutung
Verknüpfende	
<code>m.unique ()</code>	kollabiert Folgen von Elementen mit gleichem Wert
<code>m.unique (op)</code>	kollabiert Folgen von Elementen mit <code>op(e1)==op(e2)</code>
<code>m1.splice (pos,m2)</code>	verschiebt alle Elemente von m2 nach m1 vor die Position pos
<code>m1.splice (pos,m2,m2pos)</code>	verschiebt das Elemente von m2 an der Position m2pos nach m1 vor die Position pos (m1, m2 identisch ist erlaubt)
<code>m1.splice (pos,m2,m2anf,m2end)</code>	verschiebt alle Elemente von m2 aus dem Bereich [m2anf, m2end) nach m1 vor die Position pos (m1, m2 identisch ist erlaubt)
<code>m.sort()</code>	Sortiert nach <
<code>m.sort(op)</code>	Sortiert nach op
<code>m1.merge (m2)</code>	mischt sortiertes m2 in sortiertes m1 ein
<code>m1.merge (m2,op)</code>	mischt sortiertes m2 in sortiertes m1 ein, sortiert dabei nach op
<code>m.reverse()</code>	kehrt die Reihenfolge der Elemente um

11.4. STL - Mengen und - Multimengen(`#include <set>`)

Mengencontainer mit automatischer Sortierung der Elemente

set jedes Element kommt höchstens einmal vor
multiset Elemente können mehrfach enthalten sein (Bags)

wegen der automatischen Sortierung muss für den Elementtyp der Operator < definiert sein ! dieser legt auch die Gleichheitsrelation fest: zwei Elemente a und b sind gleich, wenn weder $a < b$ noch $b < a$ gilt !

man kann bei der Instanzierung von Mengentemplates auch ein anderes Ordnungskriterium angeben:

```
namespace std {  
template < class T,  
          class Compare = less<T>,  
          class Allocator = allocator > // --> später  
          // alle STL-Klassen haben diesen Parameter !  
class set;
```

Objektorientierte Programmierung in C++

```

template < class T,
           class Compare = less<T>,
           class Allocator = allocator >
class multiset;
}
    
```

z.B. `set <int, greater<int> >` **absteigend sortierte menge;**
 Implementierung typischerweise als balancierte Binärbäume (*red-black-tree*)
 --> sehr gutes Zeitverhalten beim Suchen, gutes beim Löschen und Einfügen an allen Stellen !

die Sortierung hat zur Konsequenz, dass man Elemente nicht ändern kann, realisiert dadurch, dass alle Iteratoren Zugriffe auf **const**-Objekte bereitstellen (Wert ändern = alten Wert aufsuchen und löschen, neuen einfügen)

- Spezielle Set/Multiset-Operationen

Ausdruck	Bedeutung
Spezielle Suchfunktionen (mit $O(\log n)$ Zeitverhalten)	
<code>m.count (elem)</code>	Anzahl der Elemente mit Wert elem set: 0..1; multiset 0..n
<code>m.find(elem)</code>	liefert die Position des ersten Auftretens von elem oder end()
<code>m.lower_bound(elem)</code>	erste Position an der elem eingefügt werden könnte (das erste Element \geq elem)
<code>m.upper_bound(elem)</code>	letzte Position an der elem eingefügt werden könnte (das erste Element $>$ elem)
<code>m.equal_range(elem)</code>	erste und letzte Position zum einfügen gibt ein pair<const_iterator, const_iterator> zurück
Einfügende/Löschende (alle generischen Operationen +)	
<code>m.insert (elem)</code>	fügt Element ein und liefert Position des neuen Elements, bzw. ob es geklappt hat ^{(*}
<code>m.insert (pos,elem)</code>	fügt Element ein und liefert Position des neuen Elements, bzw. ob es geklappt hat ^{(*} (pos wird nur als Hinweis verwendet)
<code>m.insert (anf, end)</code>	Elemente von [anf, end) einfügen ret??
<code>m.erase (elem)</code>	löscht alle auftreten von elem, return Anzahl der entfernten Elemente
<code>m.erase (pos)</code>	löscht Element bei pos
<code>m.erase (anf, end)</code>	löscht Bereich [anf, end) liefert Position des Folgeelements

^{(*} beim Einfügen einzelner Elemente unterscheiden sich **set** und **multiset** in ihren Rückgabewerten:

Objektorientierte Programmierung in C++

- Spezielle Map/Multimap-Operationen

Ausdruck	Bedeutung
Spezielle Suchfunktionen (mit $O(\log n)$ Zeitverhalten)	
<code>m.count (key)</code>	Anzahl der Elemente mit Schlüssel <code>key</code> map: 0..1; multimap 0..n
<code>m.find(key)</code>	liefert die Position des ersten Auftretens von <code>key</code> oder <code>end()</code>
<code>m.lower_bound(key)</code>	erste Position an der mit <code>key</code> eingefügt werden könnte (das erste Element mit Schlüssel \geq <code>key</code>)
<code>m.upper_bound(key)</code>	letzte Position an der mit <code>key</code> eingefügt werden könnte (das erste Element mit Schlüssel $>$ <code>elem</code>)
<code>m.equal_range(key)</code>	erste und letzte Position zum einfügen mit <code>key</code> gibt ein pair<const_iterator, const_iterator> zurück
Zugreifende/Iteratoren (alle generischen Operationen +)	
<code>m[key]</code> (! nur für map)	liefert eine Referenz für den Wert des Elements zu <code>key</code> , fügt das Element dabei ggf. neu in die map ein!!
Einfügende/Löschende (alle generischen Operationen +)	
<code>m.insert (elem)</code>	fügt Element ein und liefert Position des neuen Elements, bzw. ob es geklappt hat ^{(*}
<code>m.insert (pos,elem)</code>	fügt Element ein und liefert Position des neuen Elements, bzw. ob es geklappt hat ^{(*} (<code>pos</code> wird nur als Hinweis verwendet)
<code>m.insert (anf, end)</code>	Elemente von [<code>anf</code> , <code>end</code>) einfügen ret??
<code>m.erase (elem)</code>	löscht alle auftreten von <code>elem</code> , return Anzahl der entfernten Elemente
<code>m.erase (pos)</code>	löscht Element bei <code>pos</code>
<code>m.erase (anf, end)</code>	löscht Bereich [<code>anf</code> , <code>end</code>) liefert Position des Folgeelements

