Object-oriented scientific programming with C++

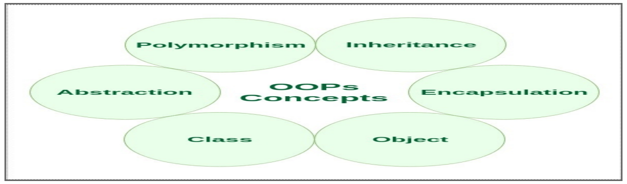
Matthias Möller, Jonas Thies, Cálin Georgescu, Jingya Li (Numerical Analysis, DIAM)
Lecture 1

What's this course about?

1. Principles of **object oriented programming** (not restricted to C++)
2. Principles of **scientific programming** (also not restricted to C++)
3. C++ 11, 14, 17, 20 and some of 23, not C++ 03 and before

Object oriented programming

Using LEGO blocks as an analogy, you can think of object-oriented programming (OOP) as a way of building complex structures (programs) by piecing together different types of blocks (objects). These are the main concepts of OOP



A first example of OOP concepts

Matlab	Python
`A = [1 2; 3 4]` `size(A)`	`A = numpy.matrix([[1, 2], [3, 4]])` `A.shape`
Here, `size()` is a standalone **functions** that is applied to the matrix A from outside. That means that `size()` must be able to deduce the matrix size. In other words, the matrix size is **publicly visible**.	Here, matrix A provides a **member attribute** to report its size from inside. The attribute is also **publicly visible** but offers more fine-grained control.

Course information

- **Lectures** (nonobligatory)
    - weeks 2.1-2.7, Wed 13:45-15:45 in lecture hall Boole
    - recordings from previous years are available on BrightSpace
- **Lab sessions** (nonobligatory)
    - weeks 2.2-2.8, Tue 8:45-12:45 (IDE Ctrl+Enter)
    - this is the time and place to ask your questions (no office hours! no reply to emails! no reply to discussion forums!)

- **WebLab (https://weblab.tudelft.nl/tw3720tu-wi4771tu/2024-2025)**
    - all demos, homework assignments and the final projects are provided via WebLab
    - weekly demos and homework assignments become available every Monday and must be submitted by Tuesday 23:59 two weeks later via WebLab
    - final projects become available in the week before Christmas and must be submitted by the end of Q2 via WebLab

- **Assessment** (3 ECTS)
  - weekly homework assignments to be worked on **individually** (1/3 of the grade)
  - final project can be worked on in **groups of 1-3 students** (2/3 of the grade)
- **Grading**
  - your code is checked automatically against **unit tests** (you get direct feedback, no human bias, no negotiation: pass=pass, fail=fail)

- **Unit tests**
  - don’t try to reverse engineer the unit tests!
  - don’t ask us to write the unit tests so that they tell you which line of your code needs to be changed and how!
  - write your code according to **all** requirements of the assignment, especially adhere to the given interfaces
  - test your code carefully and think about corner cases, e.g., math-operations between different data types
  - if you cannot find the error **ask us** during the lab sessions (and **not** 5 min before the submission deadline via email!)

- **Fraud**
    - It is **prohibited to distribute the material** in full or in parts in any form (printed and electronically). This, in particular, prohibits the transfer of the material to webservices like Bitbucket, GitHub, Gitlab, etc. and the making available of solutions to the assignments. This action is considered piggybacking and will be treated as fraud.
    - It is **not forbidden to use ChatGPT or Co-Pilot** as a source of inspiration. However, you must understand the code you submit and you must be able to explain your code when asking for help from TAs. Questions like "*This is what ChatGPT came up with, can you make it pass the unit tests?*" will not be answered.

