Hydroinformatik - SoSe 2024
HyBHW-S1-01-V8a: Pointer

Olaf Kolditz

[1]Helmholtz Centre for Environmental Research – UFZ, Leipzig
[2]Technische Universität Dresden – TUD, Dresden
[3]Center for Advanced Water Research – CAWR
[4]TUBAF-UFZ Center for Environmental Geosciences – C-EGS, Freiberg / Leipzig

Dresden, 24.05.2024

https://www.ufz.de/index.php?de=40416
https://bildungsportal.sachsen.de/opal/auth/RepositoryEntry/32518209537?10

Zeitplan: Hydroinformatik I

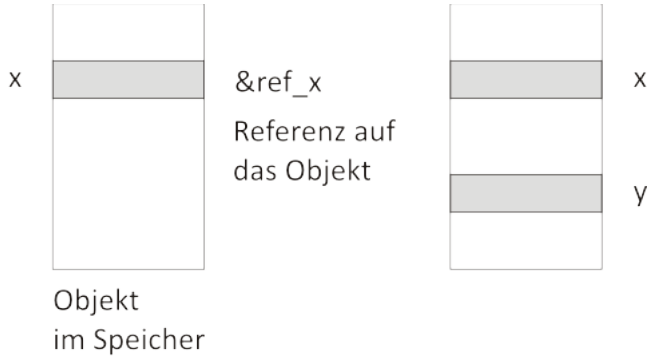
Sommersemester 2024

Nr.	KW	Datum	ID	Thema	Format
01	16	19.04.2024	HyBHW-1-01-01	Hydroinformatik - Einführung	O
02		19.04.2024	HyBHW-1-01-02	Werkzeuge (Compiler, github)	
03		19.04.2024	HyBHW-1-01-03	Jupyter, Python	
04	18	03.05.2024	HyBHW-1-01-04	Datentypen	P
05		03.05.2024	HyBHW-1-01	Installation: Compiler/Python	
06	19	10.05.2024	HyBHW-1-01-05	Klassen	O
07		10.05.2024	HyBHW-1-01-06	Input-Output (I/O)	
08	20	17.05.2024	HyBHW-1-01-07	Strings - Textverarbeitung	O
09		17.05.2024	HyBHW-1-01-08	Pointer Container	
10	22	31.05.2024	HyBHW-1-01-09	Hydrologische Modellierung	P
11		31.05.2024	HyBHW-1-01-10	BigData Water 4.0	
12	23	07.06.2024	HyBHW-1-01-11	Neuronale Netzwerke	P
13		07.06.2024	HyBHW-1-01-12	ANN / Bayes'sche Netzwerke	
14		07.06.2024	HyBHW-1-01-13	BN / Maschinelles Lernen	

Referenzen und Zeiger

Referenzen und Zeiger (pointer), d.h. die Symbole & und * sind uns bereits mehrfach begegnet, z.B. in der Parameterliste der `main()` Funktion ... Generell können Zeiger und Referenzen als einfache Variable, Parameter oder Rückgabewert (return) einer Funktion auftauchen.
In der Übung E63 haben wir bereits mit Referenzen gearbeitet und festgestellt, dass Referenzen eigentlich nichts anderes als andere Bezeichner (Alias) für eine bereits existierende Instanz eines Objekts sind. Es kann also mehrere solcher Referenzen geben. Bei der Definition eines Zeigers wird also kein neuer Speicher reserviert. Die Instanz des Objekts muss physikalisch (also speichermäßig) natürlich vorhanden sein, sonst 'verabschiedet' sich unser Programm gleich wieder.

Referenzen auf Objekte



Die Abbildung soll das Konzept des Referenzierens noch einmal verdeutlichen.

