

Advanced Programming Techniques - Zusammenfassung des Wichtigsten

Wintersemester 2015/2016

Dr. Harald Köstler

Autor:

- Christian Strate

Inhaltsverzeichnis

0	Organisatorisches	3
1	Allgemeines	4
2	Lecture 2	5
3	Lecture 3	7
4	Lecture 3a	10
5	Lecture 4	13
6	Lecture 5	16
7	Lecture 5a	17
8	Lecture 6	19
9	Lecture 7	20
10	Lecture 7a	22
	10.1 Vorstellung des Projekts	22
11	Lecture 8	24
12	Lecture 9	27
13	Lecture 9a	29
14	Lecture 10	31
15	Lecture 11	36
16	Lecture 12	39
17	Lecture 12a	41
18	Lecture 13	47

0 Organisatorisches

- Für die Klausur ist der gesamte Code des Projektes als Hilfsmittel erlaubt
- Übungsbeginn: 20.10.2015
- CIP: 00.131
- Mittwochs VL nur alle zwei Wochen
- [VL-Aufzeichnung](#)

Klausurstoff:

1. multiple choice:
 - C++-keywords
 - Concepts
 - initialisation
2. Which function is called
3. Standardlibrary (Implementation with it)
 - Vector
 - List
 - Sort
 - unique
 - for each
 - find if
4. Project, nahe an, aber weniger als 50 % der Arbeit
 - Fällt weg, da schwierig zu korrigieren: neue Typen, Funktionalitäten implementieren
 - bring your printed code!
 - Design Concepts \Rightarrow abstract and C++
 - Warum sind die verwendeten \uparrow sinnvoll
 - Oberhalb der Sprache, also unabhängig von C++

Über diese Zusammenfassung:

Diese Zusammenfassung hält fest was ich interessant hielt oder noch nicht konnte. Ferner handelt es sich hierbei lediglich um eine inoffizielle Zusammenfassung von Studenten, weshalb weder Garantie auf Vollständigkeit noch auf Korrektheit gewährleistet werden.

1 Allgemeines

- Weiterer Debugger: ddd
- Profiler: gprof

Postfix-Operatoren erfordern das Anlegen einer Kopie, wenn der Operator von den entsprechenden C++-Klasse überladen wurde. Für die C-Datentypen sollte die Performanz identisch zu prefix sein. Wenn der Operator aber inlined wird, dann sollte auch bei der postfix-Notation eine Optimierung seitens des Compilers ermöglicht werden.

Auf den Folien steht es sei nur eine main-Funktion pro C++-Programm möglich. Das ist unpräzise. Aus Binary-Sicht ist das korrekt, aus Code-Sicht nicht, denn es können sich durchaus weitere main-Funktionen in anderen Namespaces verstecken.

- `std::cin` nimmt auch eine Typprüfung vor.
- Schlüsselwort "explicit" verhindert impliziten Aufruf bei Assignment. - Sinnvoll bei Copy-Constructor, wenn das Objekt sehr groß ist, um by-value-Funktionscalls zu vermeiden
- Schlüsselwort "auto" als Typ lässt den Typ vom Compiler bei der Initialisierung bestimmen. Kann jedoch weder const, noch Referenzen repräsentieren.
- In Bitfeldern sorgt ein 0-bitiger Eintrag für Alginment auf das nächste Byte
- In Bitfeldern lassen sich mehrere Einträge zu einem zusammenfassen, z.B: `int b:5, c:11` (Zugriff erfolgt über selben Eintrag)

2 Lecture 2

- Für float kann auch die Beschreibung F , statt f verwendet werden
- Operationen auf float sind wesentlich schneller als double
- Declaration: Stellt die Definition irgendwo im Programm sicher (header)
- Definition: Fordert Speicher für einen Typ an und initialisiert diesen optional (source)
- potential scope: In dem die Variable sichtbar sein könnte, solange sie nicht überdeckt ist
- actual scope: in dem die Variable tatsächlich sichtbar ist, dh nicht komplett überdeckt
- C++-Referenz ist im Grunde ein Alias: `type &id = object;`
Der Unterschied zu einer C-Type-Referenz ist, es kann kein anderes Objekt referenzieren, nachdem es ein Objekt referenzierte
- Nonconst reference kann nur an ein Objekt des identischen Typen der Referenz gehangen werden
- const reference: kann darüberhinaus auch const object, nonconst object, Ergebnis einer Expression
Kann das verbundene Objekt nicht manipulieren

```

1 //ri ist alias für i
2 int i = 1; int &ri = i;
3
4 //funktioniert, da Referenz zu einem rvalue (geht nicht mit nur
   einem '&'), entspricht nun einer move operation
5 int &&ri3 = 10;
6 int i = 42; int &&rr = i*42;
7
8 //const funktioniert, da intern zu int konvertiert
9 double dval = 3.14; int const &ri = dval;

```

Listing 1: Referenzen

- Synonym zu “typedef type synonym”: “using synonym = type”
- decltype: Vererbt quasi den eigenen Typ weiter

```
1 //x und y vom selben Typ
2 type x; decltype(x) y;
3
4 //y ist eine Referenz auf einen Typ wie x
5 decltype((x))y;
6
7 //Referenz zu der Zuweisung
8 decltype(A=B) D = A;
```

Listing 2: decltype

3 Lecture 3

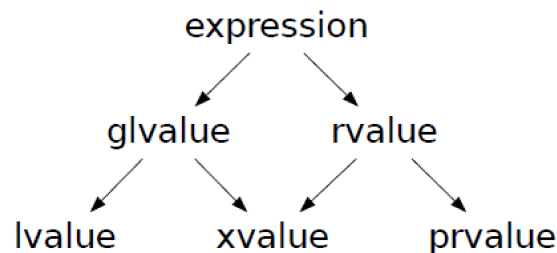


Abbildung 1: Expression-Klassen

lvalue:

Gibt Funktion oder Objekt zurück. Nonconst lvalue ist meist der linke Operand einer Zuweisung. Verfügt meist über eine Adresse. Kann nicht moved werden. Verfügen über eine Identität. Beispiele: `a = b`, `a += b`, `++a`, `a[n]`

xvalue:

Referenziert ein Objekt, das kurz vor dem Tod steht. Man möchte Ressourcen moven. Häufig in Kombination mit rvalue-Expressions. Sind somit Literale oder temporäre Objekte. Die referenzierten Variablen sind allerdings weiterhin lvalues. Können moved werden. Verfügen über eine Identität. Beispiele: `std::move(a)`, `a[n]` wobei `a` ein rvalue

glvalue:

schlicht eine Oberbezeichnung. Verfügt über eine Identität.

prvalue:

“Pure rvalue”, das kein xvalue ist. Verfügen über keine Identität. Können moved werden. Beispiele: `a++`, `a+b`, `a && b`, `a < b`, `a == b`

rvalue:

Gibt einen Wert, nicht jedoch die Adresse zurück. Kann moved werden.

