IP of the Max-Planck-Institute for Physics of Complex Systems (mpipks),
Nöthnitzer Straße 38, 01187 Dresden

# MPI PKS C++ Lecture

Bennet Becker

2019-09-09

MPI PKS

# C++ Basics

# Hello World and Program Structure

# Hello World

```
1  // my first program in C++
2  #include <iostream>
3
4  int main(int argc, const char** argv)
5  {
6    std::cout << "Hello World!" << std::endl;
7
8    return 0;
9  }
```

# Hello World

- Line 1: Comment, indicated by the two slash
- Line 2: Preprocessor Directive. interpreted by the preprocessor before compilation. In this case inclusion of standard C++ code.
- Line 4: Declaration of the main-function, which is typically the first function to be invoked after program execution
- Line 5 and 9: Begin and End of the main function's function block
- Line 6: C++ statement. Writing the string inside the quotes and **st**andar**d** **end** of **l**ine symbol to the **st**andar**d** **c**haracter **out**put device.
- Line 8: Return statement for the main function, which value is returned to the calling (parent) process

# Program Structure i

- semicolon (;) at each statement end
  - marks the end of the statement.
  - All C++ statements must end with a semicolon character.
  - One of the most common syntax errors in C++ is forgetting to end a statement with a semicolon.

# Program Structure ii

- this program is structured in different lines and with indentation
  - not necessary in C++, but makes it much easier to understand for humans.
  - In bigger Projects or in some Companies there are Style Guides. They define how you should/have to structure Program, to make it understandable for others
  - E.g.:
    - Google Style Guide `https://google.github.io/styleguide/cppguide.html`
    - C++ Core Guidelines `https://github.com/isocpp/CppCoreGuidelines`
    - LLVM `https://llvm.org/docs/CodingStandards.html`
    - Stroustrup `http://www.stroustrup.com/JSF-AV-rules.pdf`
    - ...

# Program Structure iii

- In this course we will follow these simple rules:
  - 1 statement per line
  - $\approx \leq 120$ Characters per line
  - new Block = new indentation level

# Comments

```
// line comment
/* block
comment */
```

# The Compiler

# The Compiler

- translate your program into a computer executeable form
- the computer does not understand C(++) and Programming in the Computers Language (Assembly) is not something someone want or should do

# Assembly i

The simple program from before, even more simplified. And this is C and not C++

```
#include <unistd.h>
#include <stdlib.h>

void _start(){
write(1, "Hello World!", 12);
exit(0);
}
```

# Assembly ii

```
gcc -s -nostartfiles -O3 -S hello.c
```

# Assembly iii

```
        .file   "hello.c"
        .text
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "Hello World!"
        .text
        .p2align 4,,15
        .globl  _start
        .type   _start, @function
_start:
.LFB29:
        .cfi_startproc
        leaq    .LC0(%rip), %rsi
        movl    $1, %edi
        subq    $8, %rsp
        .cfi_def_cfa_offset 16
        movl    $12, %edx
        call    write@PLT
        xorl    %edi, %edi
        call    exit@PLT
        .cfi_endproc
.LFE29:
        .size   _start, .-_start
        .ident  "GCC: (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0"
        .section        .note.GNU-stack,"",@progbits
```

# A Compiler Toolchain i

- a (C++) compiler usually only translates (C++) Code to Assembler Code. But other steps are necessary which are not part of the compiler.

# A Compiler Toolchain ii

- Preprocessor: simple text substitution tool to resolve preprocessor macros and directives
  - `#define` Substitutes a preprocessor macro.
  - `#include` Inserts a particular header from another file.
  - `#undef` Undefines a preprocessor macro.
  - `#ifdef` Returns true if this macro is defined.
  - `#ifndef` Returns true if this macro is not defined.
  - `#if` Tests if a compile time condition is true.
  - `#else` The alternative for `#if` .
  - `#elif` `#else` and `#if` in one statement.
  - `#endif` Ends preprocessor conditional.
  - `#error` Prints error message on stderr.
  - `#pragma` Issues special commands to the compiler, using a standardized method.