(*

beim Einfügen einzelner Elemente unterscheiden sich `map` und `multimap` in ihren Rückgabewerten:

multimap

Position des neuen (eingefügten) Elements

map

gibt ein **pair p** zurück: **p.second** gibt an, ob wirklich eingefügt wurde

p.first Position des eingefügten/bereits vorhandenen Elements

Objektorientierte Programmierung in C++

- Beim Einfügen in `maps/multimaps` muss der richtige Elementtyp (`const key`) eingehalten werden ! am besten mit `[multi]map<..., ...>::value_type`

```
////////////////////////////////sample/stl/wc.cc////////////////////////////////
#include <iostream>
#include <string>
#include <map>
using namespace std ;

int main ( ) {
string buf ;
map < string , int > m ;
while ( cin >> buf ) m [ buf ] ++ ;
multimap < int , string > n ;
for ( map < string , int > :: iterator p = m . begin ( ) ;
      p != m . end ( ) ; ++ p )
    n . insert ( map < int , string > :: value_type
                ( p -> second , p -> first ) ) ;
for ( map < int , string > :: iterator p = n . begin ( ) ;
      p != n . end ( ) ; ++ p )
    cout << p -> first << "\t" << p -> second << endl ;
}

```

\$ wc < wc.cc :

1	"\t"	1	value_type	4	->
1	<iostream>	1	while	4	<<
1	<map>	1	{	4	m
1	<string>	1	}	4	map
1	>>	2	!=	4	n
1	[2	=	5	.
1]	2	begin	5	<
1	cin	2	end	5	>
1	cout	2	first	6	,
1	endl	2	for	6	int
1	insert	2	iterator	6	string
1	main	2	second	10	(
1	multimap	3	#include	10)
1	namespace	3	++	10	p
1	std	3	::	11	;
1	using	3	buf		

- wann immer für spezielle Container Memberfunktionen bereitstehen, die generischen Algorithmen entsprechen, sollten diese (ersten) benutzt werden, weil sie effizienter (unter Ausnutzung der Spezifika des entsprechenden Containers) implementiert sind!

```
////////////////////////////////sample/stl/mapfind.cc////////////////////////////////
#include <map>
#include <iostream>
#include <string>
#include <algorithm>
#include <time.h> // itgendwann einmal ctime

inline bool test12345(map<int,int>::value_type elem) {
    return elem.first==12345;
}

main() {
    map <int, int> m;
    long start, elapsed;
    int f;
}

```

Objektorientierte Programmierung in C++

```
cout<<"Generating 200.000 nodes: "<<flush;
start=clock();
for(int i=0; i<100000; ++i) m[rand()]=rand();
m[12345]=12345;
for(int i=1; i<100000; ++i) m[rand()]=rand();
elapsed=clock()-start;
cout<<"in "<<elapsed/1000000.0<<" Sekunden\n";
cout<<m.size()<<" Elemente in der Map"<<endl;

cout<<"m.find(...): "<<flush;
start=clock();
for (int i=1; i<1000; ++i) m.find(12345);
f=m.find(12345)->second;
elapsed=clock()-start;
cout<<"found "<<f<<" in "<<elapsed/1000000.0<<" Sekunden\n";

cout<<"find (...): "<<flush;
start=clock();
for (int i=1; i<1000; ++i) find_if(m.begin(), m.end(), test12345);
f=find_if(m.begin(), m.end(), test12345)->second;
elapsed=clock()-start;
cout<<"found "<<f<<" in "<<elapsed/1000000.0<<" Sekunden\n";
}
/*
Generating 200.000 nodes: in 7.62 Sekunden
32697 Elemente in der Map
m.find(...): found 19134 in 0.03 Sekunden
find (...): found 19134 in 30.03 Sekunden
!!! FAKTOR ~1000 !!!
*/

////////////////////////////////sample/stl/multimapfind.cc////////////////////////////////
#include <map>
#include <iostream>
#include <string>
#include <algorithm>
#include <time.h>

using namespace std;

typedef multimap<int,int>::value_type VT;

inline bool test12345(VT elem) {
    return elem.first==12345;
}

main()    {
    multimap <int, int> m;
    long start, elapsed;
    int f;

    cout<<"Generating 200.000 nodes: "<<flush;
    start=clock();
    for(int i=0; i<100000; ++i) m.insert(VT(rand(),rand()));
    m.insert(VT(12345,12345));
    for(int i=1; i<100000; ++i) m.insert(VT(rand(),rand()));
    elapsed=clock()-start;
    cout<<"in "<<elapsed/1000000.0<<" Sekunden\n";
```

Objektorientierte Programmierung in C++

```
cout<<m.size()<<" Elemente in der Multimap"<<endl;
cout<<"m.find(...): "<<flush;
start=clock();
for (int i=1; i<1000; ++i) m.find(12345);
f=m.find(12345)->second;
elapsed=clock()-start;
cout<<"found "<<f<<" in "<<elapsed/1000000.0<<" Sekunden\n";

cout<<"find (...): "<<flush;
start=clock();
    for (int i=1; i<1000; ++i)
        find_if(m.begin(), m.end(), test12345);
f=find_if(m.begin(), m.end(), test12345)->second;
elapsed=clock()-start;
cout<<"found "<<f<<" in "<<elapsed/1000000.0<<" Sekunden\n";
}
/*
Generating 200.000 nodes: in 10.48 Sekunden
200000 Elemente in der Multimap
m.find(...): found 30080 in 0.03 Sekunden
find (...): found 30080 in 242.99 Sekunden
!!! FAKTOR ~8000 !!!
*/
```