- prefix gibt ein lvalue zurück
- postfix gibt ein rvalue zurück

```
1 //g ist lvalue
2 float &f(){float g; return g;}
3
4 //2 ist prvalue
5 float f(){return 2;}
6
7 //2 ist xvalue
8 float &&f(){return 2;}
```

Listing 3: Expression-Klassen

- Überladene Operatoren behalten ihre Ausführungspriorität. d.h.
cout << (70 < 60)? "fail": "pass";
gibt 1 oder 0 aus
- [Operatoren](#)
- Promotion ist ein Cast in einen größeren Datentyp, ohne Informationsverlust
- Conversion allgemeine Typumwandlung
- static_cast: Häufige Verwendung um implizite Umwandlung vom Compiler zu überschreiben
- const_cast: entfernt const
- dynamic_cast: Dynamischer Typ
- reinterpret_cast: Interpretation des Operations-Inhalts als einen anderen Typen.
Keine Typüberprüfung
- Rechen-Beispiele mit impliziter Typ-Konvertierung
- Einige Print-Preprocessor-Makros
- `__LINE__`


```
1 //long double
2 3.14L + 'a'
3
4 //int, beide werden zu int promoted, bevor die Operation
   ausgeführt wird
5 short+char
6
7 //int
8 unsigned short + int
9
10 //unsigned long
11 unsigned long + long
```

Listing 4: Implizite Typumwandlung

- `__FUNCTION__`
- `__TIME__`
- `__DATE__`

4 Lecture 3a

- constexpr kann zur Compilezeit ausgewertet werden
- Floating Operationen, die zur Laufzeit ausgewertet werden müssen nicht die selben Ergebnisse liefern, wie jene zur Compilezeit, dh:

```

1 char array [1+int(1+0.2-0.1-0.1)]; // compiletime
2 int size = 1+int(1+0.2-0.1-0.1); // maybe at runtime
3 sizeof(array) == size; // muss nicht wahr sein

```

Listing 5: Floatberechnung Compile vs. Runtime

```

1 struct A{
2     constexpr A(int i) : val(i){}
3     constexpr operator int() const {return val;}
4 private:
5     int val;
6 };
7 template<int> struct X{};
8
9
10 constexpr A a = 42; // constructor
11 X<a> x; // verwendet int-operator für das template

```

Listing 6: Cooles Operator-Überladen mit constexpr

```

1 constexpr int factorial(int n){return n <= 1 ? 1:
    (n*factorial(n-1));}
2
3 template<int n>
4 struct {constN(){//do stuff
5 }};
6
7 //kann trotzdem zur Laufzeit ausgewertet werden, wenn Ziel als
    volatile deklariert
8 constN<factorial(4)> result;

```

Listing 7: Template-Metaprogramming mit constexpr

literal class:

Klasse, deren Konstruktor eine constexpr liefert. Die komplette Klasse kann zur Compilezeit ausgewertet werden

```
1 func(const char (&a)[N])
```

Listing 8: Sicherstellung der Größe von dem Eingabe-String a gleich N

allocator:

Interessante Klasse, die via Template: allocator<type>name; die Speicherverwaltung von Elementen vom Typ type vornehmen kann, ohne deren Konstruktor aufrufen zu müssen

name.allocate(amount) //alloziert für amount Elemente vom Typ type den Speicher

name.construct(index) // Ruft den Konstruktor für das index-te Element auf

name.destroy(index)

```
1 uninitialized_copy(start, end, destination) //kopiert die
   Elemente des Bereichs start-end nach destination
2 uninitialized_fill_n(destination, amount, element) // füllt
   amount viele Slots von destination mit element auf
```

Listing 9: Kopieren und befüllen

```
1 allocator<string> alloc;
2 string *sp = alloc.allocate(2);
3 new (sp) string(begin, end) //Erzeugt den String in dem
   raw-Feld mit replacement
```

Listing 10: Placement new

Das Objekt wird in dem übergebenen Speicherbereich erzeugt, es wird also kein neuer Speicherplatz angefordert. Und es erfolgt auch kein unnötiger Kopiervorgang.

Bei smart pointer (löschen sich selbst sobald keine Referenz mehr darauf existiert) wird auf "use count" zurückgegriffen. Das wird automatisch bei new und delete modifiziert.

```
1 class C;
2 void operator=(C &&in){
3 //Übergebener Typ: C&, der nicht mehr movable ist
4     next_proc(in);
5 //soll es movable bleiben, ist explizites Casten notwendig
6     next_proc((C &&)in);
7 }
```

Listing 11: Move-Operator

Soll ein && weitergereicht werden, ist ein expliziter Cast erforderlich, z.B. via `std::move`

5 Lecture 4

- neue Include-Guards: # pragma once, ist nur eine Zeile und unabhängig von define-Konventionen
- `int f(int[10]);` ist äquivalent zu `int f(int*);`
- `int f(int (&arr)[10]);` hier wird tatsächlich die Größe überprüft
- Variable Parameteranzahl:
C: `void f(param_list,...)`
C++: `void f(initializer_list<string>il)`
- funktionslokale statische Variablen werden nur beim ersten Aufruf mit dem zugewiesenen Wert initialisiert und danach bleiben sie bestehen und behalten ihren Wert
- `end(array)`, `begin(array)` sind in der `stdlib` definiert
- `void Class::f() const`, die Modifier an der Stelle von `const` beziehen sich immer auf den "this"-Parameter. Dh. da kann auch eine Referenz stehen

```
1 //pointer to an array of 5 ints
2 int (*arrP)[5] = arrPtr();
3 //reference to an array of 5 ints
4 int (&arrR)[5] = arrRef();
5 //returns a reference to an array of 5 ints, nimmt ein int
   entgegen
6 int (&arrRef(int i))[5]{...}
7
8
9 array = {0,1,2,3,4};
10 decltype(array) *arrPtr(void){
11     return &array;
12 }
13 int(&arrRef(void))[5]{
14     return array;
15 }
```

Listing 12: arrPtr und arrRef

Auflösung von Überladung:

- candidate functions: korrekter Name
- viable functions: Parameterliste stimmt
- Beste wählen aus viable functions
 1. Exakte Übereinstimmung: Argument und Parameter-Typen sind identisch
 2. Promotion
 3. Standard conversions: z.B. double to int
 4. class type conversions

Conversions-Dinge:

- Arithmetische Operationen nehmen keine Typen kleiner als int an, dort erfolgt automatisch eine promotion to int. Beispiel: `char c = 1; typeid(c+c);` ist inter Tat ein int.
- Bei Floating-point conversion, also keine promotion gilt, ist der Typ nicht exakt darstellbar, sondern liegt zwischen zwei darstellbaren Werten ist die Rundung Implementierungssache (default das nähere der beiden)
- Prinzipiell ist immer nur eine Conversion auf einmal erlaubt
- NULL (nullptr) kann in jeden beliebigen Pointer konvertiert werden (einschließlich cv), was in lediglich einer Conversion möglich ist.
- cv-qualifiers (const/volatile) können in C++, anders als in C, tatsächlich zu jeder Ebene eines mehrdimensionalen Pointers hinzugefügt werden. Dafür gibt es komplexe Regeln, die besagen wann dies erlaubt ist.

