Object Oriented Scientific Programming in C++ (WI4771TU)

Matthias Möller
Numerical Analysis

TUDelft

Course information

- Lectures
  - Weeks 3.1-3.7, Tue 10:45-12:45
- Lab sessions
  - Weeks 3.1-3.7, Fri 13:45-17:45 in DW-PC1
  - This is the time and place to ask your questions (no office hours)
- Assessment (3 ECTS)
  - Weekly homework to be worked on individually (1/3 grade)
  - Final project to be worked on in teams (2/3 grade)

Learning objectives

- You will
  - learn **concepts** of object oriented (scientific) programming (classes, template meta programming, polymorphism,…)
  - learn **programming techniques** to realise OOP in C++11
  - learn to use **software tools** (Git) for programming in team
  - learn to use effective **programming workflow** (cmake, doxygen)

Disclaimer

- This is no classical C++ course
  - We will focus on concepts (and not on C++ in all its ‘beauty’)
  - We will learn C++ to the level needed for solving practical problems in scientific computing (don‘t expect GUI design or a fully formal definition of all language constructions)
- This course expresses my personal view
  - Choice of ‘best’ version control/build system depends on project admin and/or personal flavour; I will teach you mine
  - There are always multiple ways to implement an algorithm

Think …

- **Big**
  If your implementation works for a scalar integer value ask yourself what will happen for a gigabyte of data
- **Future**
  Better invest more time now to develop a future-proof implementation than rewrite your code next month
- **Fail-safe**
  Write code that can handle invalid inputs without failure

Program in week 1

- Introduction to the **version control system Git**
  – All examples/assignments will be distributed via Git only
- Introduction to the **build system CMake**
  – All examples/assignments make use of this build system and provide a predefined CMakeLists.txt file doing the job for you
- Short **crash course on C++**
  – Constants, variables, namespaces, and scopes
  – Pointers, references, and dynamic memory allocation

Distributed Version Control System

INTRODUCTION TO GIT

Git – a distributed version control system

- Helps you to **keep track of your file history**
  - Simple mechanism to organise your homework
  - Automatically detect revision that introduced a bug
  - Straightforward synchronisation of your work on multiple computers
- Helps you to **develop in a team**
  - Powerful tool to merge contributions from different team members
  - Easy handling of conflicting changes by different team members
  - Add-ons for measuring team performance
- Helps you **to share your work** with the world
  - Quasi standard for open-source projects (github, gitlab, …)

Git – further reading

- Git homepage: https://git-scm.com
- Git tutorial: https://www.atlassian.com/git/
- Git compendium: https://git-scm.com/book/en/v2
  (if you want to know all about Git and that‘s a lot)

