



---

# Parallel Processing

Winter Term 2025/26

Roland Wismüller  
Universität Siegen  
roland.wismueller@uni-siegen.de  
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: October 21, 2025



---

# Parallel Processing

Winter Term 2025/26

## 1 Repetition / Foundations



## Contents

- ➔ C/C++ for Java programmers
- ➔ Threads and synchronisation
- ➔ C++ threads

## 1.1 C/C++ for Java Programmers

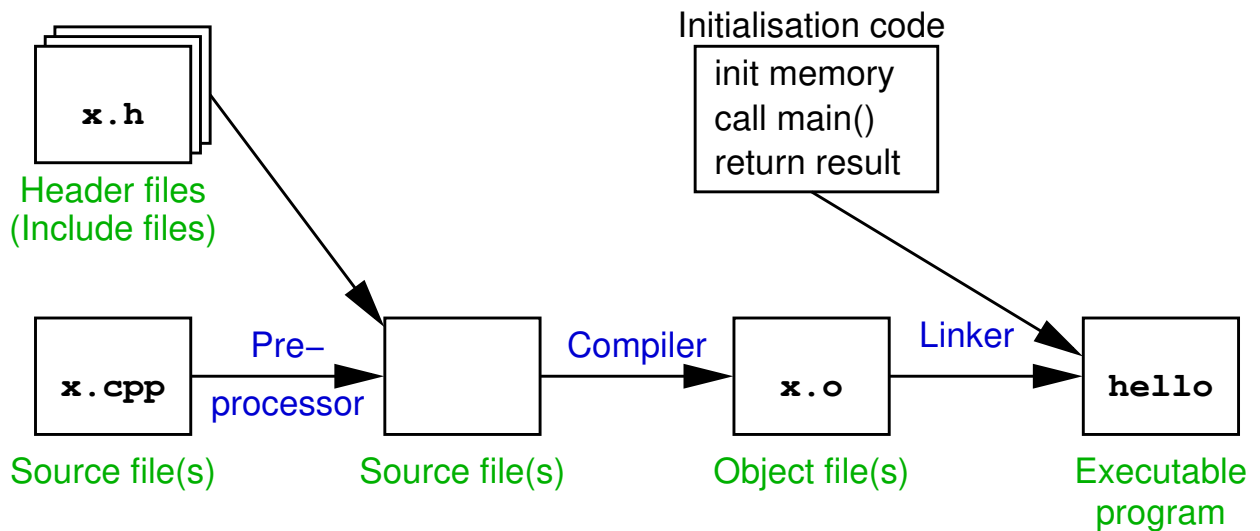


### 1.1.1 Fundamentals of C++

- ➔ Commonalities between C++ and Java:
  - ➔ imperative programming language
  - ➔ syntax is mostly identical
- ➔ Differences between C++ and Java:
  - ➔ C++ is not purely object oriented
  - ➔ C++ programs are translated directly to machine code (no virtual machine)
- ➔ Usual file structure of C++ programs:
  - ➔ header files (\*.h) contain declarations
    - ➔ types, classes, constants, ...
  - ➔ source files (\*.cpp) contain implementations
    - ➔ methods, functions, global variables



### Compilation of C++ programs



- ➔ Preprocessor: embedding of files, expansion of macros
- ➔ Linker: binds together object files and libraries



### Compilation of C++ programs ...

- ➔ Invocation of the GNU C++ compiler:
  - ➔ `g++ -Wall -o <output-file> <source-files>`
  - ➔ executes preprocessor, compiler and linker
  - ➔ `-Wall`: report all warnings
  - ➔ `-o <output-file>`: name of the executable file
- ➔ Additional options:
  - ➔ `-g`: enable source code debugging
  - ➔ `-O`: enable code optimization
  - ➔ `-l<library>`: link the given library
  - ➔ `-c`: do not execute the linker
    - ➔ later: `g++ -o <output-file> <object-files>`



