UNIVERSITY OF PASSAU

Faculty of Computer Science and Mathematics

Object-oriented Programming with C++

Prof. Dr.-Ing. Christian Hammer

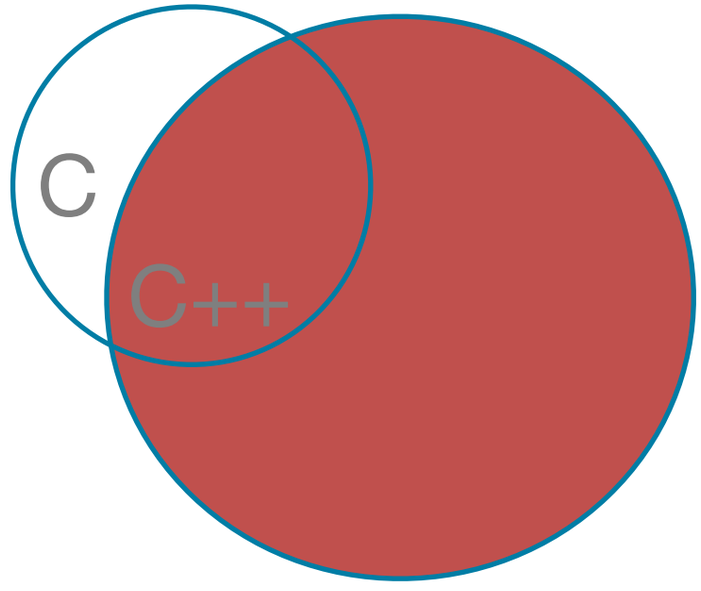
Winter 2021/22

- Lecture slides published on Stud.IP, look for course “5855V”
- Schedule
  - Lecture : Wed, 10:15 - 11:45, Zoom
  - Recitation: Wed, 12:15 - 13:45, Zoom
    Wed, 16:15 - 17:45, Zoom
    Fri, 14:15 - 15:45, Zoom
- Loosely following these books:
  - (English) Programming and Principles and Practice Using C++ (B. Stroustrup)
    - Available as hard copy in library
  - (German) Der C++ Programmierer (Ulrich Breymann) [5th ed., C++17]
    - Available as ebook & hard copy in library

- Homework
  - Posted every week, discussed in tutorial the week after
  - No submission but **present** at least 3 correct exercise solution in tutorials to qualify for examination
- Project
  - 4 short projects over the semester, pass in at least 3 projects
  - Each project contributes 33% to Final Grade
- Final Grade = Sum of score in best 3 projects

- Previous knowledge of programming any (imperative) programming language (e.g. in C, Java, Python)
- Experience in using build systems is an advantage

- 1979 Bjarne Stroustrup (AT&T): C with classes
- Mostly superset of language C
- Most prominent feature: Object Orientation
- Course Objective
  - omit C legacy and learn “modern” C++ style



Hello World

```cpp
#include <iostream> // library for I/O

int main() {
   std::cout << "Hello World!" << std::endl;
   // missing return statement in main replaced by return 0;
}
```

- Preprocessor directives with #
- System libraries with <> and no file name
- I/O via streams
- std::cout is standard output, << output operator

- namespaces allow to reuse the same name
- Definition of namespaces: later

- usage of namespace std:

```
using namespace std;
```

- or selectively

```
using std::cout;
using std::endl;
```

- then

```
int main() {
   cout << "Hello World!" << endl;
   // missing return statement in main replaced by return 0;
}
```

- Perfect static type safety
- Automatic release of general resources (perfect resource safety)
- No run-time overhead compared to good hand-crafted code (the zero-overhead principle)
- No restriction of the application domain compared to C and previous versions of C++
- Compatibility with previous versions of C++ (long-term stability is a feature)

- Perfectly matching these ideals (and more; see [Stroustrup,1994]) seemed impossible

- Low level language w/ many high level abstractions
- Developed since first implementation in 1985.
- Standard continuously developed (C++ 98, 03, 11, 14, 17, 20)
- Efficient but often complex semantics
- No automatic memory management (garbage collection)
- No automatic safety/security checks (array bounds, etc)
- Undefined behavior often (ab)used (compiler specific)
  - Not in this class!