Get started with Git

- Open a terminal and **configure** Git

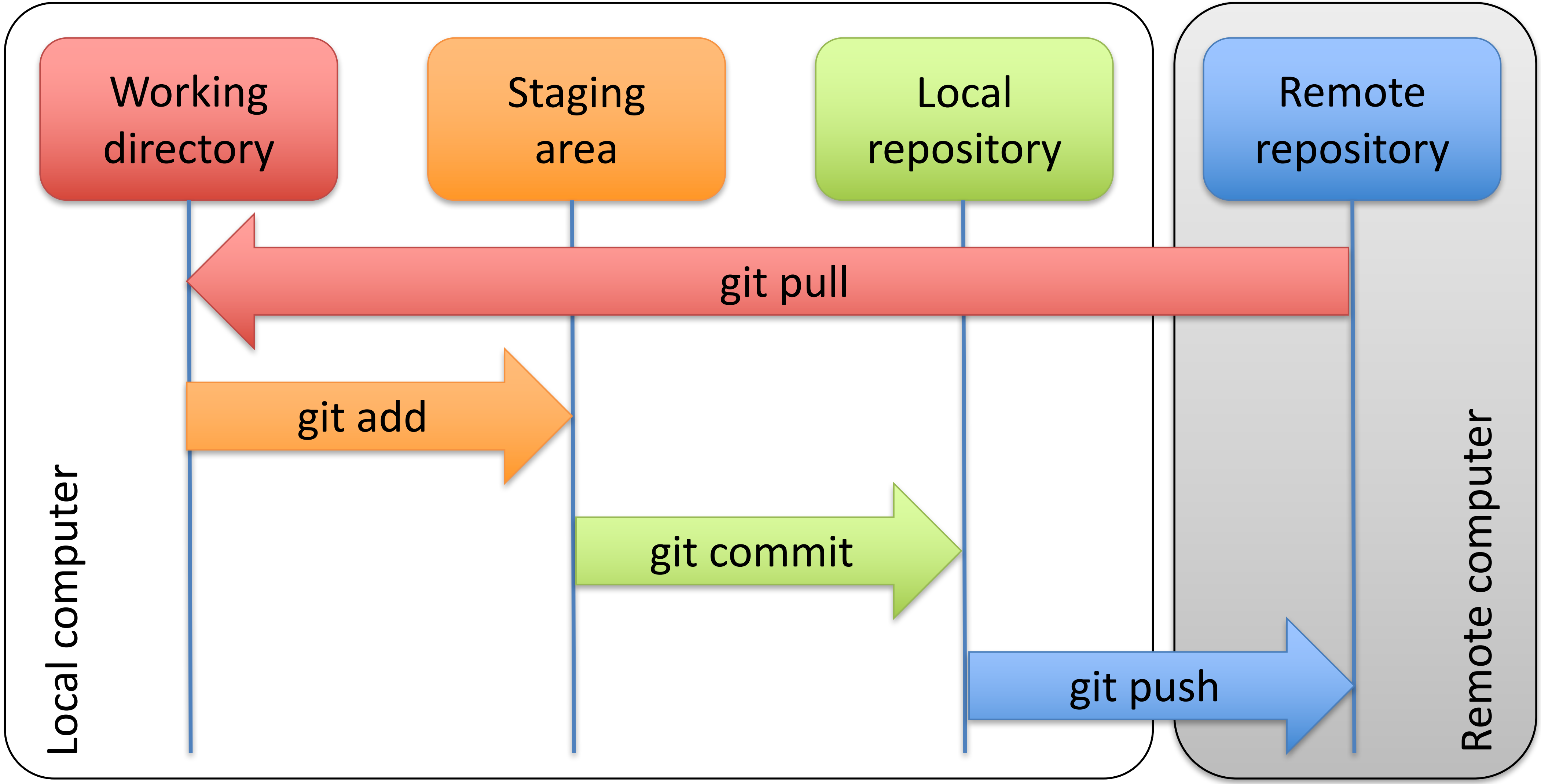
```
$> git config --global user.name “Your Name“
$> git config --global user.email you@somewhere.nl
```

- **Clone** the remote Git project for this course (only once)

```
$> git clone https://gitlab.com/mmoelle1/wi4771tu.2017.git
$> cd wi4771tu.2017.git
$> ls
README.md
01-hello
02-variables-constants
03-namespaces
04-pointers
```

TUDelft

Git workflow



TU Delft

Pull, add, commit ... that‘s it

- Update your current checkout (before each lab session)
  ```
  $> git pull
  ```
- Show status of all files in your **working directory**
  ```
  $> git status
  ```
- Add new or modified file to your **staging area**
  ```
  $> touch hello.cpp
  $> git add hello.cpp
  ```
- Commit new or modified file to your **local repository**
  ```
  $> git commit –m „Added hello.cpp“
  ```

Local -> remote repository

- Your local repository contains my files (provided via the remote repository) and your personal additions
- If you want to transfer your local contributions to the remote repository you need to push the changes
  ```
  $> git push
  ```
- Push does not work for my repository for obvious reasons (more on this topic in the first lab session)

Resolving conflicts

- Git tries to automatically resolve conflicts in a file that changed both in local and remote repository if possible