### An example: *Hello World!* (↗ 01/hello.cpp)

```
#include <iostream> // Preprocessor directive: inserts contents of file
                       // 'iostream' (e.g., declaration of cout)

using namespace std; // Import all names from namespace 'std'

void sayHello() {      // Function definition
    cout << "Hello World!\n"; // Print a text to console
}

int main() {          // Main program
    sayHello();
    return 0;        // Convention for return value: 0 = OK, 1,...,255: error
}
```

➔ Compilation: `g++ -Wall -o hello hello.cpp`

➔ Start: `./hello`



### Syntax

➔ Identical to Java are among others:

- ➔ declaration of variables and parameters
- ➔ method calls
- ➔ control statements (if, while, for, case, return, ...)
- ➔ simple data types (short, int, double, char, void, ...)
  - ➔ deviations: `bool` instead of `boolean`; `char` has a size of 1 Byte
- ➔ virtually all operators (+, \*, %, <<, ==, ?:, ...)

➔ Very similar to Java are:

- ➔ arrays
- ➔ class declarations



### Arrays

#### ➔ Declaration of arrays

- ➔ only with fixed size, e.g.:

```
int ary1[10]; // int array with 10 elements
double ary2[100][200]; // 100 * 200 array
int ary3[] = { 1, 2 }; // int array with 2 elements
```

- ➔ for parameters: size can be omitted for **first** dimension

```
int funct(int ary1[], double ary2[][200]) { ... }
```

#### ➔ Arrays can also be realized via pointers (see later)

- ➔ then also dynamic allocation is possible

#### ➔ Access to array elements

- ➔ like in Java, e.g.: `a[i][j] = b[i] * c[i+1][j];`
- ➔ but: **no** checking of array bounds!!!

## 1.1.2 Data types in C++ ...



### Classes and objects

#### ➔ Declaration of classes (typically in .h file):

```
class Example {
  private: // private attributes/methods
    int attr1; // attribute
    void pmeth(double d); // method
  public: // public attributes/methods
    Example(); // default constructor
    Example(int i); // constructor
    Example(Example &from); // copy constructor
    ~Example(); // destructor
    int meth(); // method
    int attr2; // attribute
    static int sattr; // class attribute
};
```



### Classes and objects ...

- ➔ Definition of class attributes and methods (\*.cpp file):

```
int Example::sattr = 123; // class attribute

Example::Example(int i) { // constructor
    this->attr1 = i;
}

int Example::meth() { // method
    return attr1;
}
```

- ➔ specification of class name with attributes and methods
  - ➔ separator :: instead of .
- ➔ this is a pointer (☞ 1.1.3), thus this->attr1
- ➔ alternatively, method bodies can also be specified in the class definition itself



### Classes and objects ...

- ➔ Declaration of objects:

```
{
    Example ex1; // initialisation using default constructor
    Example ex2(10); // constructor with argument
    ...
} // now the destructor for ex1, ex2 is called
```

- ➔ Access to attributes, invocation of methods

```
ex1.attr2 = ex2.meth();
j = Example::sattr; // class attribute
```

- ➔ Assignment / copying of objects

```
ex1 = ex2; // object is copied!
Example ex3(ex2); // initialisation using copy constructor
```



### Templates

- ➔ Somehow similar to generics in Java
  - i.e., classes (and methods) may have type parameters
  - however, templates are more powerful (and complex) than generics
- ➔ Main goal: allow to implement generic classes / data structures, e.g., lists
- ➔ Usage of templates:

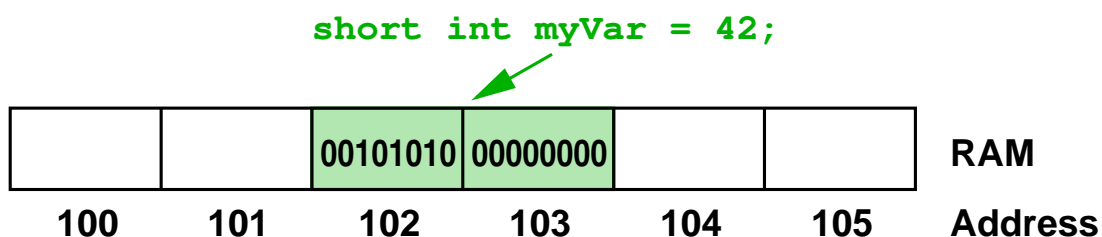
```
std::list<int> intlist;           // List of integers
intlist.push_back(42);          // Add at the end of the list
int i = intlist.front();        // First element
std::list<double> dblist;       // List of doubles
dblist.push_back(3.1415);
```

