- 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 werden primär für Container-Klassen verwendet wie etwa in der STL, zunehmend aber auch für andere Anwendungen wie etwa der Metaprogrammierung.

- 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.

- 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.

Deklaration eines Templates


```
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.

⟨template-declaration⟩ ⟶ **template**
„<“ ⟨template-parameter-list⟩ „>“
⟨declaration⟩

⟨template-parameter-list⟩ ⟶ ⟨template-parameter⟩

⟶ ⟨template-parameter-list⟩ „,“
⟨template-parameter⟩

⟨template-parameter⟩ ⟶ ⟨type-parameter⟩

⟶ ⟨parameter-declaration⟩

- Eine reguläre ⟨parameter-declaration⟩ ist nur zulässig für ganzzahlige Datentypen wie etwa **int**, Aufzählungstypen, Zeiger und Referenzen.

⟨type-parameter⟩ ⟶ **class** [„...“] [⟨identifier⟩]

⟶ **class** [⟨identifier⟩] „=“ ⟨type-id⟩

⟶ **typename** [„...“] [⟨identifier⟩]

⟶ **typename** [⟨identifier⟩] „=“ ⟨type-id⟩

⟶ **template**
„<“ ⟨template-parameter-list⟩ „>“
class [„...“] [⟨identifier⟩]

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

Tail.cpp

```
#include <iostream>
#include <string>
#include "History.hpp"

int main() {
   History<std::string> tail(10);
   std::string line;
   while (std::getline(std::cin, line)) {
      tail.add(line);
   }
   for (int i = tail.size() - 1; i >= 0; --i) {
      std::cout << tail[i] << std::endl;
   }
}
```

- Diese Anwendung gibt die letzten 10 Zeilen der Standardeingabe aus.
- *History* ist eine Container-Klasse, die sich nur die letzten n hinzugefügten Objekte merkt. Alle vorherigen Einträge werden rausgeworfen.

Tail.cpp

```
History<std::string> tail(10);
std::string line;
while (std::getline(std::cin, line)) {
   tail.add(line);
}
for (int i = tail.size() - 1; i >= 0; --i) {
   std::cout << tail[i] << std::endl;
}
```

- Mit *History<std::string> tail*(10) wird die Template-Klasse *History* 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 zu speichernden Einträge bestimmt.
- Der []-Operator wurde hier überladen, um eine Notation analog zu Arrays zu erlauben. So steht *tail*[0] für das zuletzt hinzugefügte Objekt, *tail*[1] für das vorletzte usw.

History.hpp

```
#include <vector>
template<typename Item>
class History {
   public:
      // constructor
      History(unsigned int capacity);
      // accessors
      unsigned int max_size() const; // returns capacity
      unsigned int size() const; // returns # of items in buffer
      const Item& operator[](unsigned int i) const;
         // PRE: i >= 0 && i < size()
         // i = 0: return item added last
         // i = 1: return item before last item
      // mutators
      void add(const Item& item);
   private:
      std::vector<Item> items; // ring buffer with the last n items
      unsigned int index; // next item will be stored at items[index]
      unsigned int filled; // # of items in ring buffer so far
};
```

History.hpp

```
template<typename Item>
class History {
   // ...
};
```

- Typparameter bei Templates werden in der Form **typename** T spezifiziert. Zugelassen sind nicht nur Klassen, sondern auch elementare Datentypen wie etwa **int**.
- Alternativ können Typparameter auch mit **class** T deklariert werden. Das schränkt T nicht auf Klassen ein. Diese Schreibweise wird nur noch aus historischen Gründen unterstützt.

History.hpp

```
    const Item& operator[](unsigned int i) const;
       // PRE: i >= 0 && i < size()
       // i = 0: return item added last
       // i = 1: return item before last item
```

- Per Typparameter eingeführte Klassen können innerhalb des Templates so verwendet werden, als wären sie bereits vollständig deklariert worden.
- Der []-Operator erhält einen Index als Parameter und liefert hier eine konstante Referenz zurück, die Veränderungen des Objekts nicht zulassen. Dies ist hier beabsichtigt, da eine *History* Objekte nur aufzeichnen, jedoch nicht verändern sollte.

History.hpp

```
private:
   std::vector<Item> items; // ring buffer with the last n items
   unsigned int index; // next item will be stored at items[index]
   unsigned int filled; // # of items in ring buffer so far
```

- Template-Klassen steht es frei, andere Templates zu verwenden und ggf. auch hierbei die eigenen Parameter zu verwenden.
- In diesem Beispiel wird ein Vektor mit dem Template-Parameter *Item* angelegt.

History.hpp

