

C++-Programming – Beginner

An Introduction to the Fundamentals -

Center for Scientific Computing

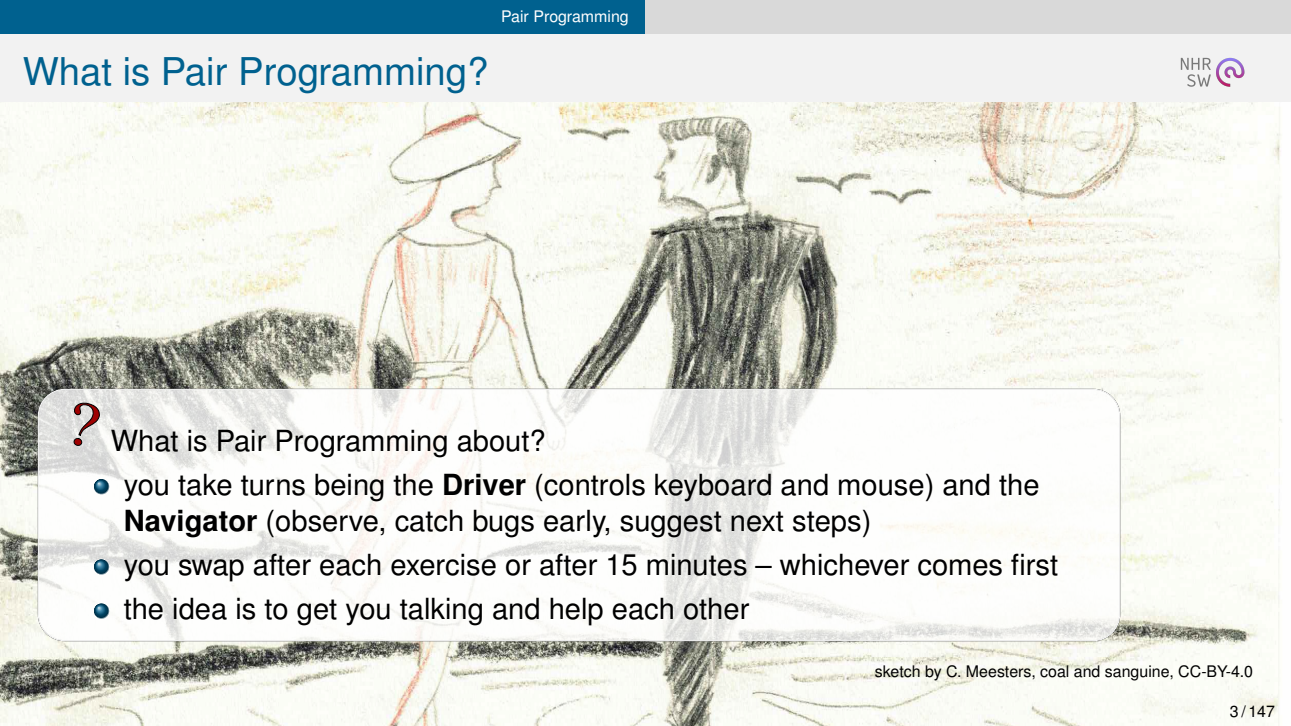
October 2026



- 1 About C++
- 2 Resources
- 3 Hello World

- 4 Comments and Code Blocks
- 5 Types and Operations
- 6 Loops and Conditionals
- 7 Types and Operations II: Containers

What is Pair Programming?

- 
- ?
- What is Pair Programming about?
- you take turns being the **Driver** (controls keyboard and mouse) and the **Navigator** (observe, catch bugs early, suggest next steps)
 - you swap after each exercise or after 15 minutes – whichever comes first
 - the idea is to get you talking and help each other

sketch by C. Meesters, coal and sanguine, CC-BY-4.0

Outline

- 1 About C++
- 2 Resources
- 3 Hello World

- 4 Comments and Code Blocks
- 5 Types and Operations
- 6 Loops and Conditionals
- 7 Types and Operations II: Containers
- 8 Functions

Section 2

About C++

This Chapter is about:

- The language itself
- The course

The authors are extremely grateful for the template material of

André Müller	HPC-group, JGU
Philipp Sitzmann	System Vertrieb Alexander GmbH

Furthermore we thank the following reviewers:

- none yet

C++? Characteristics



control	over resources (CPU, memory, ...)
abstraction	convenient & zero or very low overhead
performance	"you don't pay for what you don't ask for"

ISO Standard C++

C++98	1998	original standard
C++11	2011	many novelties, almost new language
C++14	2014	improvements in standard and implementation
C++17	2017	again many novelties, incl. new types
C++20	2020	introduction of concepts & ranges
C++23/26	upcoming	executors, reflection, networking, ...

Pros and Cons

Suggested when

high performance demands
tight memory constraints
system oriented programming

“number crunching”
massively parallel software

Discouraged when

“scripting will do”
rapid prototyping
REPLs (read-evaluate-print-loops $\Rightarrow$ interactive interpreters)

C vs. C++

C

embedded programming, OS, etc.
top-down approach
procedural
no native string type
etc.

C++

anything
bottom-up approach
procedural, OOP
vast standard library (incl. strings)
etc.

C vs. C++

C

embedded programming, OS, etc.
top-down approach
procedural
no native string type
etc.

C++

anything
bottom-up approach
procedural, OOP
vast standard library (incl. strings)
etc.

Good News

Most code written in C should also work with C++ or is often reusable as an external function.

C++ compared to other Languages I

Just a sample . . .

C++

compiled
pot. little RAM
programmer controls object
lifetime
etc.

Python

interpreted
+/-
interpreter controls object
lifetime
etc.

Java

JIT
lots of RAM
run time engine controls
object lifetime
etc.

C++ compared to other Languages I

Just a sample . . .

C++

compiled
pot. little RAM
programmer controls object
lifetime
etc.

Python

interpreted
+/-
interpreter controls object
lifetime
etc.

Java

JIT
lots of RAM
run time engine controls
object lifetime
etc.



Warning

| Each “etc.” may cause a flamewar!

C++ compared to other Languages II

- Another popular programming language in scientific computing: **Fortran**
- Fortran means FORMula TRANslator
- Fortran is especially designed to
 - translate formulas into code such that the formula remains visible
 - yield fast code
 - provide built-in capability for matrices and vectors
- Fortran is **not** designed as a “general purpose” language as C or C++
- Many scientific codes are written in Fortran

About this Course I

You will learn:

- Basics of C++:

About this Course I

You will learn:

- Basics of C++:
 - Structure of a C++-program

About this Course I

You will learn:

- Basics of C++:
 - Structure of a C++-program
 - Variables and Types

About this Course I

You will learn:

- Basics of C++:
 - Structure of a C++-program
 - Variables and Types
 - Functions and Loops

About this Course I

You will learn:

- Basics of C++:
 - Structure of a C++-program
 - Variables and Types
 - Functions and Loops
- How to compile a program and interpret compiler messages.

About this Course I

You will learn:

- Basics of C++:
 - Structure of a C++-program
 - Variables and Types
 - Functions and Loops
- How to compile a program and interpret compiler messages.
- Object-Oriented-Programming (OOP) – in subsequent course

About this Course I

You will learn:

- Basics of C++:
 - Structure of a C++-program
 - Variables and Types
 - Functions and Loops
- How to compile a program and interpret compiler messages.
- Object-Oriented-Programming (OOP) – in subsequent course
- about the Standard Library (STL) – do not re-invent the wheel! – mostly in subsequent course

About this Course I

You will learn:

- Basics of C++:
 - Structure of a C++-program
 - Variables and Types
 - Functions and Loops
- How to compile a program and interpret compiler messages.
- Object-Oriented-Programming (OOP) – in subsequent course
- about the Standard Library (STL) – do not re-invent the wheel! – mostly in subsequent course
- hopefully: How to write clean (=good) code. – in the next course

About this Course I

You will learn:

- Basics of C++:
 - Structure of a C++-program
 - Variables and Types
 - Functions and Loops
- How to compile a program and interpret compiler messages.
- Object-Oriented-Programming (OOP) – in subsequent course
- about the Standard Library (STL) – do not re-invent the wheel! – mostly in subsequent course
- hopefully: How to write clean (=good) code. – in the next course

About this Course I

You will learn:

- Basics of C++:
 - Structure of a C++-program
 - Variables and Types
 - Functions and Loops
- How to compile a program and interpret compiler messages.
- Object-Oriented-Programming (OOP) – in subsequent course
- about the Standard Library (STL) – do not re-invent the wheel! – mostly in subsequent course
- hopefully: How to write clean (=good) code. – in the next course



Warning

I You will *not* learn programming in this course – any course!

About this Course II – Organizational Aspects

- We will start with a bit of theory, but then . . .

About this Course II – Organizational Aspects

- We will start with a bit of theory, but then . . .
- You will receive tasks, along with the necessary info.

About this Course II – Organizational Aspects

- We will start with a bit of theory, but then ...
- You will receive tasks, along with the necessary info.
- Some bits of theory ...

About this Course II – Organizational Aspects

- We will start with a bit of theory, but then ...
- You will receive tasks, along with the necessary info.
- Some bits of theory ...
- A task ...