## 1.1.3 Pointers



### Variables in memory

- ➔ Reminder: variables are stored in main memory



- ➔ a variable gives a name and a type to a memory block
  - here: `myVar` occupies 2 bytes (`short int`) starting with address 102
- ➔ A **pointer** is a memory address, together with a type
  - the type specifies, how the memory block is interpreted

## Notes for slide 34:

C++ also has the concept of *references* and so called *smart pointers*. Since these concepts are not needed to solve the lab assignments, they are not discussed here.

34-1

## 1.1.3 Pointers ...

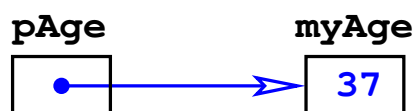


(Animated slide)

### Declaration and use of pointers

➔ Example:

```
int myAge = 25;    // an int variable
int *pAge;        // a pointer to int values
pAge = &myAge;    // pAge now points to myAge
*pAge = 37;       // myAge now has the value 37
```



- ➔ The **address operator** & determines the address of a variable
- ➔ The access to \*pAge is called **dereferencing** pAge
- ➔ Pointers (nearly) always have a type
  - ➔ e.g. int \*, Example \*, char \*\*, ...



### Passing parameters *by reference*

➔ Pointers allow to pass parameters *by reference*

➔ Instead of a value, a **pointer** to the values is passed:

```
void byReference(Example *e, int *result) {
    *result = e->attr2;
}
int main() {
    Example obj(15);           // obj is more efficiently
    int res;                  // passed by reference
    byReference(&obj, &res);  // res is a result parameter
    ...
}
```

➔ short notation: `e->attr2` means `(*e).attr2`



### void pointers and type conversion

➔ C++ also allows the use of generic pointers

➔ just a memory address without type information

➔ declared type is `void *` (pointer to void)

➔ Dereferencing only possible after a type conversion

➔ caution: no type safety / type check!

➔ Often used for generic parameters of functions:

```
void bsp(int type, void *arg) {
    if (type == 1) {
        double d = *(double *)arg; // arg must first be converted
                                    // to double *
    } else {
        int i = *(int *)arg;        // int argument
    }
}
```



### Arrays and pointers

- ➔ C++ does not distinguish between one-dimensional arrays and pointers (with the exception of the declaration)
- ➔ Consequences:
  - array variables can be used like (constant) pointers
  - pointer variables can be indexed

```
int a[3] = { 1, 2, 3 };  
int b = *a;           // equivalent to: b = a[0]  
int c = *(a+1);      // equivalent to: c = a[1]  
int *p = a;          // equivalent to: int *p = &a[0]  
int d = p[2];        // d = a[2]
```



### Arrays and pointers ...

- ➔ Consequences ...:
  - arrays as parameters are always passed *by reference!*

```
void swap(int a[], int i, int j) {  
    int h = a[i];    // swap a[i] and a[j]  
    a[i] = a[j];  
    a[j] = h;  
}  
int main() {  
    int ary[] = { 1, 2, 3, 4 };  
    swap(ary, 1, 3);  
    // now: ary[1] = 4, ary[3] = 2;  
}
```



### Dynamic memory allocation

- ➔ Allocation of objects and arrays like in Java

```
Example *p = new Example(10);  
int *a = new int[10];           // a is not initialised!  
int *b = new int[10]();         // b is initialised (with 0)
```

- ➔ allocation of multi-dimensional arrays does not work in this way

- ➔ Important: C++ does not have a garbage collection

- ➔ thus explicit deallocation is necessary:

```
delete p;    // single object  
delete[] a; // array
```

- ➔ caution: do not deallocate memory multiple times!



