C++ Basics and Applications in technical Systems

Lecture 3 - Simple IO, functions and modular design

Institute of Automation
University of Bremen

09th November 2012 / Bremen

WiSe 2012/2013

VAK 01-036

Overview

1. Organization
2. Repetition
3. Simple input and output
   - IO to screen
   - IO to files
4. Functions and arguments
   - Function structure and declaration
   - Parameter passing and overloading
   - Specification and the main-function
5. Modular design principles
   - Compiler directives and macros
   - Structure of source-files
   - Visibility and validity between modules
6. Misc
   - Assert, function templates and inline-functions
7. Exercise

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

if, else - Statement

If-Statement

Statement is executed if booleanExpression is true:

```
if (booleanExpression)
  Statement;
```

If-Else-Statement

Statement1 is executed if booleanExpression is true, otherwise Statement2:

```
if (booleanExpression)
  Statement1;
else
  Statement2;
```

If-Else-Statement with blocks

Statement1 and Statement2 are executed if booleanExpression is true, otherwise Statement3:

```
if (booleanExpression)
{
  Statement1;
  Statement2;
}
else
{
  Statement3;
}
```

Case selection with switch

- expression is evaluated, the result has to be of type integer or char
- constValueX is compared to the result of expression; if equal: statements are executed
- break has to be used to finish a case; without break the execution continues
- the statements after the label default are executed if no case fits the result of the evaluated expression

Example

```
switch (expression)
{
case constValue1:
  Statements1;
  break;

case constValue2:
  Statements2;
  break;

default:
  Statements;
}
```

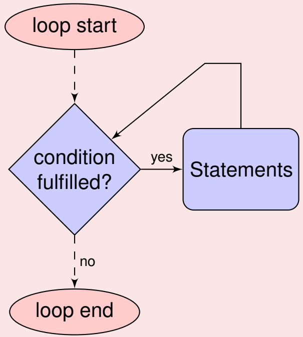
While-Do

The condition is checked before the first execution of the statements.

Example

```
while (condition)
{
  Statements;
}
```

Flow chart



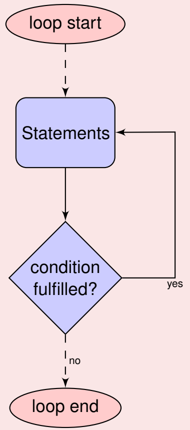
Do-While

The statements are executed ones before the first condition check is performed.

Example

```
do
{
  Statements;
} while (condition);
```

Flow chart



Loops with for

Structure

```
for (Initialization; Condition; Modification)
{
  Statements;
}
```

Example

```
const unsigned int iLIMIT = 1000;
double dArray[iLIMIT];
for (int nI = 0; nI < iLIMIT; nI++)
{
  std::cout << "Value [" << nI << "] ?";
  std::cin >> dArray[nI];
}
```

Character streams

Stream data-types

```
cout

cerr

endl

cin
```

Header file

```
#include <iostream>
```

Declaration

```
std::cout << "Hello World!"<< std::endl;

std::cerr << "There was an error..."<< std::endl;

std::cin >> nValue;
```

Input streams

- >> 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);
```

Input examples

Example

- without space characters

```
std::string sName;
std::cin >> sName;                    // Input: "Donald Duck"
std::cout << sName;                        // Output: "Donald"
```

- with space characters

```
std::string sName;
std::getline(cin, sName);             // Input: "Donald Duck"
std::out << sName;                   // Output: "Donald Duck"
```

Per default getline expects `'\n'` as line delimiter. If there is a delimiter in cin from the last input, it must be cleared with `cin.ignore(numeric_limits<streamsize>::max(),'\n');` first.

Correctly reading numbers (1)

Institute of Automation

Example

- By checking the fail bit of cin

```
#include<limits> //for numeric_limits
int iNumber;
std::cout << "Please enter an integer ";
while(!(std::cin >> iNumber)) {
  std::cin.clear();
  std::cin.ignore(std::numeric_limits<
    std::streamsize>::max(),’\n’);
  std::cout << "Please enter an integer ";
}
```

Another posibillity is to check cin.fail() for an erroneous input.

Universität Bremen

Correctly reading numbers (2)

Example

- By reading a string and converting using atoi

