



Programming in C/C++

- Fundamentals and Data Types -

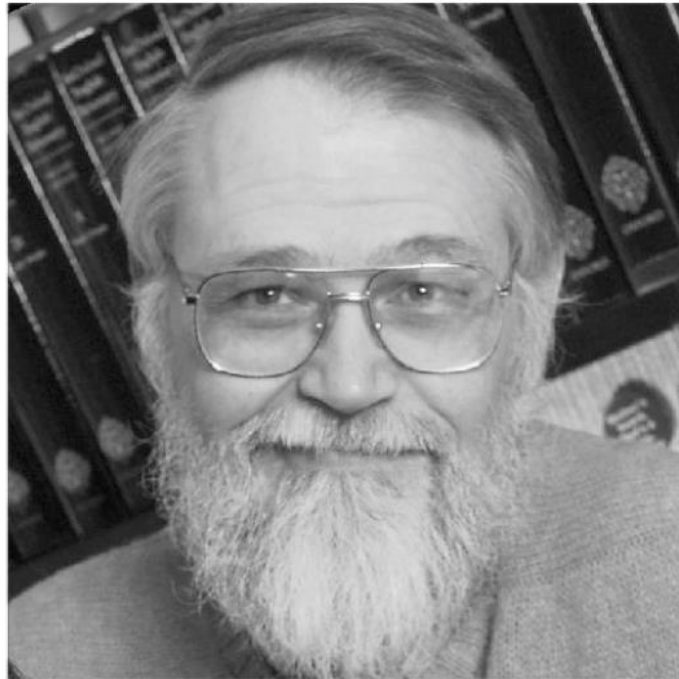


C/C++

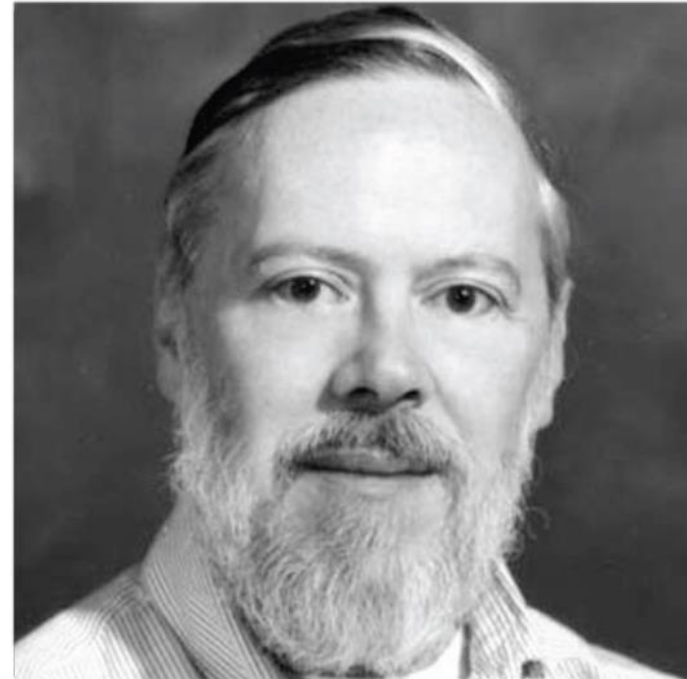
- Brief history of C/C++

C

- 1978: Brian Kernighan, Dennis Ritchie (K&R) - The C Programming Language
- Higher-level, portable language that replaced assembler for many task



Kernighan



Ritchie



- 1984: Developed by **Bjarne Stroustrup** (AT&T)
- Originated as an extension of the C programming language, or „C with classes“
- Nearly every C-program is a valid C++ program => we will focus on C++
- Active development to support modern concepts



Bjarne Stroustrup

<https://de.wikipedia.org/wiki/Bild:BjarneStroustrup.jpg>

C makes it easy to shoot yourself in the foot;

C++ makes it harder, but when you do, it blows away your whole leg.



C++ - in a nutshell

- Like C, C++ is a **compiled language**
 - source code itself is not executable, it must be translated to machine code
 - the generated machine code is platform specific (**pro**: highly optimized, **con**: nonportable)
- **General-purpose programming language**
 - Data abstraction, Object-oriented programming (e.g., classes, inheritance)
 - Generic programming (e.g., reusable containers and algorithms, operator overloading, templates)
 - Supports functional programming (e.g., lambda functions, functors)
 - Supports both low-level (close to hardware) and high-level programming
- Consists of:
 - **core language**: built-in types, loops, control flow...
 - **standard template library STL** of generic data structures and algorithms
- **statically typed**: the type of every entity must be known at compile time

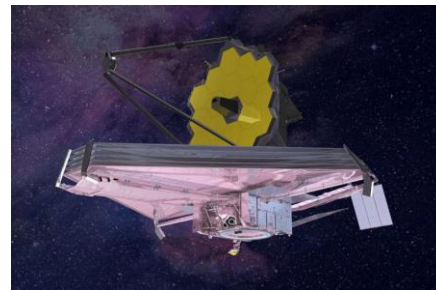


C++ - in a nutshell

- C/C++ and the STL is continuously developed and standardized
 - ISO standard offers stability over decades

Jul 2022	Jul 2021	Change	Programming Language		Ratings	Change
1	3	▲		Python	13.44%	+2.48%
2	1	▼		C	13.13%	+1.50%
3	2	▼		Java	11.59%	+0.40%
4	4			C++	10.00%	+1.98%
5	5			C#	5.65%	+0.82%

Tiobe Index (www.tiobe.com/tiobe-index/)





Timeline C/C++

1967 – 1980	Developing Unix [Ken Thompson, Denis Ritchie et al, Bell Labs].
1969 – 1973	C [Denis Ritchie, Bell Labs] based on B [Ken Thomson].
1984	C++ [Bjarne Stroustrup].
1998/99	C++ Standard ISO/IEC 14882 Update 2003
2011	C++11 "C++0x, ISO/IEC 14882:2011 (rvalue references, lambda functions, auto, multi-threading...)
2014	C++14 (reader-writer locks, generalized lambdas...)
2017	C++17 (structure bindings, parallel algorithms...)
2020	C++20 (concepts, <i>ranges</i> , <i>modules</i> ...)
2023	C++23 (<i>coroutines</i> , <i>modular STL</i> , ...)



Overview

1. Introduction and Fundamental
2. Data Structures, Iterators
3. OOP, Classes
4. Polymorphism
5. Memory
6. Files, Exceptions, Debugging
7. Class and Function Templates
8. STL Algorithms
9. Parallelism
10. Graphs
11. Graph Algorithms

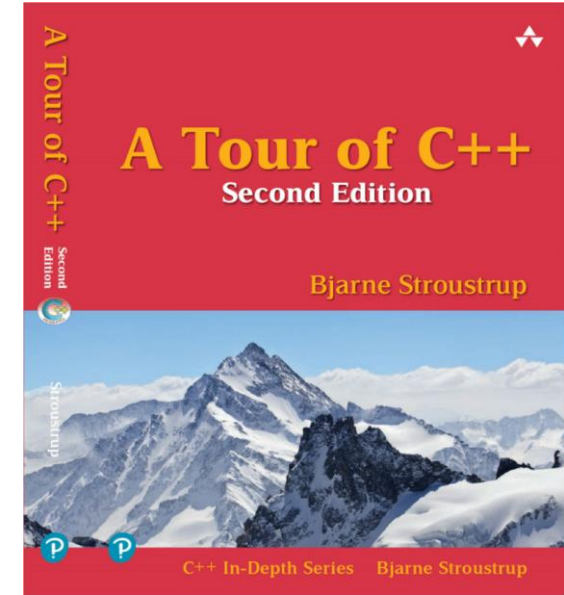


Literature



Classics:

- [1] B.W. Kernighan, D.M. Ritchie, The C Programming Language, 2nd ed, Prentice Hall, 1988.
- [2] B.W. Kernighan, D.M. Ritchie, Programming in C: With the C- Reference Manual in German, Hanser Fachbuch; Edition: 2nd, ed. (February 1, 1990)
- [3] Bjarne Stroustrup, The C++ Programming Language - 4th, updated edition: Addison-Wesley Publishers; (May 28, 2013).



[4] Bjarne Stroustrup, A Tour of C++ (second edition), Addison-Wesley, 2018

- [5] Jesse Liberty, C++ in 21 days, Markt+Technik; Edition: 3rd ed. (October 15, 1999).
- [6] Helmut Erlenkötter, C++: Objektorientiertes Programmieren von Anfang an, rororo; Edition: 15 (January 3, 2000).