After all formalities …

- … ENJOY the course, I did so in all the years
- … LEARN to write good C++ code
- … DARE to think out-of-the-box
- … ASK questions, I and my TAs are happy to answer them

Let’s get started with the fun part

Programming languages

- A **computer programming language** is a formal notation (set of instructions) for writing computer programs
- One distinguishes between
  - **Interpreted languages** : Python, JavaScript, ...
  - **Compiled languages** : C/C++, Fortran, Julia, ...
- Another distinction is between
  - Functional programming : treat computation as evaluation of math functions
  - Object-oriented programming : treat computation as living objects that have mutable data and provide methods to manipulate the data

Pros and Cons of interpreted languages

Python: Hello.py

```
1    a = 1
2    b = 2+a
3    c = 3+b
4
5    for i in range(b, 3+b):
6      a=a+1
7
8    print(c)
```

- Processes the code line by line at run-time (slow! ☹)
  - Computes 3+b twice (lines 3,5) ☹
  - Cannot know **in advance** that lines 5-6 are not 'used' at all ☹
  - Cannot reorder code ☹
- Can inject / load new code instructions at run time ☺
- Platform independent ☺
- Dynamic or no typing ☺ / ☹

Pros and Cons of compiled languages

C/C++: Hello.cxx

```
1    int a = 1;
2    int b = 2+a;
3    int c = 3+b;
4
5    for(int i=b; i<=3+b; i++)
6       a=a+1;
7
8    printf("%d\n", c);
```

- Pre-processes the code line by line at compile-time to create (optimized) run-time executable
  - Evaluates lines 2-3 at compile-time and replaces 2+a, 3+b by values ☺
  - Eliminates 'dead' code (l. 1-7) ☺
- (very) platform dependent ☹
- Compiler can check types ☺

Example of a compiled language

C/C++: Hello.cxx

```
    int a = 1;
    int b = 2+a;
    int c = 3+b;

    for(int i=b; i<=3+b; i++)
      a=a+1;

    printf("%d\n", c);
```

- Compilation by hand:

```
$> g++ -O0 --save-temps \
     Hello.cxx -o Hello.exe
```

  – Hello.ii <- #includes (later!)
  – Hello.s <- assembly code (text)
  – Hello.o <- object file (binary)
  – Hello.exe <- executable (binary)

- Run:

```
$> ./Hello.exe
6
```

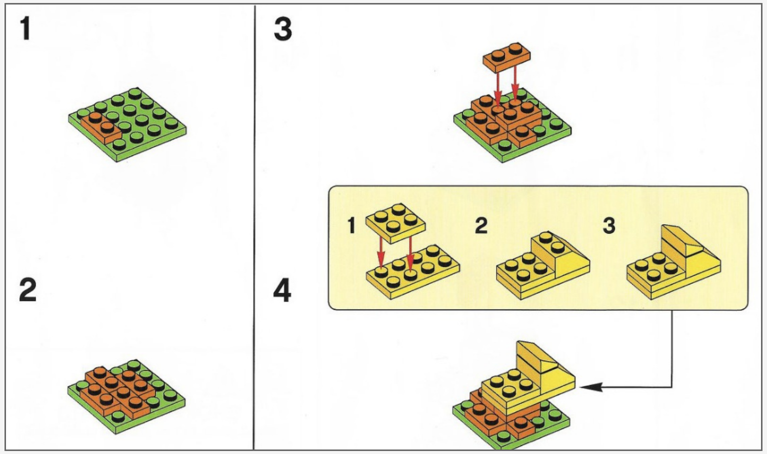
Let's get started

- Live demo in WebLab **https://weblab.tudelft.nl/tw3720tu-wi4771tu/2024-2025/**
- Where to find information on C++
    - A Tour of C++: **https://isocpp.org/tour**
    - CPlusPlus: **http://www.cplusplus.com**
    - Geeks for Geeks: **https://www.geeksforgeeks.org/c-plus-plus/**

Class vs. Object

Once again in LEGO terms ...

A **class is a blueprint**, aka an instruction manual that tells you how to create something



vs.

An **object** is a living instance created (instantiated) from the class blueprint

Let's start with the following example, and learn everything (first two weeks) by filling in this example

In [ ]:

```
// The class. Recommendation: use // for single line comments
class MyFirstClass { // ... and also at the end of a code line
};

int main() {
  /**
   * Recommendation: use this syntax for multiline comments
   * for multiline comments
   */
  MyFirstClass myObj;  // Create an object of MyFirstClass
  return 0;
}
```

Hello World!

