- Generische Module, in C++ *templates* genannt, sind unvollständige Deklarationen, die von nicht deklarierten Typparametern abhängen.
- Sie können nur in instantiierter Form verwendet werden, wenn alle Typparameter gegeben und deklariert sind.
- Unter bestimmten Umständen sind auch implizite Instantiierungen möglich, bei denen sich die Typparameter aus dem Kontext ergeben.
- Generische Module wurden zuerst von Ada unterstützt (nicht in Kombination mit OO-Techniken) und später in Eiffel, einer statisch getypten OO-Sprache.
- Generische Module werden primär für Container-Klassen verwendet wie etwa in der STL.

- 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 Typparametern) Code zu generieren ist.
- Anders als bei Makros
  - müssen sich generische Module sich an die üblichen Regeln halten (korrekte Syntax, Sichtbarkeit, Typverträglichkeiten),
  - können entsprechend einige Fehler schon vor einer Instantiierung festgestellt werden und es
  - lässt sich die Code-Duplikation im Falle zweier Instanzen mit den gleichen Typparametern vermeiden.

```
class ListOfElements {
   // ...
   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.
- Parameter 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.

Tail.C

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

using namespace std;

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

- 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.C

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

- Mit *History*< *string* > *tail*(10) wird die Template-Klasse *History* mit *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.h

```
#include <vector>
template<typename Item>
class History {
   public:
      // constructor
      History(int nitems);
      // accessors
      int max_size() const; // returns capacity
      int size() const; // returns # of items in buffer
      const Item& operator[](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
      int index; // next item will be stored at items[index]
      int nof_items; // # of items in ring buffer so far
};
```

History.h

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

- Typparameter bei Templates werden immer in der Form **typename** *T* spezifiziert. Zugelassen sind nicht nur Klassen, sondern auch elementare Datentypen wie etwa **int**.

History.h

```
    const Item& operator[](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.h

```
  private:
     std::vector< Item > items;
        // ring buffer with the last n items
     int index; // next item will be stored at items[index]
     int nof_items; // # 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.h

```
private:
   std::vector< Item > items;
      // ring buffer with the last n items
   int index; // next item will be stored at items[index]
   int nof_items; // # 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
  - in der Lage sind, die Zulässigkeit der Indizes zu überprüfen.
- Auf Basis der Template-Klasse *vector* lassen sich leicht andere Zuordnungen von Indizes zu zugehörigen Objekten umsetzen.

Eine Container-Klasse als Template

History.C

```
#include <cassert>
#include "History.h"

template<typename Item>
History<Item>::History(int nitems) :
      items(nitems), index(0), nof_items(0) {
   assert(nitems > 0);
} // History<Item>::History

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

template<typename Item>
int History<Item>::size() const {
   return nof_items;
} // 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.C

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

```
template<typename Item>
const Item& History<Item>::operator[](int i) const {
   assert(i >= 0 && i < nof_items);
   // 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 sind.

- 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.C

```
#include "History.h"

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

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

```
clonard$ make 2>&1 | fold -sw 60
CC    -c -o History.o History.C
CC    -c -o TemplateFailure.o TemplateFailure.C
"/opt/SUNWspro/prod/include/CC/Cstd/./vector", line 207:
Error: Could not find a match for Integer::Integer() needed
in std::vector<Integer>::vector(unsigned).
"History.C", line 11:     Where: While instantiating
"std::vector<Integer>::vector(unsigned)".
"History.C", line 11:     Where: Instantiated from
History<Integer>::History(int).
"TemplateFailure.C", line 11:     Where: Instantiated from
non-template code.
1 Error(s) detected.
make: *** [TemplateFailure.o] Error 1
clonard$
```

- Bei der Übersetzung von Templates gibt es ein schwerwiegendes Problem:
  - Dort, wo die Methoden des Templates stehen (hier etwa in *History.C*), ist nicht bekannt, welche Instanzen benötigt werden.
  - Dort, wo das Template instantiiert wird (hier etwa in *Tail.C*), sind die Methodenimplementierungen des Templates unbekannt, da zwar *History.h* reinkopiert wurde, aber eben nicht *History.C*.
- 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?

- Ein trivialer Ansatz übernimmt mit Hilfe einer **#include**-Anweisung grundsätzlich auch die Methoden-Implementierung in die instantiierenden Module.
- Entsprechend müsste dann beispielsweise in *Tail.C* dem **#include** "History.h" noch ein **#include** "History.C" folgen.
- 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 leider nicht.
- Der von Sun ausgelieferte C++-Übersetzer (bei uns mit *CC* aufzurufen) folgt diesem Modell.