- riesige und schwer überschaubare Möglichkeiten, gefragt sind Compiler, die einerseits die Template-Konstrukte incl. Member-Templates und Default-Template-Argumente überhaupt umsetzen können und andererseits den Benutzer besser bei der Lokalisierung von Fehlern unterstützen - Typfehler bei der Benutzungen der STL werden i.allg. mit einer Kaskade von schwer lesbaren Fehlerausschriften quittiert:

```
// Beispiel:
#include <map>
#include <iostream>
#include <string>
using namespace std;

main() {
    multimap <string, double> m;
    m["willi"]=7.7; // multimap hat keinen op[] !!
    m["otto"]=8.8;
    m["willi"]=9.9;

    for(map<string,double>::iterator p=m.begin(); p!=m.end();++p)
        cout<<p->first<<": "<<p->second<<endl;
}
```

```
err.cc: In function `int main()':
err.cc:11: invalid types
`multimap<basic_string<char,string_char_traits<char>,__default_alloc_template<false,0>
>,double,less<basic_string<char,string_char_traits<char>,__default_alloc_template<false,0> >
>,__default_alloc_template<false,0> >[char[6]]' for array subscript
err.cc:12: invalid types
`multimap<basic_string<char,string_char_traits<char>,__default_alloc_template<false,0>
>,double,less<basic_string<char,string_char_traits<char>,__default_alloc_template<false,0> >
>,__default_alloc_template<false,0> >[char[5]]' for array subscript
err.cc:13: invalid types
```

Objektorientierte Programmierung in C++

```
`multimap<basic_string<char,string_char_traits<char>,__default_alloc_template<false,0>  
>,double,less<basic_string<char,string_char_traits<char>,__default_alloc_template<false,0>  
>,__default_alloc_template<false,0> >[char[6]]' for array subscript
```

- neben den (primären) Containern gibt es in der STL einige sog. Container-Adapter: es handelt sich dabei um spezielle Anpassungen der Container für spezielle Anwendungen:

Queues: (#include <queue>)

```
template < // im namespace std  
         class T,  
         class Container = deque<T> >  
class queue;
```

FIFO-Warteschlangen (auch mittels `list` instanzierbar)

Priority Queues: (#include <queue>)

```
template < // im namespace std  
         class T,  
         class Container = vector<T>,  
         class Compare = less<Container::value_type>  
>  
class priority_queue;
```

Warteschlangen mit Prioritäten (auch mittels `deque` instanzierbar)

Stacks: (#include <stack>)

```
template < // im namespace std  
         class T,  
         class Container = deque<T> >  
class stack;
```

Kellerspeicher (auch mittels `list` und `deque` instanzierbar)

- in der STL wird auch die Standardklasse `string` zur Zeichenverarbeitung bereitgestellt:

Strings: (#include <string>)

```
template < // im namespace std  
         class charT,  
         class traits = char_Traits<charT>  
         class allocator = allocator >  
class basic_string; // noch nicht auf Zeichentyp  
                  // festgelegt  
  
// im namespace std:  
typedef basic_string<char> string; // ASCII  
typedef basic_string<wchar_t> wstring; // Unicode
```

Objektorientierte Programmierung in C++

mit Einführung von `string` wurde auch die `iostream`-Bibliothek erheblich überarbeitet, um mit `strings` zusammenarbeiten zu können, ohne dass sich deren Endnutzerschnittstelle wesentlich verändert hat, ggf. ist wichtig

```
typedef basic_ostream<char, char_traits<char> > ostream; /*!!!*/
```

Operationen über `strings` (Konstruieren, Suchen, Verknüpfen, ...) hier nicht ausgeführt (vgl. z.B. Josuttis, Kapitel 10, S.357 ff.)

- die STL hat noch zwei sog. numerische Klassen:

Komplexe Zahlen: (`#include <complex>`)

```
template < // im namespace std
          class T
        >
class complex;
```

komplexe Zahlen mit allem "drum und dran" (Arithmetik und transzendente Funktionen) für `float`, `double`, `long double` bereits vordefiniert !

Valarrays: (`#include <valarray>`)

```
template < // im namespace std
          class T
        >
class valarray;
```

Vektoren und Matrizen für numerische Operationen mit gutem Zeitverhalten (keine temporären Zwischenergebnisse) und kompakter Notation (z.B. Ausführung von Operationen auf allen Elementen, Bildung von sog. Slices, ...)

- Bitsets: (`#include <bitset>`)

```
template < // im namespace std
          size_t bits
        >
class bitset;
```

Bitvektoren (konstanter Länge)

- Allokatoren:
Separation der Speicherverwaltung für dynamische Objekte (Listen- und Baumknoten etc.) von den abstrakten Containern: Container enthalten als Typparameter eine Klasse, die die Speicherverwaltung komplett übernimmt,

Standardmäßig stellt jede Implementation einen *default allocator* in der Klasse `std::allocator` bereit, dieser verwaltet geeignete Memory-Pools --> `new` und `delete` werden von der STL nicht direkt gerufen,

Objektorientierte Programmierung in C++

alternative Allokatoren (z.B. mit *garbage collection*, oder auf verschiedenen Speichermodellen ...) sind möglich und beeinflussen die eigentliche Funktionalität der Container in keiner Weise !!

- Erweiterungen der STL (über den standardisierten Umfang hinaus)

in konkreten Implementationen meist nicht als solche ausgewiesen, sondern unterschiedslos mit den standardisierten STL-Komponenten realisiert !

maßgeblicher Vertreter (und guter Kandidat für 2nd revision des C++ -Standards) ist die SGI - STL, die folgende wichtige Zusätze implementiert (z.B. libg++2.8.1):

neue Algorithmen: "Kleinigkeiten", die sich aber nicht allgemein und effizient mit den vorhandenen generischen Algorithmen realisieren lassen: `iota` ([anf,end) mit aufsteigenden Zahlen initialisieren), `random_sample` (eine zufällige Stichprobe ohne Änderung des Originalcontainers [statt `random_shuffle`]), etc.
ein verbesserter `sort`-Algorithmus (aka introsort) der sich auch bei nahezu vorsortierten Containers optimal [$O(n \log n)$] verhält

neue Container: die assoziativen Container `[multi]set`, `[multi]map` sind immer sortiert, dies ist nicht immer nötig (kostet aber overhead) bzw. manchmal sogar unerwünscht ---> `hash_set`, `hash_map`, `hash_multiset`, `hash_multimap` in der Benutzung wie zugehörige STL-Container, aber effizienter
nicht immer braucht man **doppelt** verkettete Listen ---> `slist`
eine spezielle Stringklasse **rope** (Ropes are a scalable string implementation: they are designed for efficient operation that involve the string as a whole. Operations such as assignment, concatenation, and substring take time that is nearly independent of the length of the string. Unlike C strings, ropes are a reasonable representation for very long strings such as edit buffers or mail messages)

thread safety: originale STL ist nicht thread safe !!!