- `x` ist die Instanz eines Datentyps `T`
- `&ref_x` ist eine Referenz (Verweis) auf `x`

Übung EX08a Pointer

In der ersten Übung sehen die Definition von Referenzen und Zeigern im Quelltext

```
1 double x = 1.23456789;
2 double* ptr_x;
3 double& ref_x = x; // initialisation is needed
```

Die Bildschirmausgabe der ersten Übung zeigt Folgendes:

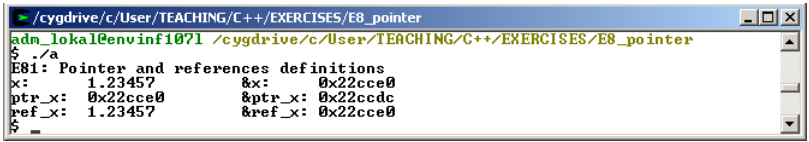


Figure: Referenzen auf Objekte

Zwischen-Summary#1

Das Verstehen und Arbeiten mit Zeigern und Referenzen ist nicht ganz einfach und braucht eine ganze Zeit der praktischen Anwendung. Der *Operator kann beispielsweise als Zeiger auf etwas und gleichzeitig für dynamische Vektoren benutzt werden. Dies scheint zunächst absolut verwirrend zu sein, beim genaueren überlegen aber: * ist lediglich die Startadresse für den Vektor. Merken sie sich erst einmal nur, dass es geschickter (und schneller) ist, nicht mit den Daten-Objekten direkt sondern mit Verweisen darauf (also deren Adressen) zu arbeiten. Die nachfolgende Tabelle und Abbildung sollen das Thema 'Referenzen und Zeiger' noch einmal illustrieren.

Zwischen-Summary#2

Syntax	Bedeutung
`double x`	Definition einer doppelt-genauen Gleitkommazahl (Speicherbedarf 8 Byte)
`double* ptr`	Zeiger auf eine doppelt-genauen Gleitkommazahl (Speicherbedarf 4 Byte)
`double& ref`	Referenz auf eine doppelt-genaue Gleitkommazahl (Speicherbedarf 8 Byte)
`double* dVektor`	Zeiger auf einen Gleitkommazahl-Vektor

Table: Zeiger und Referenzen

Nochmal eine Grafik ...

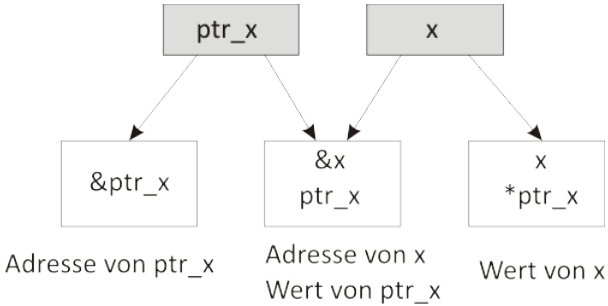


Figure: Referenzen und Zeiger

Zeiger

Zeiger sind eine der Grundideen moderner Programmiersprachen. Dabei arbeitet man nicht direkt mit den Daten-Blöcken im Speicher sondern mit deren Adressen. Dies macht insbesondere Sinn, wenn es um große Daten-Objekte geht oder bei Objekten, die erst zur Laufzeit angelegt werden (z.B. Vektoren, Listen, Strings). Das Klassen-Prinzip unterstützt die Zeigertechnik optimal, da die Klasse ihre Daten ja selber verwaltet (siehe Klassenkonstruktur) Der Zeiger (pointer) auf ein Objekt repräsentiert dessen Adresse und Daten-Typ.

Übung E72

Die nachfolgende Übung zeigt, wie wir Zeiger definieren und mit ihnen arbeiten können. Für den Zugriff auf Zeiger gibt es den sogenannten Verweisoperator '*'.

```
1 int main()
2 {
3   int i, *ptr_i; // Definitionen
4   i = 100;       // Zuweisung
5   ptr_i = &i;    // Dem Zeiger wird die Adresse der integer Variable i
6                  // zugewiesen
7   *ptr_i += 1;   // Die integer Variable i wird um Eins erhoeht (i += 1;)
8 }
```

Geben sie Werte und Adressen der Integer-Variable i und den Zeiger auf i (i_ptr) aus.

