

# TW3720TU: Object Oriented Scientific Programming with C++ (11/14/17)

Matthias Möller  
Numerical Analysis

# Course information

- Lectures
  - Weeks 2.1-2.7, Tue 15:45-17:30
- Lab sessions
  - Weeks 2.1-2.8, Thu 13:45-17:45 in IO-PC hall 2 (**week 2.2/2.6!**)
  - 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++11/14/17
  - learn to use **software tools** (Git) for programming in team
  - learn to use **software tools** (cmake) for building and testing code across different platforms (Windows/Linux/macOS)
  - Learn to use **software tools** (doxygen) for documenting code

# 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

# 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

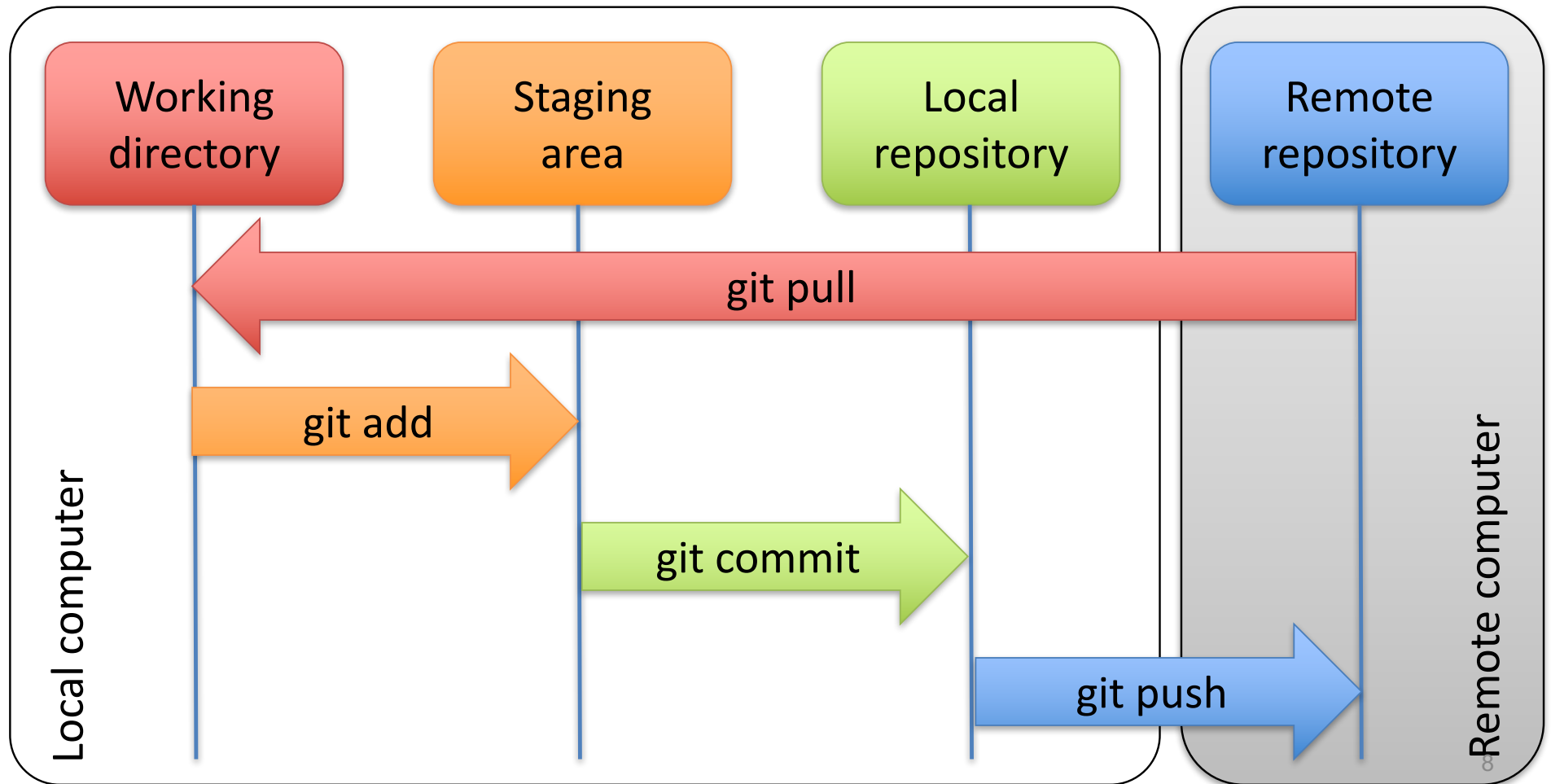
Distributed Version Control System

# INTRODUCTION TO GIT

# What is Git and why should I use it?

- **Git** is a distributed version control system
  - Automatic **backup** of your work with full development history
  - Automatic **roll back** to the last working version of your code
  - Automatic **merging** of your work with other peoples' work