12. Exceptions

12.1. Programmausnahmen

in vielen Programmiersprachen gibt es kein praktikables Konzept zur Behandlung von logischen Fehlern zur Laufzeit, übliche Ansätze sind:

- "ES WIRD SCHON NICHTS PASSIEREN": ignorieren aller möglichen erkennbaren Fehler in Erwartung, dass ein übergeordneter Mechanismus in Kraft tritt (z.B. segmentation violation, core dump), für komplexe, große, reale Anwendungen ungeeignet / gefährlich
- "VORSICHT IST DIE MUTTER DER PORZELLANKISTE": alles kann schief gehen, daher bekommt z.B. jede Funktion einen Rückkehrcode, der ihren ordnungsgemäßen Abschluß quittiert

Objektorientierte Programmierung in C++

```
int f() { ...; return OK ? 1 : 0; }
....
if (!f()) // irgendwas ist faul
```

zumeist überflüssige Test blähen den Programmcode auf bzw. die Rückkehrcodes werden ignoriert; Funktionen, die 'echte' Resultate liefern bringen Probleme

- "EINER FÜR ALLE": die typische C -Ein-Ausgabefehlertsignalisierung:

```
extern int errno;

... call_any_io_function(...);
if (errno) { perror ("Fehler"); ...; errno=0; }
// "Fehler: String, passend zu errno"
```

wird ebenfalls meist unterlassen, muss explizit zurückgesetzt werden, ist zu 'zentralistisch'

- "VERTRAUEN IST GUT, KONTROLLE IST BESSER":

```
#include <cassert>

assert (i!=0); n=1/i;

/* i=0:
Assertion failed: i!=0, file ass.cc, line 7
Abort (core dumped)
*/
```

Tests werden ebenfalls immer ausgeführt, können allerdings durch Neuübersetzung mit `#define NDEBUG` komplett eliminiert werden

keiner der beschriebenen Ansätze ist befriedigend, erschwerend kommt noch hinzu, dass die Position von Fehlerursache und geeigneter Fehlerbehandlung i. allg. in unterschiedlichen Quelltexten (unterschiedlicher Autoren) auftauchen:

```
// Modul: any_lib.cc:
int f ( int i, int j) {
// never call with j==0
    return i / j;
    // hier tritt der Fehler auf
}
```

```
-----
// Modul: Benutzung:
#include "any_lib.h"

main() {
    int n = f (1, 0);
    // hier ist der Fehler eigentlich zu behandeln
}
```

- C++ hat (wie Ada, Java) das Konzept des *exception handling*, um solchen Situationen angemessen zu begegnen: eine Folge von Anweisungen kann in einen sog. **try**-Block eingeschlossen werden, an den sich eine Folge von **catch**-Blöcken (Fehlerbehandlungsaktionen) anschließen kann.

Sobald während der Ausführung des **try**-Blockes ein Fehler "aufgeworfen" wird (u.U. aus indirekt gerufenen Funktionen), wird die Abarbeitung des **try**-Blockes beendet und in einen passenden (s.u.) **catch**-Block verzweigt !

```
// Modul: any_lib.cc:
int f ( int i, int j)
// never call with j==0
{
    if (!j) throw anException;
    return i / j;
    // hier tritt der Fehler auf
}

-----

// Modul: Benutzung:
#include "any_lib.h"

main()
{
    try {
        int n = f (1, 0);
    }
    catch (anException)
    { cerr << "Division durch Null\n" }
}
```

12.2. hierarchische Behandlung

Was ist eine Exception ? Es ist ein Objekt eines beliebigen Typs, welches vom Ort des Auftretens des Fehlers an den Behandlungsblock übergeben wird !

ein *Handler* vom Typ T, const T, T& oder const T& passt zu einem **throw**-Ausdruck mit einem Objekt des Typs E wenn

- [1] T und E der gleiche Typ sind, oder
- [2] T eine public Basis von E ist, oder
- [3] T und E Zeigertypen sind, und E kann durch Anwendung der Standardtypkonversion in T umgewandelt werden

Objektorientierte Programmierung in C++

```
class anException {
int reason;
public:
anException (int i=0): reason(i) {}
int why () {return reason; }
};

try {
.... throw int(); // Wert undefiniert
.... throw 1;
.... throw anException ();
.... throw new anException (2);
}
catch (int) { .... }
catch (anException& ex) { ... ex.why(); .... }
catch (anException* ex) { ... ex->why(); ... }
catch (...) { /* catch all others */ ....}
```

die Behandlung des Fehlers erfolgt hierarchisch im doppelten Sinne:

1. Tritt in einem `try`-Block ein Fehler auf (`throw`), für den kein passender *Handler* folgt, so wird die Ausnahme (das Exception-Objekt) an den übergeordneten Block 'weitergereicht' ----> eine Ausnahme die aus `main()` 'herausfällt' verursacht den Aufruf einer Funktion `void terminate();`, die die Programmabarbeitung mittels `void abort ();` beendet (kann mit `void *set_terminate(void (*)()) ();` geändert werden)
2. eine aufgeworfenen Exception wird immer vom ersten passenden `catch`-Block aufgefangen, so lassen sich spezielle Fehler (Ableitungen) vor allgemeinen Fehlern (Basisklassen) behandeln (es wird stets nur eine *Handler* abgearbeitet). Die Platzierung eines Basisklassen-*Handlers* vor einem *Handler* für eine Ableitung ist ein statischer Fehler. Exceptions können auch explizit weitergereicht werden: `throw;` veranlasst dies.