NULL Zeiger

Wir haben gesehen, dass Referenzen auf Daten-Objekte zwingenderweise initialisiert werden müssen (sonst streikt der Kompiler). Zeiger müssen eigentlich nicht initialisiert werden, ihnen wird bei der Definition eine 'vernünftige' Adresse zu gewiesen. Dies bedeutet noch lange nicht, dass sich hinter dieser Adresse etwas Vernünftiges verbirgt. Daher ist es ratsam, auch Zeiger zu initialisieren, um Nachfragen zu können, ob sie tatsächlich auf existierende Objekte zeigen. Zum Initialisieren von Pointern gibt es den NULL-Zeiger.

```
1 double* ptr_x = NULL;
2 ptr_x = &x;
3 if(!ptr_x) cout << "Warning: ptr_x does not have a meaningful adress";
```

Der linke Operand der Operators = muss immer auf eine 'sinnvolle' Speicherstelle verweisen. Dieser wird daher auch als so genannter L-Wert (L(eft) value) bezeichnet. Wenn eine Fehlermeldung 'wrong L-value' erscheint, wissen sie, dass keine 'sinnvolle' Adresse vergeben wurde, dies spricht wiederum für eine Initialisierung von Zeigern mit Zero-Pointer.

Zeiger und Arrays

Wie bereits gesagt, die Verzeigerungs-Technik ist eine der Besonderheiten von objekt-orientierten Sprachen, wie z.B. von C++. Mit solchen Pointern können wir auch größere Daten-Objekte wie Vektoren verwalten. Die zentrale Idee ist: Wenn Startpunkt (*id* Adresse) und Länge des Vektors bekannt sind, wissen wir eigentlich alles und können auf alle Vektor-Daten zugreifen.

Statische Objekte

Bei der Einführung der String-Klasse hatten wir bereits mit Zeichenketten zu tun. Aus der Sicht des Speichers ist eine Zeichenkette ein Vektor der eine bestimmte Anzahl von Zeichen enthält. Vektoren können natürlich für (fast) alle Datentypen definiert werden. Im nächsten Kapitel "Container" beschäftigen wir uns mit sogenannten Containern, damit können z.B. Vektoren, Listen etc. für ganze Klassen generiert werden.

```
1 char Zeichenkette[80];
2 int iVektor[50];
3 double dVektor[50];
```

Dynamische Objekte#1

Bisher haben wir uns fast ausschließlich mit Datentypen beschäftigt, deren Speicherbedarf bereits wöhrend der Kompilierung fest steht und reserviert wird (bis auf Strings-Objekte). Es gibt aber auch die Möglichkeit, den Speicherbedarf während der Programmausführung zu ändern - dies geht mit dynamischen Daten-Objekten. Die werden wir folgt deklariert.

```
1 char* Zeichenkette;
2 int* iVektor;
3 double* dVektor;
```

Dynamische Objekte#2

Nun müssen wir uns allerdings selber um die Speicherreservierung kümmern. Dies erfolgt mit dem `new`-Operator und wird in der nächsten Übung gezeigt. Wir ahnen schon, wenn es etwas zum Vereinbaren von Speicher gibt, muss es wohl auch das Gegenstück - freigeben von Speicher - geben. Natürlich habe sie Recht. Dies erledigt der `delete`-Operator.

```
1 double* dVektor;
2 dVektor = new double[1000];
3 delete [] dVektor;
```

Übung EX08a Arrays

Die Anwendung beider Operatoren zum Speichermanagement sowie eine Möglichkeit, den verbrauchten Speicher zu messen, werden in der folgenden Übung gezeigt.

```
#include <iostream>  // for using cout
#include <malloc.h>  // for using malloc_usable_size
using namespace std;

int main()
{
  char* Zeichenkette;             // Definitions
  long size_memory;               // Auxiliary variable for memory size
  Zeichenkette = new char[1000]; // Memory allocation
  size_memory = malloc_usable_size(Zeichenkette); // calculation of memory
  delete [] Zeichenkette;         // Memory release
  return 0;
}
```

Mehrdimensionale Objekte

Zeiger können auch verschachtelt werden: Zeiger auf Zeiger ... Damit lassen sich mehrdimensionale Datenstrukturen schaffen, z.B. Matrizen.