  - Automated **testing** of each commit (*Continuous Integration*) possible together with services like Travis, GitLab, etc.

# Git workflow





# Getting 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/tw3720tu.2017.git
```

```
$> cd tw3720tu.2017.git
```

```
$> ls
```

```
README.md
```

```
01-hello
```

```
02-variables-constants
```

```
03-namespaces
```

```
...
```

# 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**

```
$> edit hello.cpp
```

```
$> git add hello.cpp
```

- Commit new or modified file to your **local repository**

```
$> git commit -m „Added hello.cpp“
```

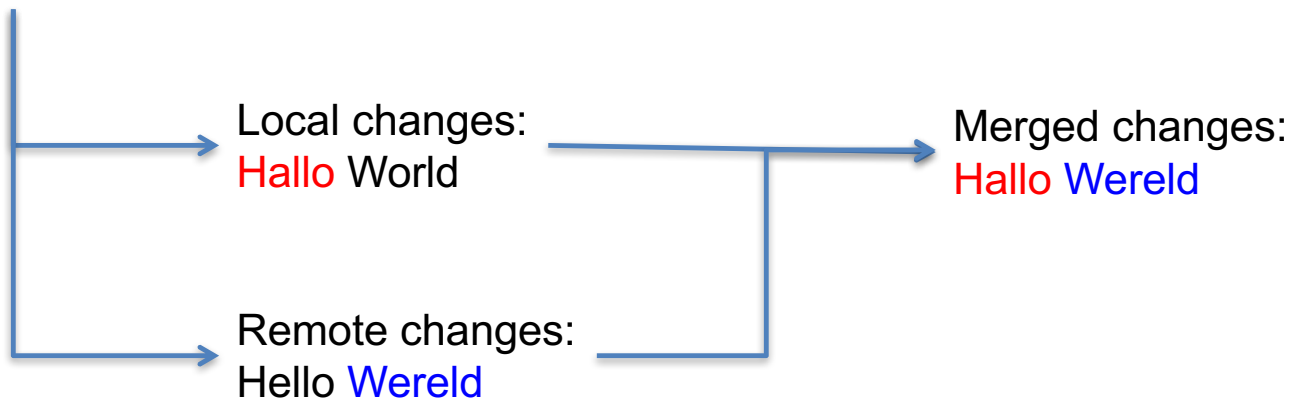
## Local -> remote repository

- 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`
- Pushing your work into my repository is not allowed for obvious reasons but with your own projects it will work

# Resolving conflicts

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

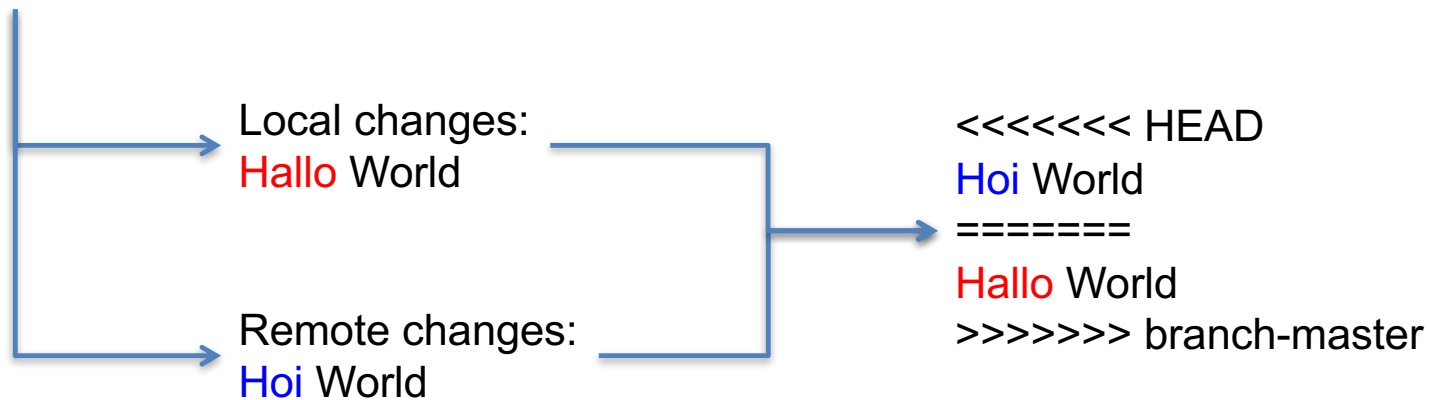
Common checkout:  
Hello World



# Resolving conflicts, cont'd

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

Common checkout:  
Hello World



# Finding the commit that broke the code

- Commit history

