Institute of Automation

C++ Basics and Applications in technical Systems

Lecture 4 - C++ Pointer

Institute of Automation
University of Bremen

16th November 2012 / Bremen

WiSe 2012/2013

VAK 01-036

Universität Bremen

Institute of Automation

Overview

1. Organization
2. Repetition
3. Technical background
   - Memory organization
   - Pointer and addresses
4. Compile-time specification
   - C-Arrays (fixed)
   - Pointer arithmetic
   - C-Strings
5. Run-time allocation / deallocation
   - Dynamic allocation / deallocation
   - C-Arrays (dynamic)
   - Parameter passing via pointer

Universität Bremen

Lecture schedule

Institute of Automation

Time schedule

- HK **26. Oct.** - Introduction / Simple Program / Datatypes ...
- HK **02. Nov.** - Flow control / User-Defined Data types ...
- CF **09. Nov.** - Simple IO / Functions/ Modular Design ...
- CF **16. Nov.** - C++ Pointer
- CF **23. Nov.** - Object oriented Programming / Constructors
- AL **30. Nov.** - UML / Inheritance / Design principles
- AL **07. Dec.** - Namespace / Operators
- AL **14. Dec.** - Polymorphism / Template Classes / Exceptions
- HK **11. Jan.** - Design pattern examples

Universität Bremen

Important dates

Institute of Automation

Submission of exercises

1-3 **16. Nov.** - Deadline for submission of Exercise I, 13:00

4-6 **07. Dec.** - Deadline for submission of Exercise II, 13:00

For admission to final exam you need at least 50% of every exercise sheet.

Final project

1-9 **15. Feb.** - Deadline for submission of final project, 13:00

Final exam

1-9 **06. Feb.** - Final exam, 10:00-12:00, H3

Universität Bremen

Character streams

Institute of Automation

Stream data-types

```
std::cout
std::cerr
std::endl
std::cin
```

Header file

```
#include <iostream>
```

Declaration

```
std::cout << "Hello World!"<< std::endl;
std::err << "There was an error..."<< std::endl;
std::cin >> nValue;
```

Universität Bremen

Character file-streams

Institute of Automation

Stream data-types

```
std::ifstream
std::ofstream
```

Header file

```
#include <fstream>
```

Usage, operators and methods

- open a file: `newFile.open("file.txt");`
- close a file: `newFile.close();`
- input: `newFile << "Text";`
- output: `newFile >> sLine;`
- read single character: `char cChar = newFile.get();`
- write single character: `newFile.put(cChar);`

Bremen

Streams

Institute of Automation

- >> ensures that the necessary reformatting is performed automatically
- leading space characters (e.g. whitespaces, tabulator '\t' or line interlacing '\v') are ignored
- space characters represent end identifier
- other characters are interpreted according to the required target data-type

To not ignore space characters use:

```
char cInput;
std::cin.get(cInput);
```

Universität Bremen

Modular design of applications

Institute of Automation

Definition

- Separation into header-file (*.h) and implementation-file (*.cpp)
- One header-file and one implementation-file form a module
- Creation of one main() function that has access to the remaining modules
- Principle for separation into modules:
  - Reuseability
  - Connection
  - Reduction of complexity

Universität Bremen

Contents of header and implementation-file

Institute of Automation

*.h and *.cpp

- Declarations, constants, user-defined types, ...
- Definitions
- Source documentation

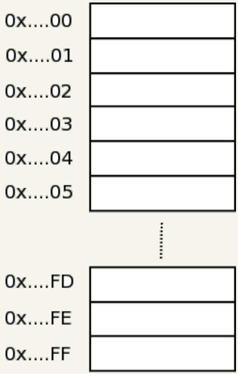
Example

```
// file "myHeader.h"
#ifndef MY_HEADER_H
#define MY_HEADER_H

int MyMax(int iNumber1,
          int iNumber2);

#endif // MY_HEADER_H
```

```
#include "myHeader.h"
int MyMax(int iNumber1,
          int iNumber2)
{
  int iMax;
  iMax = iNumber1 < iNumber2
   ? iNumber2 : iNumber1;
  return iMax;
}
```

Linear memory organization

Institute of Automation



some basics

- The system memory is organized in a linear form
- Each memory cell has an unique address
- To read/write values from/to the memory the address is used to specify the location in the memory

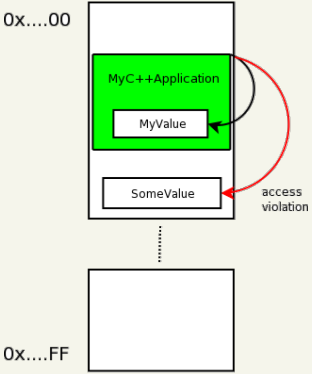
A memory address **points** to a location in the system memory.