In [ ]:

```
// 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; // or “Hello world!\n”;
  return 0; // Return code 0 to the operating system (=no error)
}
```

This can also be done by using a **class**

In [ ]:

```
#include <iostream>

// HelloWorld class
class HelloWorld {
    public:
    void PrintHelloWorld() // public member function
    {
        std::cout << "Hello World!\n";
    }
};

int main(){
    HelloWorld hello; // create an object of HelloWorld
    hello.PrintHelloWorld(); // call member function
}
```

- Extra functionality provided by the standard C++ library is defined in so-called **header files** which need to be included via

  ```
  #include &ltheaderfile>
  ```

  , e.g.,

  - ```
    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
    ```

OOP style and member function

Function that computes the sum of a Vector

---

double sum(const Vector& v) { double s = 0; for (auto i=0; i<v.length; i++) s += v.array[i]; return s; }
This is NOT really an OOP-style!

---

int main() { Vector x = { 1, 2, 3, 4, 5 }; std::cout &lt&lt sum(x) &lt&lt std::endl; }

The member function version

```
class Vector { public: double sum(){ double s = 0; for (auto i=0; i<length; i++)
s+=array[i]; return s; } }
```

This is a GOOD OOP-style!

```
int main() { Vector x = {1,2,3}; std::cout &lt&lt x.sum() &lt&lt std::endl; }
```

We will get back to this topic later

THE main function

Now let's only focus on 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

In [1]:

```
#include <iostream>

std::cout << "Hello world!" << std::endl;
```

```
Hello world!
```

Streams can be easily concatenated

In [2]:

```
std::cout << "Hello" << " " << "world!" << std::endl;
```

```
Hello world!
```

Streams are part of the standard C++ library and therefore encapsulated in the namespace

---

std; instead of using

---

std:: one can also import all functionality from the namespace by

In [3]:

```
using namespace std;
cout << "Hello world!" << endl;
```

```
Hello world!
```

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

In [4]:

```
int a = 10; // create integer variable and initialize it to 10
a = 15;     // update the value of integer variable to 15
```

Out[4]:

15

Variables can be updated, constants cannot

In [5]:

```
const int b = 20; // create integer constant and initialize it to 10

a = b;
b = a;
```

```
input_line_15:4:3: error: cannot assign to variable 'b' with const-qualified type 'const int'
b = a;
~ ^
input_line_15:2:12: note: variable 'b' declared const here
 const int b = 20; // create integer constant and initialize it to 10
 ~~~~~~~~~~^~~~~~

Interpreter Error:
```

Initialization of constants

Constants must be initialized during their definition

In [6]:

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

Initialization of variables

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

In [7]:

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

Out[7]:

```
20
```

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 **initialization** initializes 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

In [ ]:

```
int main() {
    int a = 10;  // variable a is visible here
    {
      int b = a; // variable a is visible here
    }
}
```

Variables/constants are only visible in their scope

In [ ]:

```
int main() {
    int a = 10;  // variable a is visible here
    {
      int b = a; // variable a is visible here
    }
    {
      int c = b; // variable a is visible here,
    } // b is not(!) visible here -> error
}
```

Another example

In [ ]:

```
int main() {
    int a = 10;   // variable a is visible here
    {
      int a = 20; // new variable a only visible in blue
                  // scope, interrupts scope of red one
      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.
	`float`	Precision not less than float

Floating-point type	`double`	Precision not less than float
	`long double`	Precision not less than double
Boolean type	`bool`	

Examples of C++ types

Double-precision floating-point type

In [8]:

```
double d1 = 1.0;
double d2 = 1.;    // zero is added automatically
double d3 = 1e3;   // -> 1000
double d4 = 1.5E3; // -> 1500
double d5 = 15e-2; // -> 0.15
```

Single-precision floating-point type

In [9]:

```
float f1 = 1.0f;   // or 1.0F suffix type specifier
float f2 = 1.0;    // works the same but does conversion
float f3 = 1.5e3F; // -> 1500
```

Mixing and conversion of types

Getting the type of a variable

In [10]:

```
#include <typeinfo>

float f = 1.7f; double d = 0.7;
std::cout << typeid(f).name() << std::endl; // float
std::cout << typeid(d).name() << std::endl; // double
```

```
f
d
```

Out[10]:

```
@0x7f128339fde0
```

C++ converts different types automatically

In [11]:

```
std::cout << typeid(f + d).name() << std::endl; // double
```

d

Out[11]:

@0x7f128339fde0

Check if you are happy with the result

In [12]:

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