Integer Numbers

- Types

	usual	min according to standard
short	16	
int	32	16
long	64	32
long long		64

- (u)int_fast8_t
- (u)int_fast16_t
- (u)int_fast32_t
- (u)int_fast64_t

Limits of Basic Data Integer Types on a Given System

```
/* cppbuch/k1/intlimits.cpp
   Beispiel zum Buch von U. Breymann: Der C++ Programmierer; 4. Aufl., korr. Nachdruck 2016
   Diese Software ist freie Software. Website: http://www.cppbuch.de/
*/
#include <cstddef>
#include <iostream>
#include <limits> //Has limits information
using namespace std;

int main() {
  cout << “Limits for Integer Types:" << '\n'; // = Newline, alternate use std::endl
  cout << "int Minimum =       " << numeric_limits<int>::min() << '\n';
  cout << "int Maximum =       " << numeric_limits<int>::max() << '\n';
  cout << "long Minimum =      " << numeric_limits<long>::min() << '\n';
  cout << "long Maximum =      " << numeric_limits<long>::max() << '\n';
  cout << "long long Minimum = " << numeric_limits<long long>::min() << '\n';
  cout << "long long Maximum = " << numeric_limits<long long>::max() << '\n';
  cout << "unsigned-Maxima (Minimum is 0):\n"; //new line
  cout << "unsigned int  =  " << numeric_limits<unsigned int>::max() << '\n';
  cout << "unsigned long =  " << numeric_limits<unsigned long>::max() << '\n';
  cout << "unsigned long long = " << numeric_limits<unsigned long long>::max() << '\n';

  cout << “# Bytes:\n";
  cout << "int       " << sizeof(int) << '\n';
  cout << "long      " << sizeof(long) << '\n';
  cout << "long long " << sizeof(long long) << '\n';
  cout << "size_t    " << sizeof(size_t) << '\n';
}
```

Check for Integer Overflow

- Integer under/overflow can lead to out-of-bounds access memory error, remote code execution?
- int INT_MAX = numeric_limits<int>::max()
- a + b < INT_MAX
- a < INT_MAX - b // assuming b >= 0
- a * b < INT_MAX
- a < INT_MAX / b // assuming b >= 0

- int INT_MIN = numeric_limits<int>::min()
- a - b < INT_MIN
- a < INT_MIN + b // assuming b >= 0

Floating Point Types

- float - Single Precision IEEE754 32 bit
- double - Double precision IEEE754 64bit
- long double - Usually 80bit, not confirming to IEEE754

```
cout << “float (max) =  " << numeric_limits<float>::max() << '\n';
cout << “double (max)=  " << numeric_limits<double>::max() << '\n';
cout << “long double (max) = " << numeric_limits<long double>::max() << '\n';
cout << "float  (min)=" << numeric_limits<float>::min() << '\n';
cout << “double (min)=  " << numeric_limits<double>::min() << '\n';
cout << “long double (min)=" << numeric_limits<long double>::min() << '\n';
```

- [unsigned/signed] char a = 'a'
- (usually) 8 bit character, signed or unsigned depends on system
- wchar_t for wide characters, literals start with L: L'??'
- Conversion:

```
int i {66};
char c = static_cast<char>(i);
cout << c;
c = '1';
i = static_cast<int>(c);
cout << i;
```

```
bool isCapital;
char c;
cin >> c;
isCapital = (c >= 'A') && (c <= 'Z');
cout << isCapital;
cout.setf(ios_base::boolalpha);
cout.unsetf(ios_base::boolalpha);
cout << isCapital << '\n';
```

- Type bool built into language
- Operators: !, &&, ||, !=, ==
- bool can be true(1) or false(0)

```
const float PI = 3.1415926; // better: pi<float> (later)
circumference = PI * diameter
```

- `const` for constant that cannot be changed

```
constexpr float circumference = PI * diameter
```

- `constexpr` for constant that can be determined at compile time
- not all const are constexpr (later)

Aliasing via References

```
int i {2};
int j {9};
int& r {i};
r = 10;
r = j;
```

- Reference created with type annotated with &
- Creates alias (alternative name) for right hand side var

Associativity and evaluation order

- right-associative: prefix ++ or --, ?: and assignments +=
- a = b = c is equal to a = (b = c)