```
private:
   std::vector<Item> items; // ring buffer with the last n items
   unsigned int index; // next item will be stored at items[index]
   unsigned int filled; // # of items in ring buffer so far
```

- *vector* ist eine Template-Klasse aus der STL, die anders als die regulären Arrays in C++
  - nicht wie Zeiger behandelt werden,
  - sich die Dimensionierung merken und
  - dynamisch wachsen können.
- Achtung: Es wird nicht überprüft, ob die Indizes innerhalb der zulässigen Array-Grenzen liegen.

History.tpp

```
#include <cassert>

template<typename Item>
History<Item>::History(unsigned int capacity) :
      items(capacity), index(0), filled(0) {
   assert(capacity > 0);
} // History<Item>::History

template<typename Item>
unsigned int History<Item>::max_size() const {
   return items.size();
} // History<Item>::max_size

template<typename Item>
unsigned int History<Item>::size() const {
   return filled;
} // History<Item>::size
```

- Allen Methodendeklarationen, die zu einer Template-Klasse gehören, muss die Template-Deklaration vorangehen und der Klassenname ist mit der Template-Parameterliste zu erweitern.

History.tpp

```
template<typename Item>
void History<Item>::add(const Item& item) {
   items[index] = item;
   index = (index + 1) % items.size();
   if (filled < items.size()) {
      filled += 1;
   }
} // History<Item>::add
```

- *add* legt eine Kopie des übergebenen Objekts in der aktuellen Position im Ringpuffer ab.
- Die Template-Klasse *vector* aus der STL unterstützt ebenfalls den []-Operator.

History.tpp

```
template<typename Item>
const Item& History<Item>::operator[](unsigned int i) const {
   assert(i >= 0 && i < filled);
   // we are adding items.size to the left op of % to avoid
   // negative operands (effect not defined by ISO C++)
   return items[(items.size() + index - i - 1) % items.size()];
}; // History<Item>::operator[]
```

- Indizierungsoperatoren sollten die Gültigkeit der Indizes überprüfen, falls dies möglich und sinnvoll ist.
- *items.size*() liefert die Größe des Vektors, die vom *nitems*-Parameter beim Konstruktor abgeleitet wird.
- Da es sich bei *items* um einen Ringpuffer handelt, verwenden wir den Modulo-Operator, um den richtigen Index relativ zur aktuellen Position zu ermitteln.

- 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 der Template-Klasse *History*:
  - *Default Constructor*: Dieser wird implizit von der Template-Klasse *vector* verwendet, um das erste Element im Array zu initialisieren.
  - *Copy Constructor*: Dieser wird ebenfalls implizit von *vector* verwendet, um alle weiteren Elemente in Abhängigkeit vom ersten Element zu initialisieren.
  - Zuweisungs-Operator: Ist notwendig, damit Elemente in und aus der Template-Klasse *History* kopiert werden können.
  - Destruktor: Dieser wird von der Template-Klasse *vector* verwendet für Elemente, die aus dem Ringpuffer fallen bzw. bei der Auflösung des gesamten Ringpuffers.

TemplateFailure.cpp

```
#include "History.hpp"

class Integer {
   public:
      Integer(int i) : i(i) {};
   private:
      int i;
};

int main() {
   History<Integer> integers(10);
}
```

- Hier fehlt ein Default-Konstruktor. Dieser wird auch nicht implizit erzeugt, da mit *Integer*(**int** *i*) ein Konstruktor bereits gegeben ist.
- Damit wird eine der Template-Abhängigkeiten von *History* nicht erfüllt.

Template-Abhängigkeiten


```
thales$ make 2>&1 | fold -sw 80
g++ -Wall -g -std=gnu++11   -c -o TemplateFailure.o TemplateFailure.cpp
In file included from /usr/local/gcc51/include/c++/5.2.0/vector:62:0,
                 from History.hpp:8,
                 from TemplateFailure.cpp:1:
/usr/local/gcc51/include/c++/5.2.0/bits/stl_construct.h: In instantiation of
'void std::_Construct(_T1*, _Args&& ...) [with _T1 = Integer; _Args = {}]':
/usr/local/gcc51/include/c++/5.2.0/bits/stl_uninitialized.h:519:18:   required
from 'static _ForwardIterator
std::__uninitialized_default_n_1<_TrivialValueType>::__uninit_default_n(_Forward
Iterator, _Size) [with _ForwardIterator = Integer*; _Size = unsigned int; bool
_TrivialValueType = false]'
/usr/local/gcc51/include/c++/5.2.0/bits/stl_uninitialized.h:575:20:   required
from '_ForwardIterator std::__uninitialized_default_n(_ForwardIterator, _Size)
[with _ForwardIterator = Integer*; _Size = unsigned int]'
/usr/local/gcc51/include/c++/5.2.0/bits/stl_uninitialized.h:637:44:   required
from '_ForwardIterator std::__uninitialized_default_n_a(_ForwardIterator,
_Size, std::allocator<_Tp>&) [with _ForwardIterator = Integer*; _Size =
unsigned int; _Tp = Integer]'
/usr/local/gcc51/include/c++/5.2.0/bits/stl_vector.h:1311:36:   required from
'void std::vector<_Tp, _Alloc>::_M_default_initialize(std::vector<_Tp,
_Alloc>::size_type) [with _Tp = Integer; _Alloc = std::allocator<Integer>;
std::vector<_Tp, _Alloc>::size_type = unsigned int]'
/usr/local/gcc51/include/c++/5.2.0/bits/stl_vector.h:279:30:   required from
'std::vector<_Tp, _Alloc>::vector(std::vector<_Tp, _Alloc>::size_type, const
allocator_type&) [with _Tp = Integer; _Alloc = std::allocator<Integer>;
```