Funktionen können mit sog. *exception specifications* ausgestattet werden, die im Voraus "ansagen", welche Exceptions aus dieser Funktion "herausfallen" können. Die Einhaltung dieser Vorgaben wird **zur Laufzeit !!!** geprüft, und setzt daher implizit einen RTTI-Mechanismus voraus, keine statische Prüfung a la Java (Argumentation s. ARM p. 363)

SunPro CC 4.0.x (nur unter Solaris2.x): RTTI nur für diesen Zweck (implizit) implementiert, nicht durch Nutzer verwendbar, u. U. braucht man `#include <exception.h>`

g++ 2.7.2: man braucht `#include <exception>`, `#include <typeinfo>` als erstes im Quelltext und muss mittels `g++ -frtti -fhandle-exceptions` übersetzen !!

Objektorientierte Programmierung in C++

```
// exception specification:  
void f() throw (This, That)  
{ ... g(); ...} // wird erst zur Laufzeit ausgewertet !  
void g() { if (any_thing_wrong) throw Anything(); }
```

das Aufwerfen einer "nicht angekündigten" Ausnahme verursacht den Aufruf einer Funktion `void unexpected();`, die die Programmabarbeitung mittels `void terminate ();` beendet (kann mit `void *set_unexpected(void (*)()) ();` geändert werden)

eine Funktion ohne *exception specification* reicht jede Ausnahme weiter

Exceptions und virtuelle Funktionen können gemeinsam benutzt werden: im *Handler* kann man eine virtuelle Funktion an einem Basisklassenzeiger / einer Basisklassenreferenz aufrufen, die gemäß der Natur des aufgeworfenen Exception-Objektes gebunden wird.

12.3. Programmierstil

Der Sprung vom Auftreten einer Ausnahme zu ihrer geeigneten Behandlung ist i.all. mit dem "Abräumen" von Stackframes verbunden (*stack unwinding*) in denen lokale Objekte von Klassentypen "leben" können, zur Wahrung der Konsistenz müssen für diese Objekte Destruktoren aufgerufen werden ! Dies geschieht automatisch. Lokale Objekte könnten Ressourcen belegen, die ansonsten nie freigegeben würden.

Andersherum sollten alle lokalen Ressourcenanforderungen durch lokale Objekte erfolgen, weil nur so deren Rückgabe gesichert ist. Stroustrup nennt dies "*resource acquisition is initialization*" Ansatz. Vorsicht vor Zeigern !!!!

Dies hat weitreichende Auswirkungen auf den Programmierstil insgesamt: der konsequente Einsatz von *Exception Handling* erfordert, jede Ressourcenanforderung über lokale Objekte zu veranlassen.

```
// ein Kompletbeispiel:  
#ifdef __GNUG__  
#include <exception>  
#include <typeinfo>  
#else  
#include <exception.h>  
#endif  
  
#include <iostream>  
#include <cstdlib>  
  
using namespace std;
```



.../samples/exceptions

Objektorientierte Programmierung in C++

```
class X {
    char* name;
public:
    X(char *n): name(n){cout<<"X::X() for "<<name<<"\n";}
    ~X(){cout<<"X::~X() for "<<name<<"\n";}
};

class Arithmetic_Exception {
public:
    virtual ~Arithmetic_Exception(){}
};

class Division_by_Zero: public Arithmetic_Exception
{
    const char * f;
    const int l;
public:
    Division_by_Zero(const char* file, const int line)
        :f(file),l(line){cout<<"ctor\n";}
    ~Division_by_Zero(){cout<<"dtor\n";}
    friend ostream& operator<<
        (ostream& o, Division_by_Zero& d) {
        return o<<"Division durch Null: FILE: "
            <<d.f<<" LINE: "<<d.l<<endl;
        }
};

double f(double i) throw (Division_by_Zero)
{
    X x1("x1");
    X *p = new X ("dynamic");
    if (i==0.0) throw Division_by_Zero(__FILE__, __LINE__);
    X x2("x2");
    delete p;
    return 1/i;
}

void (*old_terminate)()=abort;

void new_terminate()
{
    cout<<"Termination due to unhandled exception\n"<<flush;
    old_terminate();
}

int main() {
    cout<<"setting new termination handler\n";
    set_terminate(old_terminate);
    old_terminate=set_terminate(new_terminate);
}
```

Objektorientierte Programmierung in C++

```
try {
    cout<<"outer try>>>>>>>\n";
    try
    {
        cout<<"inner try>>>>>>>>\n";

        X x3("x3");

        f(1);
        f(0);
        cout<<"<<<<<<<<inner try\n";
    }
    // at least one handler must follow !!
    catch (Division_by_Zero& d)
    {
        cout<< d;
        /*re*/ throw;
    }
    catch (Arithmetic_Exception& d)
    {
        cout<< "any arithmetic exception\n";
    }
    cout<<"<<<<<<<outer try\n";
}
catch (Arithmetic_Exception& d)
{
    cout<< "any arithmetic exception\n";
    /**** /*re*/ throw /*once more without handler*/;
}
}
/*
setting new termination handler
outer try>>>>>>>
inner try>>>>>>>>
X::X() for x3
X::X() for x1
X::X() for dynamic
X::X() for x2
X::~X() for dynamic
X::~X() for x2
X::~X() for x1
X::X() for x1
X::X() for dynamic
ctor
X::~X() for x1
X::~X() for x3
Division durch Null: FILE: e.C LINE: 40
any arithmetic exception
dtor          // fehlt bei g++ 2.7.2
*/
```

```
/* Kommentar bei //**** raus:
....
Division durch Null: FILE: e.C LINE: 40
any arithmetic exception
Termination due to unhandled exception
Abort (core dumped)
*/
```

