Neues in C++11

Simon Brummer

HAW Hamburg

1. November 2014

Inhaltsverzeichnis

1. Sprachentwicklung in C++11
2. Neue Sprachfeatures
3. Neue Library Features
4. Kompilerunterstützung
5. Links

Sprachentwicklung in C++11

- Arbeistitel: C++0x
- ISO Standardisiert.
- Letzter Standard war C++03 (Also 8 Jahre her)
- C++ erweitert nur und erhält Sprachkompatibilität zurück bis zu C.
- Boost $\longrightarrow$ TR $\longrightarrow$ stdlib++ $\longrightarrow$ Boost ...

Neue Sprachfeatures

C++11 als Multiparadigmensprache, fügt mehrere neue Sprachfeatures dem Sprachkern hinzu, um aktuellen Programmiermoden besser zu entsprechen. Hier wird besonders der Trend zu funktionaler Programmierung unterstützt.
Alte Sprachkonstrukte bleiben, wie gewohnt, unverändert.

The Rule of Five

Erweiterung des Rule of Three Konzepts, um den Move Konstruktor und Move Assignment Operator.
Move Konstruktor und Move Assignment Operator sollten als optionale Optimierungsmöglichkeit aufgefasst werden, da sie Rechenzeit im Falle von Referenzen als Rückgabewert sparen(Besonders bei Deepcopy im Copy Konstruktor).
Beispiel: Sinnvoll bei Factory Methoden

Typinferenz (auto)

```
1  #include <vector>
2  #include <iostream>
3  using namespace std;
4
5  class ClassWithAnoyingLongName{};
6
7  int main(void){
8      vector<ClassWithAnoyingLongName> vec;
9
10     // Old
11     for(vector<ClassWithAnoyingLongName>::iterator it  = vec.begin();
12                                                    it != vec.end();
13                                                    it++)
14     {
15         // Do Something
16     }
17
18     // New
19     for(auto v: vec){
20         // Do something
21     }
22
23     return 0;
24 }
```

Anonyme Funktionen

```
1  #include <cstdio>
2
3  int main(void){
4
5      // Syntax:
6      // []=Bindung | Bindet aufrufenden Kontext an die Funktion
7      // ()= Param  | Parameterliste(Optional)
8      // ->= Return | Rueckgabewert(Optional) wg. Typinferenz
9      // {}=Function| Funktionskoerper
10     int t = 0;
11     auto f = [&t](int x) -> int { return 2*x + t; };
12
13     t = 1;
14     printf("%d\n",f(1));
15
16     t = 2;
17     printf("%d\n",f(2));
18     return 0;
19 }
```

Erweiterte Initialisierungslisten

```
1  #include <iostream>
2  #include <initializer_list>
3  #include <vector>
4  using namespace std;
5
6  class Obj{
7  public:
8      vector<int> vec;
9      Obj(initializer_list<int> l): vec(l) {};
10 };
11
12 int sum(initializer_list<int> l){
13     int sum = 0;
14     for(auto i : l){
15         sum += i;
16     }
17     return sum;
18 }
19
20 int main(void){
21     Obj o = {1,2,3};                    // Initializer als Konstruktorparamter
22     printf("%d\n", sum( {1,2,3} ));     // Initializer als Parameter
23     for(auto i : {1,2,3} ){             // Initializer als Parameter(For)
24         printf("%d\n", i);
25     }
26     return 0;
27 }
```

Alternative Funktionssyntax

```
1  #include <iostream>
2  using namespace std;
3
4  class Hand{
5  private:
6      Hand(){
7      }
8
9  public:
10     auto static build() -> Hand {
11         return Hand();
12     }
13
14     auto giveFive() -> int {
15         return 5;
16     }
17 };
18
19 auto main(int argc, char** argv) -> int {
20     Hand h = Hand::build();
21     cout << h.giveFive() << endl;
22     return 0;
23 }
```

Explizites Vererbungsverhalten(Selbstschutz)

```
1   class A{
2   public:
3       virtual auto a() -> int {
4           return 0;
5       }
6
7       virtual auto answerToAllQuestions() -> int final{
8           return 42;
9       }
10  };
11
12  class B: public A{
13  public:
14      virtual auto a() -> int override {
15          return 1;
16      }
17  };
18
19  auto main(void) -> int {
20      return 0;
21  }
```

Variadic Templates

Erweiterung des Template Mechanismus. Templates können jetzt mit einer Variablen Anzahl von Typen deklariert werden.
Beispiel: tuple

```
1  tuple<int, float>          t1(0, -1.0);
2  tuple<int, float, string> t2(0, -1.0, "Foo");
```

Neue String Literale

```
1  #include <string>
2  #include <iostream>
3  using namespace std;
4
5  auto main(void) -> int {
6      // Vorsicht mit Operationen aus dem char Array
7      char u8[]  = u8"I'm a UTF-8 string. Character: \U0001F4A9";
8      char16_t u16[] = u"This is a UTF-16 string.";
9      char32_t u32[] = U"This is a UTF-32 string.";
10
11     cout << u8 << endl;
12     // char16_t und char32_t haben noch keine Ausgabe im Standard.
13     // Evtl. hat Boost was.
14
15     return 0;
16 }
```

Static Assertions

Neue statische Assertions. Können nur für konstante Ausdrücke verwendet werden:

```
1  auto main(void) -> int {
2      const int x = 0;
3      const int y = 1;
4
5      // This throws a compiler Error
6      static_assert(x == y, "Test failed: x==y");
7
8      return 0;
9  }
```

Neue Library Features

Die Standardlibrary wurde viele Klassen erweitert, die moderneres Programmieren ermöglichen. Viel kam aus Boost, zu den Technical Reports(TR1) und von dort in die Standard Library.