```
std::vector<_Tp, _Alloc>::size_type = unsigned int; std::vector<_Tp,
_Alloc>::allocator_type = std::allocator<Integer>]'
History.tpp:9:42:   required from 'History<Item>::History(unsigned int) [with
Item = Integer]'
TemplateFailure.cpp:11:32:   required from here
/usr/local/gcc51/include/c++/5.2.0/bits/stl_construct.h:75:7: error: no
matching function for call to 'Integer::Integer()'
     { ::new(static_cast<void*>(__p)) _T1(std::forward<_Args>(__args)...); }
       ^
TemplateFailure.cpp:5:7: note: candidate: Integer::Integer(int)
       Integer(int i) : i(i) {};
       ^
TemplateFailure.cpp:5:7: note:   candidate expects 1 argument, 0 provided
TemplateFailure.cpp:3:7: note: candidate: constexpr Integer::Integer(const
Integer&)
 class Integer {
       ^
TemplateFailure.cpp:3:7: note:   candidate expects 1 argument, 0 provided
TemplateFailure.cpp:3:7: note: candidate: constexpr Integer::Integer(Integer&&)
TemplateFailure.cpp:3:7: note:   candidate expects 1 argument, 0 provided
<builtin>: recipe for target 'TemplateFailure.o' failed
make: *** [TemplateFailure.o] Error 1
thales$
```

- Bei der Übersetzung von Templates gibt es ein schwerwiegendes Problem:
  - Dort, wo die Methoden des Templates stehen (hier etwa in *History.tpp*), ist nicht bekannt, welche Instanzen benötigt werden.
  - Dort, wo das Template instantiiert wird (hier etwa in *Tail.cpp*), sind die Methodenimplementierungen des Templates unbekannt, da zwar *History.hpp* reinkopiert wurde, aber eben nicht ohne weiteres *History.tpp*.
- 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.
- Entsprechend wird in *History.hpp* am Ende auch noch *History.tpp* mit **#include** hereinkopiert. (Deswegen auch die Datei-Endung „.tpp“ anstelle von „.cpp“.)
- 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 History<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 History<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.

History.hpp

```
class History {
   public:
      History(unsigned int capacity) :
            items(capacity), index(0), filled(0) {
         assert(capacity > 0);
      }
      unsigned int max_size() const { // returns capacity
         return items.size();
      }
      unsigned int size() const { // returns # of items in buffer
         return filled;
      }
      const Item& operator[](unsigned int i) const {
         assert(i >= 0 && i < filled);
         return items[(items.size() + index - i - 1) % items.size()];
      }
      void add(const Item& item) {
         items[index] = item;
         index = (index + 1) % items.size();
         if (filled < items.size()) {
            filled += 1;
         }
      }
   private:
      std::vector<Item> items; // ring buffer with the last n items
      unsigned int index; // next item will be stored at items[index]
      unsigned int filled; // # of items in ring buffer so far
};
```

- Wenn eine Methode direkt in einer Klassendeklaration definiert wird, d.h. ihre Implementierung integriert ist, dann ist sie automatisch als **inline** deklariert. (Das ist äquivalent zu einer getrennten Implementierung, bei der das Schlüsselwort **inline** angegeben wird.)
- **inline**-Methoden geben dem Übersetzer die Möglichkeit, den Code unmittelbar beim Aufruf einer Methode zu expandieren. (Voraussetzung ist hier, dass sich alles statisch ableiten lässt.)
- Wenn alle Methoden einer Template-Klasse **inline** sind, entfällt die Notwendigkeit einer „.tpp“-Datei und entsprechend fällt die gesamte Problematik weg.
- Ferner wird damit das Optimierungspotential eröffnet, indem der Methodenaufruf und die Parameterübergabe wegfallen.
- Allerdings wird der erzeugte Code umfangreicher und die Übersetzungszeiten nehmen deutlich zu.
- Große Teile der C++-Standard-Bibliothek sind auf diese Weise realisiert.

- Analog wie bei *History* lassen sich Container-Klassen als generische Klassen ausführen.
- Dies geht auch beispielsweise mit der der bereits eingeführten *Array*-Klasse.

Array als generische Klasse

array.hpp