```
#include <stdlib.h> //for atoi
std::string sNumber;
int iNumber;
bool isValid = false;
do {
  std::cout << "Please enter an integer ";
  std::getline(cin, sNumber);
  if (iNumber = atoi(sNumber.c_str()))
    isValid = true;
} while (!isValid);
```

For conversion from number to string `itoa` can be used (caution: only on Windows). Alternatively use string streams.

Output examples

Example

- the operator << transforms the internal representation automatically into a textual representation with the necessary range, like:

```
std::cout << 7 << 11;                     // Output: "711"
```

- formatting is possible (e.g. switching to boolean output or completion with white spaces):

```
std::cout << 7;
std::cout.width(6);
std::cout << 11;                      // Output: "7    11"
```

Character file-streams

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: `unsigned char cChar = newFile.get();`
- write single character: `newFile.put(cChar);`

File-streams examples

Institute of Automation

Example

- open a file for writing (output)

```
std::ofstream outputFile;
outputFile.open("/usr/share/test.txt");
outputFile << "Letter: ";
outputFile.put('A'); // characters must be in single quotes
outputFile.close();
```

- open a file for reading (input)

```
std::ifstream inputFile;
std::string sPrefix;
inputFile.open("/usr/share/test.txt");
inputFile >> sPrefix;
char cLetter = inputFile.get();
inputFile.close();
```

Universität Bremen

Additional hints working with files I

Example

- Check if file was opened successfully

```
outputFile.open(...);
if (outputFile) {
  ...
}
else {
  std::cout << "Error opening file!" << std::endl;
}
```

- Read and write a binary file

```
// for ofstream
outputFile.open("myFile.txt",
                std::ios::binary | std::ios::out);
// for ifstream
inputFile.open("Name of File",
              std::ios::binary | std::ios::in);
```

Additional hints working with files II

Institute of Automation

Example

- Read until end of file

```
char cChar;
...
while (inputFile.get(cChar)) {
  ...
}
```

- Open file by means of a string variable

```
std::string sFileName;
...
inputFile.open(sFileName.c_str());
```

Universität Bremen

Example: print file contents to screen

```
#include <iostream>
#include <fstream>
int main() {
  char cInput;
  std::ifstream sourceFile;                    // file definition
```

```
  sourceFile.open("Test.txt");                       // open file
```

```
  if (sourceFile.is_open()) {                // check for success
```

```
    while (sourceFile.get(cInput)) {    // character wise reading
      std::cout << cInput;
    }
```

```
  }
```

```
  return 0;
}
```

Small exercise

Input and output with files

Create a program that writes the user’s input into a file. The input should be continued until the user enters ’X’. Implement the following functionalities:

- Create an empty text-file with an editor first (e.g. Kate)
- Open the existing file and write typed input to it
- Use a simple menu to control the usage of your program

Arguments and return-value (signature)

Syntax of declaration

```
type-of-return-value FunctionName(formalParameters);
```

Example

```
int Max(int iNumber1, int iNumber2);
```

Syntax of function call

```
FunctionName(currentParameters);
```

Example

```
int iInput;
std::cin >> iInput;
int iMax = Max(iInput, 42);
```

Definition

Syntax of definition

```
type-of-return-value FunctionName(formalParameters)
{
  ...
}
```

Example

```
int Max(int iNumber1, int iNumber2)
{
  int iMaxValue;
  iMaxValue = iNumber1 < iNumber2 ? iNumber2 : iNumber1;
  return iMaxValue;
}
```

Range of validity and visibility within functions

```cpp
int iNumber1, iNumber3;                       // Global variables
```

```cpp
int Max(int iNumber1, int iNumber2)           // Formal parameter
{
  int iMaxValue;                                // Local variable
  iMaxValue = iNumber1 < iNumber2 ? iNumber2 : iNumber1;
  return iMaxValue;
}
```

```cpp
int main( )
{
  int iNumber2;                                 // Local variable
  std::cin >> iNumber1 >> iNumber3;   // Input in global Variable
  iNumber2 = Max(iNumber3, iNumber1);         // actual parameter
  std::cout << std::endl << "The maximum is:" << iNumber2;
  return 0;
}
```

Functions with memory

Static variable inside method

Static variables exist in the memory before a method/function is called. They are initialized once and they are valid during the program execution.