13. Namespaces

13.1. Motivation

beim Einsatz von C++ für große Projekte, die verschiedene Bibliotheken verwenden kommt es unweigerlich zum Problem der Namenskollision im globalen Namensraum, Klasse nsind ein Hilfsmittel zur Entlastung des globalen Namensraumes, Klassennamen sind ihrerseits jedoch (zumeist) wiederum globale Bezeichner: `string`, `String`, `XtString`, `QtString`, `Matrix`, `Vector`,

- Verschieden Bibliotheken sollten verschiedene Namensräume benutzen. Lange Präfixe sind unhandlich, kurze von Dopplung bedroht Präfix muss optional sein: -> C++ namespaces
- Syntax: Deklaration wie Klasse (aber keine Vererbung, access, ...) (oder wie Funktion, aber keine Parameter), ...

```
namespace Humboldt_Universität {
    class Fachbereich { //...
    };
    class Student;
    void register(Fachbereich&, Student&)
} // ; muss hier nicht stehen im Gegensatz zu class !
```

- namespace re-opening: Zusätzliche Deklaration, fehlende Definitionen

```
namespace Humboldt_Universität {
void register(Fachbereich& f, Student& s)
{
    //...
}
// ...
} // gehört zum gleichen namespace
```

- Member-Definitionen in umhüllenden namespace

```
class Humboldt_Universität::Student {
    //...
};
```

Objektorientierte Programmierung in C++

- Aliases: verkürzte Formen

```
namespace HUB = Humboldt_Universität;
```

- Zwei Möglichkeiten der "Bereitstellung" von Elementen aus namespaces:

1. Using-Direktiven: Alle Namen werden importiert.

```
using namespace Humboldt_Universität;  
Fachbereich Informatik;  
Student Markus_Mustermann;
```

2. Using-Deklaration: Ausgewählte Namen werden importiert.

```
void doit(){  
    using HUB::register;  
    register(Informatik, Markus_Mustermann);  
}
```

- Anonyme Namensräume: Ersatz für dateiglobale (static) Objekte.

```
namespace{  
    int counter = 0;  
    void inc();  
}
```

```
main(){inc();}
```

```
namespace{  
    void inc() { counter++;}  
}
```

- Lookup: Regelwerk für das Auffinden von Namen (Block, Klasse, Namespace)

Beispiel Funktionsruf:

```
#include <iostream>  
  
void f(){cout<<"::f()\n";}  
  
namespace A{  
    void f(int = 4) {cout<<"A::f(int)\n";}  
    void f(char*) {cout<<"A::f(char*)\n";}  
    class EineKlasse {public: void g();};  
    void EineKlasse::g(){ f();}  
}  
  
main(){  
    A::EineKlasse a;  
    a.g(); // A::f(int) !!  
}
```

Objektorientierte Programmierung in C++

Reihenfolge ist immer 1. lookup, 2. overload resolution, 3.access check
Suche erfolgt „von innen nach außen“

- Lookup unqualifizierter Namen: verwende alle using-Deklaration, und rekursiv alle using-Direktiven

```
namespace A{
    void f();
}
namespace B{
    using namespace A;
    void f(int);
}
using namespace B;
// A::f() und B::f(int) verfügbar
```

- Lookup qualifizierter Namen: Alle using-Deklaration; using-Direktiven nur, wenn zuvor nichts gefunden

```
namespace A{
    void f(int);
}

namespace B{
    void f();
}

using namespace A;
using B::f;
void g1(){
    ::f(); //B::f
}

void f(char*);
void g2()
{
    ::f(3); //Fehler: f() und f(char*) verfügbar
    ::f(); //B::f
}
```

- Mehrdeutigkeit und Redefinition

using-Deklaration übernimmt Namen in den aktuellen Namensraum

```
namespace A{
    void f();
    void f(int);
}
```

Objektorientierte Programmierung in C++

```
using A::f;
void f(); // Konflikt: ::f und A::f
/*
ns3.cc:6: new declaration `void f()'
ns3.cc:2: ambiguates old declaration `void f()'
*/
```

Mehrdeutigkeiten erlaubt bei überladenen Funktionen, wenn *overload resolution* Auswahl trifft

```
namespace A{
    void f();
}
namespace B{
    void f();
    void f(int);
}
void f(char*);
using namespace A;
using namespace B;           //Ok

void g() {
    f(4);                     //Ok: A::f()
    f();                       //Fehler: A::f oder B::f?
    f("Hallo");              //Ok: ::f(char*)
}
```

- Spezielle erweiterte Form des Lookup: sog. "Koenig-Lookup":
Bei unqualifiziertem Funktions- oder Operatorruf hängt Lookup von den Parametertypen ab:

```
namespace Humboldt_Universität{
    ostream& operator <<(ostream &, Student&);
    Student* suche(Fachbereich&, char*); }
void f(char *name)
{
    HUB::Student* s = suche (name);
    // man möchte hier natürlich:
    cout << "Found" << *s << endl;
    // sagen können, die bisherigen lookup-Regeln
    // erlauben das jedoch nicht, weil << keine
    // Memberfunktion ist!
}
```

der explizite Ruf ist äußerst unhandlich:

```
void f(char *name)
{
    HUB::Student* s = HUB::suche (name);
    HUB::operator<<(cout<<"Found", *s)<<endl;
}
```

Lösung: Koenig-Lookup (Auf Vorschlag von Andrew Koenig)

betrachtet auch alle Parametertypen, bei Klassen alle Basistypen, bei Funktionspointern die Argumenttypen und den Ergebnistyp und sucht in den namespaces, in denen die Typen definiert sind (einschließlich using-Deklarationen, ausschließlich using-Direktiven) !