### Function pointers

- ➔ Pointers can also point to functions:

```
void myFunct(int arg) { ... }  
void test1() {  
    void (*ptr)(int) = myFunct; // function pointer + init.  
    (*ptr)(10);                // function call via pointer  
}
```

- ➔ Thus, functions can, e.g., be passed as parameters to other functions:

```
void callIt(void (*f)(int)) {  
    (*f)(123); // calling the passed function  
}  
void test2() {  
    callIt(myFunct); // function as reference parameter  
}
```

## 1.1.4 Strings and Output



- ➔ Like Java, C++ has a string class (`string`)
  - ➔ sometimes also the type `char *` is used
- ➔ For console output, the objects `cout` and `cerr` are used
- ➔ Both exist in the name space (packet) `std`
  - ➔ for using them without name prefix:  
`using namespace std; // corresponds to 'import std.*;' in Java`
- ➔ Example for an output:  
`double x = 3.14;`  
`cout << "Pi ist approximately " << x << "\n";`
- ➔ Special formatting functions for the output of numbers, e.g.:  
`cout << setw(8) << fixed << setprecision(4) << x << "\n";`
  - ➔ output with a field length of 8 and exactly 4 decimal places



# Parallel Processing

Winter Term 2025/26

20.10.2025

Roland Wismüller  
Universität Siegen  
roland.wismueller@uni-siegen.de  
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: October 21, 2025



## 1.1.5 Further specifics of C++



### ➔ Global variables

- ➔ are declared outside any function or method
- ➔ live during the complete program execution
- ➔ are accessible by all functions

### ➔ Global variables and functions can be used only **after** the declaration

- ➔ thus, for functions we have **function prototypes**

```
int funcB(int n);           // function prototype
int funcA() {                // function definition
    return funcB(10);
}
int funcB(int n) {           // function definition
    return n * n;
}
```

## 1.1.5 Further specifics of C++ ...



### ➔ Keyword `static` used with the declaration of global variables or functions

```
static int number;
static void output(char *str) { ... }
```

- ➔ causes the variable/function to be visible only in the local source file

### ➔ Keyword `const` used with the declaration of variables or parameters

```
const double PI = 3.14159265;
void print(const char *str) { ... }
```

- ➔ causes the variables to be read-only
- ➔ roughly corresponds to `final` in Java
- ➔ (note: this description is extremely simplified!)

## 1.1.5 Further specifics of C++ ...



➔ Passing command line arguments:

```
int main(int argc, char **argv) {  
    if (argc > 1)  
        cout << "Argument 1: " << argv[1] << "\n";  
}
```

Example invocation: `bslab1% ./myprog -p arg2`  
Argument 1: `-p`

- ➔ `argc` is the number of arguments (incl. program name)
- ➔ `argv` is an array (of length `argc`) of strings (`char *`)
- ➔ in the example: `argv[0] = "./myprog"`  
`argv[1] = "-p"`  
`argv[2] = "arg2"`
- ➔ important: check the index against `argc`

## 1.1.6 C/C++ Libraries



### Overview

➔ There are several (standard) libraries for C/C++, which always come with one or more header files, e.g.:

| Header file             | Library (g++ option)  | Description     | contains, e.g.  |
|-------------------------|-----------------------|-----------------|---|
| <code>iostream</code>   |                       | input/output    | <code>cout</code> , <code>cerr</code>                         |
| <code>string</code>     |                       | C++ strings     | <code>string</code>   |
| <code>stdlib.h</code>   |                       | standard funct. | <code>exit()</code>   |
| <code>sys/time.h</code> |                       | time functions  | <code>gettimeofday()</code>                                   |
| <code>math.h</code>     | <code>-lm</code>      | math functions  | <code>sin()</code> , <code>cos()</code> , <code>fabs()</code> |
| <code>pthread.h</code>  | <code>-pthread</code> | threads         | <code>pthread_create()</code>                                 |
| <code>mpi.h</code>      | <code>-lmpich</code>  | MPI             | <code>MPI_Init()</code>                                       |

## 1.1.7 The C Preprocessor



### Functions of the preprocessor:

- ➔ Embedding of header file

```
#include <stdio.h> // searches only in system directories
#include "myhdr.h" // also searches in current directory
```