Common checkout:
Hello World

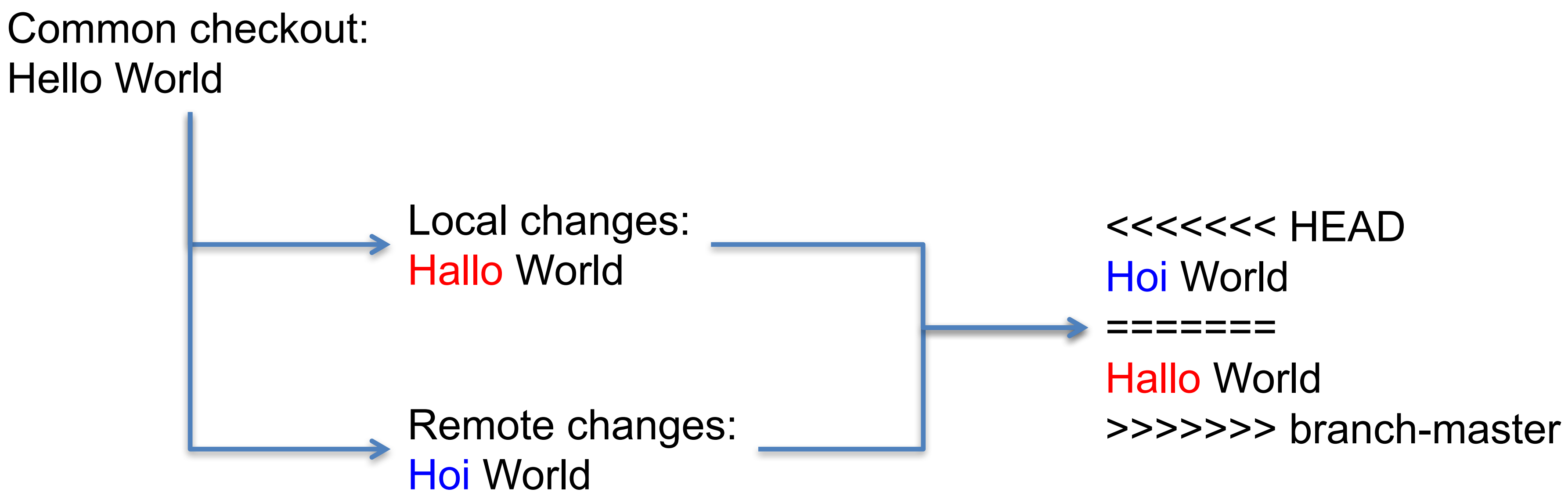
Local changes:
Hallo World

Remote changes:
Hello Wereld

Merged changes:
Hallo Wereld

Resolving conflicts, cont'd

- If Git cannot resolve conflicts automatically it is your responsibility to edit the conflicting file and commit it



Summary on git

- In the very first lab session

```
$> git config --global user.name “Your Name“
$> git config --global user.email you@somewhere.nl
$> git clone https://gitlab.com/mmoelle1/wi4771tu.2017.git
```

- At the beginning of each lab session update your checkout

```
$> git pull
```

- At the end of each lab session save your work

```
$> git status
$> git add <all new and modified files>
$> git commit –m “My changes in week X”
```

Universal Build System

INTRODUCTION TO CMAKE

CMake – a universal build system

- Platform-independent build system with simple structure
  - CMakeLists.txt file in *top-level directory* defines *global* build process (project name, compiler settings, etcetera)
  - CMakeLists.txt file in each *subdirectory* describes *local* build process (file dependency, linking of libraries, etcetera)
- Tutorial: https://cmake.org/cmake-tutorial/
- Documentation: https://cmake.org/cmake/help/v3.0/index.html

TUDelft

Top-level CMakeLists.txt

- Comments start with ‘#’
```
# Force CMake version 3.1 or above
```
- Each CMake version brings new features; define minimal version if you rely on features not available in earlier ones
```
cmake_minimum_required (VERSION 3.1)
```
- Define global project name
```
project (wi4771tu.2017)
```

Top-level CMakeLists.txt

- Write out messages
  ```
  message("Build all targets of the course wi4771tu”)
  ```
- Add subdirectories where to look for CMakeLists.txt files
  ```
  add_subdirectory(01-hello)
  add_subdirectory(02-variables-constants)
  add_subdirectory(03-pointers)
  add_subdirectory(04-passing-arguments)
  add_subdirectory(05-namespaces)
  ```
- Documentation:
  https://cmake.org/cmake/help/v3.3/index.html

TUDelft

CMakeLists.txt in subdirectory

- Create an executable named `hello` from the source file `hello.cxx` stored in the subdirectory `src`

```
add_executable(hello src/hello.cxx)
```

- If you use special C++11 (or 14) features that are not supported by all compilers (by default) tell CMake

```
target_compile_features(hello PRIVATE cxx_auto_type)
```