```
f
```

Out[12]:

```
@0x7f128339fde0
```

C++11 introduces the

auto keyword, which makes handling of mixed types very easy

In [13]:

```
auto x = f + d;
std::cout << typeid(x).name() << std::endl; // double
```

```
d
```

Out[13]:

```
@0x7f128339fde0
```

You can also explicitly cast one type into another

In [14]:

```
auto y = f + (float)d;
std::cout << typeid(y).name() << std::endl; // float

auto z = (int) (f + (float)d);
std::cout << typeid(z).name() << std::endl; // int
```

```
f
i
```

Out[14]:

```
@0x7f128339fde0
```

`auto`

vs. explicit types

- **Recommendation:** use the keyword

  ```
  auto
  ```

  - 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
  - ```
    auto a = 1.5+0.3;
    ```

    is

    ```
    (double)1.8 = 1.8
    ```
  - ```
    int b = 1.5+0.3;
    ```

    is

    ```
    (int)1.8 = 1
    ```
- ... unless the C++ standard does not allow so, e.g., return type of a (pure) virtual function (later in this course)

Use of suffix type specifiers

- **Suffix type specifiers** (termed **literals**) seem unnecessary at first glance since constants are implicitly converted
  - `float f1 = 0.67;`
- But keep in mind that

  ```
  0.67
  ```

  and

  ```
  0.67f
  ```

  are not the same
  - `std::cout << (f1 == 0.67);`

    // -> false
  - `std::cout << (f1 == 0.67f);`

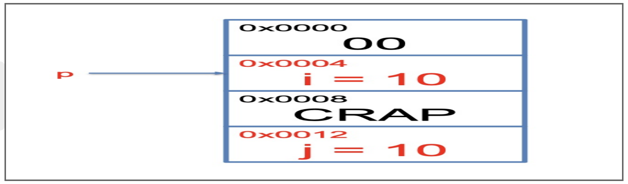
    // -> true

In [15]:

```
float f1 = 0.67;
std::cout << (f1 == 0.67);
std::cout << (f1 == 0.67f);
```

Address-of/dereference operators

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



In [31]:

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

Address-of operator (

&): returns the address of a variable (= its physical location in the computer's main memory)

In [34]:

```
std::cout << i << std::endl;
std::cout << p << std::endl;
```

```
10
1
```

Addresses are of pointer type (equal to that of the variable)

In [35]:

```
std::cout << typeid(i).name() << std::endl; // -> i
std::cout << typeid(p).name() << std::endl; // -> Pi
```

```
i
Pi
```

Out[35]:

```
@0x7f128339fde0
```

Dereference operator (

*): returns the value behind the pointer (= the value stored at the physical location in the computer's main memory)

In [36]:

```
std::cout <<  i << std::endl;
std::cout << *p << std::endl;
```

```
10
10
```

Pointers and references

Pointers can be used to have multiple variables (with different names) pointing to the same value, i.e. the same location in the computer's main memory

In [37]:

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

Let us change the value of variable

i

In [38]:

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

```
20
```

Out[38]:

```
@0x7f128339fde0
```

Dereference pointer

p and change its value

In [39]:

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

```
30
```

Out[39]:

```
@0x7f128339fde0
```

Change value of pointer

p without dereferencing it

In [40]:

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

```
1
0
```

Out[40]:

```
@0x7f128339fde0
```

Pointer hazards

Pointers that remain uninitialized can cause hazard

In [ ]:

```
int* p;
std::cout <<  p << std::endl; // prints some memory address
std::cout << *p << std::endl; // prints some random content at that address
```

C++11 introduces the new keyword

nullptr that explicitly sets a pointer to null

In [42]:

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

```
0
```

```
input_line_64:4:15: warning: null passed to a callee that requires a non-null argument [-Wnonnu
ll]
std::cout << *p << std::endl; // yields Segmentation fault
              ^


Interpreter Exception:
```

You can use

nullptr to check if a pointer can be dereferenced without problems

In [ ]:

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

Error handling with exceptions

Let us include the

exception header file

In [44]:

```
#include <exception>
```

Use

throw to signal the occurrence of an anomalous situation

In [45]:

```
throw std::runtime_error("An error occured");
```

```
Standard Exception: An error occured
```

Use

---

try and

---

catch blocks to handle exceptions gracefully

In [46]:

```
try
{
    throw std::runtime_error("An error occured");
} catch (const std::exception& e)
{
    std::cout << e.what() << std::endl; // or handle the exception
}
```

```
An error occured
```

Error handling with assertion

Let us include the

cassert header file and assert that a condition is true

In [48]:

```
#include <cassert>

float f1 = 0.67;
assert(f1 == 0.67);
```

Quiz: Why does nothing happen here?

Let us turn on debug mode by explicitly disabling the non-debug (

#undef NDEBUG) mode

In [ ]:

```
#undef NDEBUG
#include <cassert>

float f1 = 0.67;
assert(f1 == 0.67);
```

The

---

assert(expression) function evaluates the

---

expression inside parentheses. If it evaluates to

---

false,

---

assert will print an error message and then terminate the program but only if the code is compiled in debug mode.
DON'T USE IT IN WEBLAB

Error handling best practices

What are the best practices of error handling?

- Prefer exceptions for signaling errors over return codes.
- Only use exceptions for exceptional conditions, not normal flow control.
- Ensure all exceptions are caught and handled appropriately.

Debug example: divide by zero

In [ ]:

```
#include <iostream>
#include <stdexcept>

// A function that might throw an exception
int divide(int numerator, int denominator) {
    if (denominator == 0) {
        throw std::invalid_argument("Denominator cannot be zero.");
    }
    return numerator / denominator;
}

int main() {
    try {
        int a = 10;
        int b = 0; // Intentionally set to zero to cause an exception
        int result = divide(a, b);
        std::cout << "Result is: " << result << std::endl;
    } catch (const std::invalid_argument& e) {
        // Handle the exception here
        std::cerr << "Caught an exception: " << e.what() << std::endl;
    }
    return 0;
}
```

Argument passing – by value

Arguments that are passed by value are passed as a physical duplicate of the original variable

In [2]:

```
int addOneByValue(int a) { return a + 1; }

int i = 1;
int j = addOneByValue(i);
```

Quiz: What happens if variable

---

a is not of type

---

int but a vector of several gigabyte?

Argument passing – by reference

Arguments that are passed by reference give the function read and write access to the memory location of the original variable

In [3]:

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

int i = 1;
int j = addOneByReference(i);
```

Quiz: What happens if

---

addOneByReference() tries to modify the value of the passed variable?

The post-increment operator

a++ performs the operation first and incremenets variable

a afterwards

In [4]:

```
int addOneByReference1(int& a) { return a++; }

int i = 1;
int j = addOneByReference1(i);
```

The pre-increment operator

a++ incremenets variable

a first and performs the operation afterwards

In [5]:

```
int addOneByReference2(int& a) { return ++a; }

int i = 1;
int j = addOneByReference2(i);
```

Argument passing – by constant reference

Arguments that are passed by constant reference give the function read access to the memory location of the original variable

In [6]:

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

int i = 1;
int j = addOneByConstReference(i);
```

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); // RVO makes it int j = (i+1);

Argument passing

```
<p>
    If we want a function that changes the argument <strong>directly</strong>, we must pass the argument by reference.
</p>
<pre>
```

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

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

The return type void indicates that 'nothing' is returned.

Argument passing – by address

Passing by address

In [7]:

```
int addOneByAddress(int* a) { return *a+1; }

int i = 1;
int j = addOneByAddress(&i);
```

This is the old C-style to pass arguments that should be modifyable inside the function or to pass arrays, aka the first position in the computer's main memory at which the array starts.

Example

Compute the sum of the entries of an array

In [9]:

```
#include <iostream>

double sum(const int* array, int length) {

    double s = 0;

    for (auto i=0; i<length; i++)
        s += array[i];

    return s;
}

int array[5] = { 1, 2, 3, 4, 5 };
std::cout << sum(array, 5) << std::endl;
```

```
15
```

This is not OOP. DON'T, REALLY DON'T DO THIS IN C++! We will learn much better ways to pass bigger objects such as arrays by (constant) reference.
There is one exception. If you want to allocate memory dynamically inside the function and assign it to a variable defined outside the function you need to work with double pointers.

Static arrays