Universität Bremen

Applications in system memory

Institute of Automation

- During startup of a compiled C++ application the operating system loads it into the system memory and reserves additional memory for all used variables
- The application is only allowed to access that part of the memory that was reserved for it by the operating system
- If an application tries to access other parts of memory an access violation occurs (segmentation fault)



Universität Bremen

Institute of Automation

Summarization

Summarization

- Addresses of cells in the system memory can be interpreted as pointer
- An application is not allowed to access memory that was not reserved for it by the operating system

Universität Bremen

Pointer in C++

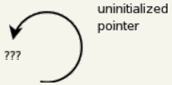
Definition

The value of a pointer variable is an address of a specific memory cell.

Declaration

```
int * pPointerToInteger;
char * pPointerToChar;
float * pPointerToFloat;
```

Attention: The pointer value is not initialized after declaration!



Pitfall using C++ pointers

Institute of Automation

Working with uninitialized pointers leads often to segmentation faults! Therefore:

Important

Initialize each pointer directly after declaration!

Declaration

Declaration with initialization:

```
char * pPointer = NULL;       // better to read
char * pPointer = 0;          // typesafe
```

Both values are possible as initial value for pointers.

Universität Bremen

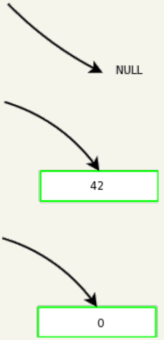
Assignment

Institute of Automation

```
int iValue = 42;               // int value
int * pPointer = NULL;    // pointer to int
```

```
pPointer = &iValue;    // let pointer point
                            // to int value
                       // '&' gives address
                            // of the value
```

```
*pPointer = 0;          // change the value
                  // the pointer points to
```



Universität Bremen

Institute of Automation

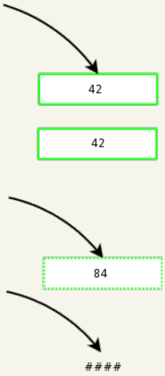
Pointer access

```
int iValue = 42;
int * pIntPointer = &iValue;
```

```
int iValue2 = *pIntPointer;   // Assignment
                   // by means of the pointer
```

```
{
  int iValue3 = 84;
  pIntPointer = &iValue3;
}                                  // end of block
                            // => iValue3 invalid
```

```
*pIntPointer = 8; // invalid memory access
```



Be careful using pointers to objects with restricted range of validity!

Universität Bremen

Type check

Institute of Automation

```
char * pPointerC;                    // pointer to char value
void * pPointerV;               // pointer to unspecified type
```

```
pPointerV = pPointerC;                              // possible
pPointerC = pPointerV;                     // impossible, error
                                       // message from compiler
```

```
pPointerC = static_cast<char *>(pPointerV);
               // possible, but should be used carefully, because
          // the type checking of the compiler is short-circuited
```

Definition

A `void`-pointer is a pointer to an unspecified data-type. It can be used to point to any object in the system memory.

Universität Bremen

Constant pointers on constant values

Institute of Automation

Pointer on constant character

```
const char * pChar;           // character fixed
```

Constant pointer on character

```
char const * pChar;            // pointer fixed
```

Constant pointer on constant character

```
const char const * pChar;         // both fixed
```

Universität Bremen

C-Array overview

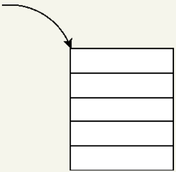
Institute of Automation

Declaration

```
type NameOfArray[FixedNumberOfElements];
```

Example

```
float table1[5];

const int iNUMBER = 5;
int table2[iNUMBER];
```



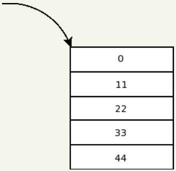
Universität Bremen

C-Array assignment and access

Institute of Automation

Assignment

```
int table1[5];
table1[0] = 0;
table1[1] = 11;
table1[2] = 22;
table1[3] = 33;
table1[4] = 44;
int table2[5] = {0, 11, 22, 33, 44};
```



