- Generische Klassen und Funktionen, in C++ *templates* genannt, sind unvollständige Deklarationen bzw. Definitionen, die von Parametern abhängen. Überwiegend handelt es sich dabei um Typparameter.
- Sie können nur in instantiierter Form verwendet werden, wenn alle Parameter gegeben und ggf. deklariert sind.
- Unter bestimmten Umständen ist auch eine implizite Festlegung eines Typparameter möglich, wenn sich dieser aus dem Kontext ergibt.
- Generische Module wurden zuerst von CLU und Ada unterstützt (nicht in Kombination mit OO-Techniken) und später in Eiffel, einer statisch getypten OO-Sprache.
- Generische Klassen wurden zunächst primär für Container-Klassen verwendet wie etwa in der STL, der Einsatz von Templates wurde aber zunehmend ausgeweitet, insbesondere nach der Einführung von C++11.

- Templates ähneln teilweise den Makros, da
  - der Übersetzer den Programmtext des generischen Moduls erst bei einer Instantiierung vollständig analysieren und nach allen Fehlern durchsuchen kann und
  - für jede Instantiierung (mit unterschiedlichen Parametern) Code zu generieren ist.
- Anders als bei Makros
  - müssen sich generische Module sich an die üblichen Regeln halten (korrekte Syntax, Sichtbarkeit, Typverträglichkeiten),
  - können Syntaxfehler schon vor einer Instantiierung festgestellt werden und es
  - lässt sich die Code-Duplikation im Falle zweier Instanzen mit identischen Parametern vermeiden.

```
class List {
   // ...
   private:
      struct Linkable {
         Element element;
         Linkable* next;
      };
      Linkable* list;
};
```

- Diese Listenimplementierung speichert Objekte des Typs *Element*.
- Objekte, die einer von *Element* abgeleiteten Klasse angehören, können nur partiell (eben nur der Anteil von *Element*) abgesichert werden.
- Entsprechend müsste die Implementierung dieser Liste textuell dupliziert werden für jede zu unterstützende Variante des Datentyps *Element*.

```
class List {
   // ...
   private:
      struct Linkable {
         Element* element;
         Linkable* next;
      };
      Linkable* list;
};
```

- Wenn Zeiger oder Referenzen zum Einsatz kommen, können beliebige Erweiterungen von *Element* unterstützt werden.
- Generell stellt sich dann aber immer die Frage, wer für das Freigeben der Objekte hinter den Zeigern verantwortlich ist: Die Listenimplementierung oder der die Liste benutzende Klient?
- Die Anwendung der Liste für elementare Datentypen wie etwa **int** ist nicht möglich. Für Klassen, die keine Erweiterung von *Element* sind, müssten sogenannte Wrapper-Klassen konstruiert werden, die von *Element* abgeleitet werden und Kopien des gewünschten Typs aufnehmen können.

Java hat nach dem Vorbild bereits damals existierender anderer objekt-orientierter Programmiersprachen[1] eine Klasse *Object* eingeführt, von der implizit alle anderen Klassen abgeleitet sind. Das hat Vor- und Nachteile:

- Container können ohne die Techniken generischer Klassen geschrieben werden, indem *Object* als Basisklasse für die enthaltenen Objekte verwendet wird.
- Die Klasse *Object* enthält zahlreiche Methoden wie beispielsweise *toString*, die von den abgeleiteten Klassen überdefiniert werden können.
- Wenn Elemente dem Container entnommen werden, ist immer eine dynamische Typkonvertierung notwendig.
- Elementare Typen wie beispielsweise **int** sind keine Erweiterungen von *Object* und benötigen daher Wrapper-Klassen wie beispielsweise *Integer*, um ganzzahlige Werte einzupacken.

- Generell haben polymorphe Container-Klassen den Nachteil der mangelnden statischen Typsicherheit.
- Angenommen wir haben eine polymorphe Container-Klasse, die Zeiger auf Objekte unterstützt, die der Klasse A oder einer davon abgeleiteten Klasse unterstützen.
- Dann sei angenommen, dass wir nur Objekte der von A abgeleiteten Klasse B in dem Container unterbringen möchten. Ferner sei C eine andere von A abgeleitete Klasse, die jedoch nicht von B abgeleitet ist.
- Dann gilt:
  - Objekte der Klassen A und C können neben Objekten der Klasse B versehentlich untergebracht werden, ohne dass dies zu einem Fehler führt.
  - Wenn wir ein Objekt der Klasse B aus dem Container herausholen, ist eine Typkonvertierung unverzichtbar. Diese ist entweder prinzipiell unsicher oder kostet einen Test zur Laufzeit.
  - Entsprechend fatal wäre es, wenn Objekte der Klasse B erwartet werden, aber Objekte der Klassen A oder C enthalten sind.

```
template<typename Element>
class List {
   public:
      // ...
      void add(const Element& element);
   private:
      struct Linkable {
         Element element;
         Linkable* next;
      };
      Linkable* list;
};
```

