- Das Lambda-Kalkül geht auf Alonzo Church und Stephen Kleene zurück, die in den 30er-Jahren damit ein formales System für berechenbare Funktionen entwickelten.
- Zu den wichtigsten Arbeiten aus dieser Zeit gehört der Aufsatz von Alonzo Church: *An Unsolvable Problem of Elementary Number Theory*, *Americal Journal of Mathematics*, Band 58, Nr. 2 (April 1936), S. 345–363.
- Diese Arbeit zeigt, dass es keine berechenbare Funktion gibt, die die Äquivalenz zweier Ausdrücke des Lambda-Kalküls feststellen kann.
- Die Turing-Maschine und das Lambda-Kalkül sind in Bezug auf die Berechenbarkeit äquivalent.
- Das Lambda-Kalkül wurde von funktionalen Programmiersprachen übernommen (etwa von Lisp und Scheme) und wird auch gerne zur formalen Beschreibung der Semantik einer Programmiersprache verwendet (denotationelle Semantik).

Da in C++ Funktionsobjekte wegen der entsprechenden STL-Algorithmen recht beliebt sind, gab es mehrere Ansätze, Lambda-Ausdrücke in C++ einzuführen:

- *boost*::*lambda* von Jaakko Järvi, entwickelt von 1999 bis 2004
- *boost*::*phoenix* von Joel de Guzman und Dan Marsden, entwickelt von 2002 bis 2005
- Integration von Lambda-Ausdrücken in den C++-Standard ISO-14882-2012.

transform2.cpp

```
int main() {
   std::list<int> ints;
   for (int i = 1; i <= 10; ++i) {
      ints.push_back(i);
   }

   std::list<int> squares;
   std::transform(ints.begin(), ints.end(),
      std::back_inserter(squares), [](int val) { return val*val; });

   for (int val: squares) {
      std::cout << val << std::endl;
   }
}
```

- Mit Lambda-Ausdrücken werden implizit unbenannte Klassen erzeugt und temporäre Objekte instantiiert.
- In diesem Beispiel ist [](**int** *val*){ **return** *val*∗*val*; } der Lambda-Ausdruck, der ein temporäres unäres Funktionsobjekt erzeugt, das sein Argument quadriert.

⟨lambda-expression⟩ ⟶ ⟨lambda-introducer⟩ [⟨lambda-declarator⟩]
⟨compound-statement⟩

⟨lambda-introducer⟩ ⟶ „[“ [⟨lambda-capture⟩] „]“

⟨lambda-capture⟩ ⟶ ⟨capture-default⟩

⟶ ⟨capture-list⟩

⟶ ⟨capture-default⟩ „,“ ⟨capture-list⟩

⟨capture-default⟩ ⟶ „&“ | „=“

⟨capture-list⟩ ⟶ ⟨capture⟩ [„...“]

⟶ ⟨capture-list⟩ „,“ ⟨capture⟩ [„...“]

Grammatik für Lambda-Ausdrücke

⟨capture⟩ ⟶ ⟨simple-capture⟩
⟶ ⟨init-capture⟩
⟨simple-capture⟩ ⟶ ⟨identifier⟩
⟶ „&“ ⟨identifier⟩
⟶ **this**
⟨init-capture⟩ ⟶ ⟨identifier⟩ ⟨initializer⟩
⟶ „&“ ⟨identifier⟩ ⟨initializer⟩
⟨lambda-declarator⟩ ⟶ „(“ ⟨parameter-declaration-clause⟩ „)“
[**mutable**] [⟨exception-specification⟩]
[⟨attribute-specifier-seq⟩]
[⟨trailing-return-type⟩]

- Die ⟨init-capture⟩ kommt mit dem C++14-Standard hinzu.

- Lambda-Ausdrücke, die in eine Funktion eingebettet sind, „sehen“ die lokalen Variablen aus den umgebenden Blöcken.
- In vielen funktionalen Programmiersprachen überleben die lokalen Variablen selbst dann, wenn der sie umgebende Block verlassen wird, weil es noch überlebende Funktionsobjekte gibt, die darauf verweisen. Dies benötigt zur Implementierung sogenannte *cactus stacks*.
- Da für C++ der Aufwand für diese Implementierung zu hoch ist und auch die Übersetzung normalen Programmtexts ohne Lamda-Ausdrücke verteuern würde, fiel die Entscheidung, einen alternativen Mechanismus zu entwickeln, der über eine *lambda-capture* spezifiziert wird.

Kopien der Variablen aus der Umgebung


```
template<typename T>
auto create_multiplier(T factor) {
   return [=](T val) { return factor*val; };
}

int main() {
   auto multiplier = create_multiplier(7);
   for (int i = 1; i < 10; ++i) {
      std::cout << multiplier(i) << std::endl;
   }
}
```

- *create_multiplier* ist eine Template-Funktion, die ein mit einem vorgegebenen Faktor multiplizierendes Funktionsobjekt erzeugt und zurückliefert.
- Die Standard-Template-Klasse *function* wird hier genutzt, um das Funktionsobjekt in einen bekannten Typ zu verpacken.
- Die *lambda-capture* [=] legt fest, dass die aus der Umgebung referenzierten Variablen beim Erzeugen des Funktionsobjekts kopiert werden.

Erzeugung einer unbenannten Klasse 73

```
template<typename T>
class Anonymous {
   public:
      Anonymous(const T& factor) : factor(factor) {}
      T operator()(T val) const {
         return factor*val;
      }
   private:
      T factor;
};

template<typename T>
auto create_multiplier(T factor) {
   return Anonymous<T>(factor);
}
```

- Der Lambda-Ausdruck führt implizit zu einer Erzeugung einer unbenannten Klasse (hier einfach *Anonymous* genannt).
- Jeder aus der Umgebung referenzierte Variable, die kopiert wird, findet sich als gleichnamige Variable der Klasse wieder, die bei der Konstruktion übergeben wird.

```
std::vector<int> values(10);
int count = 0;
std::generate(values.begin(), values.end(), [&]() { return ++count; });
```

- Alternativ können Variablen nicht kopiert, sondern per impliziter Referenz benutzt werden.
- Dann darf das Funktionsobjekt aber nicht länger leben bzw. benutzt werden, als die entsprechenden Variablen noch leben. Das liegt in der Verantwortung des Programmierers.
- *generate* steht über **#include** <algorithm> zur Verfügung und weist die von dem Funktionsobjekt erzeugten Werte sukzessiv allen referenzierten Werten zwischen dem ersten Iterator (inklusive) und dem zweiten Iterator (exklusive) zu.