- ➔ Macro expansion

```
#define BUFSIZE 100 // Constant
#define VERYBAD i + 1; // Extremely bad style !!
#define GOOD (BUFSIZE+1) // Parenthesis are important!
...
int i = BUFSIZE; // becomes int i = 100;
int a = 2*VERYBAD // becomes int a = 2*i + 1;
int b = 2*GOOD; // becomes int a = 2*(100+1);
```

## 1.1.7 The C Preprocessor ...



### Functions of the preprocessor: ...

- ➔ Conditional compilation (e.g., for debugging output)

```
int main() {
#ifdef DEBUG
    cout << "Program has started\n";
#endif
    ...
}
```

- ➔ output statement normally will not be compiled

- ➔ to activate it:

- ➔ either `#define DEBUG` at the beginning of the program
- ➔ or compile with `g++ -DDEBUG ...`



### Threads

- ➔ Activities within processes, concurrent to others
- ➔ Private resources:
  - ➔ CPU registers, including PC and stack pointer
  - ➔ local variables
- ➔ All other resources (esp. memory) are shared
- ➔ Threads are time-multiplexed to available CPU cores by the OS



### Synchronization

- ➔ Ensuring conditions on the possible sequences of events in threads
  - ➔ mutual exclusion
  - ➔ temporal order of actions in different threads
- ➔ Tools:
  - ➔ shared variables
  - ➔ semaphores / mutexes
  - ➔ monitors / condition variables
  - ➔ barriers



### Synchronization using shared variables

- ➔ Example: waiting for a result

#### Thread 1

```
// compute and
// store result
ready = true;
...
```

#### Thread 2

```
while (!ready); // wait
// read / process the result
...
```

- ➔ Extension: atomic *read-modify-write* operations of the CPU
  - ➔ e.g., *test-and-set*, *fetch-and-add*
- ➔ Potential drawback: *busy waiting*
  - ➔ but: in high performance computing we often have exactly one thread per CPU  $\Rightarrow$  performance advantage, since no system call



### Semaphores

- ➔ Components: counter, queue of blocked threads
- ➔ **Atomic** operations:
  - ➔ P() (also acquire, wait or down)
    - ➔ decrements the counter by 1
    - ➔ if counter  $< 0$ : block the thread
  - ➔ V() (also release, signal or up)
    - ➔ increments counter by 1
    - ➔ if counter  $\leq 0$ : wake up one blocked thread
- ➔ **Binary semaphore**
  - ➔ can only assume the positive values 0 and 1
  - ➔ usually for mutual exclusion



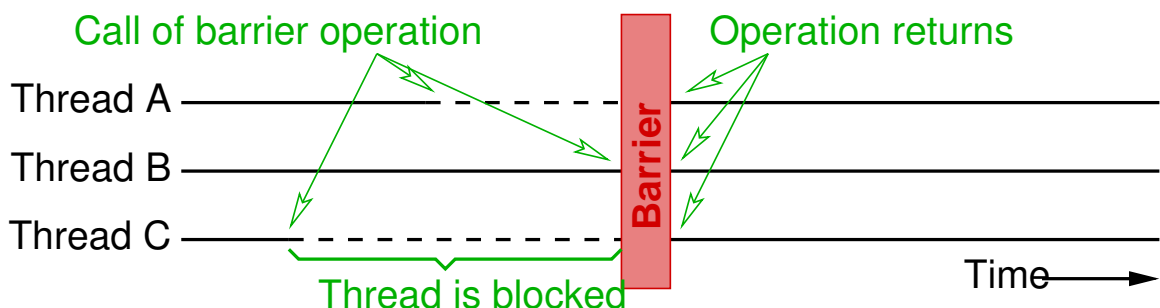
### Monitors

- ➔ Module with data, procedures and initialization code
  - access to data only via the monitor procedures
  - (roughly corresponds to a class)
- ➔ All procedures are under mutual exclusion
- ➔ Further synchronization via **condition variables**
  - two operations:
    - `wait()`: blocks the calling thread
    - `signal()`: wakes up some blocked threads
      - variants: wake up only one thread / wake up all thread
  - no "memory": `signal()` only wakes a thread, if it already has called `wait()` before



