- Funktionsobjekte sind Objekte, bei denen der **operator**() definiert ist. (Siehe ISO 14882-2003, Abschnitt 20.3.)
- Viele Algorithmen der STL akzeptieren solche Funktionsobjekte, um aus einer Menge von Objekten (repräsentiert durch Iteratoren) Objekte herauszufiltern oder eine Menge von Objekten zu transformieren.
- Es ist in vielen Fällen nicht notwendig, extra Klassen für Funktionsobjekte zu definieren, da es bereits eine Reihe vorgefertigter Funktionsobjekte gibt und mit Hilfe der Boost-Lambda-Bibliothek auch Funktionsobjekte entsprechend des λ-Kalküls frei konstruiert werden können.

transform.cpp

```
template<typename T>
class SquareIt: public unary_function<T, T> {
   public:
      T operator()(T x) {
         return x * x;
      }
};
```

- Die Klasse *SquareIt* leitet sich von *unary_function* ab und bietet einen unären Funktions-Operator an, der das Quadrat seiner Argumente zurückliefert.
- Die Template-Klasse *unary_function* ist im ISO-Standard definiert:

```
template <class _Arg, class _Result>
struct unary_function {
  typedef _Arg argument_type;
  typedef _Result result_type;
};
```

transform.cpp

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

   transform(ints.begin(), ints.end(),
      back_inserter(squares), SquareIt<int>());

   for (list<int>::iterator it = squares.begin();
         it != squares.end(); ++it) {
      cout << *it << endl;
   }
}
```

- *transform* gehört zu den in der STL definierten Operatoren, die auf durch Iteratoren spezifizierten Sequenzen arbeiten.
- Die ersten beiden Parameter von *transform* spezifizieren die zu transformierende Sequenz, der dritte Parameter den Iterator, der die Resultate entgegen nimmt und beim vierten Parameter wird das Funktionsobjekt angegeben, das die gewünschte Abbildung durchführt.

transform.cpp

```
transform(ints.begin(), ints.end(),
   back_inserter(squares), SquareIt<int>());
```

- Wenn die Sequenz-Operatoren der STL einen Iterator für die Ausgabe erhalten, dann gehen sie davon aus, dass hinter dem Ausgabe-Operator bereits Objekte existieren.
- *transform* selbst fügt also keine Objekte irgendwo ein, sondern nimmt Zuweisungen vor.
- Funktionen wie *back_inserter* erzeugen einen speziellen Iterator für einen Container, der neue Objekte einfügt (hier immer an das Ende der Sequenz).
- Bei *transform* wäre auch eine direkte Ersetzung möglich gewesen der ursprünglichen Objekte:

transform.cpp