- all other operators are left-associative
- order of subexpression evaluation is **undefined**
- **no expressions that use and define a variable**

```
int total {}; // 0
int sum = (total = 3) + total; // undefined
int i {2};
i = 3 * i++; // undefined
```

- a = b is an expression! (b is its value)
- beware:

```
if (a = b) { // Probably not what you meant
   cout << "a equals b";
}
```

- prints whenever b != 0 due to assignment
- therefore in C++ a lot of people write constants to the left:

```
if (0 == b) { // if you forget an = the compiler will warn
   cout << "a equals b";
}
```

-

```
enum class Weekdays {Sunday, Monday, Tuesday, Wednesday,
    Thursday, Friday, Saturday};
Weekdays holiday, workingday, today = Weekdays::Tuesday;


int i = static_cast<int>(Weekdays::Tuesday);
today = static_cast<Weekdays>(i);
```

```
enum class Bitmask { v1 = 1 << 0, v2 = 1 << 1, v3 = 1 << 2,
   v4 = 1 << 3, v5 = 1 << 4, v6 = 1 << 5};
```

- Usually all values are ordered starting at 0. With casts they can be converted from/to int. If a value does not exist in an enum then undefined!

- Defined in standard library file `<vector>`

```
vector<int> v(10); // round brackets
vector<int> v1 {}; // empty vector
vector<int> v2 {7, 0, 9}; // with elements 7, 0, 9
cout << v.size()  << '\n';
cout << v1.size() << '\n';
cout << v2.size() << '\n';
// 10 // 0 // 3
cout << v[0] << '\n'; // Value at index 0
cout << v.at(0) << '\n'; // same as v[0]
// 1000 is too much!
cout << v.at(1000) << '\n'; // error
```

Dynamic Growth for Vector

```
/* cppbuch/k1/dynvekt.cpp
   Beispiel zum Buch von U. Breymann: Der C++ Programmierer; 4. Aufl., korr. Nachdruck 2016
   Diese Software ist freie Software. Website: http://www.cppbuch.de/
*/
#include <iostream>
#include <vector>  // Vector from Standard Library
#include <cstddef> // size_t
using namespace std;

int main() {
  vector<int> data; // size = 0
  cout << “Please enter values\n";
  int value;
  do {
    cout << "Value (0 = End Input):";
    cin >> value;
    if (value != 0) {
      data.push_back(value);
    }
  } while (value != 0);

  cout << “You entered the following values:\n”;
  for (size_t i = 0; i < data.size(); ++i) {
    cout << i << ". Value : " << data[i] << '\n';
  }
}
```

C++ Strings

```
/* cppbuch/k1/zbstring.cpp
*/
#include <iostream>
#include <string>
#include <cstddef>
using namespace std;

int main() {

  string aString{"hallo"};
  cout << aString << '\n';
  for (size_t i = 0; i < aString.size(); ++i) {
    cout << aString[i];
  }
  cout << '\n';

  for (size_t i = 0; i < aString.length(); ++i) {
    cout << aString.at(i);
  }
  cout << '\n';

  string aStringCopy(aString);
  cout << aStringCopy << '\n';

  string thisIsNew{"neu!"};
  aStringCopy = thisIsNew;
  cout << aStringCopy << '\n';

  aStringCopy = "Buchstaben";
  cout << aStringCopy << '\n';

  aString = 'X';
  cout << aString << '\n';

  aString += aStringCopy;
  cout << aString << '\n';

  aString = aStringCopy + " ABC";
  cout << aString << '\n';

  aString = "123" + aStringCopy;
  cout << aString << '\n';

  // aString = "123" + "ABC";
  aString = string("123") + "ABC";

  // Same as strings
  string a{"Albert"};
  string z{"Alberta"};
  string b{a};
  if (a == b) {
    cout << a << " == " << b << '\n';
  } else {
    cout << a << " != " << b << '\n';
  }
  if (a < z) {
    cout << a << " < " << z << '\n';
  }
  if (z > a) {
    cout << z << " > " << a << '\n';
  }
  if (z != a) {
    cout << z << " != " << a << '\n';
  }

  string str{'a', 'b', 'c'};
  cout << “Initialization with string array:" << str
       << '\n';
}
```