About this Course II – Organizational Aspects

- We will start with a bit of theory, but then ...
- You will receive tasks, along with the necessary info.
- Some bits of theory ...
- A task ...
- Some bits of theory...

About this Course II – Organizational Aspects

- We will start with a bit of theory, but then ...
- You will receive tasks, along with the necessary info.
- Some bits of theory ...
- A task ...
- Some bits of theory...
- A task ...

About this Course II – Organizational Aspects

- We will start with a bit of theory, but then ...
- You will receive tasks, along with the necessary info.
- Some bits of theory ...
- A task ...
- Some bits of theory...
- A task ...
- ...

About this Course II – Organizational Aspects

- We will start with a bit of theory, but then ...
- You will receive tasks, along with the necessary info.
- Some bits of theory ...
- A task ...
- Some bits of theory...
- A task ...
- ...

About this Course II – Organizational Aspects

- We will start with a bit of theory, but then ...
- You will receive tasks, along with the necessary info.
- Some bits of theory ...
- A task ...
- Some bits of theory...
- A task ...
- ...

There is a folder with many example files. You may use these examples to remind yourself, how some construct worked. Find these files in

`code/examples/`

About this Course II – Organizational Aspects

- We will start with a bit of theory, but then ...
- You will receive tasks, along with the necessary info.
- Some bits of theory ...
- A task ...
- Some bits of theory...
- A task ...
- ...

There is a folder with many example files. You may use these examples to remind yourself, how some construct worked. Find these files in

`code/examples/`

You should already received the slides.

Outline

- 1 About C++
- 2 Resources**
- 3 Hello World

- 4 Comments and Code Blocks
- 5 Types and Operations
- 6 Loops and Conditionals
- 7 Types and Operations II: Containers
- 8 Functions

Section 3

Resources

This Chapter is about:

- Resources which get you further in your quest for C⁺⁺-programming
 - Books
 - Web Sites
 - Compiler
 - Tools (IDEs, Profilers, Debuggers)

Yet another Disclaimer



Info

- You will get the following slides in the handout – as any other slide (except for meta-slides like this one).
- Every following list is not a comprehensive listing. It displays our current(!) view on resources.
- Please share any idea of what should be on those lists with us.
- We will skim over the following slides, because anything else will be boring.

Books

Programming: Principles and Practice using C++

A Tour of C++

Effective C++

Discovering Modern C++

Bjarne Stroustrup

Bjarne Stroustrup

Scott Meyers

Peter Gottschling

Web Resources

- www.cppreference.com
standard library reference with examples
- www.cplusplus.com
standard library reference with examples
- <https://github.com/isocpp/CppCoreGuidelines>
programming guidelines for professional C++-developers

Don't forget a well-known Video Site ...

Basically, there are 3 possible compilers (on Linux):

- GNU g++
- LLVM clang++
clang.llvm.org/get_started.html
- Intel ICC

Basically, there are 3 possible compilers (on Linux):

- GNU g++
- LLVM clang++
clang.llvm.org/get_started.html
- Intel ICC

There is Microsoft's Visual C++ – but we are only going to discuss scientific programming, hence programming on Linux.

Editors & IDEs

You can choose between:

- editors: `vi`, `emacs`, `nano` and `gedit` – for novices we recommend `gedit`

Editors & IDEs

You can choose between:

- editors: `vi`, `emacs`, `nano` and `gedit` – for novices we recommend `gedit`
- IDEs (Integrated Development Environments): the current go-to IDE is Microsoft's Visual Studio Code (often just "`code`" on the command line)

Profiling & Debugging Tools

We will introduce advanced tools (briefly) at the end of this course.

Outline

- 1 About C++
- 2 Resources
- 3 Hello World**

- 4 Comments and Code Blocks
- 5 Types and Operations
- 6 Loops and Conditionals
- 7 Types and Operations II: Containers
- 8 Functions

Section 4

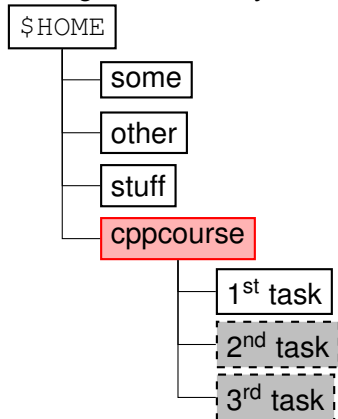
Hello World

This Chapter is about:

- Your first program – to be compiled and run.
- We will analyse its components and ...
- ...in contrast to any other task, you will be following on-screen instructions. (From here-on you will receive the necessary instructions on paper.)

Directory Layout

During the course, you best use a dedicated directory:



Task

Please create a directory `cppcourse` with subdirectory `01` (using `mkdir`).

The first Program



Task

Please follow the on-screen instructions, open an editor, type and save the following code under the name `hello.cpp`

```
#include <iostream>

int main() {
    std::cout << "Hello World from C++!" << std::endl;

    return 0;
}
```

The first Program

Task

Please follow the on-screen instructions, open an editor, type and save the following code under the name `hello.cpp`

```
#include <iostream>

int main() {
    std::cout << "Hello World from C++!" << std::endl;

    return 0;
}
```

Compiling

In order to run this program, we have to *compile* it...

Interlude – What is a compiler?

Compiler

A compiler translates source code into machine code, i.e. into a representation the hardware can deal with.



Interlude – What is a compiler?

Compiler

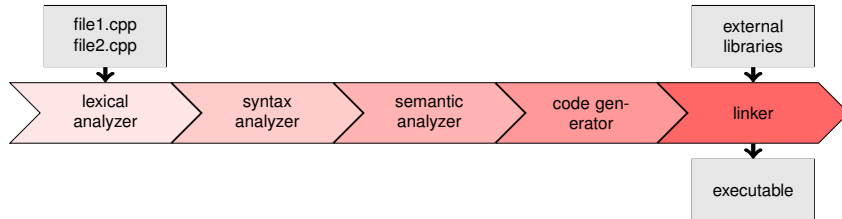
A compiler translates source code into machine code, i.e. into a representation the hardware can deal with.



Conclusion:

| The compiler is a black box!?

Interlude – How does a Compiler work?



In Essence

The compiler analyzes the source code, checks for compliance with the C++ language standard, includes the necessary external libraries and outputs the executable.

Compile and Run

For such a simple program, you have 2 options to compile:

- 1 Invoke the (GNU) compiler directly:

```
$ g++ hello.cpp -o hello
```

Use flag `-o` to indicate the desired filename for the executable.

Compile and Run

For such a simple program, you have 2 options to compile:

- 1 Invoke the (GNU) compiler directly:

```
$ g++ hello.cpp -o hello
```

Use flag `-o` to indicate the desired filename for the executable.

- 2 Run make:

```
$ make hello
```

Compile and Run

For such a simple program, you have 2 options to compile:

- 1 Invoke the (GNU) compiler directly:

```
$ g++ hello.cpp -o hello
```

Use flag `-o` to indicate the desired filename for the executable.

- 2 Run make:

```
$ make hello
```

Compile and Run

For such a simple program, you have 2 options to compile:

- 1 Invoke the (GNU) compiler directly:

```
$ g++ hello.cpp -o hello
```

Use flag `-o` to indicate the desired filename for the executable.

- 2 Run make:

```
$ make hello
```

Command `make`

`make` only works without a so-called `Makefile` for such small input, where the given keyword matches the `.cpp` file.

Running the Program

Until now, we completed the following steps:

- Write the program, i.e. writing the source code into a text file
- Compiling the program, i.e. passing the source code file to the compiler to get the executable.

Our executable is named `hello` and is a file in our directory (check with `ls`).

Running the Program

Until now, we completed the following steps:

- Write the program, i.e. writing the source code into a text file
- Compiling the program, i.e. passing the source code file to the compiler to get the executable.

Our executable is named `hello` and is a file in our directory (check with `ls`). Run the program by typing

```
$ ./hello
```

Running the Program

Until now, we completed the following steps:

- Write the program, i.e. writing the source code into a text file
- Compiling the program, i.e. passing the source code file to the compiler to get the executable.

Our executable is named `hello` and is a file in our directory (check with `ls`). Run the program by typing

```
$ ./hello
```

Expected output:

```
$ ./hello  
Hello World from C++!
```

Step by Step Analysis I

```
#include <iostream>

int main() {
    std::cout << "Hello World from C++!" << std::endl;

    return 0;
}
```

This will expose the `iostream` library to the compiler/linker:

- `iostream` points to a “header file”. It will replace the “include-line”.
- `iostream` provides the in- and output functionalities.

Step by Step Analysis II

```
#include <iostream>

int main() {
    std::cout << "Hello World from C++!" << std::endl;

    return 0;
}
```

- We have a function (`main`), taking no arguments (`()`).
- The curly braces enclose the “body” of the function
- The function should return an integer (the exit code, `int`).
- `main` is a special function, which is called upon executing the program and every C++ program should have (not valid for libraries).

Step by Step Analysis III

```
#include <iostream>

int main() {
    std::cout << "Hello World from C++!" << std::endl;

    return 0;
}
```

- We open the output stream `std::cout`.
- We use the shift operator `<<` to redirect the string `Hello World!` to the stream `std::cout` and
- finally append `std::endl` to insert a newline character and flush the output stream.
- Note the `;` at the end of the command. In C++, every command must be ended by the semicolon “`;`”.