Access by pointer

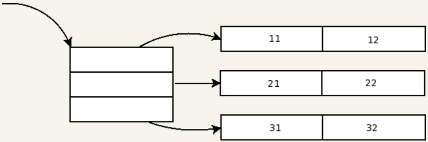
```
int iValue = table1[4];             // => iValue = 44
table1[0] = -1;
int * pPointer = table1;        // => pPointer = table1
*(pPointer + 4) = 1271;          // => table1[4] = 1271
```

Universität Bremen

Multidimensional C-Arrays

Institute of Automation

e.g. 3*x*2 Matrix




```
const int iROWS = 3;
const int iCOLUMNS = 2;
int matrix[iROWS][iCOLUMNS] = {{11, 12}, {21, 22}, {31, 32}};
```

```
for (int iI = 0; iI < iROWS; iI++) {
  for (int iJ = 0; iJ < iCOLUMNS; iJ++) {
```

```
    std::cout << matrix[iI][iJ] << std::endl;
```

```
    std::cout << *(*(matrix + iI) + iJ) << std::endl;
```

```
  }
}
```

Universität Bremen

Accessing elements of C-Arrays using method

Institute of Automation

Problem

```
void OutputTable(int table[]) {
  int iNumberBytes = sizeof(table);     // ERROR, sizeof returns
                                              // size of pointer
  int iElements = iNumberBytes
                  / sizeof(int);         // -> calculation wrong

  for (int iI = 0; iI < iElements; iI++)
    cout << table[iI] << endl;
}
```

```
int main() {
  const int iNUMBER = 5;
  int table[iNUMBER];
  OutputTable(table);
}
```

It is not possible to get the size of a C-Array!Don’t use *sizeof*

Universität Bremen

Accessing elements of C-Arrays using method

Institute of Automation

Possible Solution

Example

```
void OutputTable(int table[], int iNumber) {
  for (int iI = 0; iI < iNumber; iI++)
    cout << table[iI] << endl;
}
```

```
int main() {
  const int iNUMBER = 5;
  int table[iNUMBER];
  OutputTable(table, iNUMBER);
}
```

Pass the size of a C-Array if you need it within a function.

Universität Bremen

C-Array - exercise

Institute of Automation

- Create an application that prompts the user to input 5 numbers.
- Store this numbers into a C-Array.
- Use the type double for the numbers.
- Calculate and print the sum and the mean of the numbers.

Universität Bremen

Arithmetic operations on pointers

Institute of Automation

Definition

The term pointer arithmetic is used if arithmetic operations (e.g. + or -) are performed on pointer variables.

For all operations the pointer value is interpreted depending on the type it points to:

Example

```
int pValues[2] = {11, 22};
int * pValue = pValues;                  // *pValue -> 11
pValue++;                           // Increments address in
                                 // pValue by sizeof(int) bytes
                                         // *pValue -> 22
```

Universität Bremen

More examples on pointer arithmetic

Institute of Automation

```
double dValues[5] = {0, 10, 20, 30, 40};
```

```
double *pField1 = dValues;
double *pField2 = pField1 + 1;                    // Points to the
                                                 // address of the
                                                    // next value
```

```
std::cout << pField2 - pField1 << std::endl;        // Output = 1
```

```
std::cout << reinterpret_cast<long>(pField2) -
             reinterpret_cast<long>(pField1) << std::endl;
                                        // Output = 8 => 8 bytes
                            // after casting the calculation is
                               // done with the memory address
```

Universität Bremen

C-string declaration

Institute of Automation

Definition

A C-string is a sequence of characters from data-type `char` terminated with `'\0'` (ASCII-character with value 0)

Declaration

```
char * pString1;          // Pointer to the beginning
                          // of the string
```