```
$> git log1
3218ea8 Changed feature2
636da4e Changed feature1
34efd65 New feature2
15777b1 Changed feature1
6cb06a3 New feature1
2e38302 Initial commit
```

- 'Walk' back in time starting from the current 'bad' commit

```
$> git bisect start
$> git bisect bad
```
- If this commit is 'good' run

```
$> git bisect good
```

otherwise run

```
$> git bisect bad
```
- Once you found the commit that broke the code clean up

```
$> git bisect reset
```

# Summary on git

- In the very first lab session (or on a new computer)  

```
$> git config --global user.name "Your Name"  
$> git config --global user.email you@somewhere.nl  
$> git clone https://gitlab.com/mmoelle1/tw3720tu.2017.git
```
- At the beginning of each lab session update your checkout  

```
$> git pull
```
- At the end of each lab session save your work locally  

```
$> git status  
$> git add <all new and modified files>  
$> git commit -m "My changes in week X"
```

# 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)



Universal Build System

# INTRODUCTION TO CMAKE

# What is CMake and why should I use it?

- **CMake** is a cross-platform code building and testing tool
  - Human-readable text-based description of **build dependencies** (source files, includes, executables, libraries, documentation, etc.)
  - Automatic detection of **build environment** (hardware, compiler, OS)
  - Automatic generation of **build rules** for different build systems (Xcode, MS VisualStudio, Makefile, Ninja, CodeBlocks, etc.)
  - Easily extendable by user-defined macros (cf. [add\\_assignment](#))
  - Provides fully integrated **testing environment**

# 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 project name  
`project (tw3720tu.2017)`
- Define C++ standard globally  
`set (CXX_STANDARD 11)`

# Top-level CMakeLists.txt

- Write out messages  
`message("Build all targets of the course tw3720tu")`
- 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>

## 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)`
- The C++ standard can also be specified per target  
`set_property(TARGET hello PROPERTY CXX_STANDARD 11)`

# 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
```

# Build systems

- CMake supports different build systems, e.g.

```
$> cmake <...> -G Makefile
```

```
$> cmake <...> -G Xcode
```

```
$> cmake <...> -G CodeBlocks
```

- With out-of-source builds you can even use multiple build systems at the same time, e.g.,

```
$> mkdir build-make
```

```
$> cd build-make
```

```
$> cmake <...> -G Makefile
```

```
$> mkdir build-xcode
```

```
$> cd build-xcode
```

```
$> cmake <...> -G Xcode
```

# Testing with CTest

- CTest is enabled in the top-level CMakeLists.txt file by  
`include (CTest)`  
`enable_testing()`
- For executable `hello` the test named `hello_test` is added by  
`add_test(hello_test hello)`
- All tests can be executed by running  
`make && make test`



# 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>

Crash course on C++

# C++ THE VERY (VERY!) BASICS

# Comments

- Comments in C++

```
/**  
 * \file hello.cxx  
 *  
 * This file is part of the course tw3720tu:  
 * 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 tw3720tu:  
Object Oriented Scientific Programming in C++
```

```
\author Matthias Moller
```

```
*/
```