C++ and more

Online references:

[7] cppreference.com

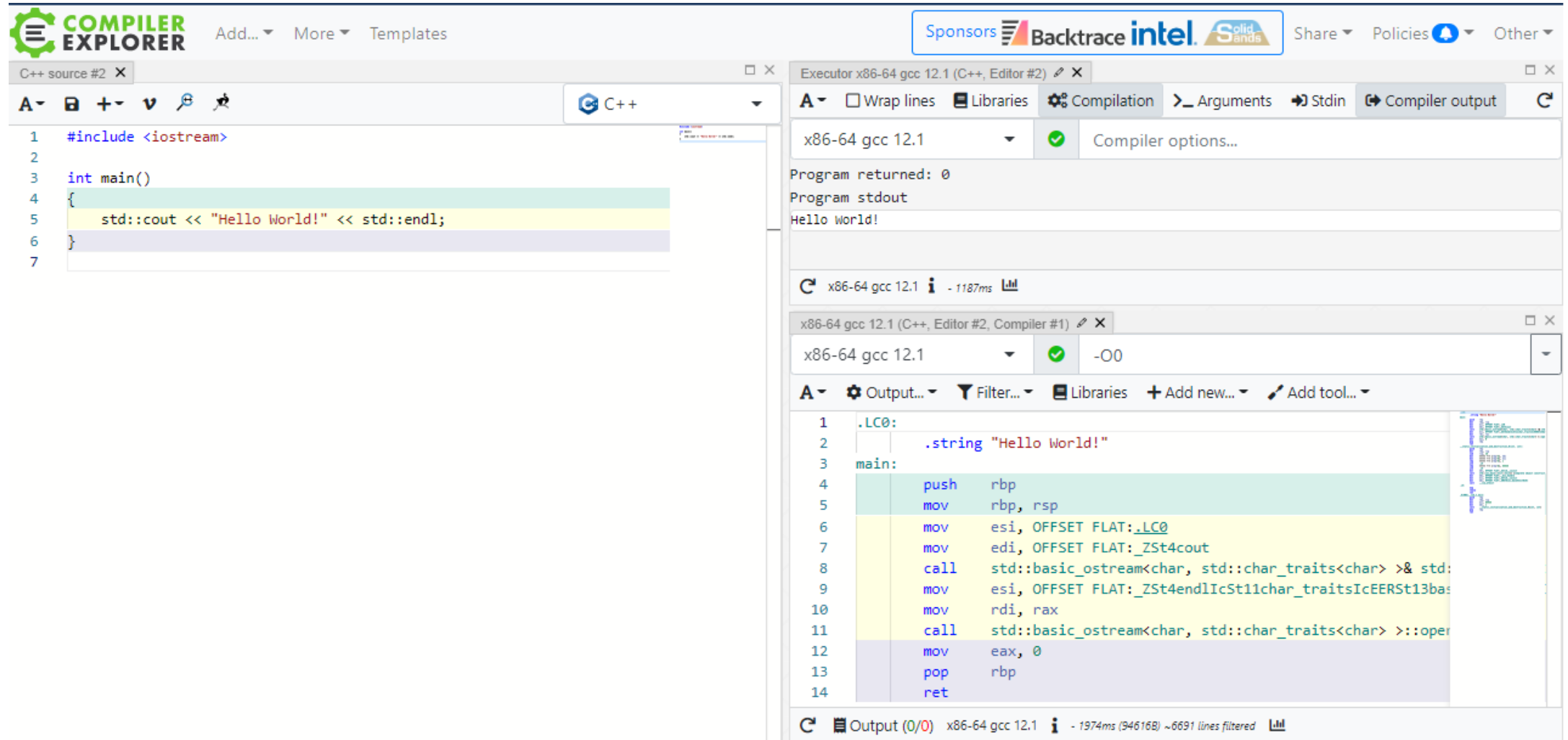
[8] cplusplus.com <https://www.cplusplus.com/doc/tutorial/>

[9] <https://isocpp.org/> Tour FAQ

Good C++ programming practice:

[10] C++ Core Guidelines <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>

Test code snippets online: <https://godbolt.org/>



The screenshot displays the Godbolt Compiler Explorer interface. On the left, the C++ source code is shown in a text editor:

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Hello World!" << std::endl;
6 }
7
```

On the right, the compilation and execution results are shown. The top panel, titled "Executor x86-64 gcc 12.1 (C++, Editor #2)", shows the compiler options and the program's output:

Program returned: 0
Program stdout
Hello World!

The bottom panel, titled "x86-64 gcc 12.1 (C++, Editor #2, Compiler #1)", shows the assembly code generated by the compiler:

```
1 .LC0:
2     .string "Hello World!"
3 main:
4     push    rbp
5     mov     rbp, rsp
6     mov     esi, OFFSET FLAT:.LC0
7     mov     edi, OFFSET FLAT:_ZSt4cout
8     call    std::basic_ostream<char, std::char_traits<char> >& std::
9     mov     esi, OFFSET FLAT:_ZSt4endlcSt11char_traitsIcEERSt13bas
10    mov     rdi, rax
11    call    std::basic_ostream<char, std::char_traits<char> >::oper
12    mov     eax, 0
13    pop     rbp
14    ret
```



The first C++ Program



"Hello World" in C++

File: HelloWorld.cpp

```
#include <iostream>           // include header with std::cout and std::endl

using namespace std;         // make names in std:: visible (without std::)

/* Hello World
   in C++ */
int main()                    // the main function
{                             // code block '{...}' starts here
    cout << "Hello World!" << endl; // print "Hello World!" in console
    return 0;                // return from main ends program
}                             // code block '{...}' ends here
```

like in C, statements end with semicolon ;



The `main` function

- The start of a C/C++ program

```
int main()
int main( int argc, char *argv[] ) // to access to command line parameters
```

- C++ program has one **main function** in one of its potentially many source files.
- A **block** `{ }` is a sequence of **expressions**, it defines the **scope** for variables defined in the block.
- Return value of main function:

```
return 0;
return EXIT_SUCCESS; // or use constants from #include <cstdlib>
```

`0` or `EXIT_SUCCESS` from `main` corresponds to error-free termination of the program



#include and namespace

```
#include <iostream> // needed for std::cout and std::endl
```

- The **header file** `<iostream>` provides the input/output streams
- The header is part of the **C++ Standard Template Library (STL)** which contains many algorithms and data structures
- Everything in this library are declared / created in the **namespace** `std` (=standard)
- A namespace groups names

To use `cout` we need to prefix the **namespace**:

```
std::cout ...
```

If we want to omit `std::`, we can

- make the name `std::cout` available in the whole file:
- or even everything in the namespace `std::`

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

```
using namespace std;
```




Standard input / output streams

Output stream `cout`

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

- Object for writing to the standard output device – the console
- Write with the **output stream operator** `<<`. Allows to chain output.
- New line with: `std::endl` “end of line”

Input stream `cin`

```
std::cin >> myVar;
```

- Object to get input from the console
- Primitive data types are read in, converted and assigned with the **input stream operator** `>>`.

Error stream `cerr`

- Unbuffered output stream for error messages



Compile and Run

C/C++ is a compiled language. You need a compiler!

Main steps:

1. **Preprocessor** processes and modified the source file. Start with **#**.
 - **#include <iostream>** -> insert header/library code at this position. The compiler otherwise doesn't know about e.g., **std::cout**.
 - **#define**, **#ifdef**, ... allow defining **processor variables** for conditional compilation (e.g., compile only parts that are needed on windows/macOS/...).
2. **Compilation** of **source code** to **object code** (machine code + meta data)
3. **Linker** combines object code into an **executable** (e.g., a .exe) or **library** (e.g., a .dll / .so file)
4. Complex projects are often divided into many files that are translated separately

Object code and executable program are **platform specific**.

E.g., an executable compiled for Linux can't be natively executed on windows.



clang++ individual steps

Individual steps can be executed manually:

```
clang++ -E HelloWorld.cpp -o preprocessed.i  run preprocessor
clang++ -S preprocessed.i -o compiled.s      generate assembler code
clang++ -c compiled.s -o assembled.o         translate into machine code
clang++ assembled.o -o HelloWorld            link and create executable
```

or shorter:

```
clang++ HelloWorld.cpp -o HelloWorld
```

running the program HelloWorld prints:

```
Hello World!
```



cmake – cross-platform build environment

cmake (build environment generator)

- Enables cross-platform development
- Takes care of complex configuration of compiler/linker flags (**Debug** vs. **Release** mode, ...)
- Evaluates the CMakeLists.txt

CMakeList.txt

```
cmake_minimum_required(VERSION 3.18)
project(hello_world)
add_executable(app main.cpp)
```

- Does e.g., compiler detection, checks if dependencies are fulfilled
- Generates the build environment in the current working directory

```
mkdir build
cd build
cmake .. -GNinja
```

ninja (build system)

- fast for incremental builds of large projects
- replaces `make`

ninja



Data Types

- Types, variables, and expressions
- Integers and floating point numbers
- Assignments and operators
- Declarations, definition, names



Motivation

- Input and processing require information to be stored temporarily
- Variables store data of a specific type

```
#include <iostream>

using namespace std;

int main() {
    cout << "Type in a number: ";

    int varA;                // declare integer variable varA
    cin >> varA;              // write into variable
    cout << "You typed " << varA << endl; // output variable to console

    return 0;
}
```



Variables

C++ is a **statically typed** language:

If we **declare** a variable, we introduce a **name** into the program and specify its **type**.

- The **type** tells us the **set of possible values** and **operations** one can perform with an object of this type. It determines the **memory requirements** and how the data in memory needs to be **interpreted**.
- The **variable** has a name and holds the data of the object

Expressions

- In expressions, things can be calculated with **operators**, **variables can be accessed**, and their **value** can be changed.

Dynamic vs Static typing