- Wenn der Klassendeklaration eine Template-Parameterliste vorangeht, dann wird daraus insgesamt die Deklaration eines Templates.
- Typparameter bei Templates sind typischerweise von der Form **typename** *T*, aber C++ unterstützt auch andere Parameter, die beispielsweise die Dimensionierung eines Arrays bestimmen.

```
List<int> list; // select int as Element type
list.add(7);
```

- Templates werden instantiiert durch die Angabe des Klassennamens und den Parametern in gewinkelten Klammern.

Deklaration eines Templates

$$
\begin{array}{rcl}
\langle\text{template-declaration}\rangle & \longrightarrow & \textbf{template} \\
 & & \text{„<“ } \langle\text{template-parameter-list}\rangle \text{ „>“} \\
 & & \langle\text{declaration}\rangle \\
\langle\text{template-parameter-list}\rangle & \longrightarrow & \langle\text{template-parameter}\rangle \\
 & \longrightarrow & \langle\text{template-parameter-list}\rangle \text{ „,“} \\
 & & \langle\text{template-parameter}\rangle \\
\langle\text{template-parameter}\rangle & \longrightarrow & \langle\text{type-parameter}\rangle \\
 & \longrightarrow & \langle\text{parameter-declaration}\rangle
\end{array}
$$

- Eine reguläre $\langle$parameter-declaration$\rangle$ ist nur zulässig für ganzzahlige Datentypen wie etwa **int**, Aufzählungstypen, Zeiger und Referenzen.

⟨type-parameter⟩ ⟶ ⟨type-parameter-key⟩ [„...“] [⟨identifier⟩]
⟶ ⟨type-parameter-key⟩ [⟨identifier⟩]
„=“ ⟨type-id⟩
⟶ **template**
„<“ ⟨template-parameter-list⟩ „>“
⟨type-parameter-key⟩ [„...“] [⟨identifier⟩]
⟶ **template**
„<“ ⟨template-parameter-list⟩ „>“
⟨type-parameter-key⟩ [⟨identifier⟩]
„=“ ⟨id-expression⟩

⟨type-parameter-key⟩ ⟶ **class**
⟶ **typename**

- Das ist die Syntax von C++17. Bei älteren Versionen ist bei Template-Template-Parametern bei ⟨type-parameter-key⟩ immer **class** anzugeben.

sample-lines.cpp

```
#include <cstdlib>
#include <iostream>
#include <string>
#include "reservoir-sampler.hpp"

int main(int argc, char** argv) {
   ReservoirSampler<std::string> r(10);
   std::string line;
   while (std::getline(std::cin, line)) {
      r.add(std::move(line));
   }
   for (std::size_t index = 0; index < r.get_size(); ++index) {
      std::cout << r(index) << std::endl;
   }
}
```

- Diese Anwendung wählt aus den Zeilen der Standardeingabe bis zu 10 Zeilen zufällig aus und gibt sie anschließend aus.
- *ReservoirSampler* ist eine Container-Klasse, die sich bis zu n Objekten merkt entsprechend dem Reservoir-Sampling-Algorithmus (hier $n = 10$).

sample-lines.cpp

```
ReservoirSampler<std::string> r(nof_lines);
std::string line;
while (std::getline(std::cin, line)) {
   r.add(std::move(line));
}
for (std::size_t index = 0; index < r.get_size(); ++index) {
   std::cout << r(index) << std::endl;
}
```

- Mit *ReservoirSampler<std::string>* wird die Template-Klasse *ReservoirSampler* mit *std::string* als Typparameter instantiiert. Der Typparameter legt hier den Element-Typ des Containers fest.
- Der Konstruktor erwartet eine ganze Zahl als Parameter, der die Zahl der auszuwählenden Einträge bestimmt.
- Der ()-Operator wurde hier überladen, um einen Zugriff auf die ausgewählten Elemente zu erlauben.

