Advanced Programming Techniques
Zusammenfassung des Wichtigsten Quellcode

Wintersemester 2015/16

Dr. Harald Köstler

Autor:

- Christian Strate

Inhaltsverzeichnis

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 ⇒ abstract and C++
   - Warum sind die verwendeten ↑ sinnvoll
   - Oberhalb der Sprache, also unabhängig von C++

Note:

Bei dieser Zusammenfassung handelt es sich um eine inoffizielle Mitschrift von Studenten. Entsprechend sind weder Garantie auf Vollständigkeit noch auf Korrektheit gewährleistet.

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 '&', da 10 ein rvalue und kein lvalue ist und ri3
      daher vom Typ rvalue reference und nicht vom Typ lvalue
      reference sein muss)
5 // rvalue references werden häufig für move Semantik verwendet
6 int &&ri3 = 10;
7 int i = 42; int &&rr = i*42;
8
9 // const funktioniert, da intern zu int konvertiert
10 double dval = 3.14; int const &ri = dval;
```

Listing 1: Referenzen

- Synonym zu `typedef type synonym` ist `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 // Typ von D ist eine Referenz, da A=B ein lvalue zurückgibt.
8 decltype(A=B) D = A;
```

Listing 2: `decltype`

3 Lecture 3

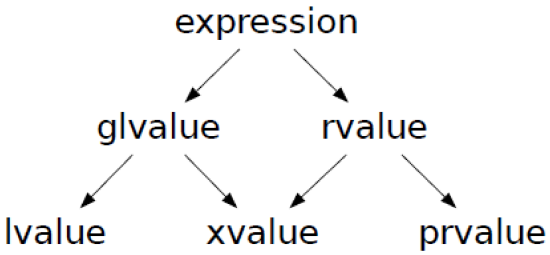

Abbildung 1: Expression-Klassen

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

```
// g ist lvalue
float &f(){float g; return g;}

// 2 ist prvalue, da ein rvalue (non reference) zurückgegeben
    wird
float f(){return 2;}

// 2 ist xvalue, da ein rvalue reference zurückgegeben wird
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

```
// long double
3.14L + 'a'

// int, beide werden zu int promoted, bevor die Operation
    ausgeführt wird
short+char

// int
unsigned short + int

// unsigned long
unsigned long + long
```

Listing 4: Implizite Typumwandlung

Einige Print-Preprocessor-Makros:

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

Listing 9: Kopieren und befüllen

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

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. Dieser 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: candidate function bei der die Parameterliste(Anzahl und Typ) stimmt
- Beste viable function wählen
  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 zu `int`. Beispiel: `char c = 1; typeid(c+c);` ist in der Tat ein `int`.
- Bei Floating-point conversion, also keine promotion, 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 - `const/volatile`), 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 zurückgegebenen 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 - s. listing 14
- 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 um z.B. eine Klasse in dieser Klasse selbst zur Beschreibung von Zusammenhängen zu verwenden
- 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

```
template<typename A,typename B, typename C>
std::function<C(A)> compose(std::function<C(B)>
    f,std::function<B(A)> g){
    // lambda
    return [f,g](A x){return f(g(x));};
}

// in C++14 geht auch das hier
template <class F, class G>
decltype(auto) compose2(F&& f, G&& g){
    return [=] (auto x){return f(g(x));};
}
```

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. D.h. conversion, die keine promotion ist funktioniert nicht

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

Listing 15: Beispiel für Listen-Konstruktor

- 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 Destructor benötigt, dann meistens auch Copy-Constructor, Assignment-Operator, Move-Constructor, 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   // points to the next free position in the new array
14   auto dest = newdata;
15   // points to the next element in the old array
16   auto elem = elements;
17   for (size_t i  0; i < size();  i++)
18      alloc.construct(dest++, std::move(*elem++));
19
20   free();
21 }
```

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-conversions direkt aufeinander folgen