```
def divide(x)
    return x / 3

divide("ups")
```

script crashes!
run time error



```
int divide(int x) {
    return x / 3;
}

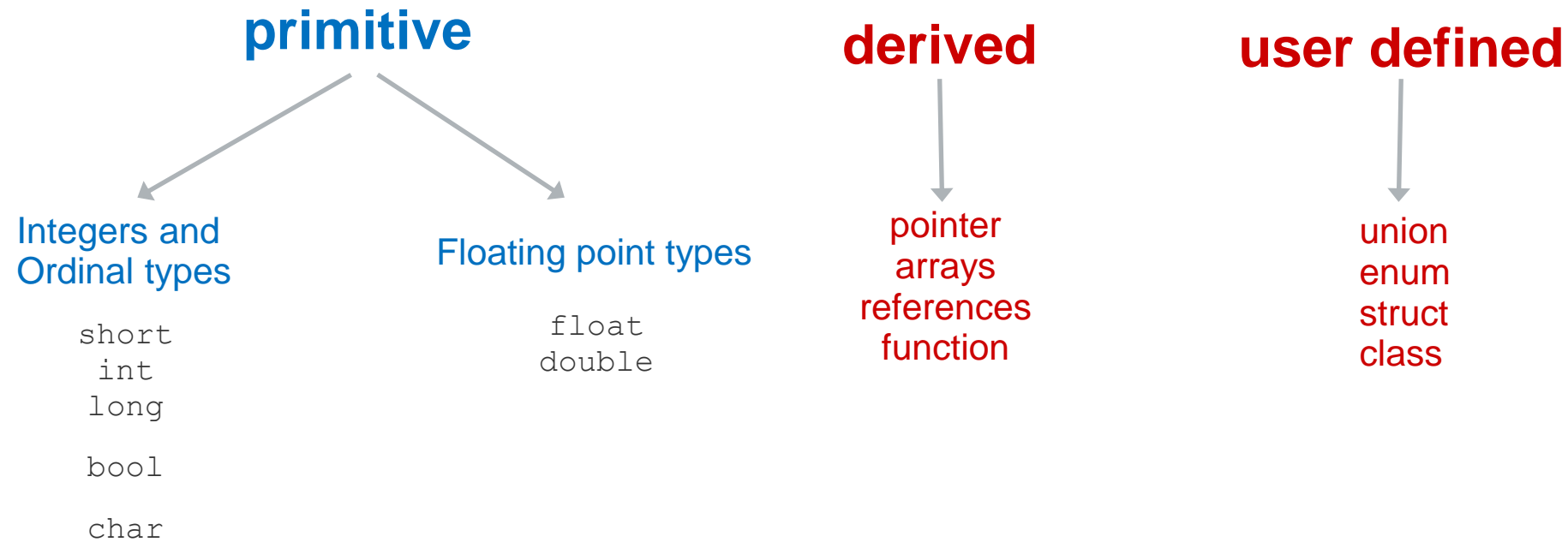
divide("ups")
```

compilation error!
No binary produced.

static typing detects errors before code is run



Taxonomy of data types



- We can define synonyms for existing types: `typedef int IndexType;`



Examples

Declarations of fundamental C++ types

```
int index;      // integer e.g, -1, 343, ...
char letter;    // e.g, 'a', '?',
bool isGreen;   // true, false
float seconds;  // single precision floating point (usually 32bit)
double price;   // double precision (usually 64bit)
std::string s;  // not a fundamental type: string class in #include <string>
```

Initialization:

```
int j{4};
int i = 3;
std::string s{"test"};
```

```
int i;

i = 5;
i = 12;
i = i + 7;
```

i:

Assignment:

```
index = 5;
letter = 'A';
isGreen = false;
price = 3.5;
s = "This is a string literal."
```

Rule-of-thumb: Always initialize your variables! If unsure, brace initialize with {}.

Integers (integer)

C++ allows controlling memory requirements by providing many types of different size.

Numbers with positive and negative value range

- Value ranges for integers, e.g.:
 - **char** [-128, ..., 127]
 - **short** [-32,768, ..., 32,767]
 - **int** [-2,147,483,648, ..., 2,147,483,647]
 - **long** [$-(2^{63})$, ..., $(2^{63})-1$] (sometimes same as int)
- Standard only guarantees how large these types **at least** are (!).
- arithmetic operators: +, -, *, /, % (modulo)
- arithmetic assignment operators: +=, -=, *=, /=, %=
- increment/decrement: ++, --
- comparison operators: ==, !=, <, <=, >, >=
- ...





Types - size and modifiers

Example: Sizes in C++ in clang on Ubuntu

compiler / OS dependent!

```
/* to determine the size */  
sizeof(var);
```

Modifiers: unsigned, signed, short, long

1 byte : bool, char, unsigned char, signed char

2 bytes: short, unsigned short

4 bytes: int, unsigned int, unsigned long, float

8 bytes: double, long, long long

12 bytes: long double



#include <stdint>

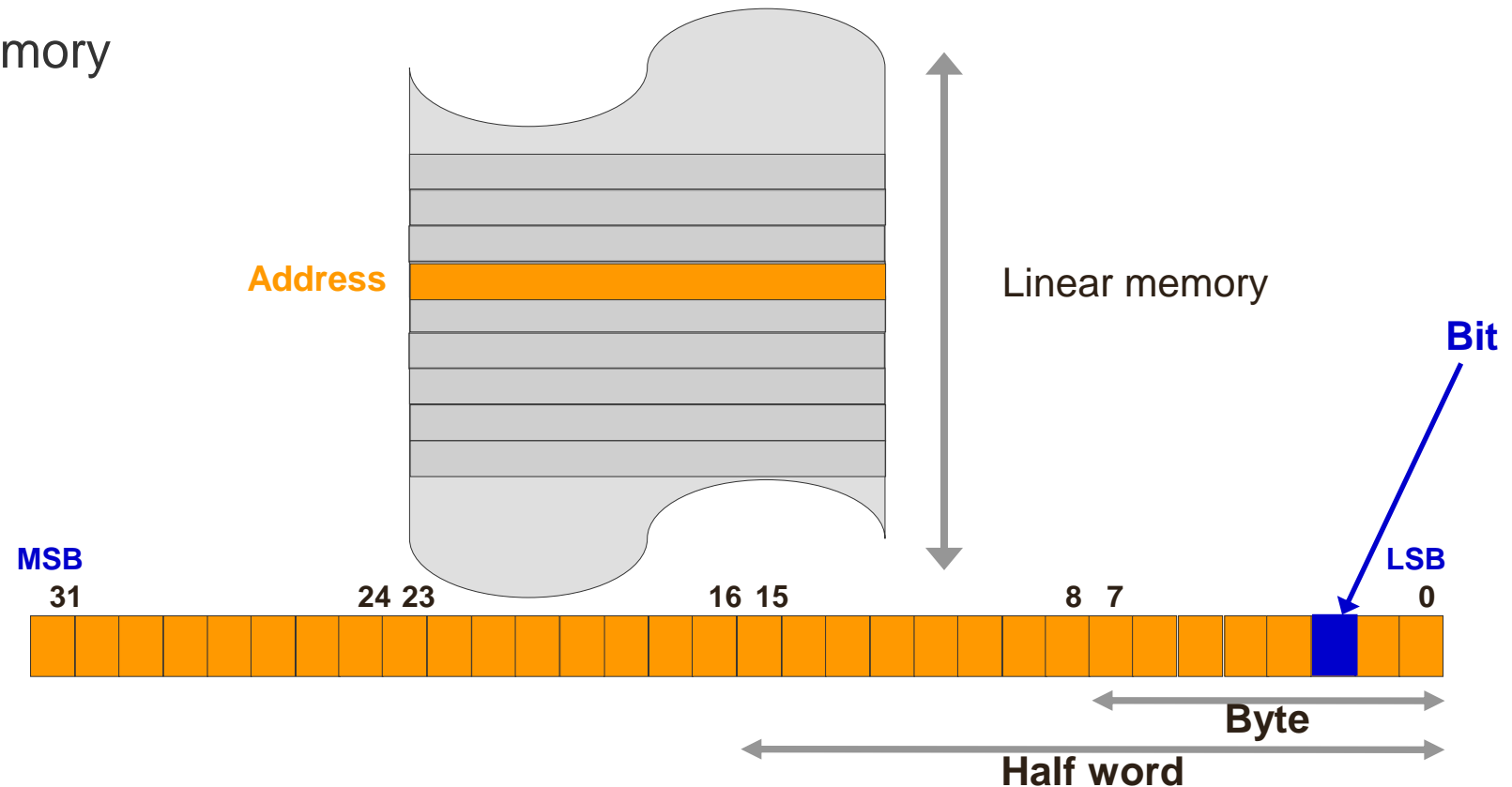
Definition of integers with exact size in memory (since C++11) possible.

<u>signed</u>	<u>unsigned</u>	
int8_t	uint8_t	
int16_t	uint16_t	8, 16, 32, (if supported 64) bits.
int32_t	uint32_t	
int64_t	uint64_t	
	size_t	Typically used for size of container/vectors (uint32_t or uint64_t)
intmax_t	uintmax_t	Integer type with the maximum width supported.

Integer representation in digital computers

Linear memory and information units (32 bit)

- One **memory element** (memory cells) with **32 bits** belongs to one **address**; this corresponds to **4 bytes** or **1 word** "width".
- Linear organized memory





Calculating with floating point numbers (float, double)

Arithmetic operations for floating point numbers: +, -, *, /

- Only a **finite number of rational numbers** can be represented from the respective value range. Many rational numbers can not be exactly represented.

- But: many integer values can be represented exactly.

- for **float**, up to 16,777,217 ($2^{24} + 1$)
- for **double**, up to 9,007,199,254,740,993 ($2^{53} + 1$)



Common source of errors: Rounding errors in arithmetic operations

Example (should be 0):

```
float y1 = 400.1f * (300.1f * 400.1f + 10.f - 200.1f * 600.1f) * 600.1f;
cout << y1 << endl; // return output: 1875.78
```