Step by Step Analysis IV

```
#include <iostream>

int main() {
    std::cout << "Hello World from C++!" << std::endl;

    return 0;
}
```

In the end we return something from the `main` function - a return code of "0" indicates a successful exit.

Step by Step Analysis IV

```
#include <iostream>

int main() {
    std::cout << "Hello World from C++!" << std::endl;

    return 0;
}
```

In the end we return something from the `main` function - a return code of "0" indicates a successful exit.



Task

| Check the return code from the `hello` executable by typing `$ echo $? .`

Outline

- 1 About C++
- 2 Resources
- 3 Hello World

- 4 **Comments and Code Blocks**
- 5 Types and Operations
- 6 Loops and Conditionals
- 7 Types and Operations II: Containers
- 8 Functions

Section 5

Comments and Code Blocks

This Chapter is about:

- The syntax and purpose of comments.
- A first glimpse on namespaces.
- What a code block is.

Comments in Programming

- Comments are ignored by the Compiler.

Comments in Programming

- Comments are ignored by the Compiler.
- You can use comments to

Comments in Programming

- Comments are ignored by the Compiler.
- You can use comments to
 - Indicate intentions prior to actual code.

Comments in Programming

- Comments are ignored by the Compiler.
- You can use comments to
 - Indicate intentions prior to actual code.
 - Describe the non-trivial portions of code.

Comments in Programming

- Comments are ignored by the Compiler.
- You can use comments to
 - Indicate intentions prior to actual code.
 - Describe the non-trivial portions of code.
 - Describe an algorithm.

Comments in Programming

- Comments are ignored by the Compiler.
- You can use comments to
 - Indicate intentions prior to actual code.
 - Describe the non-trivial portions of code.
 - Describe an algorithm.
 - Indicate a reference to literature, where the algorithm is described

Comments in Programming

- Comments are ignored by the Compiler.
- You can use comments to
 - Indicate intentions prior to actual code.
 - Describe the non-trivial portions of code.
 - Describe an algorithm.
 - Indicate a reference to literature, where the algorithm is described
- Comments are important and should be consistently used within a project.

Comments in Programming

- Comments are ignored by the Compiler.
- You can use comments to
 - Indicate intentions prior to actual code.
 - Describe the non-trivial portions of code.
 - Describe an algorithm.
 - Indicate a reference to literature, where the algorithm is described
- Comments are important and should be consistently used within a project.

Comments in Programming

- Comments are ignored by the Compiler.
- You can use comments to
 - Indicate intentions prior to actual code.
 - Describe the non-trivial portions of code.
 - Describe an algorithm.
 - Indicate a reference to literature, where the algorithm is described
- Comments are important and should be consistently used within a project.

Important Hints

- Better include more comments than no comments at all!
- If in doubt that some portion of code is easy to understand: add a comment!

Comments in C++

In C++ there are two flavours of comments:

```
// This is a comment-line  
int a = 42; // This is an inline comment.
```

or

```
/*  
This is a comment block. Every code statement  
here-in is ignored by the compiler.  
*/
```

Bad & Good Comments

```
// Calculates the X of two integers.  
int foo(int a, int b) {  
    ...  
}
```

```
/*  
 * @brief Performx X on two integers.  
 *  
 * This function takes two integer  
 * inputs and returns their X. It  
 * does not handle overflow.  
 *  
 * @param a The 1st integer is <some>.  
 * @param b The 2nd integer is <other>.  
 * @return The X-value of `a` and `b`.  
 */  
int foo(int a, int b) {  
    ...  
}
```



Task

| Your tasks from here-on: Write good comments. Always!

Good Comments

What makes a good comment?

- **Clarity:** Explains why (not just what) the code exists or behaves a certain way.

Good Comments

What makes a good comment?

- **Clarity:** Explains why (not just what) the code exists or behaves a certain way.
- **Relevance:** Avoids stating the obvious (e.g., `// Increment i for i++`). - it is OK to find certain things non-obvious as a beginner!

Good Comments

What makes a good comment?

- **Clarity:** Explains why (not just what) the code exists or behaves a certain way.
- **Relevance:** Avoids stating the obvious (e.g., `// Increment i for i++`). - it is OK to find certain things non-obvious as a beginner!
- **Brevity:** Short and to the point—no unnecessary verbosity.

Good Comments

What makes a good comment?

- **Clarity:** Explains why (not just what) the code exists or behaves a certain way.
- **Relevance:** Avoids stating the obvious (e.g., `// Increment i for i++`). - it is OK to find certain things non-obvious as a beginner!
- **Brevity:** Short and to the point—no unnecessary verbosity.
- **Context:** Highlights edge cases, assumptions, or non-obvious logic.

Good Comments

What makes a good comment?

- **Clarity:** Explains why (not just what) the code exists or behaves a certain way.
- **Relevance:** Avoids stating the obvious (e.g., `// Increment i for i++`). - it is OK to find certain things non-obvious as a beginner!
- **Brevity:** Short and to the point—no unnecessary verbosity.
- **Context:** Highlights edge cases, assumptions, or non-obvious logic.
- **Audience-aware:** Written for other developers (*or your future self!*).

Interlude: Namespaces I

Namespace

A namespace is a way to bundle objects and to avoid collisions of the names of objects. If two objects carry the same name but are within different namespaces, they are still uniquely identifiable.

Interlude: Namespaces II – Example

We wrote the following:

```
#include <iostream>  
// ...  
std::cout << "Hello World from C++!" << std::endl;
```

- `std` indicates a namespace, called `std`

Interlude: Namespaces II – Example

We wrote the following:

```
#include <iostream>  
// ...  
std::cout << "Hello World from C++!" << std::endl;
```

- `std` indicates a namespace, called `std`
- `cout` and `endl` are objects within this namespace

Interlude: Namespaces II – Example

We wrote the following:

```
#include <iostream>
// ...
std::cout << "Hello World from C++!" << std::endl;
```

- `std` indicates a namespace, called `std`
- `cout` and `endl` are objects within this namespace
- `std::cout` means “use object `cout` from the namespace `std`”

Interlude: Namespaces II – Example

We wrote the following:

```
#include <iostream>
// ...
std::cout << "Hello World from C++!" << std::endl;
```

- `std` indicates a namespace, called `std`
- `cout` and `endl` are objects within this namespace
- `std::cout` means “use object `cout` from the namespace `std`”

Interlude: Namespaces II – Example

We wrote the following:

```
#include <iostream>
// ...
std::cout << "Hello World from C++!" << std::endl;
```

- `std` indicates a namespace, called `std`
- `cout` and `endl` are objects within this namespace
- `std::cout` means “use object `cout` from the namespace `std`”

We could directly indicate the namespace and avoid the `std::` by

```
#include <iostream>

using namespace std;
// ...
cout << "Hello World from C++!" << endl;
```

C++ Code Blocks I

A code block is a block of instructions, enclosed by curly braces

```
{  
    // Here is some code  
}
```

One purpose is to structure the code, e. g. writing

```
{  
    using namespace std;  
    // Here is some code  
}
```

limits the use of the namespace `std` to only this block! Other purposes include to limit the lifetime of a variable to this block (later!) and structure your code.

C++ Code Blocks II

You already saw a code block

```
int main() { // starting brace
    // do something
    return 0;
}           // closing brace
```

Writing

```
int main() { // starting brace
    using namespace std;
    // do something
    return 0;
}           // closing brace
```

limits the availability of namespace `std` to within the `main` function.

Outline

- 1 About C++
- 2 Resources
- 3 Hello World

- 4 Comments and Code Blocks
- 5 Types and Operations**
- 6 Loops and Conditionals
- 7 Types and Operations II: Containers
- 8 Functions

Section 6

Types and Operations

This Chapter is about:

- Variables
- The basic, build-in types in C++
- And what you can do with them

Variables in C++ I

A variable is a named object. Its type will never change (it is static). Syntax:

```
type variable = value;  
// or  
type variable {value}; // since C++11
```

Variables in C++ I

A variable is a named object. It's type will never change (it is static). Syntax:

```
type variable = value;  
// or  
type variable {value}; // since C++11
```

Examples:

```
int i = 1;           // declaration and initialization  
cout << i << endl; // would print '1'  
  
int j {42};  
cout << j << endl; // would print '42'
```

Variables in C⁺ II

- The name of a variable cannot
 - start with a number,
 - equal a keyword as, e. g. `int` .
- Otherwise, the name can be freely composed out of letters, digits and underscores!
- You should stick to a clean naming guideline:
 - If your group/employer/. . . has guidelines, follow them!
 - Names should be individual
 - Names should be *meaningful*, e. g. `num_steps` instead of `i` .

Types of Variables I – Fundamental Types

```
short s      = 7;           // signed integral numbers
int i        = 12345;       //
long l       = -52214667;   //
unsigned int u = 210;        // only positive integral number

float f      = 1.476;       // single precision floating point
double d     = 3.14159;     // double precision

bool b       = true;        // boolean: true or false

char c       = 'A';         // a single character
char a       = 65;          // characters are also integers...
```

Types of Variables I – Fundamental Types

```
short s      = 7;           // signed integral numbers
int i        = 12345;       //
long l       = -52214667;   //
unsigned int u = 210;       // only positive integral number

float f      = 1.476;       // single precision floating point
double d     = 3.14159;     // double precision

bool b       = true;       // boolean: true or false

char c       = 'A';        // a single character
char a       = 65;         // characters are also integers...
```

By the Way...