9 Lecture 7

- Protected Vererbung: Kein Zugriff von friends der abgeleiteten Klasse
- Default Vererbung bei Structs ist public, bei Klassen hingegen ist es private (relevant dabei ist der Typ des erbenden Kindes)
- pure-virtual-function: `virtual <type> <func>()= 0;` implizieren, dass die Klasse abstrakt ist
- 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 Basis-Klasse ein Konstruktor in der Derived Klasse mit der selben Parameterliste angelegt wird, der die Parameter an die entsprechenden Konstruktoren der Basis-Klasse weiterreicht.
- In Assignment-Copy-Operatoren der Unterklasse muss auch entsprechender Code für die Oberklasse enthalten sein
- `dynamic_cast` castet vom Basis-Typen zu einem Derived-Typen. Erfordert anscheinend, dass die Basis-Klasse abstract ist
- `typeid`, `type_info` nützliche Dinge

```
using std::cout; using std::endl;
template<class T> struct B : A<T> {
   void print (T a)                    { cout << "6" << endl; }
   static void print (float a)         { cout << "7" << endl; }
};
template<> void B<int>::print (int a)  { cout << "8" << endl; }

void main() {
   B<int> b; B<int>* ptr = &b;
   ptr->print( 1 );
}
```

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.

```
struct A {
using std::cout; using std::endl;
   // error: static function can't be overloaded
   void print(int a, char b = '0')
      { cout << "int" << endl; }

   static void print(int a, char b = '0')
      { cout << "static int" << endl; }
};
```

Listing 18: statische Funktionen mit den selben Parametern wie non-static können nicht überladen 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 Vespin Gas $\mapsto$ 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

```
template<typename T1, typename T2> class Y{
public:
   template<typename TT> void print(TT t);
};
// zwei template-Listen, deren Reihenfolge egal ist
template<typename T1, typename T2>template<typename TT>
void Y <T1, T2>::print(TT);
```

Listing 21: Template Function in template class

```
template<typename T1> class X{
public:
   // hier wüsste der Compiler nicht ob mem eine Member-Variable oder
      ein typedef ist. Dh. er weiß nicht ob das eine Multiplikation
      oder eine Referenz auf T::mem sein soll
   // T::mem *p1;

   // beiden diese Dinge funktionieren jeweils für sich genommen,
      aber nicht gemeinsam
   static int const i = T::mem *p;  // T::mem muss eine Variable sein
   typename T::mem *p1;             // T::mem muss ein type sein
};
```

Listing 22: `typename` als Hilfe zur Konfliktauflösung

Trailing return types listing 23:

Zu Verwenden bei unbekanntem Rückgabe-Typen.

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

Listing 23: Trailing Return Type Beispiel
Der Rückgabetyp steht hinter der Parameterliste

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

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

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

Listing 24: Forwarding Beispiel

variadic template listing 25:

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

```cpp
template<typename T,typename ... Args>
void f(const T &t, const Args &rest);

template<typename T>
ostream &print(ostream &os, const T &t){
    return os<<t;
}
template<typename T, typename... Args>
ostream &print(ostream &os, const T &t, const Args&... rest){
    os<<t<<",";
    return print(os,rest...);
}
```

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 verwiesen. 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 includiert werden, werden unter Umständen mehrfach compiliert, 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.

```
template<unsigned long beg, unsigned long end,
    template<unsigned long>class T>
class Loop{
public:
    template<typename Container>
    static void run(Container &v){
        // value sind die Fibonacci-Zahlen, diese werden woanders
            berechnet (s.u.)
        v.push_back(T<beg>::value);
        Loop<beg+1, end, T>::run(v);
    }
};

// template specialisation, Abbruchbedingung
template<unsigned long end, template<unsigned long>class T>
class Loop<end, end, T>{
public:
    template<typename Container>
    static void run(Container &v){}
};


template<unsigned long N>
class Fibonacci{
public:
    static const int value =
        Fibonacci<N-2>::value+Fibonacci<N-1>::value;
// alternativ
    enum{value2 = Fibonacci<N-2>::value2 +
        Fibonacci<N-1>::value2};
};

// specialisation 0
template<>
class Fibonacci<0u>{
public:
    static const int value = 0;
    enum{value2 = 0};
};
// specialisation 1
template<>
class Fibonacci<1u>{
public:
    static const int value = 1;
    enum{value2 = 1};
};
```

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

```
ifstream is("data/outFile1");
istream_iterator<string> in (is), eof;
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.