```
double y2 = 400.1 * (300.1 * 400.1 + 10. - 200.1 * 600.1) * 600.1;
cout << y2 << endl; // returns expected output: almost 0
```



Floating point numbers: NaN, Inf

- `float`, `double`, `long double` can take on undefined values:
- **NaN**, not a number `isnan()`
 - e.g., square root of a negative number
- **Inf**, infinity `isinf()`
 - Floating point number - overflow or divided by 0.
- **Caution: NaN** always returns false for comparisons

```
#include <iostream>
#include <cmath>
int main() {
    double d = sqrt(-1.);
    cout << (d > 0.) << endl;
    cout << (d == d) << endl; // !?!
    cout << (d < 0.) << endl;
    cout << isnan(d) << endl;
    cout << isinf(d) << endl;
    cout << d << endl;
    return 0;
}
```

```
0
0
0
1
0
nan
```




Arithmetic Types – Summary

Recommendations:

- Think about what kind of number range you wish to represent.
- First question: boolean, signed integral, unsigned integral or floating point?
- Make integrals unsigned if possible!
- **Unsigned integral** and no other information? → `size_t`
- **Floating point** and no other information? → `double`
- Integral and **known range**? → fixed-width `int*_t` / `uint*_t`
- Avoid `int`, `long`, `unsigned int`, `unsigned long`

Common source of errors:

- Comparing floating point variables for equality (rounding errors!).
- Over/Underflows in arithmetic operations

```
unsigned int i = 0;
std::cout << i - 1 << std::endl; // 4294967295
```



Automatic Type Conversion

- C/C++ **implicitly** tries to convert fundamental and derived data types



Potential loss of information: common source of errors.

- Implicitly converted are e.g.,:
 - integer
 - signed and unsigned types
- in:
 - Expressions with mixed types
 - Initializations
 - Boolean expressions
- Details on:
 - <https://en.cppreference.com/w/c/language/conversion>



Conversion of Arithmetic Types

- Expected conversions (e.g, from smaller to larger types, to bool):

```
int sI = 65;
unsigned short int usI = 5;

bool b1 = sI > usI; // expected conversion of unsigned short to int
bool b2 = sI / usI; // as above + conversion of int to bool
```



Conversion of Arithmetic Types

- Expected conversions (e.g, from smaller to larger types, to bool):

```
int sI = 65;
unsigned short int usI = 5;

// implicit conversion added by compiler as explicit casts (internally):
bool b1 = sI > (int) usI; // C-style cast to int
bool b2 = static_cast<bool>(sI / static_cast<int>(usI)); // C++-style cast to int
```



Conversion of Arithmetic Types

- Expected conversions (e.g, from smaller to larger types, to bool):

```
int sI = 65;
unsigned short int usI = 5;

// implicit conversion added by compiler as explicit casts (internally):
bool b1 = sI > (int) usI; // C-style cast to int
bool b2 = static_cast<bool>(sI / static_cast<int>(usI)); // C++-style cast to int
```

- Surprising conversion to unsigned type:

```
int a{0};
unsigned int b{1};
if (a - b > 0) cout << "Didn't expect that!" << endl;
```

- Why?



Conversion of Arithmetic Types

- What we wrote:

```
int a{0};
unsigned int b{1};
if (a - b > 0) std::cout << "Didn't expect that!" << std::endl;
```

- What the compiler implicitly adds:

```
int a = {0};
unsigned int b = {1};
if((static_cast<unsigned int>(a) - b) > 0) {
    std::operator<<(std::cout, "Didn't expect that!").operator<<(std::endl);
}
```

(generated with cppinsights.io)

Recommendation:

- enable compiler warnings `-Wconversion`, `-Wsign-conversion`



const

- const as notion of immutability



Constants (from "A Tour of C++")

C++ supports two notions of immutability:

- **const**: meaning roughly "I promise not to change this value" . This is used primarily to specify interfaces, so that data can be passed to functions cannot be modified. The compiler enforces the promise made by **const**.
- **constexpr**: meaning roughly "can be evaluated at compile time". This is used primarily to specify constants, to allow placement of data in memory where it is unlikely to be corrupted, and for performance.
- (**constexpr**: evaluated at compile time.)

Immutability is a very strong promise: it makes reasoning about program code easier and allows compilers to further optimize code. Use const if possible.



Constants – Examples

```
size_t i = 5;
size_t const j = 3;
size_t constexpr k = 4;

i = 9;           // correct
// j = 7;        // error! j is a constant
// k = 7;        // error! k is a constant

size_t const l = j + 4; // may be computed at run-time
size_t const m = i + 4; // computed at run-time

size_t constexpr n = k + 4; // 4 + 4 == 8: computed at compile time
// size_t constexpr o = i + 4; // error! because i is not constexpr

size_t const p = foobar(); // works if foobar() returns size_t
size_t constexpr q = foobar(); // only works if function foobar() is constexpr
```



Naming conventions



General naming conventions

- C++ is case sensitive and does not care about newlines (except macros, comments...)

Recommendation: Decide for one style to improve readability and stick to it in a project. Otherwise working in a team can get messy.

A commonly used style (used here to distinguish from default STL style):

- **Variables**: Identifier in lower case, in compound words the respective new word is started with an uppercase letter

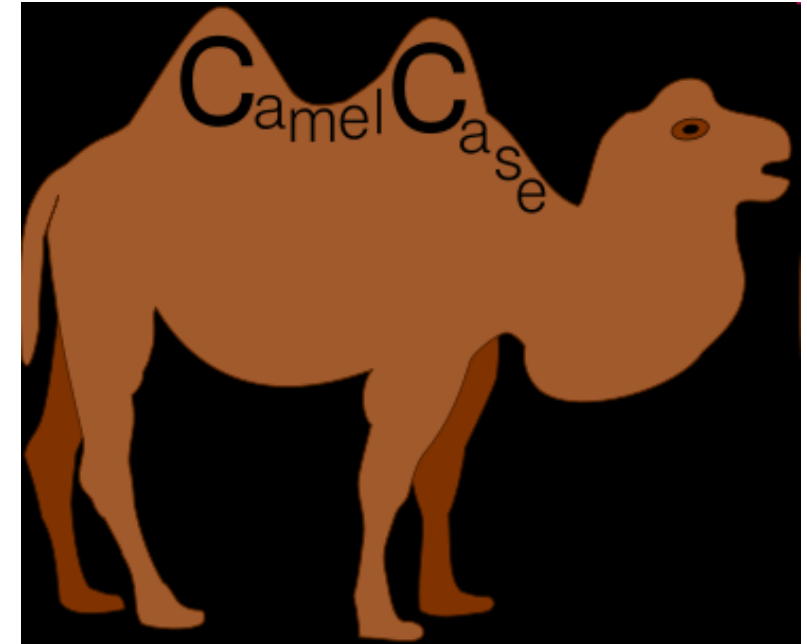
`gamesWon, otherTrack,
computersNumber, startValue`

- **Constants** (`const`): identifiers in capital letters, compound words are separated by underscore

`MAX_GUESSES, LEVEL_MAX, MAX_ITERATIONS`

General naming conventions

- This convention is called **CamelCase**
- **For compound words, each new word is started with a capital letter**
- Spaces around operators
- Use spaces to indent code blocks
- Good code is correct, efficient, easy to read and well commented



<http://en.wikipedia.org/wiki/File:CamelCase.svg>

Recommendation:

- Many IDEs support **clang format**. A great way to format your code automatically: just add a .clang-format file with the preferred style options.



Declaration vs. Definition



Declarations

Recall:

- **Declarations** introduce **names** (e.g., of a variable) in a program
- Declarations can appear at different places in the program.
- What you can declare (name):
 - Variables
 - Functions
 - Classes
 - Types
 - Structures / Unions
 - Enumerations
 - Name spaces
 - Labels
 - Preprocessor macros



Definition vs. declaration

- C++ allows declarations made in one file to be used in another
- C++ (using `#include`)
 - **Declaration introduces only the name**
 - **Definition reserves the memory space**
 - Linker ensures that a name refers exactly to one entity (**One-Definition-Rule**)



Expressions



Assignments and Expressions

- An **assignment** consists of:

```
variable = <expression>;
```

- Example:

```
v = 5;  
v = 3 * (5 + 7) ;
```

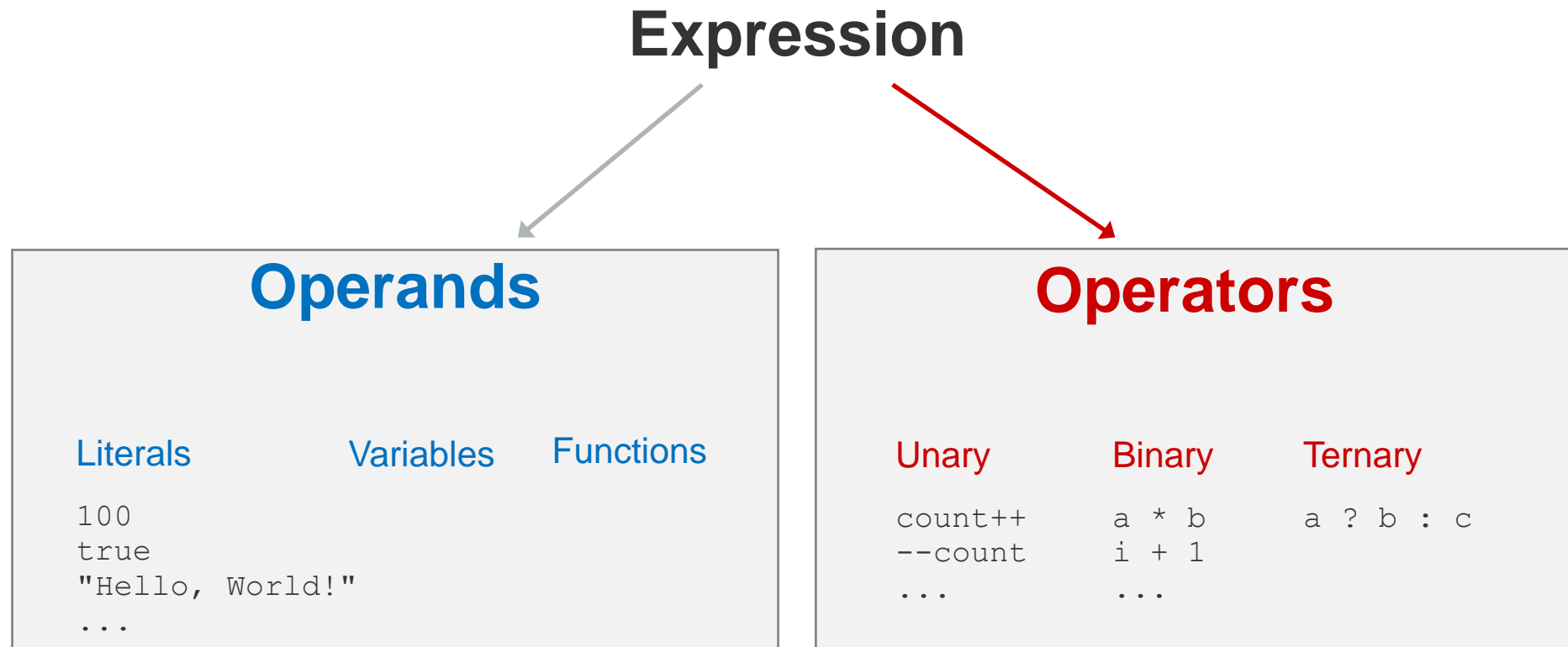
- **Consequence:** The assignment stores the value calculated from the expression (right side) in the variable (left side)
- If a variable declaration(-definition) is combined with an assignment a copy-initialization is performed.

```
<type> variable = <expression>;
```



Expressions

- Expressions are compositions of **operands** and **operators**, corresponding to mathematical equations

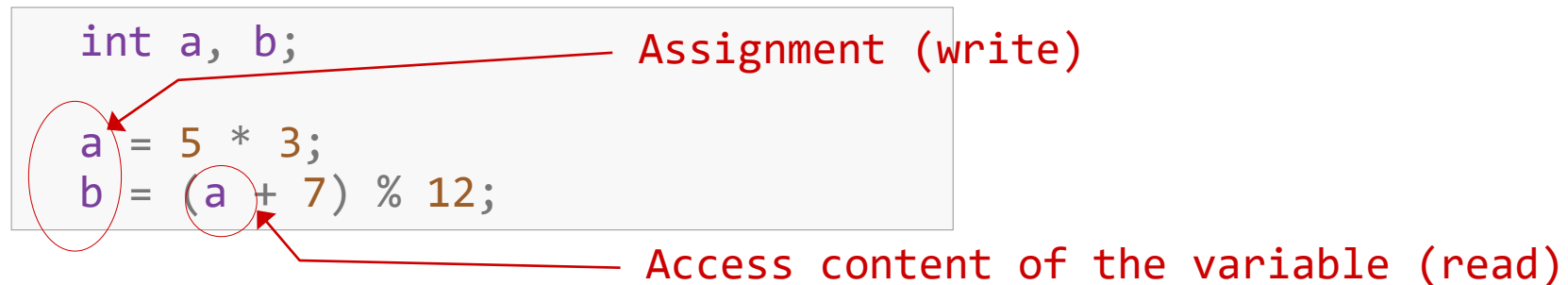




Expressions

Within an **expression** the **value of a variable** can be used by specifying the name.

- Writing happens on the left side of '=' (LHS – left hand side)
- Reading on the right side of '=' (RHS – right hand side)





Truth values (bool)

Boolean values: *True* or *False*

- **Constants for value assignments** for `bool`

- `false` and `true`
- (range of values is 0 and 1)

in C: `#include <stdbool.h>`

```
bool b;

b = true;  cout << b << endl; // 1
b = false; cout << b << endl; // 0
b = 22;    cout << b << endl; // 1, implicit conv. (0->false, otherwise true)
```

- **Boolean expressions** are formed with **logical operators**

- Negation: `!`
- AND: `&&`
- OR: `||`
- XOR: `^`



Boolean expressions

- Boolean expressions are formed with the help of
 - Comparison operations on arithmetic expressions
 - Logical operations on boolean expressions

```
const float x{5.0f};
float y{18.0f};
const int z{7};
bool a{true};
bool b = ((x > y) && (x < z)) || a;
```

b is always true
here, since a = true

- Tipp: Boolean expressions can be transformed or **simplified** (DeMorgan's laws)

$$\begin{aligned} (a \ \&\& \ b) \ || \ (a \ \&\& \ c) &\leftrightarrow a \ \&\& \ (b \ || \ c) \\ !a \ || \ !b &\leftrightarrow !(a \ \&\& \ b) \end{aligned}$$



Compound expressions

- **Compound expressions** are formed by the concatenation of operators and brackets

```
a = 5 * 3 + 5 - 6 / 4;           // Result: 19
b = (a >= 0) ^ (a < 4) && (4 < 5); // true (1)
```

- The compiler *internally* adds implicit brackets based on precedence and associativity
 - **Precedence** (ensures correct order e.g. "dot before dash")

- **Associativity**: example for hypothetical operator@: a @ b @ c

- left-associative (LR): = (a @ b) @ c
- right-associative (RL): = a @ (b @ c)

- Example:


```
cout << 5 << 6 << endl;           // is the same as
(((cout << 5) << 6) << endl);
```

Precedence rules for operators.

See https://en.cppreference.com/w/cpp/language/operator_precedence

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right →
2	a++ a-- type() type{} a() a[] . ->	Suffix/postfix increment and decrement Functional cast Function call Subscript Member access	
3	++a --a +a -a ! ~ (type) *a &a sizeof co_await new new[] delete delete[]	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT C-style cast Indirection (dereference) Address-of Size-of ^[note 1] await-expression (C++20) Dynamic memory allocation Dynamic memory deallocation	Right-to-left ←



Precedence rules for operators.

4	<code>.* ->*</code>	Pointer-to-member	Left-to-right →
5	<code>a*b a/b a%b</code>	Multiplication, division, and remainder	
6	<code>a+b a-b</code>	Addition and subtraction	
7	<code><< >></code>	Bitwise left shift and right shift	
8	<code><=></code>	Three-way comparison operator (since C++20)	
9	<code>< <= > >=</code>	For relational operators < and ≤ and > and ≥ respectively	
10	<code>== !=</code>	For equality operators = and ≠ respectively	
11	<code>a&b</code>	Bitwise AND	
12	<code>^</code>	Bitwise XOR (exclusive or)	
13	<code> </code>	Bitwise OR (inclusive or)	
14	<code>&&</code>	Logical AND	
15	<code> </code>	Logical OR	
16	<code>a?b:c</code>	Ternary conditional ^[note 2]	Right-to-left ←
	<code>throw</code>	throw operator	
	<code>co_yield</code>	yield-expression (C++20)	
	<code>=</code>	Direct assignment (provided by default for C++ classes)	
	<code>+= -=</code>	Compound assignment by sum and difference	
	<code>*= /= %=</code>	Compound assignment by product, quotient, and remainder	
	<code><<= >>=</code>	Compound assignment by bitwise left shift and right shift	
17	<code>&= ^= =</code>	Compound assignment by bitwise AND, XOR, and OR	
	<code>,</code>	Comma	Left-to-right →



Precedence Examples

```
int iVal = 7, oiVal = 3, rVal = 13;

rVal += 2 + 3 * 8 / 4 + 2;
rVal = ++iVal / oiVal--;
rVal = iVal << 2 >> 4 / 3;
rVal = (iVal & 5 || oiVal-- && 1) + 3;
rVal = iVal = oiVal = 0;
```

23
2
16
4
0

`++iVal` – first do the increment, then use the result in expression
`oiVal--` – use the current value in the expression then decrease by one

Notes:

- Precedence defines the **grouping** and **not the order** of evaluation.
- If unsure about **grouping** use brackets `()`



Do not rely on the order of execution. Brackets don't help here.

Example: `i + i++` vs. `i++ + i`; or `return a() + b() + c();`



References

- value semantics
- reference semantics



References

What happens when you initialize (or assign) one variable with another variable of the same type?