Example

```
char * pString2 = "C++ lecture";
```

Usage examples

Institute of Automation

Assume the following declaration:

```
const char * pStr = "ABC";
```

Access to single character

```
std::cout << pStr[0];          // output of first character 'A'
std::cout << pStr[iI];                                    // or
std::cout << *(pStr + iI);      // output of (iI + 1) character
```

C-Strings are write protected

```
pStr[0] = 'X';                              // runtime error!
pStr = "New";                   // new assignment is possible
                          // (address points to new C-String)
```

Universität Bremen

Basic idea

- Operator `new` and `delete` allocate / deallocate memory at runtime
- User-defined scope of validity
- `new`-operator automatically allocates amount of memory according to requested data type

Example

```
double * pValue = new double;
*pValue = 42.0;
```

Memory allocation during runtime

Institute of Automation

A small example:

```
int iNumber;
std::cout << "Give number of elements: ";
std::cin >> iNumber;
```

```
int * pValues = NULL;          // Declaration and initialization
                                                   // of pointer
```

```
pValues = new int[iNumber];                  // Array during runtime
```

```
pValues[0] = 1;                          // assignment of 1st value
pValues[1] = 27;                         // assignment of 2nd value
...
pValues[iNumber - 1] = 42;              // assignment of last value
```

Universität Bremen

Dynamically created structure

Structures can easily be used to store data.

```
struct Test_T {                                   // Declaration of struct
  int m_iNumber;
  double m_dNumber;
};
```

```
Test_T * pStruct = new Test_T;                   // Create structure during
                                                                // run-time
```

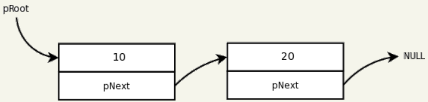
```
pStruct->m_iNumber = 1;                              // Value assignment with
pStruct->m_dNumber = 1.1;                              // arrow-operator '->'
```

```
(*pStruct).m_iNumber = 2;                            // Value assignment with
(*pStruct).m_dNumber = 2.2;                              // dot-operator '.'
```

Structures for chained lists / trees

Institute of Automation

```
struct Node_T
{
  int m_iValue;
  Node_T * m_pNext;
};
```




```
// 1. Create root "object"
Node_T * pRoot = new Node_T;

// 2. Initialize elements
pRoot->m_iValue = 10;
pRoot->m_pNext = NULL;
```

```
// 3. Create second "object"
Node_T * pTemp = new Node_T;
pRoot->m_pNext = pTemp;

// 4. Initialize elements
pTemp->m_iValue = 20;
pTemp->m_pNext = NULL;
```

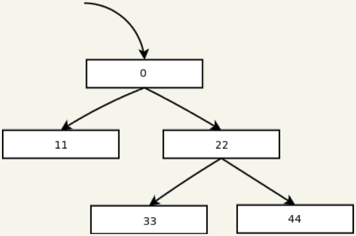
Universität Bremen

Chained tree exercise

Institute of Automation

Create an application that builds a dynamically allocated tree with five nodes. Use the following structure for the nodes:

```
struct Node_T
{
  double m_dValue;
  Node_T * m_pNext1;
  Node_T * m_pNext2;
};
```



Universität Bremen

Deallocation of memory

Institute of Automation

Memory for allocated objects has to be deallocated after usage:

```
int * pValue1 = new int;                         // Pointer to int
...                                                  // usage ...
delete pValue1;                              // Deallocate memory
pValue1 = NULL;                    // Reinitialize dangling pointer
```

```
int iNumber;
std::cin >> iNumber;
int * pValue2 = new int[iNumber];      // Create array on run-time
...                                                  // usage ...
delete [] pValue2;       // Deletes the array (deallocates memory)
pValue2 = NULL;                    // Reinitialize dangling pointer
```

Universität Bremen

Rules to remember

```
int * pInt = new int;                                  // Create int-pointer
int ** pA1 = new int*[2];                        // Create int-pointer array
                                                     // (for two pointers)
pA1[0] = pInt;
pA1[1] = NULL;
```

```
delete pA1[0];                                // equivalent to delete pInt
delete pA1[1];                                  // ok! (it points to NULL)
delete pInt;                                     // Error, already deleted
delete [] pA1;                       // ok! (deallocate memory for pointers)
```

Rules

- `delete` may be applied only once to an object.
- `delete` applied to a NULL-Pointer doesn't have an effect
- The release of arrays requires the specification of the square brackets, otherwise only the first item is released.