- Documentation: https://cmake.org/cmake/help/v3.1/prop_gbl/CMAKE_CXX_KNOWN_FEATURES.html

Out-of-source builds

- Create a build directory to keep sources clean and type

```
$> mkdir build
$> cd build
$> cmake <path to top-level CMakeLists.txt files>
```

- CMake will create a Makefile with all compiler settings; to build your entire project or individual build targets call

```
$> make <target-name>
$> make <TAB-key>      // lists all build targets
```

- When you are done, just remove the entire build directory

```
$> cd <path to build directory>/..
$> rm -rf build
```

TUDelft

Crash course on C++

C++ THE VERY BASICS

Comments

- Comments in C++

```
/**
 * \file hello.cxx
 *
 * This file is part of the course wi4771tu:
 * Object Oriented Scientific Programming with C++
 *
 * \author Matthias Moller
 */
```

- Tags (`\file`) are keywords for **Doxygen** (later in this course)

Comments, cont‘d

- Comments in C++

```
/**
   \file hello.cxx

   This file is part of the course wi4771tu:
   Object Oriented Scientific Programming in C++

   \author Matthias Moller
 */
```

TUDelft

Comments, cont'd

- Comments in C++

```
//
// \file hello.cxx
//
// This file is part of the course wi4771tu:
// Object Oriented Scientific Programming in C++
//
// \author Matthias Moller
```

- Recommendation: use `/**...*/` for multi-line comments and `//` for short comments also at the end of a code line

TUDelft

Hello.cxx

```
// Include header file for standard input/output stream library
#include <iostream>
/**
  * The global main function that is the designated start of the program
  */
int main(){
  /**
     Write the string 'Hello World' to the default output stream and
     terminate with a new line (that is what std::endl does)
    */
  std::cout << "Hello world!" << std::endl;

  return 0; // Return code 0 to the operating system (=no error)
}
```

Inclusion of header files

- Extra functionality provided by the standard C++ library is defined in so-called **header files** which need to be included
  `#include <headerfile>`
  - iostream: input/output
  - string: string types
  - complex: complex numbers
  - Good overview: http://www.cplusplus.com
- We will write our own header files later in this course

TUDelft

THE main function

- Each C++ program must provide one (and only one) global **main function** which is the designated start of the program

```
int main() { body } or
int main(int argc, char * argv[]) { body }
```

- Scope of the main function { … }
- Return type of the main function is `int` (=integer)

```
return 0;
```

- Main function cannot be called recursively

Standard output

- Stream-based output system
  `std::cout << "Hello world!" << std::endl;`
- Streams can be easily concatenated
  `std::cout << "Hello" << " " << "world!" << std::endl;`
- Streams are part of the standard C++ library and therefore encapsulated in the **namespace** `std` (later in this lecture)
- Predefined output streams
  - `std::cout`: standard output stream
  - `std::cerr`: standard output stream for errors
  - `std::clog`: standard output stream for logging

Variables and constants

- C++ is case sensitive and typed, that is, variables and constants have a value and a concrete type

```
      int a = 10; // integer variable initialised to 10
const int b = 20; // integer constant initialised to 20
```

- Variables can be updated, constants cannot

```
a = b;
b = a; // will produce compiler error since b is constant
```

- Variables and constants can be defined everywhere

```
      int A = b;
const int B = a;
```

Initialisation of variables/constants

- Constants must be initialised during their **definition**

```
const int c = 20;    // C-like initialisation
const int d(20);     // constructor initialisation
const int e = {20}; // uniform initialisation, since C++11
```

- Variables can be initialised during their **definition** or (since they are variable) at any location later in the code

```
int f = 10;          // C-like initialisation
int g(20);           // constructor initialisation
int h = {20};        // uniform initialisation, since C++11
int i;               // only declaration
i = 20;
```

Intermezzo: Terminology

- A **declaration** introduces an identifier, e.g., a variable or constant, and describes its type but does not create it
  ```
  extern int f;
  ```
- A **definition** instantiates this identifier
  ```
  int f;
  ```
- An **initialisation** initialises this identifier
  ```
  f = 10;
  ```

- Common usage: `int f = 10;`

Scope of variables/constants

- Variables/constants are only visible in their scope

```
int main() {
  int a = 10;  // variable a is visible here
  {
    int b = a; // variable a is visible here
  }
  {
    int c = b; // variable b is not visible here
  }
}
```