```
size_t i = 145;
size_t j = i; // j == 145

std::string s{"A text much longer than this example"};
std::string t{s}; // t == "A text...."
```

A **copy** is made!

- ! - C++ has **value semantics**. (Different to other programming languages)
- Copying large data is an **expensive operation**.
- - Even the time for copying many `std::string` can add up in practice.



References

- Sometimes we don't want to copy, we just want to **introduce a new name**, that's what **references** are for:

```
std::string s{"A text much longer than this example"};
std::string & t{s};
    // ^ reference symbol

std::cout << t << '\n'; // prints "A text..."

t = "a shorter text";
std::cout << s << '\n'; // prints "a shorter text"
    // ^ original variable has changed
```

- A variable of reference type behaves exactly like the original
- A variable of reference type has to be initialized when declared



References and const

- You can create a reference to `const` type, even if the original is not `const`:

```
std::string s{"A text much longer than this example"};
std::string const& t{s};
// t = "a shorter text"; // doesn't work, t is const
s = "a shorter text"; // works, also "updates" t
```

- The reverse is not possible:

```
std::string const s{"A text much longer than this example"};
// std::string & t{s}; // can not be converted to non-const
std::string const& u{s};
```



References – Summary

- **value semantics**: by default, variables are copied
- Copying small arithmetic types is fine
- Instead of copying you can create references **&** to existing variables (**reference semantics**)
- References behave just like the original (same syntax)
- The referred object **must** be set at time of declaration and **cannot** be changed later
- **const**-ness can be "added" via a reference
- **const**, **&** and **const &** are part of a variable's type, e.g. `int const & i;`
means `i` is of type "reference to integer constant"



Control Flow

- if, else, switch (enum)



Instructions and Sequences of Instructions

- Programs are consequences of **instructions**
 - During the execution of a program these sequences are **processed sequentially**
 - The execution of the instructions lead to the gradual change of the (program) state; one speaks of **imperative programming**
- Each statement must be terminated by a **semicolon: ;**
- **Exception:** If the statement ends with a curly bracket }, no ; follows.
(if you write a ; nevertheless, this is interpreted by the compiler as an **empty statement!**)

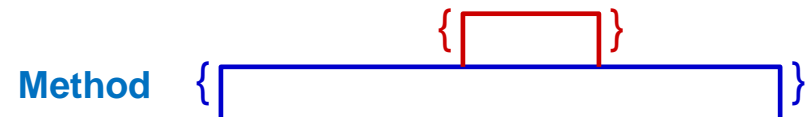


Groups of Instructions – Blocks

- Statements can be **"grouped"** into a block using {} (**compound statement**)

```
int main() {
    double d = 5.0;
    float f = 3.7f;
    {
        int i = (int) (d*f);
        i = i + 7;
    }
    return 0;
}
```

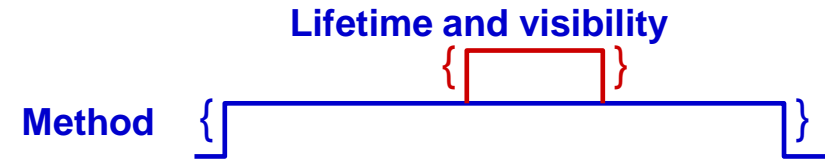
Lifetime and visibility





Blocks - Lifetime and Visibility of Variables

```
int main() {
    double d = 5.0;
    float f = 3.7f;
    {
        int i = (int) (d*f);
        i = i + 7;
    }
    int i = 4;
    return 0;
}
```



Notes:

- Variables are only valid within the block in which they are declared (e.g., the variable `i` (type `int`) is only known in the inner **block {}**).
- In enclosed blocks variables of the enclosing block can be used.
- **Variables must not be re-declared in the same block with the same name**



The `if` Statement

- The `if` statement allows separating program flow into **alternative paths** of different instructions.
- The simplest form (**simple branching**) executes a statement if the boolean expression evaluates to true.

```
if ( <boolean-expression> )
    <statement or block>
```

- Decision between **two alternatives** with **if/else**:

```
if ( <boolean-expression> )
    <statement or block>
else
    <statement or block>
```



Example: `if` Statement

- The values of two variables, `x` and `y`, are swapped if the value of `x` is greater than `y`.

```
if (x > y) {           // condition x > y true?, then execute block {...}
    int temp = x;      // temporary variable saves copy of x
    x = y;             // copy y to x
    y = temp;          // copy temp to y
}

// here: invariant x <= y holds
```



if Statement and ?-operator

- Example: max_xy should be the maximum of x and y

```
int max_xy;
if (x > y) {
    max_xy = x;
} else {
    max_xy = y;
}
```

- Syntactic sugar – the ternary ?-operator:



```
< boolean-expression > ? if_true : if_false;
int max_xy = (x > y) ? x : y;
```



Composition of Several else - if's

- Multiple `else-if` statements

```
if (<condition-1>)
    <statement-1>
else if (<condition-2>)
    <statement-2>
else if (<condition-3>)
    <statement-3>
    . . .
else if (<condition-N>)
    <statement-N>
[else
    <statement- (N+1) >]
```

// (more cases)

Like in many other languages:

- One condition after the other is evaluated **until a boolean expression returns true**
- The corresponding **statement is executed**, and all **remaining conditions are skipped**



Enumerations and switch

- Enumerations are user defined types representing a set of integer values by giving them names (prevent errors)
- **switch** is for multi-way branching
 - **case <label>**: jump to that branch
 - **default**: if none of the previous cases applied
 - **break**; end switch here
- What happens if break is omitted?

```
enum class Color
{
    red, green, blue
}; // ; is important

Color c = Color::blue;

switch (c) {
case Color::red:
    cout << "red" << endl; break;
case Color::green:
    cout << "green" << endl; break;
case Color::blue:
    cout << "blue" << endl; break;
default: // should not happen
    cout << "no color" << endl;
}
```



Enum – C vs. C++

- Strongly-typed `enum class`:

```
enum class Color {           (since C++11)
    red, green, blue
};

// scoped ↓
Color c = Color::blue;

// not implicitly convertible
// no arithmetic operators
```

- C-style `enum`:

```
enum Color {
    red, green, blue
};

// unscoped ↓
Color c = blue;

// implicitly convertible:
int i = c; // == 1

// arithmetic operators:
Color cc = red + green; // == blue
```

- **Rule-of-thumb: `enum class` is safer and thus preferred!**



Control Flow - Loops

- while, do...while, for



Loops

- Often instructions have to be executed several times
 - for different data
 - for enumerations
 - ...
 - to avoid code duplications

- Loop types:
 - `while` loop
 - `do...while` loop
 - `for` loop



The while Loop

```
while (<boolean-expression>) {
    <statements>
}
```

- When the program reaches a **while** statement, the condition is evaluated:
 - **false**: the program jumps over the **while** loop and continues with the next statement,
 - **true**: the instructions in the loop body are executed until the condition becomes false

```
int number{1};
while (number < 6) {
    cout << number << endl;
    ++number;
} // prints 1,2,3,4,5 in while loop
```



The do...while Loop

- The `do... while` loop **repeats** a statement, **as long as** a **condition** is **met**

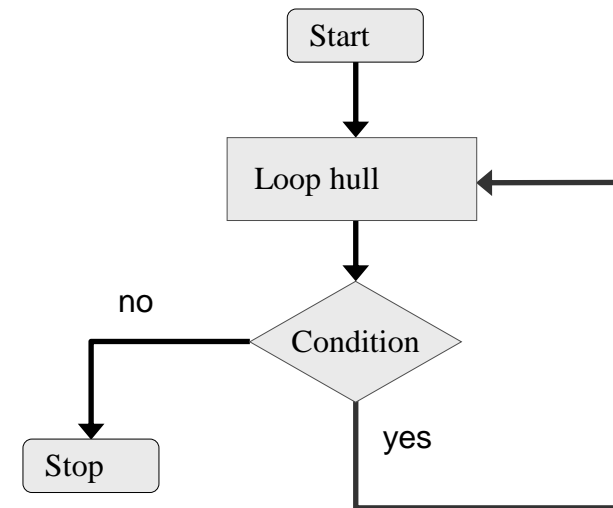
```
do {
    // loop body
} while (boolean-expression);
```

- Note:** The instruction(s) in the loop body are executed at least once!

```
int fak = k = 1;

do {
    fak = fak * k;
    ++k;
} while (k <= 7);

// fak == 5040 (7!)
```





The for loop

```
for (<preparation>; <(start) condition>; <continuation>) {
    <body>;
}
```

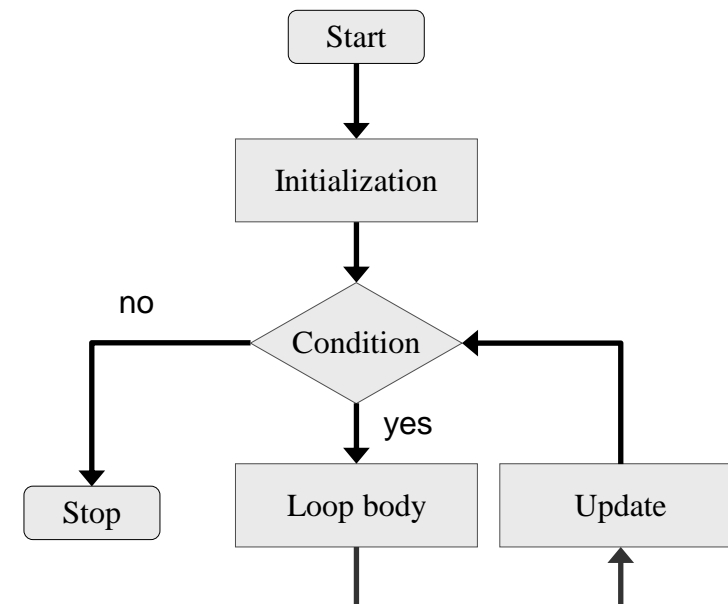
- **< Preparation> is executed once before the start of the loop**
 - one or more comma-separated **statements** (**initialization**: count variable) or a **variable declaration**
- **<(start) condition> decides whether the body is executed or not**
 - a boolean **expression**
- **<continuation> is performed after end of body is reached**
 - one statement or several statements separated by commas
- **<body> is executed once per loop pass**
 - any instruction or
 - a sequence of statements enclosed in {... } (a **block**)



Execution and Processing of a for Loop

1. The loop is **initialized** (count variable)
2. The **condition** is evaluated
3. If the condition is met, the **loop body** is executed
4. The **continuation** is executed (change of the count variable(s))
5. Repeat with 2.