Observe that the whitespaces are not important, in contrast to e.g. the declation of variables in Bash.

Types of Variables II – Why Types?

- Types indicate the allowed values to the compiler
- Types indicate the needed memory (efficiency!)
 - `short ≤ int ≤ long`
 - `float ≤ double`
- Types allow checks if some assignment is allowed, mostly for non-fundamental types (later)
- Types allow efficient pointer to variables (later)

Warning

It is possible to assign “wrong” values to a variable, causing implicit(!) conversions, e. g.

```
short s = 9999999999999; // Value too large; result: 4095  
int i   = 3.14;          // Not an integral number; result: 3  
bool b  = 3.14;          // Not a boolean; result: 1
```

Types of Variables III – Why Types?

Finally: A type allows to interpret the zeroes and ones within the physical memory!

Types of Variables III – Why Types?

Finally: A type allows to interpret the zeroes and ones within the physical memory! This is true for any object and variables are an example of an object in C++.

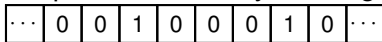
Types of Variables III – Why Types?

Finally: A type allows to interpret the zeroes and ones within the physical memory! This is true for any object and variables are an example of an object in C++.

Consider: `char a = 34;`

Variable is a named object

Object is a piece of memory, holding the value



Value is a set of bits, interpreted according to the type
 $00100010_2 = 34$

Type is a set of possible values and operations.

Pitfalls I

Pitfalls for people used to other languages

- Fundamental types are **not** initialized by default
- You **should** initialize variables at the point of declaration

Example:

```
int a = 1;           // Declared and initialized by value 1
int b;               // Declared but NOT initialized

cout << a << endl;  // Result: 1
cout << b << endl;  // Result: unpredictable
```

Pitfalls II

Pitfalls for people used to other languages

Reuse of variables is **not** possible as for example in Python

Example:

```
int a = 1;           // Declares variable a  
float a = 1;         // ERROR: a is already declared!
```

Modifiers & Literals

C++ lets you be specific:

```

int i           = 0x4A;           // hexadecimal value
int j           = 0b10110011;     // binary literal (C++14)
long l1         = -7000000000l;
long long l2    = 1234567891234ll;

unsigned u1     = 123u;           // may not be < 0
unsigned long u2 = 1234567890ul;

float f         = 1.88f;
double d        = 3.5e10;
long double ld  = 3.5e10l;       // C++11

double d3       = 123'456'789'0.123; // ' separator in C++14
  
```

Arithmetic Operations

```
int a = 4; // Declare and initialize two integer variables
int b = 3;

a = a + b; // a: 7 (add)
a += b;    // a: 10

a = a - b; // a: 7 (subtract)
a -= b;    // a: 4

a = a * b; // a: 12 (multiply)
a *= b;    // a: 36

a = a / b; // a: 12 (divide)
a /= b;    // a: 4

a = a % b; // a: 1 (remainder of division, a.k.a. 'modulo')
```

See `code/examples/05_a_types_and_operations.cpp`

Increment & Decrement Operators

- Increment Operators (increment by 1)

- Prefix: `b = ++a` equals the statement

```
a = a + 1;    // First increment, then assign  
b = a;
```

- Postfix: `b = a++` equals the statement

```
b = a;        // First assign, then increment  
a = a + 1;
```

- Decrement Operators (decrement by 1) with `--a` and `a--`

Increment & Decrement Operators

- Increment Operators (increment by 1)

- Prefix: `b = ++a` equals the statement

```
a = a + 1; // First increment, then assign
b = a;
```

- Postfix: `b = a++` equals the statement

```
b = a; // First assign, then increment
a = a + 1;
```

- Decrement Operators (decrement by 1) with `--a` and `a--`

```
int a = 4;
```

```
int b = 3;
```

```
b = a++; // a: 5    b: 4
```

```
b = ++a; // a: 6    b: 6
```

```
b = --a; // a: 5    b: 5
```

```
b = a--; // a: 4    b: 5
```

Boolean Values and Logical Operations

- `0` is always `false`
- Everything else is `true`

Boolean Values and Logical Operations

- 0 is always false
- Everything else is true

```
bool a = true;  
bool b = false;
```

```
bool c1 = a && b;    // false    (logical AND)  
bool c2 = a and b;   // false
```

```
bool d1 = a || b;    // true     (logical OR)  
bool d2 = a or b;    // true
```

```
bool e1 = !a;        // false    (logical NOT)  
bool e2 = not b;     // true
```

```
bool f1 = 12;        // true     conversion from int to bool  
bool f2 = 0;         // false    conversion from int to bool
```

Narrowing

- Implicit conversion from larger to smaller data type
- May result in *loss of information* and (hard to spot) errors
- If you code, there will be *no error*, sometimes a compiler warning

Narrowing

- Implicit conversion from larger to smaller data type
- May result in *loss of information* and (hard to spot) errors
- If you code, there will be *no error*, sometimes a compiler warning

```
double d    = 1.2345;
float  f    = 3.45 f;
unsigned u = 120u;
```

double e	= f;	// OK	float -> double
int i	= 2.5;	// NARROWING	double -> int
int j	= u;	// NARROWING	unsigned int -> int
int k	= f;	// NARROWING	float -> int

Type Cast I

- Apart from implicit type conversions, it is possible to convert explicitly
- Two ways

C style This is the only way to cast in C and also possible in C⁺.

There are no checks at all.

C++ style This is the new way to do a type cast in C⁺, where sometimes there are some basic checks. This is introduced later.

- Why do a type cast at all? Consider

```
int a    = 1;  
int b    = 2;  
float c = a / b;    // c: 0  
                // How to get the desired result 0.5 ?
```

Type Cast II – C style

- This type cast style is inherited from C
- There are no checks
- The previous example:

```
int a    = 1;
int b    = 2;
float c = a / b;    // c: 0
                // How to get the desired result 0.5 ?
float d = (float)a / (float)b;    // d: 0.5
```

- In general: Avoid any casts as far as possible!

```
int a    = 1;
int b    = 2;
float c = a / b;    // c: 0
                // How to get the desired result 0.5 ?
float a1 = a;        // This is much clearer what you want to do!
float b1 = b;
float d  = a1 / b1;  // d: 0.5
```

Variable Sizes

- The amount of memory for a fundamental type depends on the platform (hardware, on 32 vs. 64 bit and OS)
- Usually `long = int` on 32bit platforms
- Use the `sizeof` operator to get the size of a type in bytes
- See `code/examples/05_b_sizeof.cpp`

`#include <stdint.h>` C++11

gives you type names with guaranteed sizes, e. g.

- `int8_t` (int with exact 8 bits length)
- `int16_t`
- `uint64_t` (unsigned int with exact 64 bits length)
- `int_least32_t` (int with at least 32 bits length)
- ...

[more info on the reference website](#)

Constant Variables

- A constant variable has a value, that cannot change after it is assigned
- A constant variable must be initialized directly

```
const float pi = 3.14159; // Declare variable as constant  
const float j;           // ERROR: not initialized
```

Constant Variables

- A constant variable has a value, that cannot change after it is assigned
- A constant variable must be initialized directly

```
const float pi = 3.14159; // Declare variable as constant  
const float j;           // ERROR: not initialized
```



Warning

Do not declare π like this in your real program, it is actually defined in the standard library (to be introduced later)!

Constant Variables

- A constant variable has a value, that cannot change after it is assigned
- A constant variable must be initialized directly

```
const float pi = 3.14159; // Declare variable as constant
const float j;           // ERROR: not initialized
```

- Benefit: Compiler complains if you attempt to change it

```
i          = 2.0;           // ERROR: i is constant
float j = 0;
i          = j;             // ERROR: i is constant
```

- But: In contrast to Fortran, the initial value may be dynamic (vary at runtime)

```
int j = 0;
cin >> j;
const int i = j; // OK, initialized directly
```

Hands On II – Binding Input



Task

Try to work out task 2: Remember: `int x` to declare a variable holding an integer `x` and `std::cout <<` for output (see “Hello World!” example) and `std::cin >>` input.

Outline

- 1 About C++
- 2 Resources
- 3 Hello World

- 4 Comments and Code Blocks
- 5 Types and Operations
- 6 Loops and Conditionals**
- 7 Types and Operations II: Containers
- 8 Functions

Section 7

Loops and Conditionals

This Chapter is about:

- Conditionals (How to respond to specific conditions?)
- Looping (How to repeat things?)

Conditionals – Syntax

```
if (condition1) {  
    // Execute this if condition1==true  
}  
else if (condition2) {  
    // Execute this if condition2==true  
}  
else {  
    // Otherwise  
}
```

- Conditionals execute code depending on the truth value of a condition
- The condition must be (convertible to) a boolean
- Check for truth value occur from top to bottom
- Note the braces, enclosing a code block. They are not needed for a single instruction, but we strongly recommend: always use them!