Iterations

```
for(size_t i = 0; i < aVector.size(); ++i) {
    cout << aVector[i] << '\n';
}
for(int value : aVector) { //Copy every element
    cout << value << '\n';
}
for(int value : aVector) { // changeable copy
    value = 2 * value;
    cout << value << '\n';
}
for(const int value : aVector) { // unchangeable copy
    cout << value << '\n';
}
for(int& value : aVector) { // Reference variable to element
    value *= 2; // modify each element
}
for(const int& value : aVector) { // Read using a reference variable
    cout << value << '\t';
}

.
```

```
const auto a = 2;
auto b = a;
auto &c = b;

for(const auto& value : aVector) {
   cout << value << '\n';
}
```

```
for(auto& chr : astring) {
   if(chr == ' ') {
      chr = '_';
   }
}
string s1 {"Ende!"};
auto s2(s1); // not auto s2 {s1} // would be a list with s1 in it
cout << s2 << '\n';
```

```
struct BitFieldStruct {
  unsigned int a : 4; // 0-15
  unsigned int b : 3; // 0-7
};

int main() {
  BitFieldStruct x;
  x.a = 06;
  x.b = x.a | 3;
  cout << x.b << ‘\n'; // Converted to unsigned and then displayed
}
```

- unclear where in a and b the bits are being stored
- **may** save space but access incurs significant overhead
- usually subsequent bitfield are put next to each other

- cin is buffered stream (allows editing w/ backspace etc.)
- only when return pressed will OS feed the input to program
- cin uses anything delimited by whitespace (' ', 0x09 to 0x0d)
- multiple items buffered for next input

- not ignoring whitespace via get:

```
char c;
cin.get(c); // read single char
```

- reading an int and a double

```
int main() {
   int i; double d;
   while(cin >> i >> d) {
      cout << i << '\n' << d << '\n';
   }
}
```

```
int main() {
   cout << "Bitte Vor- und Nachnamen eingeben:"; string derName;
// cin >> derName; // input delimted by whitespace
   getline(cin, derName); // full input delimited by return
   cout << derName; // Donald Duck
}
```

- Output

```
cout << 7 << 11; //711

cout << 7;
cout.width(6);
cout << 11; //7     11
```

- formatting possible (more later)

```
cout << endl;

cout << '\n'; // write end of line in buffer
cout.flush(); // print buffer and empty
```

- use endl to end a line and print to screen
- otherwise output may still be in buffer when you look for bug
- can be changed via flags:

```
cout.setf(ios::unitbuf); // unbuffered, immediate output
cout.unsetf(ios::unitbuf);
```

```
/* cppbuch/k2/factorial.cpp */
#include <iostream>
using namespace std;
unsigned long factorial(unsigned int); // Function prototype (Declaration)

int main() {
  int n;
  do {
    cout << “Factorial function. Num >= 0? :";
    cin >> n;
  } while (n < 0);
  unsigned long res = factorial(n);
  cout << “The Factorial is " << res << '\n';
}

unsigned long factorial(unsigned int num) { // Function implementation(Definition)
  unsigned long fac = 1;
  for (unsigned int i = 2; i <= num; ++i) {
    fac *= i;
  }
  return fac;
}
```

```
/* cppbuch/k2/static.cpp */
#include <iostream>
using namespace std;

void func() {          // count the number of calls
  static int freq{0};
  cout << "Frequency = " << ++freq << '\n';
}
```

```
int main() {
  for (int i = 0; i < 3; ++i) {
    func();
  }
}
```

- Remembers data between calls
- Initialized before or at first call (0 if uninitialized)

Call by Reference

```
/* cppbuch/k2/per_ref.cpp */
#include <iostream>
using namespace std;

void add_7(int &y); //   int& = reference for int

int main() {
  int i{0};
  cout << i << " = Old value of i\n";
  add_7(i); // Syntax same as call by value
  cout << i << " = New value of “i” after add_7\n”;
}

void add_7(int &x) {
  x += 7; // Modified the original actual parameter
}
```

- Not a copy of value is transferred but a reference (alias) to original value

Default Parameters

```cpp
/* cppbuch/k2/preis.cpp */
#include <iostream>
#include <string>
using namespace std;

// Funktionsprototype,          2nd parameter with default
void displayPrice(const double price, const string &currency = "Euro");

int main() {
  // two calls with different parameters :
  displayPrice(12.35); // default parameter is used
  displayPrice(99.99, "US-Dollar");
}

void displayPrice(const double price, const string &currency) {
  cout << price << ' ' << currency << '\n';
}
```