```
int fak = 1;
for (int k = 1; k <= 7; ++k) {
    fak = fak * k;
}
// fak == 5040 (7!)
```





Nested Loops

Loop bodies can be composed of **any instructions**:

→ Loop bodies can also contain **loops**

Example: Output matrix with entries set to $\text{row} * \text{col}$

```
for (int row = 1; row <= 10; ++row) {
    printf("%4d | ", row);
    for (int col = 1; col <= 10; ++col) {
        // print 4-digit columns
        printf("%4d", n * row);
    }
    printf("\n"); // new line
}
```

1		1	2	3	4	5	6	7	8	9	10
2		2	4	6	8	10	12	14	16	18	20
3		3	6	9	12	15	18	21	24	27	30
4		4	8	12	16	20	24	28	32	36	40
5		5	10	15	20	25	30	35	40	45	50
6		6	12	18	24	30	36	42	48	54	60
7		7	14	21	28	35	42	49	56	63	70
8		8	16	24	32	40	48	56	64	72	80
9		9	18	27	36	45	54	63	72	81	90
10		10	20	30	40	50	60	70	80	90	100



Nested Loops (2)

- Often the **number of iterations of the inner loop** depends on the **run variable of the outer loop**

Example: Sum of all numbers 1..n

```
for (int n = 1; n <= 20; ++n) {
    int sum = 0;
    for (int k = 1; k <= n; ++k) {
        sum += k;
    }
    std::cout << n << ": " << sum << std::endl;
}
```

```
1: 1
2: 3
3: 6
4: 10
5: 15
6: 21
7: 28
8: 36
9: 45
10: 55
11: 66
12: 78
13: 91
14: 105
15: 120
16: 136
17: 153
18: 171
19: 190
20: 210
```




More Details about for Loops

- A **variable declared within the preparation** is valid only **within the** loop

```
int fak = 1;

for (int k = 1; k <= 7; ++k)
    fak = fak * k;
```

The test in the termination condition is always a source of error: *should the last value contribute to the result or not?*

- Incremental** counting loop

```
for (<variable> = <min>; <variable> <= <max>; ++<variable>) {
    <statements>;
}
```

- Decremental** counting loop

```
for (<variable> = <max>; <variable> >= <min>; --<variable>) {
    <statements>;
}
```



Loops – Use and Properties

- Each of the 3 types of loops can be replaced by both others, each has **special purpose**
- **while** vs. **do... while** loop
 - With the **do... while** loop, the loop body is executed once in any case, **even if the condition is not fulfilled at the beginning**
 - **Note:** use with care, as it is a frequent **source of errors**
- **for** vs. **while** loop
 - The **for** loop is especially good for **repeating** the loop body a **certain number of times**.
Easier to parallelize!
 - **while/do... while** harder to reason if it terminates.

```
while ( n != 1 ) {           // does it terminate for any positive n?
    if ( n % 2 == 0 ) {      // n is even
        n = n/2;
    } else {                // n is odd
        n = 3*n + 1;
    }
}
```

Example: Collatz conjecture



Example - Output of Number Sequences

Task:

- Print all even numbers between 2 and 20: (2, 4, 6, 8, 10, 12, 14, 16, 18, 20)

- **for loop** with 2 increment

```
for (int n = 2; n <= 20; n += 2)
    cout << n << " ";
```

- Equivalent **while loop** with 2 increment

```
int n = 2;
while (n <= 20) {
    cout << n << " ";
    n += 2;
}
```



Example - Output of Number Sequences (2)

- `do... while` loop with increment 2

```
int n = 0;
do {
    n += 2; // n is incremented BEFORE output
    cout << n << " ";
} while (N < 20);
```

- Which one is easier to read?

```
for (int n = 2; n <= 20; n += 2)
    cout << n << " ";
```

```
int n = 2;
while (n <= 20) {
    cout << n << " ";
    n += 2;
}
```

```
int n = 0;
do {
    n += 2;
    cout << n << " ";
} while (N < 20);
```



The continue Statement

Execution immediately jumps to the **end of the current iteration** and starts directly with the execution of the next loop pass / check of the condition

Example: Output of even numbers (naive approach)

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 1) continue; // skip odd numbers
    cout << i << endl;
}
```

```
int i = 0;
while (i < 10) {
    if (i % 2 == 1) {
        i++;
        continue;
    }
    cout << i << endl;
    i++;
}
```



The break Statement

The `break` statement

- Execution of the enclosing `while`, `do... while`, `for` or `switch` statement **is terminated immediately**
- **Example: Calculate and print prime numbers (naive approach)**

```
for (int k = 1; k < 10000; k++) {
    int j;
    for (j = 2; j < k; j++) {
        if ((k % j) == 0)
            // as soon as a divider is found
            break;
    }
    if (k == j)
        cout << " " << k << endl;
}
```

enclosing loop
(relative to the
`break`
statement)

Returns result: prime numbers from 1 to 10000



Empty statements can be sources of errors

- In a statement that consists only of ';', **no action** is performed
- Incorrectly placed semicolons can lead to working programs with semantic errors!

```
for (int k = 0; k < 10; k++);  
    cout << "Hello" << endl;
```

Such errors are often difficult to find!

- **Recommendation:** tools like clang-tidy flag code (that is formally correct) **but likely** a semantic error. Supports many checks.

Extra Clang Tools 6 documentation

CLANG-TIDY - MISC-SUSPICIOUS-SEMICOLON

« [misc-suspicious-missing-comma](#) :: [Contents](#) :: [misc-suspicious-string-compare](#) »

misc-suspicious-semicolon

Finds most instances of stray semicolons that unexpectedly alter the meaning of the code. More specifically, it looks for `if`, `while`, `for` and `for-range` statements whose body is a single semicolon, and then analyzes the context of the code (e.g. indentation) in an attempt to determine whether that is intentional.



Assertions

<cassert>

- Detect logical errors / impossible situations.
- A simple **Design by contract** technique.
- Assertions allow us to test **post-/preconditions** or **loop invariants**.

`assert(<expression>);`

- If the expression evaluates to false:
 - print expression, source filename, and line number to standard error. `abort()`.

```
assert(n == 0);           // precondition
do {
    n = n + 2;
    assert(n % 2 == 0); // invariant: n is always even
} while (n < 20);
assert(n == 2*20);       // postcondition
```

- Disabled in the optimized Release builds (no performance cost!), enabled in Debug builds



Functions



Motivation – from Block to Function

So far: blocks allow

- structuring program code
- are necessary to work with control statements and loops (**if**, **while**, **for**, ...).

But don't allow modular, reusable code.

Functions allow modularization of code and help to prevent code duplication:

```
return_type function_name(arg1_type arg1_name, arg2_type arg2_name,...) // (1)
{
// ...
}
```

- First line is called the **function signature**, everything between the { } is called **function body**
- The whole thing is called **function definition**
- A function signature without body is called a **function declaration**



Example:

```
/* Minimum of a and b */
int minimum(int a, int b) {
    if (a < b)
        return a;
    else
        return b;
}
```

- Call:

```
int main() {
    int i = minimum(3, 7);
    int x = 2, y = -4;
    cout << minimum(x, y) << endl;
}
```



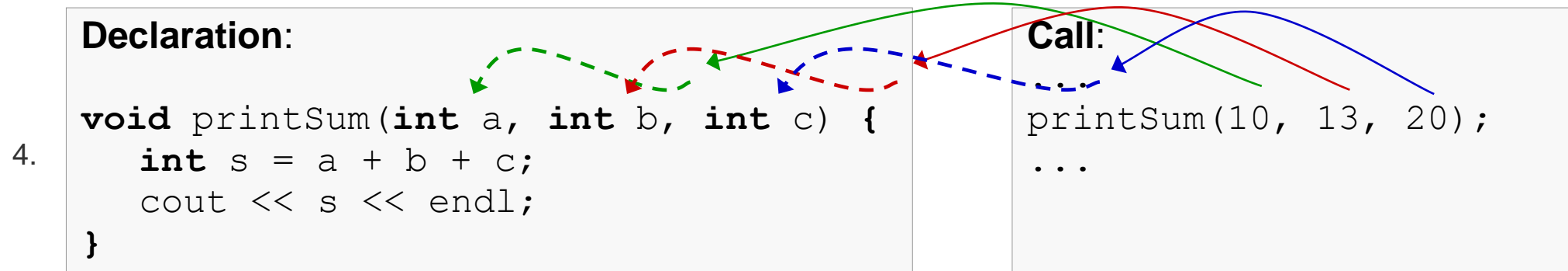
Example:

```
/* Function with default arguments */
float temp(float a, bool b = true) {
    ...
}
// call:
temp(1.0, false);
temp(1.0);           // => temp(1.0, true)
```



Function call / return value

- The parameters are passed when calling the method
- The **method call is processed** according to the following scheme:
 1. The **current parameters are evaluated**; the parameters can be variables, constants or results of the evaluation of expressions. Evaluation in unspecified order.
 2. The **parameter values** are **assigned to the formal parameters** (parameter passing and substitution);