TUDelft

Scope of variables/constants, cont‘d

- Variables/constants are only visible in their scope

```
int main() {
  int a = 10;    // variable a is visible here
  {
    int a = 20; // variable a only visible in blue scope,
                // interrupts scope of red variable a
    std::cout << a << std::endl; // is 20
  }
  std::cout << a << std::endl; // is 10
}
```

TUDelft

C++ standard types

Group	Type name	Notes on size / precision
Character types	char	Exactly one byte in size. At least 8 bits.
	char16_t	Not smaller than char. At least 16 bits.
	char32_t	Not smaller than char16_t. At least 32 bits.
Integer types (signed and unsigned)	(un)signed char	Same size as char. At least 8 bits.
	(un)signed short int	Not smaller than char. At least 16 bits.
	(un)signed int	Not smaller than short. At least 16 bits.
	(un)signed long int	Not smaller than int. At least 32 bits.
	(un)signed long long int	Not smaller than long. At least 64 bits.
Floating-point type	float	
	double	Precision not less than float
	long double	Precision not less than double
Boolean type	bool	

TUDelft

Mixing and conversion of types

- Get the type of a variable

```
#include <typeinfo>
float F = 1.7; double D = 0.7;
std::cout << typeid(F).name() << std::endl;    // float
std::cout << typeid(D).name() << std::endl;    // double
```

- C++ converts different types automatically

```
std::cout << typeid(F+D).name() << std::endl; // double
```

- Check if you are happy with the results

```
char x = 'a'; float y = 1.7;
std::cout << typeid(x+y).name() << std::endl; // float???
```

Mixing and conversion of types, cont'd

- C++11 introduces the auto keyword, which makes handling of mixed types very easy (other advantages will follow)

```
auto X = F+D;
std::cout << typeid(X).name() << std::endl;    // double
```

- You can also explicitly cast one type into another

```
auto Y = F + (float)D;
std::cout << typeid(Y).name() << std::endl;    // float
auto Z = (int) ((double)F + (float)D);
std::cout << typeid(Z).name() << std::endl;    // int
```

TUDelft

Auto vs. explicit type

- I would use the **keyword auto** as much as possible ...
  - to improve readability of the source code
  - to improve maintainability of the source code
  - to benefit from performance gains (later in this course)
- ... unless explicit conversion is required

```
int a = 1.5+0.3; // is (int)1.8 = 1
```

- Think future! Rewrite all code for change in variable?
- Think fail-safe! Explicit types may lead to implicit casts that you may not notice. Better let auto do the job!

Address-of/dereference operators

- Address-of operator (&) returns the address of a variable

```
int i = 10;
auto address = &i
std::cout << address << std::cout; // e.g. 0x7fff5def84f4
```

- The type of a variable address is an integer pointer

```
std::cout << typeid(i).name()       << std::endl; //  i
std::cout << typeid(address).name() << std::endl; // Pi
```

- Dereference operator (*) returns value behind the pointer

```
std::cout << *address << std::endl; // 10
```

Pointers and references

- Pointers can be used to have multiple variables (with different names) pointing to the same value

```
int i = 1;
int *p = &i;
```

- Change the value of variable i

```
i = 2; std::cout << *p << std::endl; // *p is 2
```

- Dereference pointer p and change its value

```
*p = 3; std::cout << i << std::endl; // i is 3
```

- Change value of pointer p without dereferencing

```
p = p+1; std::cout << *p << std::endl; // *p is 1449186548
```

Pointers and references, cont‘d

Address	Initialisation	Change i	Change *p	Change p
0x00 (i)	i = 1	i = 2	i is 3	
0x04	1449186548	1449186548	1449186548	1449186548
…				
0x10 (*p)	p -> 0x00 (*p is 1)	p -> 0x00 (*p is 2)	p -> 0x00 (*p = 3)	p -> 0x04 (*p is 1449186548)

Word of caution: When you change the pointer you may very easily point to a non-initialised location in memory which can cause all sorts of hazards.