```
transform(ints.begin(), ints.end(), ints.begin(), SquareIt<int>());
```

- 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 im kommenden C++-Standard, bereits von gcc 4.5 und weiteren C++-Übersetzern unterstützt

Church gibt eine rekursive Definition für Lambda-Ausdrücke, die in eine Grammatik übertragen werden kann:

$$
\begin{aligned}
\langle \text{formula} \rangle &\longrightarrow \langle \text{variable} \rangle \\
&\longrightarrow \text{„}\lambda\text{“ } \langle \text{variable} \rangle \text{ „[“ } \langle \text{formula} \rangle \text{ „]“} \\
&\longrightarrow \text{„\{“ } \langle \text{formula} \rangle \text{ „\}“ „(“ } \langle \text{formula} \rangle \text{ „)“}
\end{aligned}
$$

Bei Variablen werden Namen verwendet wie beispielsweise x oder y.

Später hat sich folgende vereinfachte Grammatik für Lambda-Ausdrücke durchgesetzt:

$$\begin{aligned} \langle \text{formula} \rangle &\longrightarrow \langle \text{variable} \rangle \\ &\longrightarrow \text{„}\lambda\text{“ } \langle \text{variable} \rangle \text{ „.“ } \langle \text{formula} \rangle \\ &\longrightarrow \text{„(“ } \langle \text{formula} \rangle \text{ „)“ } \langle \text{formula} \rangle \end{aligned}$$

Beispiel:

- $\lambda f.\lambda x.(f)(f)x$
  (Traditionelle Schreibweise: $\lambda f\,[\lambda x\,[\{f\}\,(\{f\}\,(x))]]$)

Variablen sind in einem Lambda-Ausdruck entweder frei oder gebunden:

- Bei „λ“ ⟨variable⟩ „[“ ⟨formula⟩ „]“ ist die hinter λ genannte Variable innerhalb der ⟨formula⟩ gebunden.
- Es liegt eine Blockstruktur vor mit entsprechendem lexikalisch bestimmten Sichtbereichen.
- Um die Lesbarkeit zu erhöhen und die textuell definierten Konvertierungen zu vereinfachen, wird normalerweise davon ausgegangen, dass Variablennamen eindeutig sind.
- Variablen, die nicht gebunden sind, sind frei.

Der Textersetzungs-Ausdruck $S_N^x M \mid$ ersetzt x global in M durch N. Hierbei ist x eine Variable.

Beispiele:

- $S_y^x \lambda x.x \mid = \lambda y.y$
- $S_{\lambda x.x}^x \lambda y.(x)(x)\, y \mid = \lambda y.(\lambda x.x)(\lambda x.x)\, y$

Textersetzungen sollten dabei keine Variablenbindungen brechen. Gegebenenfalls sind zuerst Variablennamen zu ersetzen. Das zweite Beispiel war zulässig, weil x nicht gebunden war und die im Ersatztext gebundene Variable x nicht in Konflikt zu bestehenden gebundenen Variablen steht.

- α-**Äquivalenz**: In einem Lambda-Ausdruck dürfen überall Konstrukte der Form $\lambda x.M$ durch $\lambda y.S_y^x M$ | ersetzt werden, vorausgesetzt, dass y innerhalb von M nicht vorkommt.
- Beispiel: $\lambda x.x$ ist α-äquivalent zu $\lambda y.y$
- In $\lambda x.\lambda y.(y)x$ darf y nicht durch x ersetzt werden, da x bereits vorkommt.

- Es dürfen überall Konstrukte der Form $(\lambda x.M)\,N$ durch $S_N^x M \mid$ ersetzt werden, vorausgesetzt, dass die in M gebundenen Variablen sich von den freien Variablen in N unterscheiden.
- Wenn die Voraussetzung nicht erfüllt ist, könnte zuvor bei M oder N eine α-äquivalente Variante gesucht werden, die den Konflikt vermeidet.
- Beispiel:

$$\begin{aligned}(\lambda x.\lambda y.(y)x)y &\rightarrow (\lambda x.\lambda a.(a)x)y \\ &\rightarrow \lambda a.(a)y\end{aligned}$$

- Die β-Reduktion kann mit der Auswertung eines Funktionsaufrufs verglichen werden, bei der der formale Parameter x durch den aktuellen Parameter N ersetzt wird.

- β-Reduktionen (mit ggf. notwendigen α-äquivalenten Ersetzungen) können nacheinander durchgeführt werden, bis sich keine β-Reduktion anwenden lässt.
- Nicht jeder „Funktionsaufruf“ lässt sich dabei auflösen. Beispiel: $(a)b$, wobei a eine ungebundene Variable ist.
- Der Prozess kann halten, muss aber nicht.
- Bei folgenden Beispiel führt die β-Reduktion zum identischen Lambda-Ausdruck, wodurch der Prozess nicht hält:

$$(\lambda x.(x)x)\lambda x.(x)x \quad \rightarrow \quad (\lambda x.(x)x)\lambda x.(x)x$$

Wenn mehrere β-Reduktionen zur Anwendung kommen können, welche ist dann zu nehmen?

- Satz von Church und Rosser (1936): Wenn zwei Prozesse mit dem gleichen Lambda-Ausdruck beginnen und sie beide terminieren, dann haben beide das identische Resultat. Die β-Reduktionen sind somit konfluent.
- Es kann jedoch passieren, dass die Reihenfolge, in der Kandidaten für β-Reduktionen ausgesucht werden, entscheidet, ob der Prozess terminiert oder nicht. Beispiel:

  $$(\lambda x.a)(\lambda x.(x)x)\lambda y.(y)y$$

  Dieser Ausdruck kann zu a reduziert werden, wenn immer die β-Reduktion steht, die am weitesten links steht.

- Wenn immer die am weitestens links stehende Möglichkeit zu einer β-Reduktion angewendet wird, dann handelt es sich um eine Auswertung in der Normal-Ordnung (*normal-order evaluation* oder auch *lazy evaluation*).
- Wenn der Prozess terminieren kann, dann terminiert auch die Auswertung in der Normal-Ordnung.

- Da der einfache ungetypte Lambda-Kalkül nur Funktionen als Datentypen kennt, werden skalare Werte durch Funktionen repräsentiert. Hierzu haben sich einige Konventionen gebildet.
- Die Boolean-Werte *true* und *false* werden durch Funktionen repräsentiert, die von zwei gegebenen Parametern einen aussuchen:

```
True = Lx.Ly.x
False = Lx.Ly.y
```

- (Die Syntax entspricht der eines kleinen Lambda-Kalkül-Interpreters, bei dem aus Gründen der Einfachheit λ durch `L` repräsentiert wird und Lambda-Ausdrücke über Namen referenziert werden können (hier *True* und *False*).

- Mit den Definitionen für *True* und *False* ergibt sich die Definition einer bedingten Anweisung:

```
If-then-else = La.Lb.Lc.((a)b)c
```

- Der erste Parameter (hier *a*) ist die Bedingung. Wenn sie wahr ist, wird *b* ausgewählt, ansonsten *c*.

```
(((La.Lb.Lc.((a)b)c)Lx.Ly.x)this)that
---> ((Lb.Lc.((Lx.Ly.x)b)c)this)that
---> (Lc.((Lx.Ly.x)this)c)that
---> ((Lx.Ly.x)this)that
---> (Ly.this)that
---> this
```

- Die natürliche Zahl n kann dadurch repräsentiert werden, dass eine beliebige Funktion f n-fach aufgerufen wird. Die Zahl n repräsentiert dann die n-te Potenz einer Funktion. Entsprechend werden natürliche Zahlen als Funktionen definiert, die zwei Parameter erwarten: die anzuwendende Funktion f und der Parameter, der dieser Funktion beim ersten Aufruf zugeführt wird:

```
0 = Lf.Lx.x
1 = Lf.Lx.(f)x
2 = Lf.Lx.(f)(f)x
3 = Lf.Lx.(f)(f)(f)x

> ((3)Lf.(f)hello)Lx.x
((Lf.Lx.(f)(f)(f)x)Lf.(f)hello)Lx.x
---> (Lx.(Lf.(f)hello)(Lf.(f)hello)(Lf.(f)hello)x)Lx.x
 ---> (Lf.(f)hello)(Lf.(f)hello)(Lf.(f)hello)Lx.x
  ---> ((Lf.(f)hello)(Lf.(f)hello)Lx.x)hello
  ---> (((Lf.(f)hello)Lx.x)hello)hello
  ---> (((Lx.x)hello)hello)hello
  ---> ((hello)hello)hello
```

- Wiederhole x n-mal:

```
Repeat = Ln.Lx.((n)Lg.(g)x)Ly.y
```

- Erhöhe n um 1:

```
Succ = Ln.Lf.Lx.(f)((n)f)x
```

(Es ist zu beachten, dass 3 und (Succ)2 nicht identisch aussehen, aber in der Funktionalität des Wiederholens äquivalent sind.)

- Verkleinere n um 1:

```
Pred = Ln.(((n)Lp.Lz.((z)(Succ)(p)True)
   (p)True)Lz.((z)0)0)False
```

(Das funktioniert nicht für negative Zahlen.)

- Arithmetische Operationen:

```
+ = Lm.Ln.Lf.Lx.((m)f)((n)f)x
* = Lm.Ln.Lf.(m)(n)f
```

- Test, ob eine n 0 ist:

```
Zero? = Ln.((n)(True)False)True
```

- Ein naiver Versuch, eine rekursive Funktion zur Berechnung der Fakultät von n könnte so aussehen:

```
F = Ln.(((If-then-else)(Zero?)n)1)((*)n)(F)(Pred)n
```

- Das ist jedoch nicht zulässig, da dies nicht textuell expandiert werden kann.
- Glücklicherweise lässt sich das Problem mit dem sogenannten Fixpunkt-Operator Y lösen:

```
Y = Ly. (Lx.(y)(x)x) Lx.(y)(x)x
```

- Es gilt $(Y)f \rightarrow (f)(Y)f$.
- Es ist dabei zu beachten, dass der Y-Operator nur in Verbindung mit einer Auswertung in der Normal-Ordnung funktioniert.
- Mit dem Y-Operator lässt sich nun F definieren:

```
F = (Y)Lf.Ln.(((If-then-else)(Zero?)n)1)((*)n)(f)(Pred)n
```

- ```
  > ((Repeat)(F)3)hi
  [..]
  ---> (((((hi)hi)hi)hi)hi)hi
  1114 reductions performed.
  ```
  ```

Es gibt zwei wesentliche Punkte, weswegen Lambda-Ausdrücke auch in nicht-funktionalen Programmiersprachen (wie etwa C++) interessant sind:

- Anonyme Funktionen können lokal konstruiert und übergeben werden. Ein Beispiel dafür wäre das Sortierkriterium bei *sort*. Die lokal definierte anonyme Funktion kann dabei auch die Variablen der sie umgebenden Funktion sehen (*closure*).
- Funktionen können aus anderen Funktionen abgeleitet werden. Beispielsweise kann eine Funktion mit zwei Argumenten in eine Funktion abgebildet werden, bei der der eine Parameter fest vorgegeben und nur noch der andere variabel ist (*currying*).

Grundsätzlich kann das alles auch konventionell formuliert werden durch explizite Klassendefinitionen. Aber dann wird der Code umfangreicher, umständlicher (etwa durch die explizite Übergabe der lokal sichtbaren Variablen) und schwerer lesbarer (zusammenhängender Code wird auseinandergerissen). Allerdings können Lambda-Ausdrücke auch zur Unlesbarkeit beitragen.

sortit1.cpp

```
#include <cstdlib>
#include <iostream>
#include <algorithm>
#include <vector>
#include <boost/lambda/lambda.hpp>
#include <boost/lambda/bind.hpp>

using namespace std;
using namespace boost::lambda;

int main() {
   vector<int> v(10);
   int i = 0;
   for_each(v.begin(), v.end(), _1 = var(i)++ - 5);
   cout << "unsorted:";
   for_each(v.begin(), v.end(), cout << constant(' ') << _1);
   cout << endl;

   sort(v.begin(), v.end(), _1 * _1 < _2 * _2);
   cout << "sorted:";
   for_each(v.begin(), v.end(), cout << constant(' ') << _1);
   cout << endl;
}
```

sortit1.cpp

```
vector<int> v(10);
int i = 0;
for_each(v.begin(), v.end(), _1 = var(i)++ - 5);
```

- *_1* ist ein Platzhalter für das erste Argument der anonymen Funktion, die hier konstruiert wird.
- *var*(*i*) referenziert die Variable *i*.
- Bei $\_1 = i{+}{+} - 5$ würde $i{+}{+} - 5$ nur einmal zu Beginn ausgewertet werden. Die Verwendung von *var* verschiebt die Auswertung auf den Moment, an dem der Lambda-Ausdruck bewertet wird.

sortit1.cpp

```
cout << "unsorted:";
for_each(v.begin(), v.end(), cout << constant(' ') << _1);
cout << endl;
```

- *constant*('␣') verschiebt die Auswertung der Konstante auf den Moment, an dem der Lambda-Ausdruck bewertet wird.
- Bei *cout* << '␣'<< *_1* würde sich ansonsten *cout* << '␣' sofort auswerten lassen und entsprechend würde insgesamt nur ein Leerzeichen ausgegeben werden.
- Der Aufruf von *constant* liefert eine Funktion, die keine Parameter erwartet.