Lambda Expressions:

- `[capture list](param list) ->return type {function body}`
- return type kann weggelassen werden, wird automatisch vom returnten Typ bestimmt
- capture list ist häufig leer, eine Liste lokaler Variablen, die in der umgebenen Funktion definiert werden.
Die lokale Kopie folgt den gewohnten call-by-reference und call-by-value Semantiken.
Die lokale Kopie darf anscheinend nur dann verändert werden, wenn "mutable" angegeben ist
- auto bei Lambda Funktionen ist wohl ein wenig Magie, undefinierter Typ, der häufig das repräsentiert was man will

```
1 //äquivalente Funktionsdeklarationen, geben int-pointer array
  der Größe 10 zurück
2 using arrtT = int[10];
3 arrT *func(int i);
4 auto func(int i) -> int(*)[10];
5 int(*func(int i))[10];
6
7 //Funktionen als Parameter, nehmen const string ref entgegen
  und ret. bool
8 void fct (bool (*) (const string&));
9 void fct (bool (const string&));
10
11 //Funktion nimmt ein int entgegen und returnt einen Pointer auf
   eine Funktion, die ihrerseits (int*, int) entgegen nimmt
   und ein int returnt
12 int (*ff(int))(int*,int);
13 //äquivalent
14 typedef int (*PF) (int*,int); PF ff(int);
```

Listing 13: verschiedene Funktionsdeklaration

6 Lecture 5

- Forward declaration, incomplete types
Es können lediglich Referenzen oder Pointer dieses Types verwendet werden, keine Objekte.
Nützlich zB um in der Klasse selbst diese zur Beschreibung von Zusammenhängen zu beschreiben
- Die Constructor-Initialiser-List vermeidet in einigen Fällen Doppel-Initialisierung
- Die Initialisierungsreihenfolge ist hierbei IMMER entsprechend der Reihenfolge der Klasse und nicht in der der Constructor-Initialiser-List
- ClassName() = default; baut den Default-Konstruktor, der nicht mehr vom Compiler bereitgestellt wird, sobald ein anderer Konstruktor existiert (benötigt -std=c++11)
- Delegating Constructor - Aufruf eines Konstruktors von einem anderen via Constructor-Initialiser-List
- Es gibt die implizite Konvertierung bei Zuweisung zu einem Constructor-Call, zumindest sofern kein Assignment-Operator überladen wurde
- ↑ dies lässt sich vermeiden, indem der entsprechende Konstruktor als “explicit” deklariert wird
- mutable Data-Member: ist niemals const, auch nicht wenn es ein Member eines Const-Objekts ist
- Sowohl const, als auch Reference-Typen müssen nach dem Verlassen des Konstruktors initialisiert sein
- Friend-Klassen, -Funktionen haben die selben Zugriffsrechte, wie Member
- std::move() - castet zu einem rvalue

7 Lecture 5a

```

1  template<typename A, typename B, typename C>
2  std::function<C(A)> compose(std::function<C(B)>
   f, std::function<B(A)> g){
3      //lambda
4      return [f,g](A x){return f(g(x));};
5  }
6
7  //in C++14 geht auch das hier
8  template <class F, class G>
9  decltype(auto) compose2(F&& f, G&& g){
10     return [=] (auto x){return f(g(x));};
11 }

```

Listing 14: Compose-Funktion die zwei Funktionen verknüpft ausführt: $f(g(x))$

Initialisierungsformen:

- Direct initialisation
- Copy initialisation
- List initialisation - erfordert einen List-Constructor, bei Aufruf dessen darf keine Information verloren gehen. Dh conversion, die keine promotion ist funktioniert nicht

```

1  Class(std::initializer_list<type> h){}

```

Listing 15: Beispiel für Listen-Konstruktor

- Coole Sache: Boolean-Vektoren werden intern mit einer Bitmap gespeichert, was dazu führt, dass man keine Forward-Iteratoren erhalten kann und ein Großteil der Algorithmen aus Bibliotheken nicht funktionieren. Darüberhinaus beeinflusst dies massiv die Größe.
- Argument depended lookup: Ist das Argument einer Funktion einem Namespace zugeteilt und wird einer Funktion übergeben, dann ist der Namespace dieses Arguments ebenfalls für die Funktion ein gültiger.
- Rule of Five:
Wird ein nicht-trivialer Destruktor benötigt, dann meistens auch Copy-Konstruktor, Assignment-Operator, Move-Konstruktor, Move-Assignment-Operator

```
1 std::pair<std::string*,std::string*>
2 StrVec::alloc_n_copy(const std::string *b, const std::string
   *e){
3     auto data = alloc.allocate(e-b);
4     //initialise and return a pair constructed from data and the
       value returned by unitialised data
5     return {data, unitialized_copy(b,e,data)};
6 }
7
8
9 void StrVec::reallocate(){
10    auto newcapacity = size() ? 2 * size() : 1;
11    auto newdata = alloc.allocate(newcapacity);
12    //move the data from the old memory to the new
13    auto dest = newdata; //points to the next free position in
       the new array
14    auto elem = elements; //points to the next element in the
       old array
15    for (size_t i 0; i < size(); i++)
16        alloc.construct(dest++, std::move(*elem++));
17
18    free();
19 }
```

Listing 16: Beispiele einer Vektor-Implementierung

8 Lecture 6

- Prefix-Operator `Type& operator++()`
- Postfix-Operator `Type operator++(int)`
- der `int`-Parameter wird benötigt um die beiden auseinander zu halten, kann allerdings nur beim expliziten `.operator++()` verwendet werden
- es können nicht mehrere implizite `Class-type conversion` direkt aufeinander folgen

9 Lecture 7

- Protected Vererbung: Kein Zugriff von friends der abgeleiteten Klasse
- Default Vererbung bei Structs ist public
- pure-virtual-function: `virtual <type><func>() = 0;` sorgt für abstrakte Klasse
- Wenn Funktionen nicht virtual sind, dann wird der statische Klassen-Typ verwendet:
Base b* = new Derived d;
b->func();
ruft func() der Base auf, wenn diese non-virtual ist
virtual Funktionen sind auch in sämtlichen Unterklassen implizit virtual
- Destructors von abstrakten Klassen sollten stets virtual sein. Dies ist nicht zwingend notwendig, wenn die Klasse pure-abstract ist, weil dann überhaupt keine Objekte erzeugt werden können.
- `class Class final;`
Verhindert Vererbung von Class
- `using Base::Base;`
sorgt dafür, dass für jeden Konstruktor der Base-Klasse ein Konstruktor in der Derived Klasse mit der selben Parameterliste angelegt wird, der die Parameter an die entsprechenden Constructoren der Base-Klass weiterreicht.
- In Assignment-Copy-Operatoren der Unterklasse muss auch entsprechender Code für die Oberklasse enthalten sein
- `dynamic_cast` castet vom Base-Type zu einem Derived-Type. Erfordert anscheinend, dass die Base-Klasse abstract ist
- `typeid`, `type_info` nützliche Dinge

```
1 using std::cout; using std::endl;
2 template<class T> struct B : A<T> {
3     void print (T a)                { cout << "6" << endl; }
4     static void print (float a)     { cout << "7" << endl; }
5 };
6
7 template<> void B<int>::print (int a) { cout << "8" << endl; }
8
9 void main() {
10     B<int> b; B<int>* ptr = &b;
11     ptr->print( 1 );
12 }
```

Listing 17: Bei Template-Dingen, selbst wenn die Funktion templatisiert innerhalb der templatisierten Klasse bereits existiert und der Typ passt, dann wird falls extern eine template-specialisation Funktion definiert ist diese aufgerufen, weil diese spezieller ist.

```
1 struct A {
2     using std::cout; using std::endl;
3     void print(int a, char b = '0')
4         { cout << "int" << endl; }
5     void print(float a, char b)
6         { cout << "float" << endl; }
7
8     static void print(double a)
9         { cout << "static double" << endl; }
10    static void print(int a, char b = '0')
11        { cout << "static int" << endl; }
12 };
```

Listing 18: statische Funktionen mit den selben Parametern wie non-static können nicht überschrieben werden