# A Compiler Toolchain iii

- Assembler: Translates the Assembler- to Object Code. Mnemonics (seen before) are replaced with the corresponding opcode (just a binary string) for your CPU architecture
  - Also the Mnemonics themselves differ from architecture to architecture
- Linker: Links the Libraries required by your program to the executable

- these 3 with the compiler are often referred to as a compiler toolchain.

# A Compiler Toolchain iv

- There are many C++ Compiler Toolchains for various Purposes and OSes
  - **GNU Compiler Collection** (GCC/G++). Free and Open-Source for most OSes and architectures.
  - **Clang** with LLVM. Modern Free and Open-Source Compiler for several OSes and architectures.
  - **Intel C++ Compiler** (icc). Proprietary Compiler optimized for Intel CPUs (and probably also GPUs in the future)
  - **AMD Optimizing C/C++ Compiler** (aocc). Open-Source Compiler by AMD, optimized for AMD CPUs (and GPUs)
  - **Turbo C++** (tcc). Proprietary (but Free) Borland Compiler
  - **Nvidia CUDA Compiler** (nvcc). Proprietary compiler by Nvidia intended for use with CUDA to run on CPUs, GPUs and GPGPUs.

# A Compiler Toolchain v

- **Microsoft Visual C++ compiler** (msvc). Proprietary Compiler for Visual C++ on Windows.
- ...and many more ...

# Compiling i

Compile a program: `g++ helloworld.cpp -o helloworld`
Steps in between `g++ helloworld.cpp -save-temps -o helloworld`.
Yields:

- `helloworld.cpp` - Your Code
- `helloworld.ii` - intermediate file from the preprocessor
- `helloworld.s` - Assembly
- `helloworld.o` `helloworld` - Object file and Executable

# Compiling ii

This is for a single file. But what about bigger multi-file projects?

- `helloworld.cpp` includes `test.h` with `test.cpp`
- ```
  g++ -c -Wall -Werror -fPIC test.cpp
  g++ -shared -o libtest.so test.o
  g++ -L. -Wall -o helloworld helloworld.cpp -ltest
  ```
- tedious for many libraries


  ```

# CMake

# CMake i

- CMake is an open-source, cross-platform family of tools designed to build, test and package software

# CMake i

- CMake is an open-source, cross-platform family of tools designed to build, test and package software
- Basic example. Inside the Folder with your sources create a file called `CMakeLists.txt` (case-sensitive)

# CMake ii

with the content

```
cmake_minimum_required (VERSION 3.10)
project (helloworld)
add_executable(helloworld helloworld.cpp)
```

# Easy Program Versioning i

CMakeLists.txt

```
cmake_minimum_required (VERSION 3.10)
project (helloworld)
# The version number.
set (helloworld_VERSION_MAJOR 1)
set (helloworld_VERSION_MINOR 0)

# configure a header file to pass some of the CMake settings
# to the source code
configure_file (
"${PROJECT_SOURCE_DIR}/helloworld.h.in"
"${PROJECT_BINARY_DIR}/helloworld.h"
)

# add the binary tree to the search path for include files
# so that we will find helloworld.h
include_directories("${PROJECT_BINARY_DIR}")

# add the executable
add_executable(helloworld helloworld.cpp)
```

# Easy Program Versioning ii

helloworld.h.in

```
// the configured options and settings for Tutorial
#define helloworld_VERSION_MAJOR @helloworld_VERSION_MAJOR@
#define helloworld_VERSION_MINOR @helloworld_VERSION_MINOR@
```

# Easy Program Versioning iii

helloworld.cpp

```
// my first program in C++
#include <iostream>
#include "helloworld.h"

int main(int argc, const char** argv)
{
        std::cout << "Hello World!" << std::endl;
        std::cout << "Version "
                  << helloworld_VERSION_MAJOR << "."
                  << helloworld_VERSION_MINOR << std::endl;