```
#ifndef ARRAY_HPP
#define ARRAY_HPP

#include <cassert>
#include <cstdlib>
#include <utility>

template<typename T>
class Array {
   public:
      Array() : nof_elements(0), p(nullptr) {
      }
      Array(std::size_t nof_elements) :
            nof_elements{nof_elements},
            p{new int[nof_elements] {}} {
      }
      friend void swap(Array& a1, Array& a2) {
         std::swap(a1.nof_elements, a2.nof_elements);
         std::swap(a1.p, a2.p);
      }
      Array(const Array& other) :
            nof_elements{other.nof_elements},
            p{new int[nof_elements]} {
         for (std::size_t i = 0; i < nof_elements; ++i) {
            p[i] = other.p[i];
         }
      }
```

array.hpp

```
      Array(Array&& other) : Array() {
         swap(*this, other);
      }
      ~Array() {
         delete[] p;
      }
      Array& operator=(Array other) {
         swap(*this, other);
         return *this;
      }
      std::size_t size() const {
         return nof_elements;
      }
      int& operator()(std::size_t i) {
         assert(i < nof_elements);
         return p[i];
      }
      const int& operator()(std::size_t i) const {
         assert(i < nof_elements);
         return p[i];
      }
   private:
      std::size_t nof_elements;
      T* p;
};

#endif
```

testit.cpp

```
int main() {
   std::cout << "Test..." << std::endl;
   Array<int> a{10};
   std::cout << "a = "; print_array(a); std::cout << std::endl;
   for (std::size_t i = 0; i < a.size(); ++i) {
      a(i) = i;
   }
   std::cout << "a = "; print_array(a); std::cout << std::endl;
   Array<int> b{a};
   b(1) = 77;
   std::cout << "b = "; print_array(b); std::cout << std::endl;
   std::cout << "a = "; print_array(a); std::cout << std::endl;
   Array<int> c = Array<int>{10};
}
```

- Hier wurde jeweils *Array* durch *Array*<**int**> ersetzt.
- Wie ist aber *print_array* anzupassen?

testit.cpp

```
template<typename T>
void print_array(const Array<T>& a) {
   for (std::size_t i = 0; i < a.size(); ++i) {
      std::cout << " " << a(i);
   }
}
```

- C++ unterstützt auch generische Funktionen.
- Hier unterstützt *print_array* beliebige Instantiierungen von *Array*.
- Wenn sich die Typparameter einer Template-Funktion durch die aktuellen Parameter herleiten lassen, müssen sie nicht mehr explizit angegeben werden.

testit2.cpp

```
Array<int> a{42};
a(24) = 77;
```

- Wenn dies ohne Optimierung übersetzt wird, dann werden alle Methodenaufrufe, seien sie implizit oder explizit in naheliegender Form umgesetzt:

```
        pushl   $42
        leal    -16(%ebp), %eax
        pushl   %eax
        call    _ZN5ArrayIiEC1Ej        ; Array<int> a{42}
        addl    $16, %esp
        subl    $8, %esp
        pushl   $24
        leal    -16(%ebp), %eax
        pushl   %eax
        call    _ZN5ArrayIiEclEj        ; a(24)
        addl    $16, %esp
        movl    $77, (%eax)             ; a(24) = 77;
```

testit2.cpp

```
Array<int> a{42};
a(24) = 77;
```

- Bei einer Übersetzung mit „-O2“ werden die Methodenaufrufe durch expandierten Code ersetzt.
- Die Abstraktionsgrenzen existieren dann nur noch zur Übersetzzeit und sind auf Assembler-Ebene nicht mehr sichtbar.

```
        pushl   $168                    ; 168 = 42 * sizeof(int)
        call    _Znaj                   ; new int[42]
        leal    168(%eax), %ecx
        movl    %eax, %edx
        addl    $16, %esp
.L2:
        movl    $0, (%edx)              ; Initialisierung mit Nullen
        addl    $4, %edx
        cmpl    %ecx, %edx
        jne     .L2
        subl    $12, %esp
        movl    $77, 96(%eax)           ; a(24) = 77
```