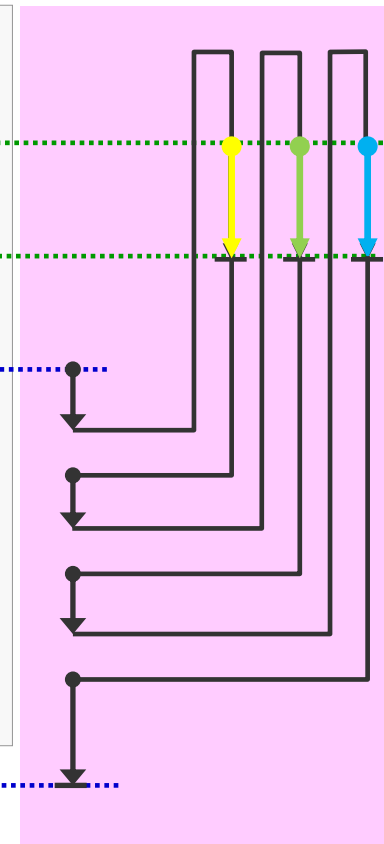
5. The body of the called method (subroutine) is executed, i.e. as if a block with instructions is inserted into the program flow at the call position.
6. The (optional) **return value** of the function is the result of the evaluation of the function call



Program text

```
void foo(...) {  
    ...  
}  
  
int main(...) {  
    ...  
    foo(...);  
    ...  
    foo(...);  
    ...  
    foo(...);  
    ...  
}
```

Program sequence





Example: sum of digits

```
int sumOfDigits(int num) {
    int sum = 0;
    while (num) {
        sum += (num % 10);
        num /= 10;
    }
    return sum;
}

int main() {
    int a = 1234;
    cout << sumOfDigits(11) << endl;
    cout << sumOfDigits(a) << endl;
    a = sumOfDigits(12 + 42 + 12 * 22);
    cout << a << endl;
}
```

num = 11

return 2

sumOfDigits(11)
-> 2



Example: sum of digits

```
int sumOfDigits(int num) {
    int sum = 0;
    while (num) {
        sum += (num % 10);
        num /= 10;
    }
    return sum;
}

int main() {
    int a = 1234;
    cout << sumOfDigits(11) << endl;
    cout << sumOfDigits(a) << endl;
    a = sumOfDigits(12 + 42 + 12 * 22);
    cout << a << endl;
}
```

num = 1234

return 10

sumOfDigits(1234)
->10



Example: sum of digits

```
int sumOfDigits(int num) {
    int sum = 0;
    while (num) {
        sum += (num % 10);
        num /= 10;
    }
    return sum;
}

int main() {
    int a = 1234;
    cout << sumOfDigits(11) << endl;
    cout << sumOfDigits(a) << endl;
    a = sumOfDigits(12 + 42 + 12 * 22);
    cout << a << endl;
}
```

num = 318

return 12

sumOfDigits(318)
-> 12



Type Checking

```
int sumOfDigits(int num) {
    ...
    return sum;
}

int main() {
    double a = 1234.3;
    char myChar[] = "asdf";

    cout << sumOfDigits(a) << endl;           // ok : automatic conversion to int -> 10
    cout << sumOfDigits("asdf") << endl;      // error: invalid conversion from 'char*' to 'int'
    cout << sumOfDigits(a) << endl;

    a = sumOfDigits(11);                       // ok : automatic conversion int to double -> 2.
    myChar = sumOfDigits(11);                  // error: incompatible types in assignment of 'int'
    to 'char [5]'
}
```



Example: `int main(int argc, char *argv[])`

Any number of parameters can be passed to a program:

- `argc` contains the number of parameters (always ≥ 1)
- `argv` is an array of char strings (next lecture)
- `argv[0]` is always the called filename (executable)
- (`argv[1]` if given: char string of the first call parameter, ...)



Example: `int main(int argc, char *argv[])`

```
#include <iostream>
#include <stdlib.h>

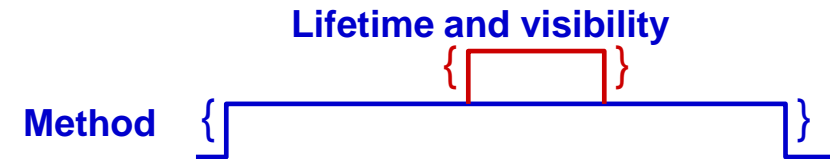
int main(int argc, char *argv[]) {

    for (int i = 0; i < argc; ++i) {
        std::cout << "Arg " << i
                    << ": " << argv[i] << std::endl;
    }
    const int xDim = atoi(argv[1]); // convert char* to int
    const int yDim = atoi(argv[2]);
    std::cout << "x: " << xDim
                << "y: " << yDim << std::endl;

    return 0;
}
```

Recall: Blocks - Lifetime and Visibility of Variables

```
int main() {  
    double d = 5.0;  
    float f = 3.7f;  
    {  
        int i = (int) (d*f);  
        i = i + 7;  
    }  
    int i = 4;  
    return 0;  
}
```



Notes:

- Variables are only valid within the block in which they are declared (e.g. the variable `i` (type `int`) is only known in the inner **block {}**).
- In enclosed (inner) blocks variables of the enclosing (outer) block may be used
- **Variables must not be re-declared in the same block with the same name**



Global and Local Variables

A function can:

- use the **formal parameters**
- introduce new **local variables**
- access all currently visible variables: **global variables**
- introduce new **static variables** that are represented only once and exist until the end of the program execution (later more...)



```
#include <iostream>
#include <string.h>
#include <cstring>

using namespace std;

int globalNum; // a global variable

int myFunc( char * txt ) {
    static int statNum = 0;    // a novel static variable
    int localNum = strlen( txt ); // a local variable

    statNum += localNum;
    globalNum++;

    return statNum;
}

int main(int argc, char * argv[]) {
    globalNum = 0;    // globalNum is also global to main
    int letters;

    for (int i = 0; i < argc; ++i ) {
        letters = myFunc( argv[i] );
    }

    cout << "number of arguments: " << argc << endl ;
    cout << "number of letters: " << letters << endl;
    cout << "number of calls to myFunc: " << globalNum << endl;
}
```



Function Parameters and References

```
// "parameter" ↓
double square(double const d) {
    return d * d;
}
```

```
// "argument" ↓
double de = square(3.4);
```

```
void print(std::string const &str) {
    std::cout << str << '\n';
}
```

```
std::string s{"test"};
print(s);
```

- **Parameters** declare new variables in the scope of the function that are initialized with the respective **arguments** of the function call.
- All rules we discussed to declaring variables apply, especially:
- **If you forget the &, values are copied on the stack to be available in the function! With '&' only its address!**
- **No big deal for small types but copying strings can already lead to bottlenecks.**
- Hint: make parameters **const** if possible



Function Parameters and References

argument	copy	can be changed in function	potential side effect?
<code>X x</code>	yes	yes	no
<code>const X& x</code>	no	no	no
<code>X& x</code>	no	yes	yes

Recommendations: Ask yourself: do I want to have change the variable?

no:

1. primitive type? → no real disadvantage to pass by value (**with or without `const`**).
2. else copy could be expensive → **`const &`**

yes:

1. so that change is visible outside? → we need **`&`**
2. change only inside function? → neither **`const`** nor **`&`** (**copy**)



Function Parameters – more Examples

- Arithmetic

```
double square1(double const d)
{
    return d * d;
}
double de = square1(3.4);
```

- Call-by-reference

```
double square2(double & const d)
{
    return d * d;
}
void square3(double & d)
{
    d = d * d;
}
double df{3};
double de = square2(df); // after call: df == 3
square3(df);           // after call: df == 9
```

- General

```
void append_foo(std::string str)
{
    str += "foo";
    std::cout << str << '\n';
}

std::string s{"test"};
append_foo(s); // prints "testfoo", s remains "test"
```

- Call-by-reference

```
void append_foo(std::string & str)
{
    str += "foo";
    std::cout << str << '\n';
}

std::string s{"test"};
append_foo(s); // prints "testfoo ", s is now "testfoo"
```



Functions as function arguments

standard call:

```
int add(int x, int y) { // Function: add two numbers
    return x + y;
}
double sum = add(3, 5) // call add(...)
```

A function can
also be an argument
to another function:

```
int add(int x, int y) { // Function: add two numbers
    return x + y;
}

int multiply(int x, int y) { // Function: multiply two numbers
    return x * y;
}

int apply(int x, int y, int (*f)(int,int)) {
    return f(x, y); // call f(x, y) and return result
}

double sum = apply(3, 5, &add); // sum == 8
double product = apply(3, 5, &multiply); // product == 15
```



Extra: Formatted output



printf for formatted output - syntax

Older C++ code still uses the C-style function `printf` for printing formatted strings:

```
printf( <formatstring>, <list of variables> );
```

Format string:

- Type specification
 - %d or %i - Output of integer values
 - %u - Output of unsigned values
 - %f - Output of floating point numbers
- Formatting:
 - %< number>[d|u|f] - right justified with < number> digits
 - %.<number>f - number of decimal places
 - %.<number>d - number of output digits
 - \n - new line
- Detailed syntax: <https://www.cplusplus.com/reference/cstdio/printf/>



printf for formatted output

- Simple formatting / tabular output easier than with cout
- C-style → newer C++ use stream manipulators, `std::format` (since C++20) or `fmt`

```
#include <stdio.h>
int main() {
    int varA = 3;
    double varD = 3.32314;
    printf("int      out %6d \n", varA);
    printf("double out %6.3f \n", varD);
    return 0;
}
```



Summary



Summary

- Variables and types:
`int, float, char, ...`
- Instructions and operators:
`=, *, -, /, ...`
- Control flow:
`if, else`
`while, for, do .. while`
- Functions
passing parameters, references, const

With these ingredients you can already implement simple algorithms