        return 0;
}
```

# More...

- Testing
- Adding Libraries
- ...

# Datatypes and Variables

# Datatypes and Variables

- Variables, portions of memory that store a value
- each variable has a *distinct* Name and a type.

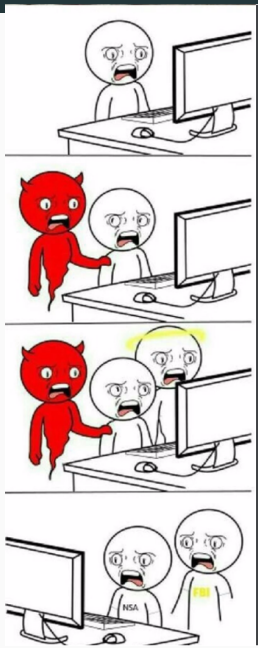
# Identifiers i

- Name can be any valid C++ identifier
  - An identifier is an arbitrarily long sequence of digits, underscores, lowercase and uppercase Latin letters, and most Unicode characters
  - valid identifier must begin with a non-digit character and can not begin with characters in the following unicode ranges:
    - U+0300 - U+036F Combining Diacritical Marks
    - U+1DC0 - U+1DFF Combining Diacritical Marks Supplement
    - U+20D0 - U+20FF Combining Diacritical Marks for Symbols
    - U+FE20 - U+FE2F Combining Half Marks

# Identifiers ii

- identifiers that are keywords cannot be used for other purposes
- identifiers with a double underscore anywhere are reserved
- identifiers that begin with an underscore followed by an uppercase letter are reserved
- identifiers that begin with an underscore are reserved in the global namespace
- using reserved identifiers may result in , undefined behavior
- Identifiers are case-sensitive

# Identifiers iii




```
namespace 🔵 = std;
using 🔢 = int;
using 💀 = void;
using 🕐 = time_t;
using 👌 = bool;
#define 💡 auto
#define 🎲 enum
#define 👎 false
#define 👍 true
#define 😈 "evil"
#define 💪 🔵::make_shared
#define 🍸 virtual
#define 🖥️ 🔵::cout
#define 🔚 🔵::endl
template<class 🤡>
using 📚 = 🔵::vector<🤡>;
template<class 🤡>
using 🐈 = 🔵::shared_ptr<🤡>;

🎲 🍬 { 🍎, 🍊, 🍋, 🍇 };
🔢 🎲() { return 🔵::rand(); }
👌 😎() { return 👎; }

struct 🍴 { 🍸 💀 👀() = 0; };
struct 🍎 : 🍴 { 🍸 💀 👀() {  🖥️ << "🍎" << 🔚; }; };
struct 🍊 : 🍴 { 🍸 💀 👀() {  🖥️ << "🍊" << 🔚; }; };
struct 🍋 : 🍴 { 🍸 💀 👀() {  🖥️ << "🍋" << 🔚; }; };
struct 🍇 : 🍴 { 🍸 💀 👀() {  🖥️ << "🍇" << 🔚; }; };
struct 🍌 : 🍴 { 🍸 💀 👀() {  🖥️ << "🍌" << 🔚; }; };
struct 🍅 : 🍴 { 🍸 💀 👀() {  🖥️ << "🍅" << 🔚; }; };

🔢 main()
{
    if (😎() == 👎)
        🖥️ << "🤮" << 🔚;

    📚<🐈<🍴>> 🥭 = { 💪<🍎>(), 💪<🍊>(), 💪<🍋>(), 💪<🍌>(), 💪<🍅>() };

    for (💡 🍏 : 🥭)
        🍏->👀();