```
#include <iostream>
void MemoryFunction()
{

  static int iMinBrain = -1;    // static variable stays valid in
                                  // function during full runtime

  std::cout << ++iMinBrain << std::endl;
}

int main()
{
  for (int i = 0; i < 10; i++)
  {
    MemoryFunction();                 // Output: 0 1 2 3 4 ... 9
  }
```

Recursive functions

Recursive functions

A recursive function calls itself inside its body. Usualy these functions have parameters that change with each recursive call. They can for example specify, when the recursion ends.

Example

```
unsigned int Faculty( unsigned int Number )
{
    if ( Number < 2 )
    {
        return 1;
    }
    else
    {
        return ( Number * Faculty( Number - 1 ) );
    }
}
```

Interfaces for data transfer

Definition

- Unique description by means of signature given in declaration
- Two different types of parameters
  - **Per value** - Copy of parameter on stack. The value of the passed variable will not be changed within the function.
  - **Per reference** - Direct access to passed variable. Value of passed parameter can be changed within function.
  - **Per pointer** - Pointer access to passed variable. Value of passed parameter can be changed within function. Will be explained in later lecture.

Parameter passing per value

Example

```cpp
#include <iostream>
int Faculty(int iNumber)
{
  int iResult = 1;
  while (iNumber > 0)
  {
    iResult *= iNumber--;
  }
  return iResult;
}
```

```cpp
int main()
{
  int iN;
  int iFac;
  std::cout << "Input number: ";
  std::cin >> iN;
  iFac = Faculty(iN);
  std::cout << "Faculty of "
            << iN << " is "
            << iFac
            << std::endl;
  return 0;
}
```

Parameter passing per reference

Institute of Automation

Example

```cpp
#include <iostream>
void Faculty(int iNumber,
             int & iFac)
{
  iFac = 1;
  while (iNumber > 0)
  {
    iFac *= iNumber--;
  }
}
```

```cpp
int main()
{
  int iN;
  int iFac;
  std::cout << "Input number: ";
  std::cin >> iN;
  Faculty(iN, iFac);
  std::cout << "Faculty of "
            << iN << " is "
            << iFac << endl;
  return 0;
}
```

Universität Bremen

Institute of Automation

Predetermined parameter values (default-values)

Definition

Declaration with default value:

```
void OpenComPort(int iComNr, int iBaudrate = 9600);
```

Example

```
int main()
{
  OpenComPort(1);
  .....
}
```

```
int main()
{
  OpenComPort(1, 38400);
  .....
}
```

Universität Bremen

Overloading of functions

Definition

Two or more declared functions with the same signature, except different parameters, are called overloaded functions.

```
int iNumber1, iNumber2;
std::string sName1, sName2;
```

```
// Declarations:
bool Equal(int iA, int iB);
bool Equal(std::string sStr1, std::string sStr2);
```

```
// Function calls (e.g. within main):
if (Equal(sName1, sName2)) ...
if (Equal(iNumber1, iNumber2)) ...
if (Equal(iNumber1, sName2)) ...                 // Error, no fitting
                                                // signature declared
```

Specification of functions (documentation purpose)

Definition

To ease the usage of your functions do not forget to create an appropriate specification for each function upon declaration.

```
// --------------------------------------------------------------
// Preconditions:  Which preconditions have to be
//                 fulfilled, so that the function can work
//                 correctly (e.g. the allowed parameter range)?
// Postconditions: What are the return values? What is the
//                 range of the returned parameters?
// Semantic:       The meaning of the function?
// --------------------------------------------------------------
int Max(int iNumber1,                            // First Number
        int iNumber2);                          // Second Number
// --------------------------------------------------------------
```

Passing data to your main()-function

Definition

Each application is able to work on data that was passed during startup (via command line arguments).

Example

```
int main(int argc, char ** argv)
{
  std::cout << "Number of passed values " << argc << std::endl;
  std::cout << "Name of application " << argv[0] << std::endl;
}
```