```
vector<int> vec;
// insert elements at the tail
auto it = back_inserter(vec);
// equivalent to vec.push_back(42)
*it = 42;
// 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(beg, end, params);
  - alg(beg, end, dest, params);
  - alg(beg, end, beg2 params); (zwei Iteratoren)
  - alg(beg, end, beg2, end2, params); (zwei Iteratoren)

```
vector<int> vi;
int i;
while(cin >> i)
   vi.push_back(i);

// läuft durch den ersten Iterator und ersetzt im selben Index
   des zweiten Iterator den Wert mit dem absoluten Wert
transform(vi.begin(), vi.end(), vi.begin(),
          [](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 einen 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

```
vector<int> vec(10);              // initialises to 0 per default
fill(vec.begin(), vec.end(), 1); // set all values to 1

// c{begin,end} returnt einen const_iterator, dieser kann die
    Werte des Containers nicht verändern, auch nicht wenn
    dieser non-const ist.
// aufsummieren aller Werte, wobei die Summe mit 0 beginnt
int sum = accumulate(vec.cbegin(), vec.cend(), 0);
// falls vec ein vector<double> ist, dann müsste der Startwert
    angepasst werden:
int sum2 = accumulate(vec.cbegin(), vec.cend(), 0.);

// erzeuge 10 Elemente am Ende des Vektors mit dem Wert 42
fill_n(back_inserter(vec), 10, 42);

// accumulate funktioniert auch auf Strings
vector<string> v;
...
string concat = accumulate(v.cbegin(), v.cend(), string(""));


// Initialisierung mittels istream_iterator
istream_iterator<int> in_iter(cin), eof;
vector<int> vec(in_iter, eof);

// Ausgabe mittels ostream_itertor
ostream_iterator<int> out_iter(cout, " " );
copy(vec.begin(), vec.end(), out_iter);


// alternativ
for(auto e: vec)
   *out_iter++ = e;
```

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

```cpp
// count the number of times each word occurs in the input
// map from string to size_t
map<string, size_t> word_count;
string word;
while(cin >> word)
    ++word_count[word];

for(const auto &w : word_count)
    cout << w.first << " occurs "  << w.second << " times" <<
        endl;
```

Listing 35: Container-Dinge `first, second`

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 $\ldots : try\{\ldots\}\ catch(\ldots)\{\}$
- 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

```
void f(shared_ptr<T1>, shared_ptr<T2>);
shared_ptr<T1> t1(new T1);
shared_ptr<T2> t2(new T2);
f(t1,t2);
```

Listing 36: Exception safe Code mit shared pointers.
Es ist wichtig, dass die Objekte NICHT in dem Aufruf angelegt werden.

```
// nicht exception safe
template<typename T> T Stack<T>::pop(){
   if(!vused)
      throw std::runtime_error("pop from empty stack");
   else{
      // man verliert bei einem Fehler Elemente
      T result = v[vused-1];
      // ich weiß nicht genau wo hier eine Exception auftreten
         können sollte, aber falls eine Exception nach dem
         decrease auftritt, so wird das Objekt nicht
         zurückgegeben, aber vused dennoch dekrementiert. =>
         Objekt effektiv verloren
      --vused;
      return result;
   }
}


// Also in einzelne Operationen zerteilen
template<typename T> T& Stack<T>::top(){
   if(!vused)
      throw std::runtime_error("empty stack");
   return v[vused-1];
}

template<typename T> void Stack<T>::pop(){
   if(!vused)
      throw std::runime_error("pop from empty stack");
   else
      --vused;
}
```

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. Zur Übersetzungszeit resultiert dies höchstens in einer Compile-Warnung. Zur Laufzeit terminiert dies das Programm

16 Lecture 12

```
int a, b;
int first, second, third, fourth;
auto tup3 = tie(first, second, third, fourth) =
    std::make_tuple(1001, 1002, 1003, 1004);
tie(std::ignore, a, std::ignore, b) = tup3;
// mutation des Elements first
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` den Wert 2. Das erste `tie` ist nicht Notwendig verdeutlicht aber dessen Funktionalität.

```
emailDescription = ".......";
// regex for email address
string regExprStr(R"((\w+(\.|_)?\w*)@(\w+\.\w+)+))");
// regex holder
regex rgx(regExprStr);
// search result holder
smatch smatch;
// search, dies matcht auf das LETZTE Vorkommen des Regex, dh.
    vorherige Treffer werden überschrieben
if(regex_search(emailDescription, smatch, rgx)){
    cout << "whole adress: " << smatch[0] << "\n";
    cout << "local part (before @): " << smatch[1] << "\n";
    cout << "domain name (after @): " << smatch[3] << "\n";
    cout << "string before the match: " << smatch.prefix() <<
        "\n";
    cout << "string after the match: " << smatch.postfix() <<
        "\n";
}

// um alle Matches zu bekommen:
sregex_token_iterator itBegin(emailDescription.begin(),
    emailDescription.end(), regExprStr);
const sregex_token_iterator itEnd;
while(itBegin != itEnd)
    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