Conditionals – Examples

```
if ( true ) {                // Simple if-else construct
    cout << "yes" << endl;
} else {
    cout << "no" << endl;
}
```

Conditionals – Examples

```
if ( true ) { // Simple if-else construct
    cout << "yes" << endl;
} else {
    cout << "no" << endl;
}
```

Testing a condition:

```
int x = 0;
if ( x == 0 ) { // x == 0 is the condition (which evaluates to 'true')
    cout << "x is zero" << endl;
}
```

Conditionals – Examples

```
if ( true ) { // Simple if-else construct
    cout << "yes" << endl;
} else {
    cout << "no" << endl;
}
```

Testing a condition:

```
int x = 0;
if ( x == 0 ) { // x == 0 is the condition (which evaluates to 'true')
    cout << "x is zero" << endl;
}
```

Braceless shortcut for one-liners. only:

```
if ( x == 0 ) cout << "x is zero" << endl;
```

Conditionals – Examples

```
if ( true ) { // Simple if-else construct
    cout << "yes" << endl;
} else {
    cout << "no" << endl;
}
```

Testing a condition:

```
int x = 0;
if ( x == 0 ) { // x == 0 is the condition (which evaluates to 'true')
    cout << "x is zero" << endl;
}
```

Braceless shortcut for one-liners. only:

```
if ( x == 0 ) cout << "x is zero" << endl;
```

Full if-else construct:

```
int y = 10;
if ( y <= 5 ) {
    cout << "y smaller or equal to 5" << endl;
} else {
    cout << "y larger than 5" << endl; }
}
```

Conditionals – Tertiary Conditional Operator

- There is a shortcut for an if-else statement:

`Result = Condition ? If-Expression : Else-Expression`

- These are often used, but less obvious in code $\Rightarrow$ Error prone

```
int i = 8;
int j = i > 10 ? 1 : 2;    // j: 2

int k = 20;
int l = (k > 10) ? 1 : 2;  // l: 1

bool b = true;
double d = b ? 2.0 : 3.0;  // d: 2.0
double e = !b ? 2.0 : 3.0; // e: 3.0
```

Switch – Syntax

```
// Assume m is an integer
switch (m) {
  case 0:
    // Code for case m==0
    break;
  case 1:
    // Code for case m==1
  case 2:
    // Code for case m==2; will also be executed if m==1
    break;
  default:
    // Code if nothing has matched
}
```

- Switch over values of integral types, e. g. `char`, `int`, ...
- Executes everything(!) between matching case and next break
- More readable compared to heavily nested if-else construct

Hands On

- Please solve the exercise No. 3 on your exercise sheet.

Loops – Syntax

- For loop: do a pre-defined number of iterations

```
for(initialization ; condition ; step){  
    // The work  
}
```

Loops – Syntax

- For loop: do a pre-defined number of iterations

```
for(initialization ; condition ; step){  
    // The work  
}
```

- While loop: do an unknown number of iterations until something gets false

```
while (condition) {  
    // The work to do until condition==false  
}
```

Loops – Syntax

- For loop: do a pre-defined number of iterations

```
for(initialization ; condition ; step){  
    // The work  
}
```

- While loop: do an unknown number of iterations until something gets false

```
while (condition) {  
    // The work to do until condition==false  
}
```

- Do-While loop: The same as a While loop, except do the work at least once

```
do {  
    // The work executed at least once and afterwards until  
    // condition==false  
} while (condition);
```

Loops – Examples

```
for( int i = 0 ; i <= 10 ; ++i ){ // Variable i is only valid within the
    cout << i << endl;           // loop body. Prints 0, 1, ..., 10
}

int j = 5;
while (j < 10) {                 // Iterates until j >= 10
    cout << j << endl;           // Prints 5, 6, 7, 8, 9
    ++j;
}

do{
    cout << j << endl;           // Prints 10, 9, ..., 1
    --j;
}while (j > 0);                  // Iterates until j<=0
```

- Please solve the exercises belonging to the section “Loops and Conditionals” on your exercise sheet, first only exercises 4 and 5.

Hints

remember the

```
var = var + 1;  
# or  
var += 1;
```

syntax

remember the

```
for ( int i = 0; i < n; i++) {  
    // code block  
}  
// or  
while (var < 10) {  
    // code block  
}
```

examples.

- Please solve the exercises belonging to the section “Loops and Conditionals” on your exercise sheet. Now, the pyramid task, No. 6. – if we have the time.

**Hint**

Nest for-loops. Build the pyramid from the top, row by row. Each row starts with padding characters, e.,g. white space or dots up to the zeros. One loop is responsible for the rows, one for the columns.

There are several possible solutions. For instance one with three loops.

Outline

- 1 About C++
- 2 Resources
- 3 Hello World

- 4 Comments and Code Blocks
- 5 Types and Operations
- 6 Loops and Conditionals
- 7 Types and Operations II: Containers**
- 8 Functions

Section 8

Types and Operations II: Containers

This Chapter is the continuation on Types. It is about:

- Arrays and enums
- Datatype “string” for text

Arrays I

Array

An array is a collection of multiple variables of the same type and name.

Example:

```
int iarr[5];    // Declare C-array of 5 integers  
  
iarr[0] = 4;    // Set the value of the first variable  
iarr[2] = 1;    // Set the value of the third variable  
  
cout << "Third variable is: " << iarr[2] << endl;
```

Initialization of the array may also be done as

```
int iarr[5] = {4, 0, 1, 0, 10};
```

Arrays II – Warning on C-arrays

CAUTION

C-arrays do **NOT** provide **any** checks, e.g. bound-checking of the index.

The following works, but the program behavior and output are unpredictable:

```
iarr[100] = 42;                // May result in a segmentation fault  
  
cout << iarr[100] << endl;    // Output unpredictable
```

Arrays II – Warning on C-arrays

CAUTION

C-arrays do **NOT** provide **any** checks, e.g. bound-checking of the index.
The following works, but the program behavior and output are unpredictable:

```
iarr[100] = 42;                // May result in a segmentation fault  
  
cout << iarr[100] << endl;    // Output unpredictable
```

? Why is this output unpredictable?

|

Arrays II – Warning on C-arrays

CAUTION

C-arrays do **NOT** provide **any** checks, e.g. bound-checking of the index.
The following works, but the program behavior and output are unpredictable:

```
iarr[100] = 42;                // May result in a segmentation fault  
  
cout << iarr[100] << endl;    // Output unpredictable
```

? Why is this output unpredictable?

| Well, we only defined `int iarr [5];` ! Any index > 4 will be undefined.

Arrays III – The Safe Way

- The standard library provides a datastructure `std::vector`
- Simply include the appropriate header: `#include <vector>`
- This implementation provides checks on the range of indices
- `std::vector` knows its size
- Unlike a C-array, the size a `std::vector` can be changed at runtime



Warning

Many people in High Performance Computing pretend that using a C array instead of a `std::vector` is faster. That *might* be true, but most of the time it is not.

Arrays IV – Vector

```
#include <vector>                                // Here, std::vector is defined

std::vector<int> v = {4, 0, 1, 0, 10};           // Initialize with 5 values
std::cout << v.size() << std::endl;              // Print size of vector: 5
std::cout << v[0] << std::endl;                  // Print the first element: 4
std::cout << v[1] << std::endl;                  // Print the second element: 0
std::cout << v[100] << std::endl;                // ERROR

std::cout << v.front() << std::endl;             // First element: 4
std::cout << v.back() << std::endl;              // Last element: 10
```

Note the difference with initialization:

```
std::vector<int> v1 {5,2};                       // Vector {5,2} with 2 entries
std::vector<int> v2 (5,2);                       // Vector {2,2,2,2,2} with 5 entries, each
                                                // with default value 2
```

Arrays V – Adding an Element

Use `push_back(Element)` to add an element to a vector:

```
vector<int> v;           // Empty vector: {}  
cout << v.size() << " \n"; // Size: 0  
  
v.push_back(2);         // {2}  
cout << v.size() << " \n"; // Size: 1  
  
v.push_back(31);        // {2, 31}  
cout << v.size() << " \n"; // Size: 2  
  
v.push_back(47);        // {2, 31, 47}  
cout << v.size() << " \n"; // Size: 3  
  
v.clear();              // Delete all contents  
cout << v.size() << " \n"; // Size: 0
```

Arrays VI – Avoiding Reallocation

If you know in advance how many elements your vector should have, tell the library about it to avoid any reallocation!

```
vector<int> v;    // Declare empty vector  
v.reserve(100);  // Tell the library to allocate space for at  
                 // least 100 int  
  
// Check current capacity of the vector  
// Note: capacity >= size  
cout << v.capacity() << endl;  
cout << v.size() << endl;
```

Scoped Enumerations I (C++11)

- Enumerations map an enumerator to an integral number between 0 and $N - 1$

```
enum class name { enumerator1, enumerator2, ..., enumeratorN };
```

- Example: The day of a week

```
enum class day { mon, tue, wed, thu, fri, sat, sun };
```

```
// This provides a new type
```

```
day d = day::mon;    // Monday, internally the integer 0
```

```
d = tue;             // ERROR: tue is only known within enum "day"
```

Scoped Enumerations II

The mapping may also be made explicit, maybe easier to deal with:

```
enum class month {  
    none = 0,  
    jan = 1, feb = 2, mar = 3, apr = 4, may = 5, jun = 6, jul = 7,  
    aug = 8, sep = 9, oct = 10, nov = 11, dec = 12  
};  
  
// Type conversions will work  
cout << int(month::jul << endl);    // Output: 7  
  
// Assignment from underlying type  
int i = 7;  
month m = month(i);                // July
```

How to Represent Text?