10 Lecture 7a

virtual inheritance:

Mehrfachvererbung, wobei die Unterklasse eine einzige Kopie einer Basis-Klasse enthält, die innerhalb der Vererbungs-History mehrfach enthalten ist.

virtual base class:

- Eine Basis-Klasse, die mittels “virtual” vererbt wurde. Diese ist lediglich einmal in dem Unterklassen-Objekt enthalten, selbst bei häufigerem Vorkommen als virtual Base.
- Bei nicht-virtual-Vererbung initialisiert der Konstruktor womöglich lediglich die direkte Basis-Klasse.
- Die Initialisierung von virtuellen Basis-Klassen erfolgt durch die niedrigste Unterklasse, die entsprechende Initialisierungen explizit vornehmen muss.

constructor order:

- Aufruf der Konstruktoren erfolgt in der Reihenfolge der Derivation-List (nicht constructor-initialiser list).
- Diese Initialisierung sollte explizit durch die constructor-initialiser-list erfolgen.
- Konstruktoren virtueller Basisklassen werden vor den anderen aufgerufen

10.1 Vorstellung des Projekts

- Gebäude \mapsto Production facilities
 - Name
 - Kosten
 - Bereitgestellte Unterkunft
 - Bauzeit
 - Produzierbare Einheiten
 - Bedingte Gebäude, um dieses zu errichten
 - Mögliche Transformation in andere Gebäude
- Einheiten \mapsto Products
 - Name
 - Kosten
 - Bereitgestellte/Benötigte Unterkunften

- Bauzeit
 - In welchem Gebäudetypen werden sie erzeugt
 - Welche Gebäude werden zur Produktion benötigt
 - Mögliche Transformationen in andere Einheiten/Gebäude
- Minerals and Vespine Gas \leftrightarrow Resources

Forward Simulation:

Für jeden Schritt:

1. Update globale Ressourcen
2. Starte neue Produktion (sofern möglich), hänge diese in die Active-Action-List. Suche Einheiten und weise diesen zu den freien Stellen in einer geeigneten production facility zu
3. Aktualisiere Active-Action-List. Produzierte Einheiten aus der Action-Liste entfernen. Entsprechende Veränderungen buchen.

Optimisation of Build Lists:

Geeignete Ansätze

- Exhaustive Search
- Branch and Bound
- Genetic Algorithm

11 Lecture 8

- class template vs function template
- nontype parameter (value) vs type parameter
- implizit mögliche Conversions:
 - const conversion
 - array/function to pointer conversion

Promotions werden nicht gemacht? Kann sein, sobald das Argument selbst templatisiert ist. sonst müsste es gehen

-
- Template Specialisation: Extra Fall für spezielle Typen

```

1 template<typename T1, typename T2> void ff() {}
2 //anscheinend muss man hier tatsächlich alle explizit angeben
3 template <> void ff<int,int>() {}
4 //und deswegen geht das hier auch nicht, da partial
  specialisation nicht erlaubt ist
5 template <typename T1> void ff<T1,int>() {}
6
7 //bei Klassen hingegen geht das
8 template<typename T1, typename T2> class Y{};
9 template<typename T1> classY<T1, int>{};

```

Listing 19: Template Specialisation - Partial Specialisation ist nur bei Klassen erlaubt

```

1 template<typename T>void f3(T t){}
2 template <typename T> void f3(T*) {}
3 template <> void f3<int*>(int* a) {}
4
5 int* ip = new int [2];
6 //ruft T* auf, weil die Specialisation eine Spezialisierung der
  ERSTEN Funktion ist und beim Finden des "best match"
  betrachtet der Compiler die beiden Generic-Funktionen,
  wovon die T* den besseren Match liefert.
7 f3(ip);

```

Listing 20: Template Specialisation kann sehr tricky sein


```

1 template<typename T1, typename T2>class Y{
2 public:
3     template<typename TT>void print(TT t);
4 };
5 //zwei template-Listen, deren Reihenfolge egal ist
6 template<typename T1, typename T2>template<typename TT>
7 void Y <T1, T2>::print(TT);

```

Listing 21: Template Function in template class

```

1 template<typename T1>class X{
2 public:
3     //hier wüsste der Compiler nicht ob mem eine Member-Variable oder
4     //ein typedef ist. Dh. er weiss nicht ob das eine Multiplikation
5     //oder eine Referenz auf T::mem sein soll
6     //T::mem *p1;
7     //das hier sind die beiden Dinge, die für sich genommen
8     //funktionieren
9     static int const i = T::mem *p; //T::mem muss eine Variable sein
10    typename T::mem *p1; //T::mem muss ein type sein
11 };

```

Listing 22: typename als Hilfe zur Konfliktauflösung

Trailing return types listing 23:

Zu Verwenden bei unbekanntem Return-Typen.

```

1 template<typename It>
2 auto fcn(It beg, It end)->decltype(*beg){return *beg;}

```

Listing 23: Trailing Return Type Beispiel

Der Rückgabotyp steht hinter der Parameterliste

Forwarding `std::forward<Type>(arg)` listing 24:

Ein Argument unverändert an die nächste Funktion weiterreichen. Dh const, non-const, lvalue, rvalue, ... bleibt bestehen

```

1 //hier würde man sonst const und references verlieren
2 template<typename F, typename T1, typename T2>
3 void flip(F f, T1 && t1, T2 && t2){
4     f(std::forward<T2>(t2), std::forward<T1>(t1));
5 }

```

Listing 24: Forwarding Beispiel

variadic template listing 25:

Template-Funktion/Klasse, die eine variable Anzahl Parameter entgegen nehmen kann.

```

1 template<typename T, typename ... Args>
2 void f(const T & t, const Args & rest);
3
4 template<typename T>
5 ostream &print(ostream &os, const T &t){
6     return os<<t;
7 }
8 template<typename T, typename... Args>
9 ostream &print(ostream &os, const T &t, const Args&... rest){
10     os<<t<<" ";
11     return print(os, rest...);
12 }

```

Listing 25: Variadic Template-Funktion. Erlaubt auch ziemlich coole Rekursionen

Member Template:

Kann nicht virtual sein. Entweder ein Klassen-Member oder ein Class-Template, das ein Funktions-Template ist. Man könnte es indirekt virtual machen, indem sie von einer Virtual-Funktion aus aufgerufen wird

12 Lecture 9

- IO-Objekte können nicht assigned werden
- condition state. Flag der Stream-Klassen, das angibt, ob der Stream korrumpiert ist oder valide
 - fail: initialisation-fail, format-Error, Beispiel String to int, ermöglicht weitere Lese-Versuche mit anderen Parametern
 - bad: Böse Dinge sind passiert, Verwendung des Streams nicht mehr möglich
 - strm::iostate, strm::badbit, strm::failbit, strm::eofbit
 - s.good(), s.bad(), s.fail(), s.eof()
 - s.clear(), s.setstate(flag), s.rdstate()
- stringstream scheint recht nützlich zu sein
- Wenn ein Output-Stream an ein Input-Stream gebunden (`stream.tie(&stream2)`) wird, dann wird dieser immer dann geflusht, wenn vom Input-Stream gelesen wird
- Beim Program-Crash wird nicht geflusht
- Deque (Double-ended queue) scheint wohl schneller zu sein, kann schnell vorne und hinten anfügen.
Es erfolgt beim Einfügen/Löschen keine Verschiebung von Elementen

Schließt dies auch den Reallocation-Fall mit ein?