- Default used whenever parameter is missing

inline Functions

```
inline int quadrat(int x) {
   return x*x;
}
```

- inline functions are directly inserted into the invoking function
- overhead of function calls (new stack frame, call, return, saving of registers) omitted

constexpr

```
unsigned long factorial0(unsigned int num)
{
  unsigned long fac = 1;
  while (num > 1) {
    fac *= num--;
  }
  return fac;
}

constexpr unsigned long
factorial1(unsigned int num) {
  return num < 2 ?
    1 : num * factorial1(num - 1);
}

unsigned int getValue() { return 3; }

int main() {

  // no \tt{constexpr}
  const unsigned long res0 = factorial0(3);
  std::cout << "Result = " << res0 << '\n';

  // no \tt{constexpr} value
  unsigned int num = 4; // no \tt{const}
  unsigned long res1 = factorial1(num);
  std::cout << "Result = " << res1 << '\n';

  // no \tt{constexpr} value
  const unsigned int cnum1 = getValue();
  unsigned long res2 = factorial1(cnum1);
  std::cout << "Result = " << res2 << '\n';

  // \tt{constexpr} value
  const int cnum2 = 3; // literal
  constexpr unsigned long res3 =
    factorial1(cnum2);
  std::cout << "Result = " << res3 << '\n';

  // \tt{constexpr} value
  constexpr unsigned long res4 =
    factorial1(3); // literal
  std::cout << "Result = " << res4 << '\n';
}
```

- must be determined at compile time! No running of code possible

main and Trailing Return Type

```
int main( int argc, char* argv[]) { // argv = Command Line Arguments
    ...
    return 0; // Exit-Code
}
```

- Careful: C-style arrays and strings

- Alternative: trailing return type

```
unsigned long factorial(unsigned int); // normal syntax
auto factorial(unsigned int) -> unsigned long; // alternate syntax
```

- becomes important later

```
// a.h
void func_a1();
void func_a2();

// a.cpp
#include "a.h"
void func_a1() {
//code for func_a1
}
void func_a2() {
   //code for func_a2
}

// b.h
void func_b();
```

```
// b.cpp
#include "b.h"
void func_b() {
// Programmcode zu func_b
}

// meinprog.cpp
#include "a.h"
#include "b.h"
int main() {
   func_a1( );
   func_a2( );
   func_b( );
}
```

- Separation of interface and implementation

Per File Global Variables

```cpp
namespace { // anonymous Namespace
    int global;
}
int main() {
    global = 17;
}
```

- Global variable only visible in that file

```cpp
// file1.cpp
extern const float CONSTANT = 42.12345; // Declaration and definition

// file2.cpp
// Declaration without Definition
extern const float CONSTANT; // without initialization
```

- Global constants require declaration of extern (non-const does not!)

One Definition Rule

```
// c.h
#ifndef C_H
#define C_H
void func_c1();
void func_c2();
enum class ColorType {red, green, blue, yellow};
#endif // C_H
```

- ensure that definition is only used once

Assertions

```cpp
#include <cassert>

constexpr int MAX = 100;
int index;
// Calculation for index
// Test for a valid index
assert(index >= 0 && index < MAX);
```

- NO TEST if NDEBUG is defined
  (via #define or via g++ -DNDEBUG …)

```cpp
static_assert(sizeof(long) > sizeof(int), "long has no more Bits than int!");
```

- Compile time check via reserved keyword!

Templates

```
template<typename T> void swap(T& a, T& b) {
    const T temp = a;
    a = b;
    b = temp;
}
```

- Defines a piece of code that can be instantiated for all possible types T
- Contrast to Java: also for basic types like int, bool, …
- if template is supposed only for class types one usually writes class instead of typename

Specializing Templates

```
template<typename T>
bool lt(const T& a, const T& b) { // comparison
    return a < b; //
}


// #include <cstdlib> for abs()
template<>
bool lt<int>(const int& a, const int& b) {
    // <int> in lt<int> may be omitted (type deduction)
    return abs(a) < abs(b); //
}
```

- Disadvantage: globally for all int comparisons