- Up to now, you know the fundamental type `char`, representing a single character
- Its internal representation is a number between -128 and 127
- Literals are enclosed in single quotes and automatically converted to numbers
`'a', 'b', 'c', ...`

```
char c1 = 'A';           // Character as literal
char c2 = 65;            // Character as number
cout << c1 << endl;      // Result: A
cout << c2 << endl;      // Result: A
```

In the old C age:

Text is represented as an array of `char`

This is possible but extremely tedious to work with...

Library Type `std::string`

- Use header `#include <string>`
- Represents an array as a dynamic array of characters
- Much functionality!

```
#include <string>           // Appropriate header
// Some code

string test = "Hello";
test += " World";          // Append string " World" to test

cout << test    << endl;    // Result: Hello World
cout << test[1] << endl;    // Result: e
```

Library Type `std::string` – Some Manipulations

Assume: `string s = "My mobile";`

- `s.insert(4, "new ")`
Result: "My new mobile"
- `s.erase(5, 3)`
Result: My mle
- `s.replace(4, 6, "Garden")`
Result: My Garden
- `s.substr(4, 6)`
Result: "mobile"
- ...and many more, see the documentation!

Final Comments on Strings

- Expression `"xyz"` has type `const char[]` ("C string")
- Expression `"xyz"s` has type `std::string` (C++14)
- Special characters
 - `\n` means new line: `Line1\nLine2\nLine3`
 - `\t` means tab: `Column1\tColumn2\tColumn3`

Hands On

- Please solve the exercise belonging to the section “Basic I/O” on your exercise sheet.

Outline

- 1 About C++
- 2 Resources
- 3 Hello World

- 4 Comments and Code Blocks
- 5 Types and Operations
- 6 Loops and Conditionals
- 7 Types and Operations II: Containers
- 8 Functions**

Section 9

Functions

This Chapter is about Functions and Memory:

- Do I really need to code everything several times?
- Default parameters
- Overloading
- Recursion
- Dependencies: Declaration vs. Definition
- Mathematical functions
- Memory management and scope of variables
- Pointers

Why Functions?

- In programming: do not reinvent the wheel! Reuse well-tested code!
- Instead of rewriting always the same sequence of commands, we want to bundle them
- BUT: do not put too much into a single function. Each function should only have one purpose

Function

A function is a callable set of commands, which may return a result.

Analogy with Mathematics

- In mathematics, functions are “rules” to compute a value from an input, e.g. $f(x) = x^3$, where x is the input and $f(x)$ the result

Analogy with Mathematics

- In mathematics, functions are “rules” to compute a value from an input, e.g. $f(x) = x^3$, where x is the input and $f(x)$ the result
- Evaluating the function for a given value of x means, substituting this value for x in the function’s rule

Analogy with Mathematics

- In mathematics, functions are “rules” to compute a value from an input, e.g. $f(x) = x^3$, where x is the input and $f(x)$ the result
- Evaluating the function for a given value of x means, substituting this value for x in the function’s rule
- Example: Let $x = 2$, then the result $f(2)$ is computed as

$$f(2) = 2^3 = 8$$

Analogy with Mathematics

- In mathematics, functions are “rules” to compute a value from an input, e.g. $f(x) = x^3$, where x is the input and $f(x)$ the result
- Evaluating the function for a given value of x means, substituting this value for x in the function’s rule
- Example: Let $x = 2$, then the result $f(2)$ is computed as

$$f(2) = 2^3 = 8$$

- Functions in programming language may be understood similarly. . .

Analogy with Mathematics

- In mathematics, functions are “rules” to compute a value from an input, e.g. $f(x) = x^3$, where x is the input and $f(x)$ the result
- Evaluating the function for a given value of x means, substituting this value for x in the function’s rule
- Example: Let $x = 2$, then the result $f(2)$ is computed as

$$f(2) = 2^3 = 8$$

- Functions in programming language may be understood similarly...
- ...however they may not only consist of a single “rule”...

Analogy with Mathematics

- In mathematics, functions are “rules” to compute a value from an input, e.g. $f(x) = x^3$, where x is the input and $f(x)$ the result
- Evaluating the function for a given value of x means, substituting this value for x in the function’s rule
- Example: Let $x = 2$, then the result $f(2)$ is computed as

$$f(2) = 2^3 = 8$$

- Functions in programming language may be understood similarly. . .
- . . . however they may not only consist of a single “rule”. . .
- . . . but do other things (“side-effects”; e.g. printing something to screen)

Functions – Syntax

```
return_type name(parameter1, parameter2, ...) {  
    // Function body; the "computation rule"  
}
```

- `return_type` denotes the data type, which is returned from the function. Use `void` if nothing is returned.

Functions – Syntax

```
return_type name(parameter1, parameter2, ...) {  
    // Function body; the "computation rule"  
}
```

- `return_type` denotes the data type, which is returned from the function. Use `void` if nothing is returned.
- Only one value may be returned

Functions – Syntax

```
return_type name(parameter1, parameter2, ...) {  
    // Function body; the "computation rule"  
}
```

- `return_type` denotes the data type, which is returned from the function. Use `void` if nothing is returned.
- Only one value may be returned
- `name` is the name of the function

Functions – Syntax

```
return_type name(parameter1, parameter2, ...) {  
    // Function body; the "computation rule"  
}
```

- `return_type` denotes the data type, which is returned from the function. Use `void` if nothing is returned.
- Only one value may be returned
- `name` is the name of the function
- `parameter1` is an (optional) parameter with syntax `type name`

Functions – Syntax

```
return_type name(parameter1, parameter2, ...) {  
    // Function body; the "computation rule"  
}
```

- `return_type` denotes the data type, which is returned from the function. Use `void` if nothing is returned.
- Only one value may be returned
- `name` is the name of the function
- `parameter1` is an (optional) parameter with syntax `type name`

Functions – Syntax

```
return_type name(parameter1, parameter2, ...) {  
    // Function body; the "computation rule"  
}
```

- `return_type` denotes the data type, which is returned from the function. Use `void` if nothing is returned.
- Only one value may be returned
- `name` is the name of the function
- `parameter1` is an (optional) parameter with syntax `type name`

```
int add(int a, int b) {    // Function has name add and returns an int  
    return (a + b);        // This is the value to return  
}  
  
int main() {  
    cout << add(2, 3) << endl;    // Call the function with a = 2, b = 3  
    return 0;  
}
```

Functions – Examples I

```
double cube(double x) {           // Think of the mathematical function
    return ( x * x * x );          // f(x) = x^3
}

int max(int x, int y) {
    if (x > y) {return x;} else {return y;}
}

void print_cubes(int n) {
    for (int i = 0 ; i <= n ; ++i) {
        cout << cube(i) << endl; // Note: implicit type conversion!
    }
}

void say_hello() {                // Empty parameter list!
    cout << "Hello!" << endl;
}
```

Functions – Examples II and Hints

```
auto foo(int a, double x) {  
    return a;  
    // Some code  
    return x;           // COMPILE ERROR: return types do not match  
}  
  
int foo(int a, const int b) {  
    a = a + 5;           // OK, int type  
    b = b + 10;          // COMPILE ERROR: b is constant!  
    return ( a * b );  
}
```

- Beware of side-effects: functions should not modify something that is not expected. Second example above indicates a way to ensure that something remains unchanged.
- Use meaningful names (“What is `foo`”???)

Default Parameters I

- Default parameters render specification of some parameters optional
- Example: We want to compute the power x^n , but if n is unspecified, the result should be x^3

```
double power(double x, int n) { // Compute x^n
    double res = 1.0;
    for (int i=0 ; i <= n ; ++i) {
        res = res * x;
    }
    return res;
}

int main() {
    power(2, 3);    // Result: 8
    power(2);       // COMPILE ERROR: Second argument unspecified

    return 0;
}
```

Default Parameters II

- Default parameters render specification of some parameters optional
- Example: We want to compute the power x^n , but if n is unspecified, the result should be x^3

```
double power(double x, int n=3) { // Compute x^n or x^3
    double res = 1.0;
    for (int i=0 ; i <= n ; ,++i) {
        res = res*x;
    }
    return res;
}

int main() {
    power(2, 3); // Result: 8
    power(2);    // OK, n unspecified, use n=3

    return 0;
}
```

Default Parameters III – Examples

```
void foo(int i = 0) { ... }  
// OK, one default parameter
```

```
void foo(int i, int j = 10) { ... }  
// OK, one required and one default parameter
```

```
void foo(int i, int j = 10, double a = -1.5) { ... }  
// OK, one required and two default parameters
```

```
void foo(int i, int j = 10, double a) { ... }  
// COMPILE ERROR: Each parameter after first  
// default must have default value!
```

Overloading

Overloading

Functions with identical names but different parameter lists are overloaded

```
int abs( int i ) {  
    return ((i<0) ? -i : i);  
}
```

```
double abs(double d) {  
    return ((d<0) ? -d : d);  
}
```

```
int square(int i) {  
    return ( i*i );  
}
```

```
double square(int i) {  
    return ( i*i );    // COMPILE ERROR: Cannot overload on return type!  
}
```

Overloading using Templates

The tedious way:

```
int foo(int x)
{...; return val;}
long int foo(long int x)
{...; return val;}
float foo(float x)
{...; return val;}
double foo(double x)
{...; return val;}
etc;
etc;
```



Warning

Doing this for every type is not feasible
(and error prone).