```
<p>Definition and creation of a static array</p>
<pre>int array[5];</pre>
```

Definition, creation and initialization of a static array:

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

Access of individual array positions:

In [10]:

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

```
1
2
3
4
5
```

Remember that C++ **starts indexing at 0**
Static arrays are **destroyed automatically** at the end of scope

Quiz: Static arrays

What happens?

In [ ]:

```
auto array = { 1, 2, 3, 4, 5 };
```

In [ ]:

```
auto array{ 1, 2, 3, 4, 5 };
```

Quiz: char* argv[]

What is char* argv[]?
Example use case:

In [ ]:

```
main(int argc, char* argv[]) {
     for (int i=0; i<argc; i++)
        std::cout << i << “-Argument is “ << argv[i] << “\n”;
}
```

Dynamic arrays

- Definition and allocation of dynamic array

```
int* array = new int[5];
```

- Definition, allocation and initialization of dynamic array

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

- Explicit deallocation of dynamically allocated array needed

```
delete[] array;
```

- Think fail-safe! Because it still points to an invalid address

```
array = nullptr;
```

Example of a dynamic array

In [11]:

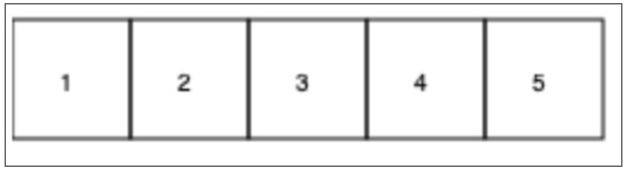
```
#include <iostream>

int* array = new int[5]{ 1, 2, 3, 4, 5 };

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

delete[] array;
array = nullptr;
```

```
1
2
3
4
5
```



Static vs. dynamic arrays

Static arrays require the size to be known at compile time

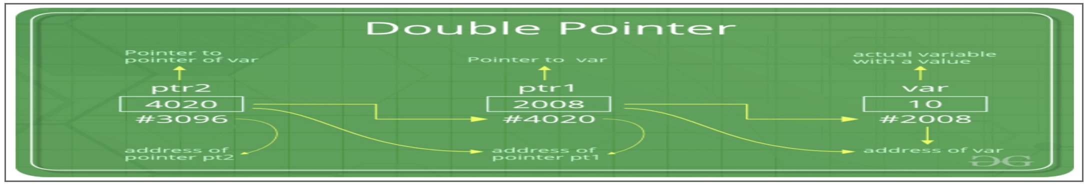
In [ ]:

```
int array[5];
constexpr int k=5; // constexpr tells the compiler that the
int array[k];      // expression is available at compile time
int k=5;
int array[k];      // Gives a compiler error !!!
```

Dynamic arrays allow variable sizes at run-time

In [ ]:

```
int k = std::atoi(argv[1]);
int* array = new int[k];
```



Double Pointer
Pointer to
pointer of var
ptr2
4020
#3096
address of
pointer pt2
Pointer to var
ptr1
2008
#4020
address of
pointer pt1
actual variable
with a value
var
10
#2008
address of var

Double pointer

---

void func(int v) // can use value of variable v

---

void func(const int& v) // can use value of variable v

In [ ]:

```

```

Namespaces

Namespaces, like

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

In [ ]:

```
namespace tudelft {
    void hello() {
      std::cout << “Hello TU Delft\n”;
    }
}
```

In [ ]:

```
namespace other {
    void hello() {
      std::cout << “Hello other\n”;
    }
}
```

Functions implemented in a namespace can be called by

- providing the namespace explicitly

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

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

- importing the namespace into the scope

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

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

Namespaces can be nested

In [ ]:

```
#include <iostream>

namespace tudelft {
    void hello() { std::cout << "Hello TU Delft\n"; }
    namespace eemcs {
      void hello() { std::cout << "Hello EEMCS\n"; }
    }
}

tudelft::hello();
tudelft::eemcs::hello();
```

Leading

:: goes back to outermost unnamed namespace

In [ ]:

```
namespace tudelft {
    void hello() { ::hello();
                   std::cout << "TU Delft\n"; }
    namespace eemcs {
      void hello() { ::hello();
  std::cout << "EEMCS at";
  ::tudelft::hello();
            }
    }
}

void hello() { std::cout << "Hello "; }
```