- List (Double-Linked List) ist prinzipiell relativ schnell bei Einfüge-/Löschoperationen
Invalidierung erfolgt nur für diese Iteratoren, die auf entfernte Elemente verweisen. Bei Vektor werden alle Iteratoren, die auf ein beliebiges Element verweisen invalidiert (nicht jedoch bei swap).
- push: Speichert eine Kopie in dem Container
emplace: Legt das Objekt erst im Container an

```
1 //string, flush
2 cout << " " << flush;
3
4 //string, null, flush
5 cout << " " << ends;
6
7 //aktiviere unmittelbare Flush-Operationen nach input
8 cout << unitbuf;
```

Listing 26: Stream-Flush

13 Lecture 9a

- Im Header stehende Template-Funktionen, wobei die Header in mehrere Source-Files inkludiert werden, werden unter Umständen mehrfach kompiliert, nur um später weggeworfen zu werden.
Dies lässt sich vermeiden, indem die entsprechende Funktion mittels `extern` und `template`-Angaben in allen bis auf einer Datei bekannt gemacht wird.
- Type traits (Type-Checking `is_<type>`, z.B. `is_pointer`) läuft zur Compile-Zeit durch
- Man kann Class-Templates auch via verschachtelte-Class-Templates in die Class-Definition einer verwendenden Klasse angeben (vgl. listing 27)
- Nette Fibonacci-Template-Metaprogramming (vgl. listing 28)

```
1 template<template<unsigned long> class T>
2 class Loop{ void function(); };
3
4 template<unsigned long N> class Fibonacci<N>{...};
5
6 Loop<Fibonacci>::function();
```

Listing 27: Verschachtelte Class-Templates

`fibonacci` ist also das innere Template. Die innere Template-Class erspart es uns jedesmal `Fibonacci<unsigned long>` zu schreiben.

```

1  template<unsigned long beg, unsigned long end,
   template<unsigned long>class T>
2  class Loop{
3  public:
4      template<typename Container>
5      static void run(Container &v){
6          //value sind die Fibonacci-Zahlen, diese werden woanders
           berechnet (s.u.)
7          v.push_back(T<beg>::value);
8          Loop<beg+1, end, T>::run(v);
9      }
10 };
11
12 //template specialisation, Abbruchbedingung
13 template<unsigned long end, template<unsigned long>class T>
14 class Loop<end, end, T>{
15 public:
16     template<typename Container>
17     static void run(Container &v){}
18 };
19
20
21 template<unsigned long N>
22 class Fibonacci{
23 public:
24     static const int value =
           Fibonacci<N-2>::value+Fibonacci<N-1>::value;
25 //alternativ
26     enum{value2 = Fibonacci<N-2>::value2 +
           Fibonacci<N-1>::value2};
27 };
28
29 //specialisation 0
30 template<>
31 class Fibonacci<0u>{
32 public:
33     static const int value = 0;
34     enum{value2 = 0};
35 };
36 //specialisation 1
37 template<>
38 class Fibonacci<1u>{
39 public:
40     static const int value = 1;
41     enum{value2 = 1};
42 };

```

Listing 28: Template-Meta-Programming Fibonacci mit Klassen

14 Lecture 10

```

1 sort(words.begin(), words.end());
2 //Überschreiben von aufeinanderfolgenden Duplikaten mit dem
  nächsten von den Duplikaten unterschiedlichen Wert. Das
  Ende bleibt unverändert, somit auch die Größe.
3 vector<string>::iterator endunique =
  unique(words.begin(), words.end());
4 //das Ende (die bereits vorgezogenen Werte) löschen
5 words.erase(endunique, words.end());

```

Listing 29: Duplikate entfernen

Dank des Sortiervorgangs entfernt dies tatsächlich sämtliche Duplikate. Ohne diesen würden lediglich die aufeinanderfolgenden Duplikate entfernt.

```

1 //syntax: auto newCallable = bind(callable, arg_list)
2 #include <functional> //std::ref
3 using namespace std::placeholders; //_1
4
5 string s;
6 vector<string> words;
7 while(cin >> s)
8     words.push_back(s);
9 for_each(words.begin(), words.end(),
10         bind(print, ref(cout), _1, ' '));
11
12 //for_each funktioniert auch mit lambda-Funktionen (s.u.
  Algorithmen-Syntax)
13 for_each(words.begin(), words.end(),
14         [](string s){cout << s << " ";});
15
16 ostream &print(ostream &os, const string &s, char c){
17     return os << s << c;
18 }

```

Listing 30: Parameter an eine Funktion mappen, Parameter sind call by value, sofern nicht using ref(arg) oder cref(arg) (für const ref) verwendet wird

```

1 ifstream is("data/outFile1");
2 istream_iterator<string> in (is), eof;
3 for_each(in, eof, bind(print, ref(cout), _1, '\n'));

```

Listing 31: Auslesen eines Inputfile(Stream)s via Iterator, Ausgabe erfolgt Wortweise, jeweils in einer neuen Zeile.

```

1 vector<int> vec;
2 //insert elements at the tail
3 auto it = back_inserter(vec);
4 //equivalent to vec.push_back(42)
5 *it = 42;
6 //front_inserter verwendet push_front

```

Listing 32: Insert iterator

Iterators:

- Iterators sollten allgemein nicht dereferenziert werden, da sie auf nicht-existierende Elemente verweisen können.
- reverse iterator - bewegt sich von hinten durch die Folge und invertiert entsprechend die Bedeutung von ++ und --
- Iterator-Hierarchie
 1. input iterator - kann Elemente nur lesen, nicht schreiben
 2. output iterator - kann Elemente nur schreiben, nicht lesen
 3. forward iterator - kann Elemente lesen und schreiben, aber "--" ist nicht unterstützt
 4. bidirectional iterator - forward iterator mit Support für "--"
 5. random-access iterator - bidirectional iterator mit der Fähigkeit Iterator-Werte mit relationalen Operatoren zu vergleichen und arithmetische Operationen auf dem Iterator auszuführen. Entsprechend wird random access unterstützt.
- Algorithmen-Syntax
 - alg(begin, end, params);
 - alg(begin, end, dest, params);
 - alg(begin, end, beg2, params); (zwei Iteratoren)
 - alg(begin, end, beg2, end2, params); (zwei Iteratoren)


```
1 vector<int> vi;
2 int i;
3 while(cin >> i)
4     vi.push_back(i);
5
6 //läuft durch den ersten Iterator und ersetzt im selben Index
   //des zweiten Iterator den Wert mit dem absoluten Wert
7 transform(vi.begin(), vi.end(), vi.begin(),
8           [](int i){return i < 0 ? -i : i;});
```