# Comments, cont'd

- Comments in C++

```
//  
// \file hello.cxx  
//  
// This file is part of the course tw3720tu:  
// 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

# File 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

# 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 in the code (not just at the beginning) and they are valid from the moment that they are defined

# 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
const int f{20};
```

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

```
int g = 10;          // C-like initialisation
int h(20);           // constructor initialisation
int i = {20};        // uniform initialisation, since C++11
int j;               // only declaration (arbitrary value!)
j = 20;              // assignment of value
```

# Intermezzo: Terminology

- A **declaration** introduces the name of the variable or constant, and describes its type but does not create it  
`extern int f;`
- A **definition** instantiates it (=creates it)  
`int f;`
- An **initialisation** initialises it (=assigns a value to it)  
`f = 10;`
- All three steps can be combined, e.g., `int f{10}` or split across different source/header files (later in this course)

# 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  
    }  
}
```

# 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  
}
```

# 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	

# 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 result

```
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
```

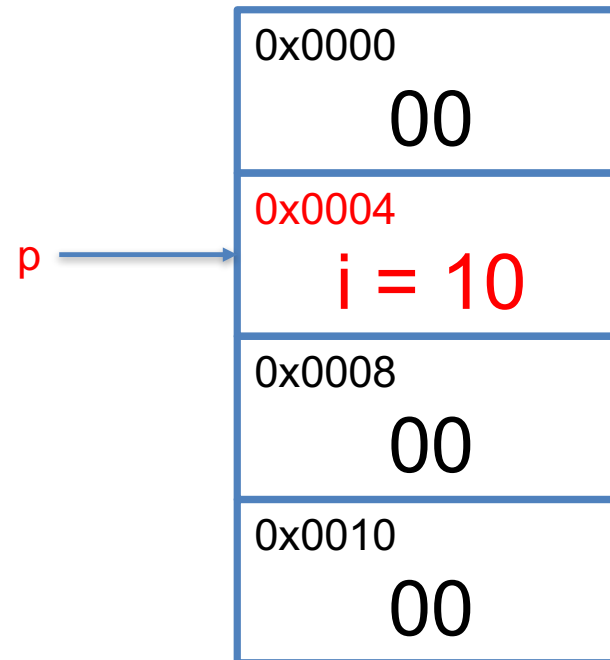
# Auto vs. explicit types

- 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
```

# Address-of/dereference operators

- Integer variable  
`int i = 10;`
- **Pointer** to its address  
`auto p = &i;`
- **Dereference** to its value  
`*p`



# Address-of/dereference operators

- **Address-of operator (&)** returns the address of a variable (= its physical location in the computer's main memory)  

```
int i = 10;  
auto p = &i  
std::cout << p << std::cout; // 0x0004
```
- Addresses are of **pointer** type (equal to that of the variable)  

```
std::cout << typeid(i).name() << std::endl; // i  
std::cout << typeid(p).name() << std::endl; // Pi
```
- **Dereference operator (\*)** returns value behind the pointer  

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

# Pointers and references

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

```
int i = 10;  
int *p = &i;
```

- Change the value of variable i

```
i = 20; std::cout << *p << std::endl; // *p is 20
```

- Dereference pointer p and change its value

```
*p = 30; std::cout << i << std::endl; // i is 30
```

- Change value of pointer p without dereferencing

```
p = p+1; std::cout << *p << std::endl; // *p is 0x0008
```

# Pointer hazards

- Pointers that remain uninitialised can cause hazard

```
int *p;  
std::cout << p << std::endl; // is 0x0008  
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++**

# 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 would give compiler error.

# C++ return value optimization (RVO)

- Most C++ compilers support RVO, that is, no temporary variable for the return value is created inside the function

```
int addOne(const int& a) { return a+1; }
```

but the return value is immediately assigned to variable j.

```
int i = 1;
```

```
int j = addOne(i); -> int j = (i+1);
```

- 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)
```

# Static arrays

- Definition of a static array

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

- Definition and initialisation of a static array

```
int array[5] = { 1, 2, 3, 4, 5 }; // since 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 = nullptr;
```

# 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();  
}
```