```
list<int> allOdd;
deque<int> allEven;
vector<int> myVec = ...;
// odd ints to list allOdd and even ints to deque allEven
partition_copy(myVec.begin(), myVec.end(),
               back_inserter(allOdd), back_inserter(allEven),
               [](int i){return i%2;});

// weitere funktionen
// true falls alle Elemente ungerade
bool b = all_of(myVec.begin(), myVec.end(),
                [](int i){return i%2;});
// true falls alle Elemente gerade
bool c = none_of(myVec.begin(),myVec.end(),
                 [](int i){return i%2;});
// kopiert alle ungeraden Elemente nach cout
copy_if(myVec.begin(), myVec.end(),
        ostream_iterator<int>(cout, " "),
        [](int i) {return i%2;});
```

Listing 41: lambda und `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

```
// expression abstract base class -> dynamic polymorphism
struct Expression{
   virtual double operator() (double) = 0;
   virtual Expression* clone() = 0;
   virtual ~Expression(){}
};
// functor for Constant_1 expression
struct Constant_1 : Expression{
   Constant_1 (const double &d): d_(d){}
   double operator()(double){ return d_; }
   const double d_;

   Expression *clone(){ return new Constant_1(*this); }
};
// functor for Variable_1 expressions
struct Variable_1 : Expression{
   double operator()(double x){ return x; }

   Expression *clone(){ return new Variable_1(*this); }
};
// functor for complex expressions
template<class Op>
struct ComplexExpression : Expression{
   // subtrees
   Expression *l_;
   Expression *r_;

   ComplexExpression(Expression &l, Expression &r) :
      l_(l.clone()), r_(r.clone()){}

   ~ComplexExpression(){ delete l_; delete r_; }
   double operator()(double d){
      // apply Operator vom template
      return Op::apply((*l_)(d), (*r_)(d));
   }
};
```

Listing 42: Expression Functor, run-time

```
struct ExpressionRef{
   ExpressionRef(Expression *ptr) : sp_(ptr), ref_cnt_(1){}
   ~ExpressionRef(){
      if(--ref_cnt_ == 0){
         delete sp_;
         sp_ = nullptr;
      }
   }

   ExpressionRef(const ExpressionRef &source) : sp_(source.sp_){
      ++ref_cnt_;
   }

   inline ExpressionRef &operator=(const ExpressionRef &rhs){
      if(this != &rhs){
         if(--ref_cnt_ <= 0){
            delete sp_;
            sp_ = nullptr;
         }
         sp_ = rhs.sp_;
         ++ref_cnt_;
      }
      return *this;
   }

   double operator()(double d){
      return (*sp_)(d);
   }

   ExpressionRef clone(){
      ExpressionRef copy(sp_->clone());
      return copy;
   }

   inline Expression *get(){
      return sp_;
   }
   inline Expression &getref(){
      return *sp_;
   }
   inline operator void*(){
      return sp_;
   }

private:
   Expression *sp_;
   int ref_cnt_;
};
```

Listing 43: Erweiterung zu listing 42

```
// integration function (average of upper bound and lower bound sums)
template<class Func>
double integrate(Func f, double low, double high, double epsilon=0.001){
   assert(low <= high);
   double auc1 = 0, auc2 = 0;
   double running_x = low;
   while(running_x < high){
      auc1 += f(running_x) * epsilon;
      running_x += epsilon;
      auc2 += f(running_x)* epsilon;
   }
   return (auc1+auc2)/2
}