Listing 33: Algorithmen transform

- `inserter(<List>, <List-Insert-Start-Position>)` - Keine Ahnung weshalb hier beides benötigt wird
- associative container - Object-Collection mit effizientem key-lookup (nach Key sortiert)
set, map, multiset, multimap
 - multiset, multimap
 - * `lower_bound(k)` - Iterator zu dem ersten Element, dessen Key-Wert mindestens k ist. Falls also k nicht enthalten ist, dann verweist dies auf das erste Element mit einem Größeren Key als k
 - * `upper_bound(k)` - Erste Element mit einem Key größer k
 - * `equal_range(k)` - Ein Iterator-Paar das `lower_bound(k)`, `upper_bound(k)` entspricht
- unordered container - Verwenden Hashing, und keine Ordnung der Schlüsselwerte
`unordered_map`, `unordered_set`, `unordered_multimap`, `unordered_multiset`

```
1 vector<int> vec(10); //initialises to 0 per default
2 fill(vec.begin(), vec.end(), 1); //set all values to 1
3
4 //c{begin,end} returnt einen const_iterator, dieser kann die
   Werte des Containers nicht verändern, auch nicht wenn
   dieser non-const ist.
5 //aufsummieren aller Werte, wobei die Summe mit 0 beginnt
6 int sum = accumulate(vec.cbegin(), vec.cend(), 0);
7 //falls vec ein vector<double> ist, dann müsste der Startwert
   angepasst werden:
8 int sum2 = accumulate(vec.cbegin(), vec.cend(), 0.);
9
10 //erzeuge 10 Elemente am Ende des Vektors mit dem Wert 42
11 fill_n(back_inserter(vec), 10, 42);
12
13 //accumulate funktioniert auch auf Strings
14 vector<string> v;
15 ...
16 string concat = accumulate(v.cbegin(), v.cend(), string(""));
17
18
19 //Initialisierung mittels istream_iterator
20 istream_iterator<int> in_iter(cin), eof;
21 vector<int> vec(in_iter, eof);
22
23 //Ausgabe mittels ostream_itertor
24 ostream_iterator<int> out_iter(cout, " ");
25 copy(vec.begin(), vec.end(), out_iter);
26
27
28 //alternativ
29 for(auto e: vec)
30     *out_iter++ = e;
```

Listing 34: Vector-Dinge: fill, fill_n, accumulate, copy

```
1 //count the number of times each word occurs in the input
2 //map from string to size_t
3 map<string, size_t> word_count;
4 string word;
5 while(cin >> word)
6     ++word_count[word];
7
8 for(const auto &w : word_count)
9     cout << w.first << " occurs " << w.second << " times" <<
    endl;
```

Listing 35: Container Dinge

15 Lecture 11

- Vorteil von Exception gegenüber Fehlercodes ist im Grunde die Trennung von Fehler-Auftreten und -Behandlung.
- Im Destruktor sollten keine Exceptions geworfen werden, da sonst die Destruktoren, die diesen verwenden ihre exception-safety verlieren.
- catch anything mit "...": trycatch(...)
- function-try-block: `f(int i): g(i)catch;` Fängt auch den Aufruf der *g*-Funktion der Constructor-initialiser list ab.

Exception Safe:

Programm verhält sich auch bei geworfenen Exceptions korrekt.

Basic guarantee:

Sämtliche Objekte können nach einer Exception entweder weiter korrekt verwendet oder zerstört werden.

Strong guarantee:

commit/rollback Semantik. Keine Seiteneffekte, z.B. werden Iteratoren nicht invalidiert.

Nofail guarantee:

Es kann keine Exception auftreten

- Diese Garantien können mittels Objekten gegeben werden. Im Constructor werden Ressourcen angefordert und im Destructor wieder freigegeben.
- `f(new T1, new T2);` kann zu einem Memory-Leak führen: Speicher für beide Objekte wird angefordert und beim erzeugen von T1 fällt eine Exception.
Also stattdessen: listing 36

```
1 void f(shared_ptr<T1>, shared_ptr<T2>);
2 shared_ptr<T1> t1(new T1);
3 shared_ptr<T2> t2(new T2);
4 f(t1, t2);
```

Listing 36: Exception safe Code mit shared pointers.

Es ist wichtig, dass die Objekte NICHT in dem Aufruf angelegt werden.

```
1 //nicht exception safe
2 template<typename T> T Stack<T>::pop(){
3     if(!vused)
4         throw std::runtime_error("pop from empty stack");
5     else{
6         //man verliert bei einem Fehler Elemente
7         T result = v[vused-1];
8         //ich weiß nicht genau wo hier eine Exception auftreten
           können sollte, aber ein underflow ist möglich,
           wodurch Objekte verloren werden können. Dies ist auch
           der allgemeine Gedanke hier: Fail nach dem decrease
           => Objekt nicht returnt, aber decreased.
9         --vused;
10        return result;
11    }
12 }
13
14
15 //Also in einzelne Operationen spalten
16 template<typename T> T&Stack<T>::top(){
17     if(!vused)
18         throw std::runtime_error("empty stack");
19     return v[vused -1];
20 }
21
22 template<typename T> void Stack<T>::pop(){
23     if(!vused)
24         throw std::runtime_error("pop from empty stack");
25     else
26         --vused;
27 }
```

Listing 37: Exception safe durch Trennung der Instruktionen

```
1 void f() noexcept {  
2     throw exception();  
3 }
```

Listing 38: noexcept

Das Versprechen keine Exception zu werfen wird gebrochen. Das gibt trotzdem höchstens eine Compile-Warnung. Zur Laufzeit terminiert dies das Programm

16 Lecture 12

```

1 int a, b;
2 int first, second, third, fourth;
3 auto tup3 = tie(first, second, third, fourth) =
    std::make_tuple(1001, 1002, 1003, 1004);
4 tie(std::ignore, a, std::ignore, b) = tup3;
5 //mutation des Elements first
6 get<0>(tup3) = 2;

```

Listing 39: tuple, tie, make_tuple, ignore

first bis fourth enthalten die einzelnen Tupel-Komponenten, a und b das zweite bzw. vierte. Ganz zum Schluss enthält first die 2. Das erste tie ist nicht Notwendig verdeutlicht aber dessen Funktionalität.

```

1 emailDescription = ".....";
2 //regex for email address
3 string regExprStr(R"((\w+(\.|\_)?\w*)@(\w+\.\w+)+)");
4 //regex holder
5 regex rgx(regExprStr);
6 //search result holder
7 smatch smatch;
8 //search, dies matcht auf das LETZTE Vorkommen des Regex, dh.
    vorherige Treffer werden überschrieben
9 if(regex_search(emailDescription, smatch, rgx)){
10     cout << "whole address: " << smatch[0] << "\n";
11     cout << "local part (before @): " << smatch[1] << "\n";
12     cout << "domain name (after @): " << smatch[3] << "\n";
13     cout << "string before the match: " << smatch.prefix() <<
        "\n";
14     cout << "string after the match: " << smatch.postfix() <<
        "\n";
15 }
16
17 //um alle Matches zu bekommen:
18 sregex_token_iterator itBegin(emailDescription.begin(),
    emailDescription.end(), regExprStr);
19 const sregex_token_iterator itEnd;
20 while(itBegin != itEnd)
21     cout << *itBegin++ << endl;

```

Listing 40: Regex

Eine nette Möglichkeit im 14er Standard zur Compile-Zeit die korrekte Funktion zu bestimmen sind restrictions und type traits.

Für Templates sollten nach Möglichkeit stets `static_assert`s verwendet werden

```
1 list<int> allOdd;
2 deque<int> allEven;
3 vector<int> myVec = ...;
4 //odd ints to list allOdd and even ints to deque allEven
5 partition_copy(myVec.begin(), myVec.end(),
6               back_inserter(allOdd), back_inserter(allEven),
7               [](int i){return i % 2;});
8
9 //weitere funktionen
10 //true falls alle Elemente ungerade
11 bool b = all_of(myVec.begin(), myVec.end(),
12               [](int i){return i%2;});
13 //true falls alle Elemente gerade
14 bool c = none_of(myVec.begin(), myVec.end(),
15                 [](int i){return i%2;});
16 //kopiert alle ungeraden Elemente nach cout
17 copy_if(myVec.begin(), myVec.end(),
18         ostream_iterator<int>(cout, " "), [](int i) {return
19         i%2;});
19 // }}}}))) <- fix vim-highlighting
```

Listing 41: lambda foo mit `back_inserter`, `partition_copy`, `all_of`, `none_of`, `copy_if`
Kopiert alle ungeraden Elemente von `myVec` in `allOdd` und alle geraden in `allEven`.