Overloading using Templates

The tedious way:

```
int foo(int x)
{...; return val;}
long int foo(long int x)
{...; return val;}
float foo(float x)
{...; return val;}
double foo(double x)
{...; return val;}
etc;
etc;
```

Templates to the rescue:

```
template <class T>
T foo(T x)
{...; return val;}
```

Only ONE implementation!

The general form is:

```
template <class Name,
          class Name2,
          ... >
```



Warning

Doing this for every type is not feasible
(and error prone).

Recursion

- Idea: A function calling itself
- Needs a break condition
- Maybe slower than loops and has limited depth

```
int factorial(int n) {  
    if (n < 2) {  
        return 1; // break condition  
    }  
    else {  
        return ( n * factorial(n - 1) ); // Calls itself  
    }  
}
```

Dependencies – Declaration vs. Definition I

- Order of definitions matters for compilation
- Following will **not** compile

```
bool even(int a) {  
    return ( !odd(a) );    // odd unknown!  
}  
  
bool odd(int a) {  
    return ( a % 2 );  
}  
  
int main() {  
    int i = /*Get random number*/;  
    if( odd(i) ) { cout << "Input is odd" << endl; } // odd unknown!  
    if( even(i) ) { cout << "Input is even" << endl; } // even unknown!  
}
```

Dependencies – Declaration vs. Definition II

- Order of definitions matters for compilation
- Following will compile: functions are already declared

```
// Declare functions to the compiler with their signature
bool even(int);
bool odd(int);

int main() {
    int i = /* Get random number */;
    if( odd(i) ) { cout << "Input is odd" << endl; } // OK
    if( even(i) ) { cout << "Input is even" << endl; } // OK
}

bool even(int a) {
    return ( !odd(a) );    // OK, odd declared
}

bool odd(int a) {
    return ( a % 2 );
}
```

(Some) Mathematical Functions

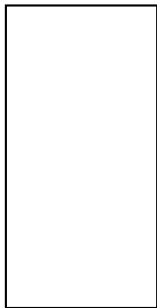
- Use library: `#include <cmath>`
- `double sqrt(double x)` computes $\sqrt{x}$
- `double pow(double a, double b)` computes a^b
- `double abs(double x)` computes $|x|$
- `double sin(double x)` computes $\sin(x)$
- `double cos(double x)` computes $\cos(x)$
- `double exp(double x)` computes $\exp(x)$
- `double log(double x)` computes $\log(x)$
- `double floor(double x)` rounds x down
- `double ceil(double x)` rounds x up
- `double fmod(double x, double y)` computes remainder of x/y

Memory I – Management Wishlist

- Lots of space: we want to use all RAM
- Fast allocation and access
- Indefinite lifetime of objects, e. g. variables
- Automatic release of objects as soon as they are not needed any more
- Avoid runtime overhead due to memory management

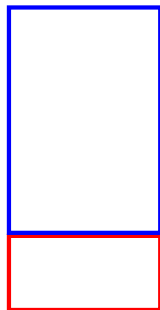
Memory II – Layout

- This is the main memory...



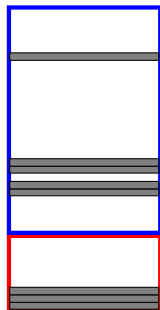
Memory II – Layout

- This is the main memory...
- ...partitioned into **stack** and **heap**
 - Heap or “Free Store”: most of the RAM, many GB
 - Stack: small, a few MB



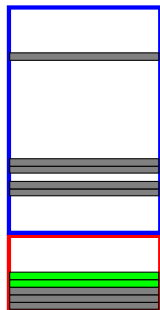
Memory II – Layout

- This is the main memory...
- ...partitioned into **stack** and **heap**
 - Heap or “Free Store”: most of the RAM, many GB
 - Stack: small, a few MB
- Each memory has some occupied space

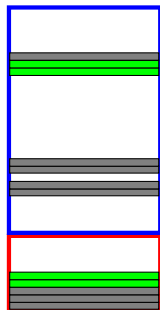


Memory II – Layout

- This is the main memory...
- ...partitioned into **stack** and **heap**
 - Heap or “Free Store”: most of the RAM, many GB
 - Stack: small, a few MB
- Each memory has some occupied space
- Per default, each **new object** (e. g. variable) is placed on top of the **stack**
 - ⇒ Fast allocation and access
- Automatic destruction of objects in reverse order of allocation



Memory II – Layout



- This is the main memory...
- ...partitioned into **stack** and **heap**
 - Heap or “Free Store”: most of the RAM, many GB
 - Stack: small, a few MB
- Each memory has some occupied space
- Per default, each **new object** (e. g. variable) is placed on top of the **stack**
 - ⇒ Fast allocation and access
- Automatic destruction of objects in reverse order of allocation
- New objects on the **heap** need search for free, contiguous memory
 - ⇒ Slow allocation on fragmented memory
- Needs explicit (!) destruction

Scope I

Scope

By “scope”, one refers to the lifetime and visibility of a variable.

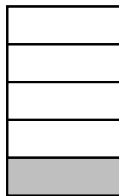
Example:

```
int main() {  
    int i = 21;  
  
    for (int j = 0; j < i; j++) {  
        if ( j > 2 ) {  
            int k = 2 * j;  
            cout << k;  
        }  
  
        cout << ' ', ' ';  
    }  
    cout << ' ' finished ' ' << endl;  
}
```

Scope II

```
int main() {  
    int i = 21;  
  
    for (int j = 0; j < i; j++) {  
        if ( j > 2 ) {  
            int k = 2 * j;  
            cout << k;  
        }  
  
        cout << ' ', ' ';  
    }  
    cout << ' ' finished ' ' << endl;  
}
```

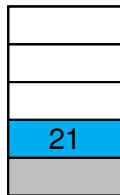
This is the stack:



Scope II

```
int main() {  
    int i = 21;  
  
    for (int j = 0; j < i; j++) {  
        if ( j > 2 ) {  
            int k = 2 * j;  
            cout << k;  
        }  
  
        cout << ' ', ' ';  
    }  
    cout << ' ' finished ' ' << endl;  
}
```

This is the stack:

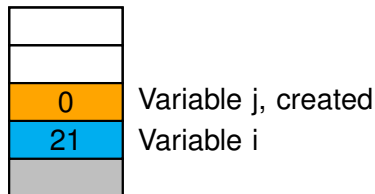


Variable i, created

Scope II

```
int main() {  
    int i = 21;  
  
    for (int j = 0; j < i; j++) {  
        if ( j > 2 ) {  
            int k = 2 * j;  
            cout << k;  
        }  
  
        cout << ' ', ' ';  
    }  
    cout << ' ' finished ' ' << endl;  
}
```

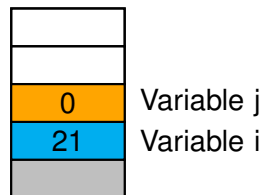
This is the stack:



Scope II

```
int main() {  
    int i = 21;  
  
    for (int j = 0; j < i; j++) {  
        if ( j > 2 ) {  
            int k = 2 * j;  
            cout << k;  
        }  
  
        cout << ' ', ' ';  
    }  
    cout << ' ' finished ' ' << endl;  
}
```

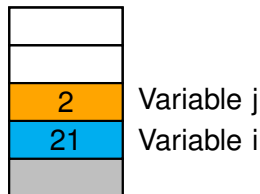
This is the stack:



Scope II

```
int main() {  
    int i = 21;  
  
    for (int j = 0; j < i; j++) {  
        if ( j > 2 ) {  
            int k = 2 * j;  
            cout << k;  
        }  
  
        cout << ' ', ' ';  
    }  
    cout << ' ' finished ' ' << endl;  
}
```

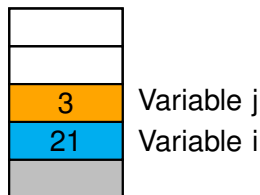
This is the stack:



Scope II

```
int main() {  
    int i = 21;  
  
    for (int j = 0; j < i; j++) {  
        if ( j > 2 ) {  
            int k = 2 * j;  
            cout << k;  
        }  
  
        cout << ' ', ' ';  
    }  
    cout << ' ' finished ' ' << endl;  
}
```

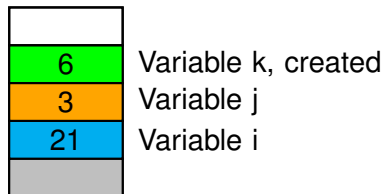
This is the stack:



Scope II

```
int main() {  
    int i = 21;  
  
    for (int j = 0; j < i; j++) {  
        if ( j > 2 ) {  
            int k = 2 * j;  
            cout << k;  
        }  
  
        cout << ' ', ' ';  
    }  
    cout << ' ' finished ' ' << endl;  
}
```