```
user@host$ ./myApplication FirstArg ...
```

More on the special meaning of `char ** argv` and `argv[0]` in the next lecture.

Modular design of applications

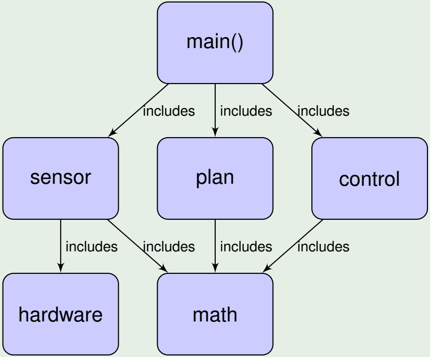
Definition

- 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

Example: Module structure tree

- One main()-module uses different sub-modules
- Sub-modules also use some different or common sub-modules
- The usage depth is not limited

Example



Prevention of multiple header inclusion

Problem

If one does not prevent the compiler to include header-files multiple time (this is possible in structures with an include-depth bigger than 2) unexpected compile problems will be the result.

Example

Add the following preprocessor statements to all of your header files to prevent multiple inclusion:

```
// MyFile.h
#ifndef MY_FILE_H
#define MY_FILE_H
... your code ...
#endif // MY_FILE_H
```

Inclusion of header files

Differentiate between system-headers and own-headers

As already mentioned, external modules are included using the #include-statement. Avoid absolute path names and use #include "" for non-system headers.

Example

```
#include <SystemHeader.h>
#include <SystemHeader2>
#include "MyHeader1.h"
#include "../MyHeader2.h"
#include "/home/C++/MyHeader3.h"      // No abolute path names!
                                      // Very hard to maintain!
```

Contents of header-file

*.h

- Declarations, constants, user-defined types that will be used within other modules
- Do not place definitions in header-files!

Example

```
// Example for header
// 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 main()
{
  int iMax, iZ1, iZ2;
  ...
  iMax = MyMax(iZ1, iZ2);
}
```

Contents of implementation-file

*.cpp

- Definitions
- Source documentation

Example

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

Keywords static and extern

Example

```
// Main program
int igGlobal;
static int iMLocal;

int main()
{
  ...
  return 0;
}
```

Example

```
// Module 1
extern int igGlobal;
int func1()
{
  igGlobal = 5;
}
```

Example

```
// Module 2
extern int iMLocal;
int func2()
{
  iMLocal = 5; // Error
}
```

„One-Definition“-rule

Definition

Each variable, function, structure, constant etc. in a program has exactly one definition!

Things to remember:

- A pure declaration introduces a name to a program and gives a meaning to the name.
- A definition is also responsible for the reservation of storage space.

Institute of Automation

Verification of logical assumptions with assert

Example

```
#include <cmath>
#include <cassert>
void PositionValues(float fX, float fY)
{
  const float MAX_POS = 600.0;
  assert((fabs(fX) <= MAX_POS) && (fabs(fY) <= MAX_POS));
  ...
}
```

- Application in test/debug phases
- Realization depends on the implementation
- Deactivation with `#define NDEBUG`

Universität Bremen

Function-templates

- Function templates for unspecified data types
- Generic usable algorithms can be made available
- **Declaration and definition must be in the header-file!**

Definition

```
template <class tType>
void Swap(tType &A, tType &B)
{
  tType Temp = A;
  A = B;
  B = Temp;
}
```

Example

```
int iNumber1;
int iNumber2;
iNumber1 = 1;
iNumber2 = 2;

Swap(iNumber1, iNumber2);
```

Inline-functions

- Theoretical reduction of execution time due to saving of jump statements (call replaced by definition)
- Only a recommendation for the compiler
- **Declaration and definition must be in header file!**

Example

```
inline int Signum(int iNumber)
{
  if (iNumber > 0)
    return 1;
  if (iNumber < 0)
    return -1;
  return 0;
}
```

Exercise

A simple math-module

Create a module, which holds a function for dividing two double values and giving the result as return-value:

- Write a specification for each function
- Use `assert()` to avoid division by 0
- Write a `main()` function to test your function/module, also for divisor = 0
- Test the behavior of your application without and with the `#define NDEBUG` statement (hint: insert `#define NDEBUG` (just) before the `#include <cassert>`-statement)
- overload the function for usage with float and integer values