    return 🎲();
}
```

# Fundamental Datatypes

- Single Characters (`char`)
- Signed Integer (`short`/`int16_t`, `int`/`int32_t`, `long`/`int64_t`)
- Unsigned Integer (`unsigned short`/`uint16_t`, `unsigned int`/`uint32_t`, `unsigned long`/`uint64_t`)
- Floating Point with single and double precision (`float`, `double`)
- *Typeless chunk of memory, `void`*

# Datatypes i

| Group | Type | Size in Byte (Bit) | Range | Note |
|---|---|---|---|---|
| Character Types | *signed* char | 1 | *-128* to *127* | Exactly one byte in size |
| | unsigned char | 1 | *0* to *255* | |
| | char16_t | 2 | *0* to *65'535* | Not smaller than char. At least 16 bits. |
| | char32_t | 4 | *0* to *4'294'967'295* | Not smaller than char16_t. At least 32 bits. |
| | wchar_t | 4 | *-2'147'483'648* to *2'147'483'647* | Can represent the largest supported character set |
| Integer Types | *signed* short *int* | 2 (16) | -32'768 to 32'767 | At least 16 bits. But Compiler and Platform dependent |
| | *signed* int | 4 | -2'147'483'648 to 2'147'483'647 | Not smaller than short. At least 16 bits. |
| | *signed* long *int* | 8 | -9'223'372'036'854'775'808 to 9'223'372'036'854'775'807 | Not smaller than int. At least 32 bits. |
| | *signed* long long *int* | 8 | -9'223'372'036'854'775'808 to 9'223'372'036'854'775'807 | Not smaller than long. At least 64 bits. |
| | unsigned short *int* | 2 | 0 to 65'535 | Same as their signed counterparts |
| | unsigned int | 4 | 0 to 4'294'967'295 | |
| | unsigned long *int* | 8 | 0 to 18'446'744'073'709'551'615 | |
| | unsigned long long *int* | 8 | 0 to 18'446'744'073'709'551'615 | |
| | int8_t | 1 | -128 to 127 | signed 8 bit fixed width integer |

# Datatypes ii

| | | | | |
|---|---|---|---|---|
| | int16_t | 2 | 32'768 to 32'767 | signed 16 bit fixed width integer |
| | int32_t | 4 | -2'147'483'648 to 2'147'483'647 | signed 32 bit fixed width integer |
| | int64_t | 8 | -9'223'372'036'854'775'808 to 9'223'372'036'854'775'807 | signed 64 bit fixed width integer |
| | uint8_t | 1 | 0 to 255 | unsigned 8 bit fixed width integer |
| | uint16_t | 2 | 0 to 65'535 | unsigned 16 bit fixed width integer |
| | uint32_t | 4 | 0 to 4'294'967'295 | unsigned 32 bit fixed width integer |
| | uint64_t | 8 | 0 to 18'446'744'073'709'551'615 | unsigned 64 bit fixed width integer |
| Floating-point types | float | 4 | 1.17549e-38 to 3.40282e+38 | single precision floating point |
| | double | 8 | 2.22507e-308 to 1.79769e+308 | double precision floating point (not less than float) |
| | long double | 16 | 3.3621e-4932 to 1.18973e+4932 | precision not less than double |
| Boolean Type | bool | 1 | 0 (false) or 1 (true) | |
| Void type | void | - | - | incomplete type. no storage |
| Null pointer | decltype(nullptr) | 8 | - | Datatype of Null pointer (pointing to memory address 0x00). length is address memory length |

# Fundamental Datatypes

- properties of fundamental types can be retrieved with `std::numeric_limits<type>`
  - min, max, number of digits, round error, ...
- all above types are fundamental types
- characters, integers, floating-point, and boolean are collectively known as arithmetic types
- also fundamental types: void, which is the lack of a type and nullptr which is a special pointer
- all other types are compound datatypes

# Declaration and Initialization of variables i

- C++ is a strongly-typed language, and requires every variable to be **declared** *with its type* **before** its first use.
- straightforward variable declaration: type followed by name

```
int a;
float number;
```

- multiple at once

```
int a, b, c;
```

# Declaration and Initialization of variables ii

- best practice: declare **all** variables you use at the beginning of a function.
- also best practice: initialize variables on declaration.
- uninitialized variable may hold a *random**/undetermined value until they are assigned a value for the first time. (* not truly random, not even pseudorandom, but whatever was there before)

# Declaration and Initialization of variables iii

- in C++ there are 3 way for initialization. They are all equivalent and are reminiscent of the evolution of the language over the years
  - c-like initialization: `type indentifier = value;`
  - constructor initialization: `type indentifier (value);`
  - uniform initialization: `type indentifier {value};`

# Special feature - Type deduction

- when a variable is initialized the compiler is able to figure out the type automatically by the initializer

```
int foo = 0;
auto bar = foo;  // the same as: int bar = foo;
```

- Variables that are not initialized can also make use of type deduction, to "copy" the type of another variable

```
int foo = 0;
decltype(foo) bar;  // the same as: int bar;
```

# Datatype-Quiz i

Which is the appropriate datatype for the following values?

- `var = 300`
- `var = true`
- `var = 4294967294`
- `var = '6'`

# Datatype-Quiz i

Which is the appropriate datatype for the following values?

- `var = 300`