Eine Container-Klasse als Template

reservoir-sampler.hpp

```
#ifndef RESERVOIR_SAMPLER_HPP
#define RESERVOIR_SAMPLER_HPP

#include <algorithm>
#include <cassert>
#include <cstdlib>
#include <random>
#include <utility>

template<typename T>
class ReservoirSampler {
   public:
      /* ... */
   private:
      std::mt19937 engine;
      std::size_t size;
      std::size_t taken;
      T* data;
};

#endif
```

reservoir-sampler.hpp

```
std::mt19937 engine;
std::size_t size;
std::size_t taken;
T* data;
```

- Um Elemente zufällig aussuchen zu können, verwenden wir ein *std*::*mt19937*-Objekt als Generator für Pseudo-Zufallszahlen.
- *size* gibt die Zahl der gewünschten Elemente an.
- *taken* gibt die Zahl der mit *add* hinzugefügten Elemente an.
- *data* zeigt auf ein Array mit *size* Elementen.
- Bei *data* sind nur $min(size, taken)$ Elemente tatsächlich existent. Wir werden hier noch sehen, wie wir in C++ mit der Situation umgehen können, nur die tatsächlich existierenden Elemente zu konstruieren. Um das zu erreichen, wird das Belegen und Freigeben von Speicher von dem Konstruieren und Abbauen von Objekten völlig getrennt.
- Die Methode *get_size* liefert die Zahl der verfügbaren Elemente, also $min(size, taken)$.

reservoir-sampler.hpp

```
template<typename T>
class ReservoirSampler {
   public:
      ReservoirSampler(std::size_t size) :
         engine(std::random_device()()),
         size(size), taken(0),
         /* allocate raw memory without constructing anything */
         data(static_cast<T*>(operator new[](sizeof(T) * size))) {
      }
      /* ... */

   private:
      /* ... */
};
```

- Das ist der „normale“ Konstruktor, mit dem ein Reservoir einer gegebenen Größe angelegt werden kann.
- Um das Konstruieren noch nicht vorhandener Elemente des Typs T zu vermeiden, legen wir nur den Speicher an, ohne eines der Objekte zu konstruieren. Dies geht mit der expliziten Verwendung von **operator new**[], wobei dann die Größe in Bytes anzugeben ist.

reservoir-sampler.hpp

```
ReservoirSampler(const ReservoirSampler& other) :
      engine(std::random_device()()),
      size(other.size), taken(other.taken),
      /* allocate raw memory without constructing anything */
      data(static_cast<T*>(operator new[](sizeof(T) * size))) {
   for (std::size_t index = 0; index < other.get_size(); ++index) {
      /* copy-construct already constructed elements of other */
      new (data + index) T(other.data[index]);
   }
}
```

- Der Kopierkonstruktor konstruiert nur die n Elemente, die bei *other* bereits existieren.
- Die Methode *get_size* liefert uns n, d.h. die Zahl existierender Elemente.
- Das Konstruieren auf bereits vorhandenem Speicher geht mit **new**, wenn vor dem Datentyp in Klammern die Adresse angegeben wird (*placement*).

reservoir-sampler.hpp

```
~ReservoirSampler() {
   /* as we allocated raw memory we need to deconstruct
      all elements ourselves */
   for (std::size_t index = 0; index < get_size(); ++index) {
      data[index].~T();
   }
   /* release raw memory */
   operator delete[](data);
}
```

- Da das Belegen von Speicher und das Konstruieren getrennt erfolgte, müssen wir beim Abbau auch beides getrennt vornehmen.
- Der *destructor* wird hier für die existierenden Elemente explizit aufgerufen.
- Mit **operator delete**[](*data*) wird dann nur der Speicher freigegeben.

reservoir-sampler.hpp

```
friend void swap(ReservoirSampler& rs1, ReservoirSampler& rs2) {
   /* there is no need to swap the engines */
   std::swap(rs1.size, rs2.size);
   std::swap(rs1.taken, rs2.taken);
   std::swap(rs1.data, rs2.data);
}
```

- Die *swap*-Funktion wird wie gewohnt implementiert – wir verzichten hier aus pragmatischen Gründen auf das Vertauschen der Pseudo-Zufallsgeneratoren.

reservoir-sampler.hpp

```
ReservoirSampler(ReservoirSampler&& other) : ReservoirSampler() {
   swap(*this, other);
}
ReservoirSampler& operator=(ReservoirSampler other) {
   swap(*this, other);
   return *this;
}
```