Rules to remember - range of validity

Institute of Automation

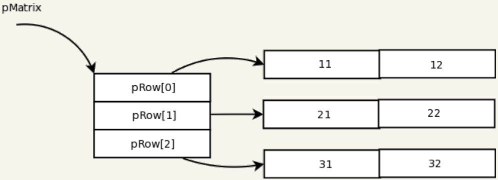
Objects created with the `new`-statement are not subject to the range of validity rules of variables. They exist until they are deleted with `delete`.

```
{
  int * pValue = new int;
  *pValue = 2;
}
```

```
                  // pValue is not accessible outside the block!
                              // => here delete is impossible
                                       // => memory leak
```

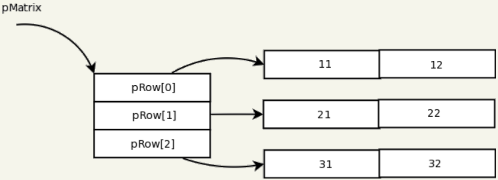
Universität Bremen

Creation of multi-dimensional arrays




```
int iRows = 3;
int iColumns = 2;

// Allocate memory for pointers to rows
int ** ppMatrix = new int*[iRows];

// Allocate memory for each row
for (int iI = 0; iI < iRows; iI++) {
  int * pRow = new int[iColumns];
  ppMatrix[iI] = pRow;
}
```

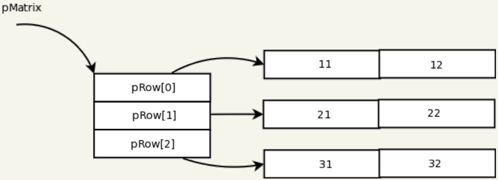
Initialization of multi-dimensional arrays




```
// Example to fill a matrix
for (int iI = 0; iI < iRows; iI++) {
  for (int iJ = 0; iJ < iColumns; iJ++) {
    ppMatrix[iI][iJ]= (iI + 1) * 10 + iJ + 1;
    std::cout << ppMatrix[iI][iJ] << std::endl;
  }
}
```

Deallocation of multi-dimensional arrays

Institute of Automation




```
// Deallocate rows first
for (int iI = 0; iI < iRows; iI++) {
  delete [] ppMatrix[iI];
}

// Deallocate array of int pointer
delete [] ppMatrix;
```

Pointer as function argument

Institute of Automation

- Change a passed object by passing a pointer to the object.
- The function works with a copy of the pointer, which points to the same object as the original pointer.
- Modification of the object the pointer refers to, no modification of the pointer itself.

```
void Times2(int * pTemp);

int main ()
{
  int * pValue = new int;
  *pValue = 42; // access here
  std::cout << *pValue << std::endl;
  Times2(pValue);
  std::cout << *pValue << std::endl;
  return 0;
}

void Times2(int * pTemp)
{
  *pTemp *= 2; // access here
}
```

Universität Bremen

Pitfall: Pointer to local object

When returning pointers, it has to be assured that they do not refer to local objects, which disappear after the function call. (Similar to return of references")

```
int * Function() {
  int iX = 123;
  return &iX; // Error! Return of the address of a local variable
                      // that only exists during the function call
}
```

```
void main() {
  int * pPointer = Function();                  // Function call
  std::cout << *pPointer
            << std::endl;          // Error! Non-existing object
}
```

Rules for pointers as function argument

Institute of Automation

If returning an dereferenced object, the object is copied and the original is no longer attainable for a `delete`.

- No objects should be returned that have been produced with the `new`-operator within a function.
- Thus only a pointer to the object may be returned.
- The user has the responsibility to delete the object outside of the function

```
int BadFunction() {
  int * pValue = new int;
  *pValue = 42;
  return *pValue;
}

int * GoodFunction() {
  int * pValue = new int;
  *pValue = 42;
  return pValue;
}
```

Universität Bremen

Final exercise

Enhance the last exercise the following way:

- Create a function that takes the root pointer of your tree and prints the values of all nodes to the screen.
- Create a function that uses the root pointer to search for a specific element within the tree.