// add and multiply (schlechte Realisierung, da duplicated code), Entfernung unten mittels type traits
struct Add{
   static double apply(double l, double r){ return l+r; }
};
struct Multiply{
   static double apply(double l, double r){ return l*r; }
};
// operatoren
ExpressionRef operator*(Expression &l, Expression &r){
   return ExpressionRef(new ComplexExpression<Multiply>(l, r));
}
ExpressionRef operator*(Expression &l, Expression &r){
   return l.getref()*r.getref();
}
ExpressionRef operator*(Expression &l, const double d){
   return l * Constant_1(d);
}
ExpressionRef operator*(const double d, Expression &r){
   return Constant_1(d * r);
}
ExpressionRef operator*(ExpressionRef &l, Expression &r){
   return l.getref()*r;
}

// Verwendung mittels dynamic polymorphism
// ruft call operator auf, Auswertung zur Laufzeit, da virtual function
Variable_1 x;
((2*x*x + 3*x + 3)*(2*x*x + 3*x + 3))(2);
integrate((2*x*2x + 3*x + 3), 0, 1);
integrate((2*x*x + 3*x + 3)*(2*x*x + 3*x + 3), 0,  1);
integrate(x*x, 0, 7);
```

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

```
struct Constant{
   double d_;
   Constant(double d): d_(d){}
   double operator() (double){ return d_; }
};
struct Variable{
   double operator() (double d){ return d; }
};
// für non-terminal, complex expression
template<class E1, class E2, class Op>
struct ComplexExpr{
   ComplexExpr(E1 l, E2 r): l_(l), r_(r){}
   double operator() (double d){
      return Op::apply(l_(d), r_(d)); }
   E1 l_;E2 r_;
};
// kapselt alle expression objekte, bool gibt an ob non-/literal
template<class E, bool b>
struct Expr{
   E e_;
   Expr(1& e) : e_(e){}
   double operator()(double d){ return e_(d); }
};
// specialisation
template<class E>
struct Expr<E, false>{
   double e_;
   Expr(double const &e) : e_(e){}
   double operator()(double d){ return e_(d); }
};
template<class E>
struct Expr<E, true>{
   double e_;
   Expr(double const &e) : e_(e){}
   double operator()(double d){ return e_; }
};
// arithmetic operators - nur noch einer, da type tracing
template<class E1, class E2>
Expr<ComplexExpr<Expr<E1,is_arithmetic<E1>::value>,
   Expr<E1, is_arithmetic<E2>::value>, Add>,
   is_arithmetic<ComplexExpr<Expr<E1, is_arithmetic<E1>::value>,
                Expr<E2, is_artihmetic<E2>::value>, Add>>::value>
operator+(E1 e1, E2 e2){
   typedef ComplexExpr<Expr<E1, is_arithmetic<E1>::value>,
             Expr<E2, is_arithmetic<E2>::value>, Add>ExprType;
   return Expr<ExprType, is_arithmetic<ExprType>::value>(
             ExprType(Expr<E1, is_arithmetic<E1>::value(e1),
                      Expr<E2,is_arithmetic<E2>::value(e2)));
}
```

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

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

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

```
class Map{
private:
    struct Impl; // forward decl
    shared_ptr<Impl> pimpl_;
};
```

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 ausgetauscht(swap) werden, warum auch immer man das tun wollte
- aktueller Thread: `std::this_thread`

```
int calculate(){ return 42; }
// startet Ausführung sofort
auto future = async(launch::async, calculate, <parameter>);
// startet später, meistens, wenn man das Ergebnis erfragt
auto future = async(launch::deferred, calculate, <parameter>);
// unspecified policy
auto future = async(calculate, <parameter>);

// cout thread-safe gestalten
cout.sync_with_stdio(true);

atomic<int> counter(0);
++counter;
```

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

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

```
std::unique_lock<std::mutex>guard1(<someMutex1>,
    std::defer_lock);
std::unique_lock<std::mutex>guard2(<someMutex2>,
    std::defer_lock);
// lockt beide gleichzeitig
std::lock(guard1, guard2);


std::mutex mutex_;
std::condition_variable condVar;
bool dataReady;
std::unique_lock<std::mutex> lck(mutex_);
// wartet bis dataReady true ist
condVar.wait(lck, []{return dataReady;});

// notify the waiting ones (in einer anderen Funktion)
std::lock_guard<std::mutex> lck(mutex_);
dataReady = true;
condVar.notify_all();
```

Listing 49: `unique_lock`

19 Abbildungsverzeichnis

Abbildungsverzeichnis

20 Listingverzeichnis

List of Listings

Liste der noch zu erledigenden Punkte