### Barrier

- ➔ Synchronization of groups of processes or threads, respectively
- ➔ Semantics:
  - thread which reaches the barrier is blocked, until all other threads have reached the barrier, too



- ➔ Used to structure concurrent applications into synchronous phases



### Synchronization errors

- ➔ Insufficient synchronization: *race conditions*
  - result of the calculation is different (or wrong), depending on temporal interleaving of the threads
  - important: do not assume FIFO semantics of the queues in synchronization constructs!
- ➔ Deadlocks
  - a group of threads waits for conditions, which can only be fulfilled by the other threads in this group
- ➔ Starvation (unfairness)
  - a thread waiting for a condition can never execute, although the condition is fulfilled regularly



### Example for *race conditions*

- ➔ Task: synchronize two threads, such that they print something alternately
- ➔ **Wrong** solution with semaphores:

```
Semaphore s1 = 1;  
Semaphore s2 = 0;
```

#### Thread 1

```
while (true) {  
    P(s1);  
    print("1");  
    V(s2);  
    V(s1);  
}
```

#### Thread 2

```
while (true) {  
    P(s2);  
    P(s1);  
    print("2");  
    V(s1);  
}
```



- ➔ Part of the C++ language standard since 2011 (C++-11)
  - implemented by the compiler and the C++ libraries
  - independent of operating system
- ➔ Programming model:
  - at program start: exactly one (master) thread
  - master thread creates other threads and should wait for them to finish
  - process terminates when master thread terminates
    - when other threads are still running, an error is raised



### Creating threads

- ➔ Class `std::thread`
  - represents a running thread
- ➔ Creation of a new thread (both C++-object and OS thread):  
`std::thread myThread(function, args ...);`
  - with this declaration, the C++ object (and the OS thread) is automatically destroyed when the current scope is left
  - *function*: the function that should be executed by the thread
  - *args* ...: any number of parameters, which will be passed to *function*
  - *function* cannot have a return value
    - use result parameters instead



### Methods of class `thread` (incomplete)

- ➔ `void join()`
  - ➔ waits until the thread execution has completed
  - ➔ after this method returns, the thread can be destroyed safely
- ➔ `void detach()`
  - ➔ detach the OS thread from the C++ thread object
  - ➔ the OS thread will continue its execution, even when the thread object is destroyed
  - ➔ the thread cannot be joined any more



### Example: Hello world (📄 01/helloThread.cpp)

```
#include <iostream>
#include <thread>

void sayHello()
{
    std::cout << "Hello World!\n";
}

int main(int argc, char **argv)
{
    std::thread t(sayHello);
    t.join();
    return 0;
}
```

## 1.3 C++ Threads ...



### Example: Summation of an array with multiple threads

(👉 01/sum.cpp)

```
#include <iostream>
#include <thread>

#define N 5
#define M 1000

/* This function is called by each thread */
void sumRow(int *row, long *res)
{
    int i;
    long sum = 0;
    for (i=0; i<M; i++)
        sum += row[i];

    *res = sum;    /* return the sum. */
}
```

## 1.3 C++ Threads ...



```
/* Initialize the array */
void initArray(int array[N][M])
{
    ...
}

/* Main program */
int main(int argc, char **argv)
{
    int array[N][M];
    int i;
    std::thread threads[N];
    long res[N];
    long sum = 0;

    initArray(array);    /* initialize the array */
}
```

```
/* Create a thread for each row and pass the pointer to the row and the
pointer to the result variable as an argument */
for (i=0; i<N; i++) {
    threads[i] = std::thread(sumRow, array[i], &res[i]);
}

/* Wait for the threads' termination and sum the partial results */
for (i=0; i<N; i++) {
    threads[i].join();
    sum += res[i];
}

std::cout << "Sum: " << sum << "\n";
}
```

### Compile and link the program

```
➔ g++ -o sum sum.cpp -pthread
```

### Notes for slide 63:

In C++, the statement

```
threads[i] = std::thread(...);
```

actually means:

1. create a new (temporary) object of class `std::thread` by calling the proper constructor,
2. copy the object into the array,
3. delete the temporary object from step 1.