13.2. namespace std

- Erweiterbarkeit der Bibliothek: neue Standardklassen können ohne Konfliktgefahr eingeführt werden
- Kompatibilität mit nichtkonformen Bibliotheken: ostream und std::ostream sind verschiedene Dinge

alt:

```
#include <iostream.h>
int main()
{
    cout << "Hello, world!" << endl;
}
```

standardkonform:

```
#include <iostream>
int main()
{
    std::cout << "Hello, world!" << std::endl;
}
```

oder:

```
#include <iostream>
using std::cout; using std::endl;
int main()
{
    cout << "Hello, world!" << endl;
}
```

13.3. Namespaces in g++

- 1993 sind *namespaces* in den C++-Standard erstmals aufgenommen worden; ein Feature mit weitreichenden Konsequenzen hinsichtlich der Benutzung von C++!
- 17. November 1994: Mike Stump erweitert g++-Syntax (namespace, using, type-name, and, ...), fortgesetzt 6. Dezember
- 16. Mai 1995: erste Version für g++ 2.7 (16. Juni 95)
- 12. März 1996: Jason Merrill deaktiviert Implementierung.
- 5. Juni 1997: std:: ist :: für g++ 2.8 (7. Jan 1998)
- 24. Februar 1998: neue Implementierung von M. v. Löwis angefangen

Objektorientierte Programmierung in C++

- Sommer 98: in egcs1.1 übernommen
- Lookup in g++:
Bisher: Identifier werden über Hashtabelle gefunden; jeder Identifier verweist auf (block-) lokale und globale Definition

```
double i;  
void f(){  
    int i;  
    for(i=0;i<10;i++){  
        char *i = "Hallo\n";  
        printf(i);  
        ::i = 3.14;  
    }  
    i = 42;  
}
```

Jetzt: globale Definition ist Liste von Tupeln (namespace, Definition). Aber: Typen

- Name mangling
Klassennamen haben Längenprefix, Parameter von links

```
Hallo::f(Welt, i, char const*)  
wird zu: f__5Hallo4WeltiPCc
```

Verschachtelte Namen werden mit Q qualifiziert

```
std::ostream::ostream  
    (int, std::streambuf *, std::ostream *)  
wird zu:  
__Q23std7ostreamiPQ23std9streambufPQ23std7ostream
```

Rückbezug auf Typen (T)

```
__Q23std7ostreamiPQ23std9streambufPT0
```

Rückbezug auf Qualifizierungen (K) (squangling, nur in egcs)

```
__Q23std7ostreamiPQ2K09streambufPT0
```

- Binärkompatibilität: wenn Klassen in std:: sind, ändert sich die Qualifizierung
- std wird weiterhin ignoriert, wenn nicht explizit angefordert (-fno-std, -fnew-abi)
- Funktionsüberladung
bisher: Für jeden Identifier war Liste der globalen Kandidaten eingetragen (Methoden sind anders realisiert)
jetzt: Kandidatenliste wird temporär beim Lookup berechnet

```
namespace A{ }  
using namespace A;
```

```
namespace B{
    bool f(bool);
}
using B::f;
namespace A{
    void f();
}
void f(int);
// f ist A::f() oder ::f(int) oder B::f(bool)
// ::f ist ::f(int) oder B::f(bool)
```

14. RTTI und neue Castoperatoren

14.1. Typidentifikation



zur Motivation ♦ 7.5.

Harmonisierung unterschiedlicher RTTI-Ansätze durch compilergestützte RTTI im Kontext von Exception Handling ohnehin zumindest intern erforderlich (s.o.)

RTTI wird in zwei Stufen unterstützt:

1. einerseits durch einen (laufzeit-) kontrollierten Cast-Operator `dynamic_cast`, dieser entspricht semantisch dem GUARD-Makro aus 7.5, wird jedoch vom Compiler implizit umgesetzt: Umwandlung eines Zeigers auf eine Basisklasse auf eine abgeleitete, falls ein solches zur Laufzeit vorliegt, sonst 0

- `dynamic_cast` kann nur mit Zeiger- und Referenztypen verwendet werden:

```
class A {
public: virtual void needed () {}
};
```

```
class B: public A {public int i;};
class C: public B {public int j;};
```

```
A *pa = new B;
B *pb = dynamic_cast<B*>(p);
if (pb) pb->i = 12345; // ok, es ist ein B
C *pc = dynamic_cast<C*>(p);
if (pc) pc->j = 54321; // wird nicht ausgefuehrt
```

- da zur Umsetzung von `dynamic_cast` offenbar zusätzliche Informationen in dem über den Zeiger / die Referenz erreichten Objekt enthalten sein muss, würde das Layout aller Objekte verändert werden ---> nur anwendbar auf Klassen, die virtuelle Funktionen enthalten (Stroustrup nennt diese in D&E: *polymorphe Klassen*)
- in jedem Fall braucht man `#include <typeinfo>` bzw. `#include <typeinfo.h>`

Objektorientierte Programmierung in C++

- ein `dynamic_cast` mit einem 'falschen' Referenztyp wirft zur Laufzeit die Exception `bad_cast` (es gibt keine "Nullreferenz"), g++ will daher neben `-frtti` auch noch `-fhandle-exceptions`, wenn Referenzen dynamisch ge'cast'ed werden (CC 4.0.x kann kein RTTI)

```
B b;
A& ra=b;

try {
    B& rb = dynamic_cast<B&>(ra); // ok keine exc.
}
catch (bad_cast){cout<<"bad cast\n";}

try {
    C& rc = dynamic_cast<C&>(ra);
}
catch (bad_cast){cout<<"bad cast\n";} // ! exc.
```

2. andererseits kann man Typidentität direkt abfragen, dazu existiert der Operator `typeid` (wie `sizeof` vom Compiler umgesetzt und nicht überladbar), der eine (vergleichbare) Struktur des Typs `type_info` liefert, auf einer solchen ist die Funktion `name` definiert, die einen Klarnamen der Klasse (nicht notwendig identisch mit dem Klassennamen) erzeugt:

```
#include <typeinfo>
#include <iostream>
using namespace std;

class A {
    virtual void () {}
};

class B: public A {
};

class C: public A {
};

void check (A* p) {
    if (typeid(*p)==typeid(A))
        { cout << "es ist ein A\n"; return; }
    if (typeid(*p)==typeid(B))
        { cout << "es ist ein B\n"; return; }
    cout << "weder A noch B\n";
}

const char* get_name(A* p)
{
    return typeid(*p).name();
}
```