  ```
  short var;
  ```

- `var = true`

- `var = 4294967294`

- `var = '6'`

# Datatype-Quiz i

Which is the appropriate datatype for the following values?

- `var = 300`
  `short var;`
- `var = true`
  `bool var; or int var;`
- `var = 4294967294`
- `var = '6'`

# Datatype-Quiz i

Which is the appropriate datatype for the following values?

- `var = 300`
  `short var;`
- `var = true`
  `bool var;` or `int var;`
- `var = 4294967294`
  `unsinged int var;` or `long var;`
- `var = '6'`

# Datatype-Quiz i

Which is the appropriate datatype for the following values?

- `var = 300`
  `short var;`
- `var = true`
  `bool var;` or `int var;`
- `var = 4294967294`
  `unsinged int var;` or `long var;`
- `var = '6'`
  `char var;`

# Datatype-Quiz ii

- `var = 42`
- `var = 4711L`
- `var = 2.718281`
- `var = 3.14159265359`
- `var = "Hello World!"`

# Datatype-Quiz ii

- `var = 42`
  `char var = '"';`
- `var = 4711L`
- `var = 2.718281`
- `var = 3.14159265359`
- `var = "Hello World!"`

# Datatype-Quiz ii

- `var = 42`
  `char var = '"';`
- `var = 4711L`
  `long var;`
- `var = 2.718281`
- `var = 3.14159265359`
- `var = "Hello World!"`

# Datatype-Quiz ii

- `var = 42`
  `char var = '"';`
- `var = 4711L`
  `long var;`
- `var = 2.718281`
  `float var;`
- `var = 3.14159265359`

- `var = "Hello World!"`

# Datatype-Quiz ii

- `var = 42`
  `char var = '"';`
- `var = 4711L`
  `long var;`
- `var = 2.718281`
  `float var;`
- `var = 3.14159265359`
  `double var;`
- `var = "Hello World!"`

# Datatype-Quiz ii

- `var = 42`
  `char var = '"';`
- `var = 4711L`
  `long var;`
- `var = 2.718281`
  `float var;`
- `var = 3.14159265359`
  `double var;`
- `var = "Hello World!"`
  `char var[12];`

# Constants

# Constants

- expressions with a fixed value.
- most obvious: Literals
  - most obvious kind of constants
  - express particular values within the source code
  - Literal constants can be classified into: integer, floating-point, characters, strings, Boolean, pointers, and user-defined literals.
  - Integer Numerals
  - Floating Point Numerals
  - Character and string literals
  - Other Literals are keyword literals: `true` and `false` for `bool` and `nullptr` , the null pointer value

# Integer Numerals

- numerical constants that identify integer values
- not enclosed in quotes or any other special character
- Easiest example: `0` `8` `15` `-42`
- Also ocatal and hexadecimal $75_{10} = 0113_{8} = 0x4b_{16}$
- all these have a type. by default `int`. But changeable via suffixes
  - `u` *or* `U` for `unsigned`
  - `l` *or* `L` for `long`
  - `ll` *or* `LL` for `long long`

```
42          // int
42u         // unsigned int
42l         // long
42ul        // unsigned long
42lu        // unsigned long
```

# Floating Point Numerals

- real values, with decimals and/or exponents
- include either a decimal point, an e character
- default type for floating-point literals is `double`. But changeable via suffixes
  - `f` *or* `F` for `float`
  - `l` *or* `L` for `long double`
- float is an approximation

# Character and string literals i

- Character and string literals are enclosed in quotes