Threads

Kompilieren mit “-pthread“ Parameter!

```
1  #include <iostream>
2  #include <thread>
3  #include <unistd.h>
4  using namespace std;
5
6  bool alive = true;
7
8  auto th_func(void) -> void {
9      while(alive){
10         cout << "I am Alive" << endl;
11     }
12 }
13
14 auto main(void) -> int {
15     thread t1(th_func);
16     sleep(2);
17     alive = false;
18     t1.join();
19     cout << "Finished" << endl;
20     return 0;
21 }
```

Tupel

```
1  #include <tuple>
2  #include <cstdio>
3  using namespace std;
4
5  tuple<int,int> func(){
6        tuple<int,int> t(0,-1);
7        return t;
8  }
9
10 int main(void){
11       int val;
12       int err;
13       auto ret = func();
14       tie(val,err) = func();
15
16       printf("Val:%2d\n", val);
17       printf("Err:%2d\n", err);
18       printf("Val:%2d\n", get<0>(ret) );
19       printf("Err:%2d\n", get<1>(ret) );
20       printf("Val:%2d\n", get<0>(func()) );
21       return 0;
22 }
```

Hashtables

Eigene Klassen müssen eine eigene Hash Funktion mitliefern.

```
1  #include <unordered_map>
2  #include <cstdio>
3  #include <string>
4  using namespace std;
5
6  int main(void) {
7      unordered_map <string, int> answer;
8      answer["to all questions"] = 42;
9      printf("The Answer to all Questions: %d\n", answer["to all questions"]);
10     return 0;
11 }
```

Regular expressions

```
1  #include <string>
2  #include <regex>
3  #include <iostream>
4  #include <iterator>
5  using namespace std;
6
7  auto main(void) -> int {
8      string s = "This is a short example Text";
9      regex w("(\\S+)");
10
11     // Check if Regex matches give string
12     if( regex_search(s,w) ){
13         cout << "Regex matches" << endl;
14     }
15
16     // Print Everything that Matches
17     auto wStr = sregex_iterator(s.begin(), s.end(), w);
18     auto wEnd = sregex_iterator();
19     for(auto i = wStr; i != wEnd; ++i){
20         smatch match = *i;
21         cout << match.str() << endl;
22     }
23
24     return 0;
25 }
```

Smart Pointer

C++11 unterstützt unique_ptr, shared_ptr und weak_ptr.
unique_ptr sollte statt auto_ptr verwendet werden.
Der unique_ptr nutzt move semantik, kein copy und assignment.

```
1  #include <memory>
2  #include <iostream>
3  class Obj{
4  public:
5  Obj(){  std::cout << "Obj()" << std::endl; }
6  ~Obj(){ std::cout << "~Obj()" << std::endl; }
7  };
8
9  auto main(void) -> int{
10 std::unique_ptr<Obj> unique(new Obj());
11 std::shared_ptr<Obj> shared(new Obj());
12 std::weak_ptr<Obj> weak = shared;
13 return 0;
14 }
```

Random Number Facility

Trennung in Engine(Erzeugungsalgorithmus) und Distribution(Verteilung des Zufalls) ermöglichen verschiedenste Zufallsarten.

```
1  #include <random>
2  #include <functional>
3  #include <iostream>
4  using namespace std;
5
6  auto main(void) -> int{
7      int seed = 42;
8      default_random_engine engine(seed);
9      uniform_int_distribution<int> dist(1,6);
10
11     // Einfach
12     int dice_roll = dist(engine);
13     cout << dice_roll << endl;
14
15     // Verbund
16     auto dice = bind (dist, engine);
17     cout << dice() << dice() << dice() << endl;
18
19     return 0;
20 }
```

Wrapper Reference

```
1  #include <iostream>
2  #include <functional>
3  using namespace std;
4
5  int main () {
6    int a = 10;
7    int b = 20;
8    int c = 30;
9
10   // an array of "references":
11   reference_wrapper<int> refs[] = {a,b,c};
12
13   cout << "refs:";
14   for (int& x : refs) {
15   cout << ' ' << x;
16   }
17   cout << '\n';
18
19   return 0;
20 }
```

Kompilerunterstützung

Sprachstandard einschalten: -std=c++11
Full Feature Support:

- gcc(ab v4.8), Momentics nutzt v4.7.3 :(
- clang(ab v3,3)

Partial Feature Support:

- VS Nov 2013 CTP (Visual Studio Compiler)
- Intel 14.0

Support:

- avr-g++(Full Feature)
- arm-none-eabi-g++(Full Feature Support, default in RIOT)

Weiterführende Links

- Wikipedia: http://en.wikipedia.org/wiki/C++11
- http://www.cplusplus.com/reference/
- cpprocks: cpprocks.com/c1114-compiler-and-library-shootout
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3551.pdf