- Entsprechend dem *copy and swap idiom* lassen sich der *move constructor* und der Zuweisungsoperator wie gewohnt leicht implementieren.

reservoir-sampler.hpp

```
void add(T value) {
   if (taken < size) {
      /* move-construct new element in reservoir */
      new (data + taken) T(std::move(value));
   } else {
      std::size_t select = std::uniform_int_distribution<std::size_t>
         (0, taken)(engine);
      if (select < size) {
         using std::swap;
         /* use argument-dependent lookup (ADL) */
         swap(value, data[select]);
      }
   }
   ++taken;
}
```

- Wenn *taken* $<$ *size*, dann ist ein weiteres Element zu konstruieren. Das erfolgt hier wieder mit **new** unter Verwendung eines *placement* auf *data* + *taken*.
- Da die lokale Variable *value* danach nicht mehr benötigt wird, können wir hier *std*::*move*(*value*) verwenden, so dass ggf. der *move constructor* zum Einsatz kommt.

reservoir-sampler.hpp

```
if (select < size) {
   using std::swap;
   /* use argument-dependent lookup (ADL) */
   swap(value, data[select]);
}
```

- Wenn ein bereits existierendes Element auszutauschen ist, erledigen wir das mit *swap*.
- Wir benutzen hier *swap* ohne Qualifikation, damit ggf. die *swap*-Funktion gefunden wird, die im Namensraum von T liegt.
- Mit **using** *std*::*swap* wird sichergestellt, dass notfalls *std*::*swap* verwendet wird, wenn keine passendere *swap*-Funktion vorliegt.
- Da die Suche nach dem passenden *swap* den Datentyp der Argumente berücksichtigt, wird dies als *argument-dependent lookup* (ADL) bezeichnet.

reservoir-sampler.hpp

```
std::size_t get_taken() const {
   return taken;
}
std::size_t get_size() const {
   return std::min(size, taken);
}
const T& operator()(std::size_t index) const {
   assert(index < get_size());
   return data[index];
}
```

- Die verbleibenden Funktionen sidn trivial zu implementieren.
- Die Minimumfunktion *std*::*min* wird von **#include** <algorithm> geliefert und akzeptiert beliebig viele Argumente.

- Template-Klassen können nicht ohne weiteres mit beliebigen Typparameter instantiiert werden.
- C++ verlangt, dass *nach* der Instantiierung die gesamte Template-Deklaration und alle zugehörigen Methoden zulässig sein müssen in C++.
- Entsprechend führt jede neuartige Instantiierung zur völligen Neuüberprüfung der Template-Deklaration und aller zugehörigen Methoden unter Verwendung der gegebenen Parameter.
- Daraus ergeben sich Abhängigkeiten, die ein Typ, der als Parameter bei der Instantiierung angegeben wird, einzuhalten hat.

- Folgende Abhängigkeiten sind zu erfüllen für den Typ-Parameter T der Template-Klasse *ReservoirSample*:
  - Es wird wird ein Kopierkonstruktor für T benötigt. Dieser ist zwingend für den Kopierkonstruktor von *ReservoirSample* notwendig. In den anderen Fällen kann auch alternativ ein *move constructor* bzw. *swap* zum Zuge kommen, falls vorhanden.
  - Destruktor: Für den Abbau der Objekte (entweder beim Austauschen oder beim Abbau des gesamten Reservoirs) wird dieser benötigt.

template-failure.cpp

```
#include "reservoir-sampler.hpp"

struct Test {
   int i;
   Test(int i) : i(i) {}
   Test(const Test& other) = delete;
};

int main() {
   ReservoirSampler<Test> r(5);
   Test val{1};
   r.add(val);
}
```

- Hier wurde der Kopierkonstruktur explizit unterbunden, womit implizit auch der *move constructor* wegfällt.
- Damit wird eine der Template-Abhängigkeiten von *ReservoirSample* nicht erfüllt.

```
theon$ g++ -o template-failure template-failure.cpp 2>&1 | head -7
template-failure.cpp: In function 'int main()':
template-failure.cpp:12:13: error: use of deleted function 'Test::Test(const Test&)'
    r.add(val);
             ^
template-failure.cpp:6:4: note: declared here
    Test(const Test& other) = delete;
    ^~~~
theon$
```

- Die Fehlermeldungen können bei nicht erfüllten Template-Abhängigkeiten ungemein umfangreich werden.
- Es lohnt sich hier aber ein Blick auf die ersten Meldungen, die das Problem recht treffend beschreiben.