```
1 'z'
2 'p'
3 "Hello world"
4 "How do you do?"
```

- first 2 are *single-character literals*. denoted with single quotes. using more than one character in single quotes is undefined-behavior
- last 2 are string literals. denoted with double quotes.

# Character and string literals ii

- special characters with escape sequences
  - `\n` newline
  - `\r` carriage return
  - `\t` tab
  - `\v` vertical tab
  - `\b` backspace
  - `\f` form feed (page feed)
  - `\a` alert (beep)
  - `\'` single quote (')
  - `\"` double quote (")
  - `\?` question mark (?)
  - `\\` backslash (\)

# Character and string literals iii

- All the character literals and string literals described above are made of characters of type `char`. Different Type with **pre**fixes
  - `u` for `char16_t`
  - `U` for `char32_t`
  - `L` for `wchar_t`
- additional encoding prefixes
  - `u8`, The string literal is encoded in the executable using UTF-8
  - `R`, The string literal is a raw string

# Other Literals

- Other Literals are keyword literals: `true` and `false` for `bool` and `nullptr` , the null pointer value

# Other Constants

- Typed constant expressions. with keyword `const` . a constant variable requires an initializer
- Preprocessor definitions. `#define identifier replacement`
  - every occurrence of identifier in the code is interpreted as replacement.
  - replacement can be any sequence of characters
  - replacement by preprocessor, blindly before program is compiled. without any type checking

# Operators

# Operators i

- Assignment: `=`
- Arithmetic: `+` (addition and unary plus) `-` (subtraction and unary minus) `*` `/` `%`
- Compound assignment: `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `^=` `|=`
- Increment/Decrement: `++` `--`
  - difference between prefix `++x` and suffix `x++` form

# Operators ii

```
x = 3;
y = ++x;
// x contains 4, y contains 4

x = 3;
y = x++;
// x contains 4, y contains 3
```

- Relational and comparison: `==` `!=` `>` `<` `>=` `<=`
- Logical: `!` `&&` `||`
- Conditional ternary: `condition ? result_true : result_false`
- NEW: C++20 Three-way comparison `<=>`

# Operators iii

- The expression returns an object such that
  - `(a <=> b) < 0` if `lhs < rhs`
  - `(a <=> b) > 0` if `lhs > rhs`
  - `(a <=> b) == 0` if `lhs` and `rhs` are equal/equivalent.
- If one of the operands is of type bool and the other is not, the program is ill-formed.
- If both operands have arithmetic types, or if one operand has unscoped enumeration type and the other has integral type, the usual arithmetic conversions are applied to the operands, and then
- If a narrowing conversion is required, other than from an integral type to a floating point type, the program is ill-formed.

# Operators iv

- Comma: used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the right-most expression is considered.
  ```
  a = (b=3, b+2); // a: 5
  ```
- Bitwise: `&` (AND), `|` (OR), `^` (XOR), `~` (NOT), `<<` (shift left), `>>` (shift right)
- Explicit type casting `(type)`
  - `type_t a = (type_t) b;`
  - casting `float` to `int`, cuts fractional part (round down)
- Streaming `<<` `>>`
- sizeof: returns size in byte of object or type
- Scope: `::` see OOP

# Operators v

- Member Access: `[]` `*` `&` `.` `->` `.*` `->*` see OOP
- (de)allocation: `new` `delete` see OOP

# Basic IO

# Basic IO

- C++ uses abstraction called *streams* for sequential media
- a program can insert or extract characters from a stream sequentially
- Streams from the standard library `#include <iostream>`
  - `std::cin` Standard input stream
  - `std::cout` Standard output stream
  - `std::cerr` Standard error output stream
  - `std::clog` Standard logging output stream

# Standard Output

- screen / terminal / (fd 1)
- insert text with insertion operator `<<` . Chaining possible
- newline `\n` (for Linux, other plattforms: `\r\n` (windows) or `\r` (mac)) or `std::endl` (appropriate for plattform)
- output is buffered. `std::endl` flushes the buffer, forces all remaining characters to be written.

# Standard Input

- keyboard / terminal (fd 0)
- read text with extraction operator `>>` followed by a variable to store the value
- input is automatically interpreted depending on variable type. fails silently on malformed input -> introduces ub
- chaining is possible. filling fist variable first. separation by space, tab, or new-line
- "Problem" with strings: separation by space characters lets you only read a single word, not whole sentences or lines. Solution:
  `std::getline(std::cin, mystr);`

# String Streams

- stringstreams from `#include <sstream>` allows strings to be treated as streams. Allowing to insert and extract from, which is useful but no limited to converting strings to numeric values.
- String-Conversion also with
  - `std::stoi()` : String to Int
  - `std::stol()` : String to Long
  - `std::stof()` : String to Float
  - `std::stod()` : String to Double
  - `std::to_string()` : X to String