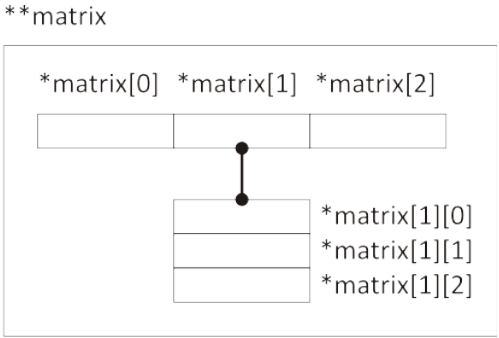


Figure: Die Definition einer Matrix mittels Pointer

Hydroinformatik - SoSe 2024
HyBHW-S1-01-V8b: Container

Olaf Kolditz

[1]Helmholtz Centre for Environmental Research – UFZ, Leipzig
[2]Technische Universität Dresden – TUD, Dresden
[3]Center for Advanced Water Research – CAWR
[4]TUBAF-UFZ Center for Environmental Geosciences – C-EGS, Freiberg / Leipzig

Dresden, 24.05.2024

https://www.ufz.de/index.php?de=40416
https://bildungsportal.sachsen.de/opal/auth/RepositoryEntry/32518209537?10

Zeitplan: Hydroinformatik I

Sommersemester 2024

Nr.	KW	Datum	ID	Thema	Format
01	16	19.04.2024	HyBHW-1-01-01	Hydroinformatik - Einführung	O
02		19.04.2024	HyBHW-1-01-02	Werkzeuge (Compiler, github)	
03		19.04.2024	HyBHW-1-01-03	Jupyter, Python	
04	18	03.05.2024	HyBHW-1-01-04	Datentypen	P
05		03.05.2024	HyBHW-1-01	Installation: Compiler/Python	
06	19	10.05.2024	HyBHW-1-01-05	Klassen	O
07		10.05.2024	HyBHW-1-01-06	Input-Output (I/O)	
08	20	17.05.2024	HyBHW-1-01-07	Strings - Textverarbeitung	O
09		17.05.2024	HyBHW-1-01-08	Pointer Container	
10	22	31.05.2024	HyBHW-1-01-09	Hydrologische Modellierung	P
11		31.05.2024	HyBHW-1-01-10	BigData Water 4.0	
12	23	07.06.2024	HyBHW-1-01-11	Neuronale Netzwerke	P
13		07.06.2024	HyBHW-1-01-12	ANN / Bayes'sche Netzwerke	
14		07.06.2024	HyBHW-1-01-13	BN / Maschinelles Lernen	

Fahrplan für heute ...

- bisher nur mit einzelnen Elementen (Zahlen, Text, Klassen) beschäftigt
- stimmt nicht ganz: Felder (Arrays) `double*` etc.
- heute: Arbeiten mit vielen Elementen

Fahrplan für heute ...

- bisher nur mit einzelnen Elementen (Zahlen, Text, Klassen) beschäftigt
- stimmt nicht ganz: Felder (Arrays) `double*` etc.
- heute: Arbeiten mit vielen Elementen

Fahrplan für heute ...

- bisher nur mit einzelnen Elementen (Zahlen, Text, Klassen) beschäftigt
- stimmt nicht ganz: Felder (Arrays) `double*` etc.
- heute: Arbeiten mit vielen Elementen

STL `std::`

Wir haben bereits mit der **String-Klasse** gesehen, dass C++ neben der Basisfunktionalität einer Programmiersprache auch sehr nützliche Erweiterungen, wie das Arbeiten mit Zeichenketten, anbietet. Diese Erweiterungen gehören zum Standard von C++, gehören also zum 'Lieferumfang' des Compilers dazu und heißen daher auch Standard-Klassen. Diese Klassen bilden die sogenannte **Standard-Bibliothek** (STL - STandard Library).