sortit2.cpp

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>

using namespace std;
using namespace boost::phoenix;
using namespace boost::phoenix::arg_names;

int main() {
   vector<int> v(10);
   int i = 0;
   for_each(v.begin(), v.end(), arg1 = ref(i)++ - 5);
   cout << "unsorted:";
   for_each(v.begin(), v.end(), cout << val(' ') << arg1);
   cout << endl;

   sort(v.begin(), v.end(), arg1 * arg1 < arg2 * arg2);
   cout << "sorted:";
   for_each(v.begin(), v.end(), cout << val(' ') << arg1);
   cout << endl;
}
```

sortit2.cpp

```
vector<int> v(10);
int i = 0;
for_each(v.begin(), v.end(), arg1 = ref(i)++ - 5);
cout << "unsorted:";
for_each(v.begin(), v.end(), cout << val(' ') << arg1);
cout << endl;
```

- Die Bibliothek ähnelt auf dem ersten Blick der älteren Lambda-Bibliothek. Statt *_1* und *_2* werden hier *arg1* und *arg2* verwendet, statt *var* kommt *ref* zum Zuge.
- Im Unterschied zur älteren Bibliothek unterstützt diese jedoch auch verschachtelte Lambda-Ausdrücke.

sortit3.cpp

```
#include <cstdlib>
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main() {
   vector<int> v(10);
   int i = 0;
   for_each(v.begin(), v.end(),
      [&i](int& a) {
         return a = i++ - 5;
      });
   cout << "unsorted:";
   for_each(v.begin(), v.end(),
      [](int a) -> ostream& {
         return cout << ' ' << a;
      });
   cout << endl;

   sort(v.begin(), v.end(),
      [](int a, int b) {
         return abs(a) < abs(b);
      });
   cout << "sorted:";
   for_each(v.begin(), v.end(),
      [](int a) -> ostream& {
         return cout << ' ' << a;
      });
   cout << endl;
}
```

sortit3.cpp

```
vector<int> v(10);
int i = 0;
for_each(v.begin(), v.end(),
   [&i](int& a) {
      return a = i++ - 5;
   });
```

- In C++0x beginnen anonyme Funktionen mit einem []-Konstrukt, das festlegt, was von der lokalen Umgebung (*closure*) wie zur Verfügung gestellt wird.
- [&*i*] bedeutet, dass die anonyme Funktion die lokale Variable *i* als Referenz erhält.
- Wenn dieses Verfahren gewählt wird, sollte die anonyme Funktion nicht mehr länger verwendet werden, sobald die umgebende Funktion nicht mehr existiert.
- Alternativ können mit [=] Variablen auch kopiert werden.
- Der **return**-Typ der anonymen Funktion wird automatisch abgeleitet von der **return**-Anweisung.

sortit3.cpp

```
for_each(v.begin(), v.end(),
   [](int a) -> ostream& {
      return cout << ' ' << a;
   });
cout << endl;
```

- Der **return**-Typ kann auch explizit vorgegeben werden.
- Dann muss dazu jedoch die neue Syntax verwendet werden, bei der der **return**-Typ der Parameterliste folgt (*trailing return type*).
- Die Angabe ist in diesem Beispiel zwingend notwendig, da sonst *ostream* gewählt wird und *ostream*-Objekte nicht kopiert werden dürfen.
- Hinweis: C++0x bietet elegantere **for**-Schleifen, die *for_each*-Konstrukte in vielen Fällen ablösen. (Allerdings sind die bei gcc 4.5 noch nicht implementiert.)

sortit3.cpp

```
   vector<int> v(10);
   int i = 0;
   for_each(v.begin(), v.end(),
      [&i](int& a) {
         return a = i++ - 5;
      });
```

- Allen Implementierungen ist gemein, dass bei Lambda-Ausdrücken Klassen entstehen, von denen Funktionsobjekte instantiiert werden können:

```
class AnonymousFunction {
   public:
      // constructor accepts the captured environment
      AnonymousFunction(int& _i) : i(_i) {
      }
      // body of the anonymous function
      int operator()(int& a) {
         return a = i++ - 5;
      }
   private:
      // captured variables from the environment
      int& i;
};
```