- Bei der Übersetzung von Templates gibt es ein schwerwiegendes Problem:
  - Dort, wo die Methoden einer Template-Klasse implementiert sind, ist nicht bekannt, welche Instanzen benötigt werden.
  - Dort, wo das Template instantiiert wird sind die Methodenimplementierungen der Template-Klasse unbekannt, wenn diese nicht in der entsprechenden Header-Datei stehen.
- Folgende Fragen stellen sich:
  - Wie kann der Übersetzer die benötigten Template-Instanzen generieren?
  - Wie kann vermieden werden, dass die gleiche Template-Instanz mehrfach generiert wird?

- Beim Inclusion-Modell wird mit Hilfe einer **#include**-Anweisung auch die Methoden-Implementierung hereinkopiert, so dass sie beim Übersetzung der instantiierenden Module sichtbar ist.
- Das funktioniert grundsätzlich bei allen C++-Übersetzern, aber es führt im Normalfall zu einer Code-Vermehrung, wenn das gleiche Template in unterschiedlichen Quellen in gleicher Weise instantiiert wird.
- Das Borland-Modell sieht hier eine zusätzliche Verwaltung vor, die die Mehrfach-Generierung unterbindet.
- Der *gcc* unterstützt das Borland-Modell, wenn jeweils die Option *-frepo* gegeben wird, die dann die Verwaltungsinformationen in Dateien mit der Endung *rpo* unterbringt. Dies erfordert die Zusammenarbeit mit dem Linker und funktioniert beim *gcc* somit nur mit dem GNU-Linker.

- Der elegantere Ansatz vermeidet zusätzliche **#include**-Anweisungen. Entsprechend muss der Übersetzer selbst die zugehörige Quelle finden.
- Hierfür gibt es kein standardisiertes Vorgehen. Jeder Übersetzer, der dieses Modell unterstützt, hat dafür eigene Verwaltungsstrukturen.
- *gcc* unterstützt dieses Modell jedoch nicht.
- Der von Sun ausgelieferte C++-Übersetzer (bei uns mit *CC* aufzurufen) folgt diesem Modell.
- Im C++-Standard von 2003 wurde dies explizit über das Schlüsselwort **export** unterstützt.
- Da dies jedoch von kaum jemanden implementiert worden ist, wurde dies bei C++11 gestrichen. Entsprechend ist das Inclusion-Modell das einzige, das sich in der Praxis durchgehend etabliert hat.

```
template class ReservoirSampling<std::string>;
```

- Die Kontrolle darüber, genau wann und wo der Code für eine konkrete Template-Instantiierung zu erzeugen ist, kann mit Hilfe expliziter Instantiierungen kontrolliert werden.
- Eine explizite Instantiierung wiederholt die Template-Deklaration ohne das Innenleben, nennt aber die Template-Parameter.
- Dann wird an dieser Stelle der entsprechende Code erzeugt.
- Das darf dann aber nur einmal im gesamten Programm erfolgen. Sonst gibt es Konflikte beim Zusammenbau.
- Seit C++11 ist es möglich, so eine explizite Instantiierung mit dem Schlüsselwort **extern** zu versehen. Dann wird die Generierung des entsprechenden Codes unterdrückt und stattdessen die anderswo explizit instantiierte Fassung verwendet.

```
extern template class ReservoirSampling<std::string>;
```

- Der Vorteil expliziter Instantiierungen liegt in der Vermeidung redundanten Codes, ohne sich auf entsprechende implementierungsabhängige Unterstützungen des Übersetzers verlassen zu müssen.
- Ein weiterer Vorzug ist die kürzere Übersetzungszeit, da die Template-Implementierung dann nur noch dort benötigt wird, wo explizite Instantiierungen vorgenommen werden.
- Diese Vorgehensweise nötigt den Programmierer jedoch, selbst einen Überblick zu behalten, welche Instantiierungen alle benötigt werden. Das wird sehr schnell sehr unübersichtlich.
- Das liegt an der sogenannten *one-definition-rule* (ODR), d.h. Objekte dürfen beliebig oft deklariert, aber global nur einmal definiert werden. Bei impliziten Instantiierungen ist das ein Problem des Übersetzters, bei expliziten Instantiierungen übernimmt der Programmierer die Verantwortung hierfür.
- Diese Technik wird daher typischerweise nur in isolierten Fällen benutzt.