Container

In diesem Kapitel beschäftigen wir uns mit sogenannten **Containern** mit denen Daten organisiert, gespeichert und verwaltet werden können, z.B. Listen und Vektoren. In der Abbildung sind die wichtigsten Elemente der Container dargestellt. Der Begriff Container soll verdeutlichen, dass Objekte des gleichen Typs z.B. in einer Liste gespeichert werden. Und wie immer bei Klassen, werden natürlich auch die Methoden für den Zugriff oder die Bearbeitung von diesen Objekten bereitgestellt. Die Speicherverwaltung für Container-Elemente erfolgt dynamisch zur Laufzeit (siehe Kapitel Speicherverwaltung). Die Abbildung zeigt auch, welche verschiedenen **Typen** von Containern es gibt:

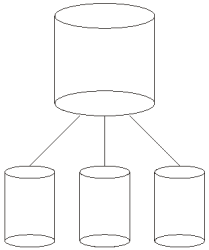
Container

(größtes Containerschiff, über 400m lang, 60 m breit, 19224 Container, ...)

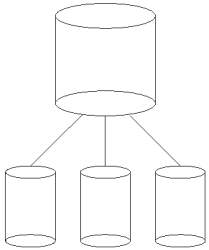
Container

Container

sequential Container



associative Container



Container

Sequentielle Container: sind z.B. Vektoren, Listen, Queues und Stacks. Vektoren sind Felder (arrays) in denen die einzelnen Elemente nacheinander angeordnet sind. Bei einer Liste kennt jedes Element nur seine Nachbarn (Vorgänger- und Nachfolgeelement). Queues (wir kennen dies von Druckern) sind Warteschlangen, die nach dem FIFO-Prinzip (first in, first out) funktionieren. Das heißt, dass zuerst eingefügte Element wird auch zuerst verarbeitet (z.B. Druckjobs). Stacks sind sogenannte Kellerstapel, die nach dem LIFO-Prinzip (last in, first out) funktionieren. Das bedeutet, das zuletzt eingefügte Element wird zuerst verarbeitet.

Assoziative Container: sind z.B. Maps (Karten) und Bitsets. Die Elemente sind nicht wie bei sequentiellen Container linear angeordnet sondern in bestimmtem Strukturen (z.B. Baumstrukturen). Dies ermöglicht einen besonders schnellen Datenzugriff, der über eine Verschlüsselung erfolgt.

Container

Die Besonderheit der C++ Container ist die **dynamische Speicherverwaltung**. Das heisst, Einfügen und Entfernen von Container-Elementen ist sehr einfach, wir brauchen uns nicht selbst um das Speichermanagement zu kümmern. Dies übernehmen die Konstruktoren und Destruktoren der jeweiligen Container-Klasse.

Sequentielle Container

Klassen-Template	Include-Files
vector<T,Allocator>	<vector>
list<T,Allocator>	<list>
stack<T,Container>	<stack>
queue<T,Container>	<queue>

Table: Klassentemplates für sequentielle Container

Container

Ein Klassentemplate ist nichts weiter als eine formale Beschreibung der Syntax.

- vector: Schlüsselwort
- $<$ ... $>$: Parameterliste
- T: erster Parameter, Datentyp des Vektors
- Allocator: zweiter Parameter, Speichermodell des Vektors

Container

Die Methoden sequentieller Container sind sehr einfach, intuitiv zu bedienen und sie sind gleich für Vektoren, Listen etc. Ein Auswahl der wichtigsten gemeinsamen Methoden finden sie in der nachfolgenden Tabelle.

Methode	Bedeutung
`size()`	Anzahl der Elemente, Länge des Containers
`push_back(T)`	Einfügen eines Elements am Ende
`pop_back()`	Löschen des letzten Elements
`insert(p,T)`	Einfügen eines Elements vor p
`erase(p)`	Löschen des p-Elements (an der Stelle p)
`clear()`	Löscht alle Elemente
`resize(n)`	Ändern der Containerlänge

Table: Methoden für sequentielle Container

Vektoren

The standard vector is a template defined in `namespace std` and presented in `<vector>`. The vector container is introduced in stages: member types, iterators, element access, constructors, list operations etc. The structure of a vector is illustrated in Fig. 1.