17 Lecture 12a

Expression Templates:

dienen dazu vom Compiler generierte temporäre Variablen zu vermeiden. Baut zur Compilezeit einen Syntax-Tree
Dafür wird ein die Baum-Logik benötigt in Form von terminals and non-terminals

Expression Functors:

Nesting function objects und äußerer operator() ruft inneren operator() auf functional objects. siehe listing 42

- push_back konstante Zeit-Komplexität, wohingegen vector::insert lineare hat
- niemals mehrere Dinge in einem Statement allozieren (Exceptions)
- ValueClass - sollte vorweisen: Copy-controll, keine Baseklasse sein und keine virtual functions, Verwendung auf dem Stack oder in anderen Klassen
- BaseClass - virtual destructor, nonpublic copy-controll, virtual functions, Verwaltung dynamisch auf dem Heap. Verwendet mittels smart-Pointer in Form eines Objekts der abgeleiteten Klasse.
- TraitsClass - Templates, die Type-Informationen mit sich führen, ausschließlich typedefs und static functions, keine veränderbaren Status oder virtuals, keine Instantiierung
- PolicyClass - Fragmente in Form von Templates, keine Instantiierung
- ExceptionClass - ungewöhnlicher Mix von Value und reference-Semantiken, thrown by value sollte von caught by reference aufgefangen werden, no-fail constructors, virtuelle Funktionen, erbt häufig virtuell von std::exception
- Composition sollte tight coupling (friend function, ...) vorgezogen werden, da dies bei Vererbungen besser ist
- private Members sind “nicht zugreifbar”, nicht jedoch “unsichtbar”, dies kann mittels Pimpl idiom erreicht werden, was für Information-Hiding und Compiler-Grenzen sorgt. Versteckt hinter opaque pointer (optimalerweise smart pointer) Kann buildtime verringern, vermeidet ambiguity, Beispiel listing 47
- dynamic polymorphism - static type checking, dynamic binding, separate compilation
- static polymorphism - static type checking, static binding, verschiedene Typen, compile-time evaluation → Optimierungen

```
1 //expression abstract base class -> dynamic polymorphism
2 struct Expression{
3     virtual double operator() (double) = 0;
4     virtual Expression* clone() = 0;
5     virtual ~Expression(){}
6 };
7 //functor for Constant_1 expression
8 struct Constant_1 : Expression{
9     Constant_1 (const double& d): d_(d){}
10    double operator()(double){ return d_; }
11    const double d_;
12
13    Expression *clone(){ return new Constant_1(*this); }
14 };
15 //functor for Variable_1 expressions
16 struct Variable_1 : Expression{
17     double operator()(double x){ return x; }
18
19     Expression *clone(){ return new Variable_1(*this); }
20 };
21 //functor for complex expressions
22 template<class Op>
23 struct ComplexExpression : Expression{
24     //subtrees
25     Expression *l_;
26     Expression *r_;
27
28     ComplexExpression(Expression& l, Expression& r) :
29         l_(l.clone()), r_(r.clone()){}
30
31     ~ComplexExpression(){ delete l_; delete r_; }
32     double operator()(double d){
33         //apply Operator vom template
34         return Op::apply((*l_)(d), (*r_)(d));
35     }
36 };
```

Listing 42: Expression Functor, run-time

```
1 struct ExpressionRef{
2     ExpressionRef(Expression *ptr) : sp_(ptr), ref_cnt_(1){}
3     ~ExpressionRef(){
4         if(--ref_cnt_ == 0){
5             delete sp_;
6             sp_ = 0;
7         }
8     }
9
10    ExpressionRef(const ExpressionRef &source) : sp_(source.sp_){
11        ++ref_cnt_;
12    }
13
14    inline ExpressionRef &operator=(const ExpressionRef &rhs){
15        if(this != &rhs){
16            if(--ref_cnt_ <= 0){
17                delete sp_;
18                sp_ = 0;
19            }
20            sp_ = rhs.sp_;
21            ++ref_cnt_;
22        }
23        return *this;
24    }
25
26    double operator()(double d){
27        return (*sp_)(d);
28    }
29
30    ExpressionRef clone(){
31        ExpressionRef copy(sp_>clone());
32        return copy;
33    }
34
35    inline Expression *get(){
36        return sp_;
37    }
38    inline Expression &getref(){
39        return *sp_;
40    }
41    inline operator void*(){
42        return sp_;
43    }
44
45 private:
46     Expression *sp_;
47     int ref_cnt_;
48 };
```

Listing 43: Erweiterung zu listing 42

```

1 //integration function (average of upper bound and lower bound
  sums)
2 template<class Func>
3 double integrate(Func f, double low, double high, double
  epsilon = 0.001){
4   assert(low <= high);
5   double auc1 = 0, auc2 = 0;
6   double running_x = low;
7   while(running_x < high){
8     auc1 += f(running_x) * epsilon;
9     running_x += epsilon;
10    auc2 += f(running_x)* epsilon;
11  }
12  return (auc1+auc2)/2
13 }
14
15 //add and multiply (schlechte Realisierung, da duplicated
  code), Entfernung unten mittels type traits
16 struct Add{
17   static double apply(double l, double r){ return l+r; }
18 };
19 struct Multiply{
20   static double apply(double l, double r){ return l*r; }
21 };
22 //operatoren
23 ExpressionRef operator*(Expression &l, Expression &r){
24   return ExpressionRef(new ComplexExpression<Multiply>(l, r));
25 }
26 ExpressionRef operator*(Expression &l, Expression &r){
27   return l.getref()*r.getref();
28 }
29 ExpressionRef operator*(Expression &l, const double d){
30   return l * Constant_1(d);
31 }
32 ExpressionRef operator*(const double d, Expression &r){
33   return Constant_1(d * r);
34 }
35 ExpressionRef operator*(ExpressionRef &l, Expression &r){
36   return l.getref()*r;
37 }
38
39 //Verwendung mittels dynamic polymorphism
40 //ruft call operator auf, Auswertung zur Laufzeit, da virtual
  function
41 Variable_1 x;
42 ((2*x*x + 3*x + 3)*(2*x*x + 3*x + 3))(2);
43 integrate((2*x*2x + 3*x + 3), 0, 1);
44 integrate((2*x*x + 3*x + 3)*(2*x*x + 3*x + 3), 0, 1);
45 integrate(x*x, 0, 7);