Objektorientierte Programmierung in C++

```
main()
{
    A *p;

    p = new A;
    check (p);
    cout << get_name(p) << endl;

    p = new B;
    check (p);
    cout << get_name(p) << endl;

    p = new C;
    check (p);
    cout << get_name(p) << endl;
}
/* erzeugt nach g++ -frtti ti.cc:
es ist ein A
TD.1A
es ist ein B
TD.1B
weder A noch B
TD.1C
*/
/* fehlt virtual A::needed() so ergibt sich
(offenbar fehlerhaft):
es ist ein A
TD.1A
es ist ein A
TD.1A
es ist ein A
TD.1A
*/
```

- wenn möglich sollte Variante 1 (`dynamic_cast`) anstelle von Variante 2 verwendet werden !
- mit Hilfe der eindeutigen `type_info` kann man sich weitere Laufzeitinformationen an Objekte 'hängen' z.B. Stroustrup D&E pp. 406:

```
class Extended_type_info { /* all you wish */ };

extern Map<type_info*, Extended_type_info> EITable;
Extended_type_info& ext_typeid(A* p)
{
    return EITable[&typeid(*p)];
}
// + entsprechende Extended_type_info-Objekte pro Klasse
// und Map-Eintrag durch Konstruktoren
```

Objektorientierte Programmierung in C++

- mit `dynamic_cast` lässt sich auch das Problem der verbotenen Umwandlung von einem Zeiger auf eine virtuelle Basisklasse in einen Zeiger auf eine Ableitung lösen:

```
Werkstudent *pw;
Person      *pp; // virtuelle Basisklasse
Werkstudent martin ("Martin", "Schmidt",
                   123456789, 700, 900);
pp=pw;      // erlaubt, da virtuelle Basisklasse
// pw=(Werkstudent*)pp; // nicht erlaubt aber:
pw=dynamic_cast<Werkstudent*>(pp);
// virtual Person::print() !!!!
```

14.2. Typumwandlung

die aus C übernommene Typumwandlung durch Typcasts birgt potentielle Fehlerquellen in sich, man kann damit zugleich unkritische (sinnvolle) Operationen

```
2 / double(3)
```

```
class B: public A {...};
A *pa = new B; B* pb = (B*)pa;
```

```
cout << (void*)pa;
```

aber auch äußerst kritische Operationen

```
int *pi = new int; int i=int(pi); // ???
```

```
const X x; X* px=(X*)&x; // ???
```

```
class A{}; class B{};
A *pa = new A; B* pb=(B*)pa; // ???
```

ausdrücken, die anhand des Quelltextes sowohl für den Programmierer als auch für den Compiler nicht voneinander zu unterscheiden sind. (vgl. Zitat aus D&E S.38)

Um diese Situation zu verbessern, schlägt der C++ -Standard 3 neue (statische) Typcast-Operatoren vor, die Typumwandlungen einerseits deutlicher im Quelltext hervortreten lassen, und die andererseits die Absicht der Umwandlung deutlich machen

```
static_cast<T> (e) // hinreichend gutartige Typumwandlung
reinterpret_cast<T> (e) // Typumwandlungen, die zurückgenommen
// werden müssen, um sicher verwendet
// werden zu können
const_cast<T> (e) // "Kappen" von const
```

Objektorientierte Programmierung in C++

B. Stroustrup in D&E, p. 420: "Die etwas lang geratenen Bezeichner und die Template-ähnliche Syntax schrecken viele Leute von den neuen Typumwandlungsoperatoren ab. Das mag gar nicht das Schlechteste sein, da es den Programmierer beständig daran erinnert, auf welch risikoreiches Geschäft er sich einlässt und das verschiedenartige Gefahren hinter den unterschiedlichen Operatoren lauern."

Derzeit werden die neuen Operatoren (wenn überhaupt [z.B. g++ 2.7.2]) neben den herkömmlichen C-Casts $(\tau)e$ und der C++-Variante `typename(e)` unterstützt. Dies kann/wird sich in Zukunft ändern !

- `static_cast` soll in allen Umwandlungen eingesetzt werden, die dem Muster `Base*/& <---> Derived*/&` genügen (davon ist nur eine Richtung immer sicher) das Ergebnis eines solchen Casts kann ohne weitere Typumwandlung benutzt werden (ander bei `reinterpret_cast`), `static_cast` ist nicht so sicher, wie `dynamic_cast`, letzteres wird aber erst zur Laufzeit geprüft und setzt Typinformation voraus

mit einem `static_cast` kann man sich nicht (wie mit einem C-Cast) Zugang zu privaten Basiskomponenten verschaffen ! (Konsistenz des Schutzsystems wird verbessert)

- `reinterpret_cast` ist für Typumwandlungen der Art `char*` nach `int*` oder `Some_class*` nach `Unrelated_class*` gedacht, die naturgemäß unsicher und implementierungsabhängig sind. Bei Hierarchiebezogenen Umwandlungen nimmt `reinterpret_cast` keine Adressumrechnung vor wie `static_cast`. Das Ergebnis lässt sich nicht ohne Sicherheitsrisiko verwenden, außer man 'interpretiert' zurück auf den ursprünglichen Typ. Andere Verwendungsarten sind auf alle Fälle nicht portabel !! Sollte daher nur in Ausnahmefällen benutzt werden
- `dynamic_cast`, `static_cast` und `reinterpret_cast` erlauben keine "Beschneidung" der `const`-Eigenschaft. `const_cast` ist "der traurigste Zweck, Ersatz für die althergebrachten Typumwandlungen zu finden" (Stroustrup, D&E). Eigentlich sollte eine `const`-Eigenschaft nie angetastet werden, die Verwendung von `const_cast` also auf ein Minimum reduziert werden.
- eine empfohlene Strategie der Migration besteht darin, zunächst alle `T(e)/T(e)` - Casts durch `static_cast<T>(e)` zu ersetzen und die Meldungen des Compilers abzuwarten. Unter genau geprüften Umständen (besser unter deren Eliminierung) kann man auf `const_cast` und `reinterpret_cast` zurückgreifen.