Figure: Structure of vectors

Vektoren

Following lines are necessary in order to use the vector container in your sources code.

```
1 #include <vector>
2 using namespace std;
```

Vektoren

Types: Very different types can be used in vectors. A vector e.g. of integers can be declared in this way.

```
1 // Declaration and construction
2   vector<int> v;
```

Vektoren

Stack operators: The functions `push_back` and `pop_back` treat the vector as a stack by adding and removing elements from its end.

```
// Input elements to vector
  for(i=0;i<9;i++)
  {
    v.push_back(i+1);
  }
```

Container

Iterators: can be used to navigate containers without the programmer having to know the actual type to identify the elements, e.g. for inserting and deleting elements. A few number of key functions allow to step through the elements, such as `begin()` pointing to first element and `nd() ~ pointing to th` one-past-last element. An example of a loop through all vector elements using the iterator is given below.

```
1 // Navigating vector elements
2   for(vector<int>::iterator it=v.begin(); it!=v.end(); ++it)
3   {
4     cout << *it << '\n';
5   }
```

Vektoren: Zugriff

Vector elements can be addressed directly by element number. Fast data access is an advantage of vectors.

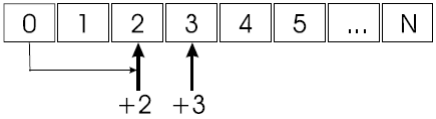


Figure: Access to vector elements

Vektoren: Elemente löschen

In contrast, vectors are not as flexible as lists. Deleting and adding of elements within the vector is complicated (Fig. 3).

Figure: Deleting elements

Vektoren: Übungen

- EX08b-vector: Vektoren anlegen
- EX08b-data-base: Verwalten von Datenbank-Objekten (Studenten-Klasse)

Vektoren: Übung EX08b

```
...
#include <vector>    // for using vectors
using namespace std; // for std functions

int main()
{...
  vector<long>my_vector;
  //-----------------------------------------------------------------
  cout << "Current vector size is: " << my_vector.size() << endl;
  for(long i=1;i<101;i++)
  {
     my_vector.push_back(i*i*i*i);
  }
  cout << "Current vector size is: " << my_vector.size() << endl;
  //-----------------------------------------------------------------
  for(long i=0;i<(long)my_vector.size();i++)
  {
    cout << my_vector[i] << endl;
  }
  //-----------------------------------------------------------------
  my_vector.clear();
  cout << "Current vector size is: " << my_vector.size() << endl;
...}
```

Vektoren

In der nächsten Übung werden einige Dinge, die wir bisher gelernt haben, zusammenbringen. Sie werden sehen, dass 'gut' programmierte Dinge einfach 'zusammengebaut' werden können: Objekte, IO-Methoden, Stringverarbeitung und natürlich Container.

Vektoren

```
1 #include <iostream>  // for using cout
2 #include <fstream>   // for using ifstream / ofstream
3 #include <string>    // for using string
4 #include <vector>    // for using vectors
5 #include "student.h" // for using CStudents
6 using namespace std; // for std functions
```

Listen

A list is a sequence optimized for insertion and deletion of elements. They allow a very flexible organization of elements. But the price for flexibility is a relatively slow access to data.
List provides bidirectional iterators. This implies that a STL list will typically be implemented using some form of a doubly-linked list.

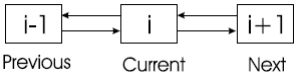


Figure: Structure of lists

Listen

Following lines are necessary in order to use the list container in your sources code.

```
1 #include <list>
2 using namespace std;
```

Arbeiten mit Listen

- Übung E83
- Studenten-Datenbank lesen: Übung EX08b-data-base/E84
- Warum: Erweiterung der Übung (Verknüpfen von Vektoren und Listen-Funktionalitäten)

Ende des Programmierteils (C++)