```

Listing 44: Erweiterung2 zu listings 42 and 43, run-time

```

1 struct Constant{
2     double d_;
3     Constant(double d): d_(d){}
4     double operator() (double){ return d_; }
5 };
6 struct Variable{
7     double operator() (double d){ return d; }
8 };
9 //für non-terminal, complex expression
10 template<class E1, class E2, class Op>
11 struct ComplexExpr{
12     ComplexExpr(E1 l, E2 r): l_(l), r_(r){}
13     double operator() (double d){
14         return Op::apply(l_(d), r_(d)); }
15     E1 l_;E2 r_;
16 };
17 //kapselt alle expression objekte, bool gibt an ob non-/literal
18 template<class E, bool b>
19 struct Expr{
20     E e_;
21     Expr(l& e) : e_(e){}
22     double operator()(double d){ return e_(d); }
23 };
24 //specialisation
25 template<class E>
26 struct Expr<E, false>{
27     double e_;
28     Expr(double const &e) : e_(e){}
29     double operator()(double d){ return e_(d); }
30 };
31 template<class E>
32 struct Expr<E, true>{
33     double e_;
34     Expr(double const &e) : e_(e){}
35     double operator()(double d){ return e_; }
36 };
37 //arithmetic operators - nur noch einer, da type tracing
38 template<class E1, class E2>
39 Expr<ComplexExpr<Expr<E1, is_arithmetic<E1>::value>,
40     Expr<E1, is_arithmetic<E2>::value>, Add>,
41     is_arithmetic<ComplexExpr<Expr<E1, is_arithmetic<E1>::value>,
42     Expr<E2, is_artihmetic<E2>::value>, Add>>::value>
43 operator+(E1 e1, E2 e2){
44     typedef ComplexExpr<Expr<E1, is_arithmetic<E1>::value>,
45     Expr<E2, is_arithmetic<E2>::value>, Add>ExprType;
46     return Expr<ExprType, is_arithmetic<ExprType>::value>(
47         ExprType(Expr<E1, is_arithmetic<E1>::value(e1),
48         Expr<E2, is_arithmetic<E2>::value(e2)));
49 }

```

Listing 45: Erweiterung3 zu listings 42 and 43, compile-time + templates

```
1 //Verwendung mittels static polymorphism
2 Variable y;
3 (<+3.1)(2);
4 ((2*y*y+3*y+3.0)*(2*y*y+3*y+3))(2);
5 integrate((2.0*y*y+3*y+3), 0, 1);
6 integrate((2*y*y+3*y+3)*(2*y*y+3*y+3), 0, 1);
7 integrate(y*y, 0, 7);
```

Listing 46: Erweiterung zu listing 45, compile-time + templates

```
1 class Map{
2 private:
3     struct Impl; //forward decl
4     shared_ptr<Impl> pimpl_;
5 };
```

Listing 47: pimpl

18 Lecture 13

- condition variable erlaubt Kommunikation mehrerer Prozesse mithilfe von Mutexen.
- `std::shared_future` ist wie `std::future`, aber zusätzlich können mehrere Threads auf den selben Zustand warten
- Threads können sogar gewaped werden, warum auch immer man das tun wollte
- aktueller Thread: `std::this_thread`

```
1 int calculate(){ return 42; }
2 //startet Ausführung sofort
3 auto future = async(launch::async, calculate, <parameter>);
4 //startet später, meistens, wenn man das Ergebnis erfragt
5 auto future = async(launch::deferred, calculate, <parameter>);
6 //unspecified policy
7 auto future = async(calculate, <parameter>);
8
9 //cout thread-safe gestalten
10 cout.sync_with_stdio(true);
11
12 atomic<int> counter(0);
13 ++counter;
```

Listing 48: `async` - erlaubt keine Kommunikation über mehrere Prozesse

- `lock_guard`: Verwendet mutex, lock im constructor, free im destructor. Ist schneller als `unique_lock`
- `unique_lock`: genauso, aber kann auch später gelockt, unlocked und verschoben werden. Das ist notwendig wenn es sie condition variablen verwendet werden. Ist flexibler als `lock_guard`

```
1  std::unique_lock<std::mutex>guard1(<someMutex1>,
   std::defer_lock);
2  std::unique_lock<std::mutex>guard2(<someMutex2>,
   std::defer_lock);
3  //lockt beide gleichzeitig
4  std::lock(guard1, guard2);
5
6
7  std::mutex mutex_;
8  std::condition_variable condVar;
9  bool dataReady;
10 std::unique_lock<std::mutex> lck(mutex_);
11 //wartet bis dataReady true ist
12 condVar.wait(lck, []{return dataReady;});
13
14 //notify the waiting ones (in einer anderen Funktion)
15 std::lock_guard<std::mutex> lck(mutex_);
16 dataReady = true;
17 condVar.notify_all();
```

Listing 49: unique_lock

Listings

1	Referenzen	5
2	decltype	6
3	Expression-Klassen	8
4	Implizite Typumwandlung	9
5	Floatberechnung Compile vs. Runtime	10
6	Cooler Operator-Überladen mit constexpr	10
7	Template-Metaprogramming mit constexpr	10
8	Sicherstellung der Größe von dem Eingabe-String a gleich N	11
9	Kopieren und befüllen	11
10	Placement new	11
11	Move-Operator	12
12	arrPtr und arrRef	13
13	verschiedene Funktionsdeklaration	15
14	Compose-Funktion die zwei Funktionen verknüpft ausführt: $f(g(x))$	17
15	Beispiel für Listen-Konstruktor	17
16	Beispiele einer Vektor-Implementierung	18
17	Bei Template-Dingen, selbst wenn die Funktion templatisiert innerhalb der templatisierten Klasse bereits existiert und der Typ passt, dann wird falls extern eine template-specialisation Funktion definiert ist diese aufgerufen, weil diese spezieller ist.	21
18	statische Funktionen mit den selben Parametern wie non-static können nicht überschrieben werden	21
19	Template Specialisation	24
20	Template Specialisation kann sehr tricky sein	24
21	Template Function in template class	25
22	typename als Hilfe zur Konfliktauflösung	25
23	Trailing Return Type Beispiel	25
24	Forwarding Beispiel	26
25	Variadic Template-Funktion. Erlaubt auch ziemlich coole Rekursionen	26
26	Stream-Flush	28
27	Verschachtelte Class-Templates	29
28	Template-Meta-Programming Fibonacci mit Klassen	30
29	Duplikate entfernen	31
30	Parameter an eine Funktion mappen, Parameter sind call by value, sofern nicht using ref(arg) oder cref(arg) (für const ref) verwendet wird	31
31	Auslesen eines Inputfile(Stream)s via Iterator und Ausgabe	32
32	Insert iterator	32
33	Algorithmen transform	33
34	Vector-Dinge: fill, fill_n, accumulate, copy	34
35	Container Dinge	35
36	Exception safe Code mit shared pointers	37

37	Exception safe durch Trennung der Instruktionen	37
38	noexcept	38
39	tuple, tie, make_tuple, ignore	39
40	Regex	39
41	lambda foo mit back_inserter, partition_copy, all_of, none_of, copy_if . . .	40
42	Expression Functor, run-time	42
43	Erweiterung zu listing 42	43
44	Erweiterung2 zu listings 42 and 43, run-time	44
45	Erweiterung3 zu listings 42 and 43, compile-time + templates	45
46	Erweiterung zu listing 45, compile-time + templates	46
47	pimpl	46
48	async n stuff	47
49	unique_lock	48

Liste der noch zu erledigenden Punkte

- Auf den Folien steht es sei nur eine main-Funktion pro C++-Programm möglich.
Das ist unpräzise. Aus Binary-Sicht ist das korrekt, aus Code-Sicht nicht,
denn es können sich durchaus weitere main-Funktionen in anderen Name-
spaces verstecken. 4
- Promotions werden nicht gemacht? Kann sein, sobald das Argument selbst
templatisiert ist. sonst müsste es gehen 24
- Schließt dies auch den Reallocation-Fall mit ein? 27