For threads, this sequence works due to the special implementation of the `std::thread` class, that overrides the assignment operator in such a way, that the OS thread is *not* copied and/or destroyed along with the C++ object.

A consequence of this assignment is that if the array `threads` is destroyed (because execution leaves the block where it has been declared), the threads are also destroyed.



### Remarks on the example

- ➔ When creating the thread, any number of parameters can be passed to the thread function
  - the thread function will store its result there
  - caution: since `res` is a local variable, the threads must be joined before the method exits
- ➔ No synchronization (other than `join()`) is required
  - each thread stores to a different element of `res`
- ➔ With `join()`, we can only wait for a specific thread
  - inefficient, when the threads have different execution times



### Synchronization: mutex variables

- ➔ Behavior similar to a binary semaphore
  - states: locked, unlocked; initial state: unlocked
- ➔ Declaration (and initialization):  
`std::mutex mutex;`
- ➔ To lock the mutex, create an object of class `std::unique_lock`:
  - `std::unique_lock<std::mutex> lock(mutex);`
  - the mutex is automatically unlocked when `lock` is destroyed, i.e., when execution leaves the current block
- ➔ Class `mutex` does not allow recursive locking
  - i.e., the same thread cannot lock the mutex twice
  - use class `recursive_mutex` for this purpose

## Notes for slide 65:

Please see the C++ reference for more information about the mutex and lock classes, e.g. <http://www.cplusplus.com/reference/mutex/>.

There are, for instance, also classes `timed_mutex` and `recursive_timed_mutex` which enable to have a timeout when trying to lock the mutex.

65-1

## 1.3 C++ Threads ...



### Synchronization: condition variables

- ➔ Declaration (and initialization):  
`std::condition_variable cond;`
- ➔ Important methods:
  - ➔ wait: `void wait(unique_lock<mutex>& lock)`
    - ➔ thread is blocked, the mutex wrapped by `lock` will be unlocked temporarily
    - ➔ signaling thread keeps the mutex, i.e., the signaled condition may no longer hold when `wait()` returns!
    - ➔ typical use: `while (!condition_met) cond.wait(lock);`
  - ➔ signal just one thread: `void notify_one()`
  - ➔ signal all threads: `void notify_all()`

## Notes for slide 66:

The syntax `unique_lock<mutex>& lock` indicates that the argument of the method `wait()` is passed by reference rather than by value.

66-1



---

# Parallel Processing

Winter Term 2025/26

21.10.2025

Roland Wismüller  
Universität Siegen  
roland.wismueller@uni-siegen.de  
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: October 21, 2025

### Example: simulating a monitor with C++ threads

```
#include <thread> (👉 01/monitor.cpp)
#include <mutex> // Defines std::mutex
#include <condition_variable> // Defines std::condition_variable

std::mutex mutex;
std::condition_variable cond;
volatile int ready = 0;
volatile int result;

void storeResult(int arg) {
    std::unique_lock<std::mutex> lock(mutex);
    result = arg; /* store result */
    ready = 1;
    cond.notify_all();
    // The 'lock' object is destroyed when the method ends, thus unlocking the mutex!
}
```

#### Notes for slide 67:

The keyword `volatile` at the beginning of the declarations of global variables indicates the **compiler** must actually perform any read and write operation programmed in the source code (in the given order), i.e., the compiler must not apply any optimizations to this variable (especially “caching” the value in a register). This is necessary here, as the variables can be modified at any time by another thread.

Note that `volatile` does **not** imply sequential consistency, since it only imposes a restriction on the compiler, not on the CPU.

### Example: simulating a monitor with C++ threads ...

```
int readResult()  
{  
    std::unique_lock<std::mutex> lock(mutex);  
    while (ready != 1)  
        cond.wait(lock);  
    return result; // mutex unlocked automatically when 'lock' is destroyed.  
}
```

- ➔ `while` is important, since the waiting thread unlocks the `mutex`
- ➔ another thread could destroy the condition again before the waiting thread regains the `mutex`  
(although this cannot happen in this concrete example!)

#### Notes for slide 68:

Note that the C++ standard allows `wait()` to return even in cases where the condition has **not** been signalled. Thus, you always must use a `while` loop!