Pointer hazards

- Pointers that remain uninitialised can cause hazard

```
int *p;
std::cout <<  p << std::endl; // is 0x7fff50e39500
std::cout << *p << std::endl; // is 1449186548
```

- C++11 introduces the new keyword nullptr

```
int *p = nullptr;
std::cout <<  p << std::endl; // is 0x0
std::cout << *p << std::endl; // yields Segmentation fault
```

- Think fail-safe!

```
std::cout << (p ? *p : NULL) << std::endl;
```

Argument passing – by value

- Arguments are passed by value (C++ default behaviour)

```
int i = 1;              // first  variable i=1
int j = addOne(i);      // second variable j=2 (i=1 still)

int addOne(int a)       // third variable a (=copy of i)
{
   return a+1;          // increment a by one and copy result
                        // to variable j on return
}
```

- Think big! Do you want to copy gigabytes of data?

Argument passing – by reference

- Arguments can be passed by reference

```
int i = 1;             // first variable  i=1
int j = addOne(i);     // second variable j=2 (i=1 still)

int addOne(int& a)     // a is reference to first variable
{
    return a+1;        // copy result of a+1
                       // to variable j on return
}
```

- Think big! Fine, we saved one copy.
- Think fail-safe! What if addOne tries to hijack a(=i)?

Argument passing – by reference, cont‘d

- Arguments can be passed by reference

```
int i = 1;            // first variable  i=1
int j = addOne(i);    // second variable j=1 (i=2)

int addOne(int& a)    // a is reference to first variable
{
   return a++;        // copy value of a to variable j on
                      // return and increment a by one
}
```

- **Post-increment operator** a++

TUDelft

Argument passing – by reference, cont‘d

- Arguments can be passed by reference

```
int i = 1;              // first variable  i=1
int j = addOne(i);      // second variable j=2 (i=2)

int addOne(int& a)      // a is reference to first variable
{
   return ++a;          // increment value of a by one and
                        // copy result to variable j on return
}
```

- **Pre-increment operator** ++a

Argument passing – by const reference

- Arguments can be passed by constant reference

```
int i = 1;            // first variable  i=1
int j = addOne(i);    // second variable j=2 (i=1 still)

int addOne(const int& a)  // a is const reference to a
{
   return a+1;          // copy result of a+1
                        // to variable j on return
}
```

- Think big! Fine, we saved one copy.
- Think fail-safe! Fine, a++ and ++a yield compiler error.

TU Delft

C++ return value optimization (RVO)

- Most C++ compilers support RVO, that is, they eliminate the creation of a temporary object to hold a function's return value. The previous example thus reduces to

```
int i = 1;             // first variable  i=1
int j = i+1;           // second variable j=2 (i=1 still)
```

- Think big! We finally saved two copies.

Argument passing, cont'd

- If we want a function that changes the argument directly we must pass the argument by reference

```
int i = 1;        // i=1
addOne_Val(i);    // i=1 (still)
addOne_Ref(i);    // i=2

void addOne_Val(int a)  { a++; } // increment local copy
void addOne_Ref(int& a) { a++; } // increment a(->i)
```

TUDelft

Static arrays

- Definition of static array

```
int array[5];
```

- Definition and initialisation of static array

```
int array[5] = { 1, 2, 3, 4, 5 }; // in C++11
```

- Access of individual array positions

```
for (auto i=0; i<5; i++)
   std::cout << array[i] << std::endl;
```

- Remember that C++ starts indexing at 0

Dynamic arrays

- Definition and allocation of dynamic array
```
int *array = new int[5];
```
- Definition , allocation and initialisation of dynamic array
```
int *array = new int[5] = { 1, 2, 3, 4, 5 }; // in C++11
```
- Deallocation of dynamically allocated array
```
delete[] array;
```
- Think fail-safe!
```
array = nullprt;
```

TUDelft

Namespaces

- Namespaces, like `std`, allow to bundle functions even with the same function name (and interface) into logical units

```
namespace tudelft {
    void hello() {
        std::cout << "Hello TU Delft" << std::endl;
    }
}
namespace other{
    void hello() {
        std::cout << "Hello other" << std::endl;
    }
}
```

Namespaces, cont'd

- Functions in namespaces can be called
  - with explicit use of namespace

```
tudelft::hello();
other::hello();
```

  - with keyword `using` (here visible only in respective scope {...})

```
{
    using namespace tudelft;
    hello();
}
{
    using namespace other;
    hello();
}
```

TUDelft