This is the stack:



Scope II

```
int main() {  
    int i = 21;  
  
    for (int j = 0; j < i; j++) {  
        if ( j > 2 ) {  
            int k = 2 * j;  
            cout << k;  
        }  
  
        cout << ' ', ' ';  
    }  
    cout << ' ' finished ' ' << endl;  
}
```

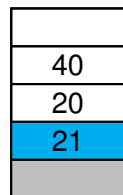
This is the stack:

6	Variable k, destroyed
3	Variable j
21	Variable i

Scope II

```
int main() {  
    int i = 21;  
  
    for (int j = 0; j < i; j++) {  
        if ( j > 2 ) {  
            int k = 2 * j;  
            cout << k;  
        }  
  
        cout << ' ', ' ';  
    }  
    cout << " finished" << endl;  
}
```

This is the stack:



Variable j, destroyed

Variable i

Scope III – Conclusions

- Variables and objects have finite lifetime
- Variables and objects, declared within a structured block are only available within this block

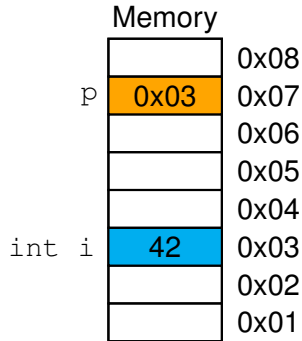
Best Practices

- Declare a variable as late as possible
- Limit the lifetime of a variable to where it is really needed
- Avoid global variables!

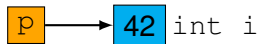
What is a Pointer?

(Raw) Pointer

A (raw) pointer is a variable, holding the memory address of an object, e.g. of a variable, hence it “points” somewhere into memory.



The pointer `p` points to a variable `i` of type `int` :



Why Pointers??



Hint

- Pointers are essential to "old" C++.
- They are still in use in lots of code examples.
- They are essential to loop efficiently.

In other words: Understanding pointers is essential.

Why Pointers??

Hint

- Pointers are essential to "old" C++.
- They are still in use in lots of code examples.
- They are essential to loop efficiently.

In other words: Understanding pointers is essential.

Hint

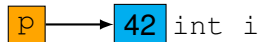
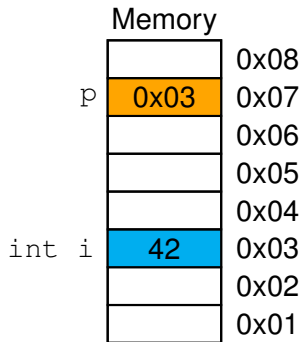
Pointers to `float` |, `double` |, etc. need to be declared like those to `int` |. This is **cumber** and **error prone**.

↪ the follow-up course covers C++'s options to use pointers without the hassle (and sometimes not seeing that you use pointers at all).

Pointer – Syntax and Address Operator

```
int i = 42;           // Declare and initialize  
                      // the variable  
  
int* pint;           // Declare pointer to  
                      // an int  
  
pint = &i;            // Fill p with memory  
                      // address of i
```

- Raw pointer of type `T*` can store address to an object of type `T`
- Reference operator `&` returns memory address



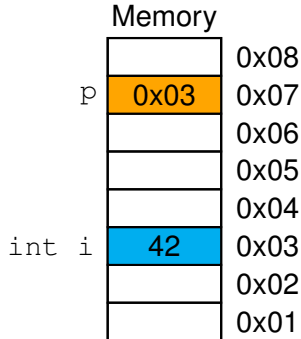
Pointer – Dereferencing Operator

```
int i = 42;           // Declare and initialize
                      // the variable

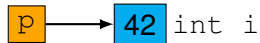
int* pint;            // Declare pointer to
                      // an int

p = &i;               // Fill p with memory
                      // address of i

cout << p << endl;    // 0x03
cout << *p << endl;    // 42
```



- Reference operator `&` returns memory address
- Dereference operator `*` accesses value at the address, where pointer points to



Raw Pointer – Remarks and Pitfalls

Raw pointers are independent of the pointed-to object

- Pointer can switch the pointed-to object

```
int a = 0;
int b = 1;

int* p = &a;
cout << *p << endl;    // Output: 0

p = &b;
cout << *p << endl;    // Output: 1
```

- Pointer can exist even if the object is already destroyed

```
int* p;
{
    int c = 42;
    p = &c;
}
cout << *p << endl;    // c is destroyed
                        // Undefined behaviour
```

Raw “dummy” pointer: `nullptr`

- `nullptr` has implicit conversion to `false`
- Initialize each pointer with valid address or `nullptr`
- Always check if pointer is `nullptr` before dereferencing

```
int* p = nullptr;           // Initialize to nullptr
{
    int c = 42;              // Assign valid address
    p = &c;

    if(p) { *p = 45; }       // Check for nullptr

    p = nullptr;
}                             // End of block: c is destroyed

if(p) { ... }               // p was safely set to nullptr
```

Arrays and Pointers

- We already met arrays, e.g.

```
int a[5];
```

- Internally, the array is represented as a pointer to the first element of a contiguous part of memory, each element exactly the memory to hold the desired data type



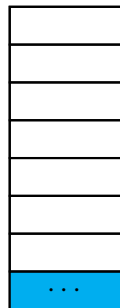
- Consequently, adding the amount of memory for the datatype to the pointer results in the next element (“pointer arithmetic”, **dangerous**), i.e.

```
int a[5] = {1, 2, 3, 4, 5};  
cout << *a      << endl;      // Output: 1  
cout << *(a+1)  << endl;      // Output: 2  
cout << *(a+2)  << endl;      // Output: 3
```

What happens when calling a function? I

```
int square(int p) {  
    int y;  
    y = p * p;  
    return y;  
}  
  
int main() {  
    int x = 2;  
    int i = square(x);  
    cout << i << endl;  
}
```

Stack



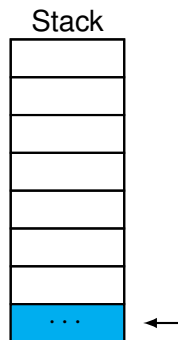
- Program is stored in memory
- Each instruction has an address (“program counter”)
- All stored in stack; contiguous memory
- Last In–First out

What happens when calling a function? II

```
int square(int p) {  
    int y;  
    y = p * p;  
    return y;  
}
```

```
int main() {  
    int x = 2;  
    int i = square(x);  
    cout << i << endl;  
}
```

// ← Program counter

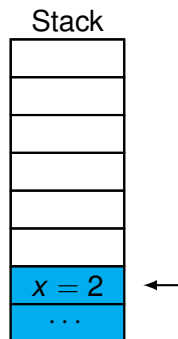


- Program counter starts somewhere on top of the stack

What happens when calling a function? III

```
int square(int p) {  
    int y;  
    y = p * p;  
    return y;  
}
```

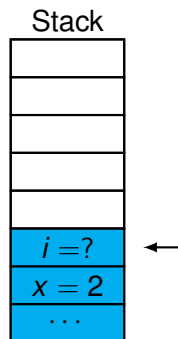
```
int main() {  
    int x = 2;           // <- Program counter  
    int i = square(x);  
    cout << i << endl;  
}
```



- Allocate variable `x` on top of stack

What happens when calling a function? IV

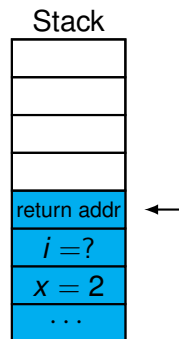
```
int square(int p) {  
    int y;  
    y = p * p;  
    return y;  
}  
  
int main() {  
    int x = 2;  
    int i = square(x); // <- Program counter  
    cout << i << endl;  
}
```



- Allocate variable `i` on top of stack, yet with unknown value

What happens when calling a function? V

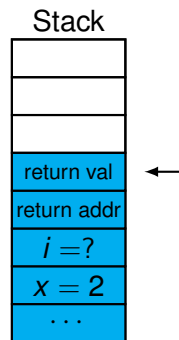
```
int square(int p) {  
    int y;  
    y = p * p;  
    return y;  
}  
  
int main() {  
    int x = 2;  
    int i = square(x); // <- Program counter  
    cout << i << endl;  
}
```



- To perform the function call, store the return address for CPU

What happens when calling a function? VI

```
int square(int p) {  
    int y;  
    y = p * p;  
    return y;  
}  
  
int main() {  
    int x = 2;  
    int i = square(x); // <- Program counter  
    cout << i << endl;  
}
```

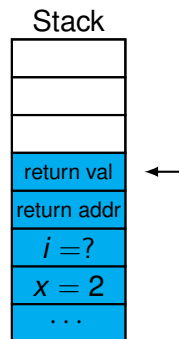


- To perform the function call, store the return address for CPU
- Allocate memory for the return value of the function

What happens when calling a function? VII

```
int square(int p) {           // <- Program counter
    int y;
    y = p * p;
    return y;
}

int main() {
    int x = 2;
    int i = square(x);
    cout << i << endl;
}
```

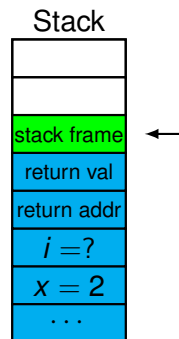


- Enter the function with the program counter

What happens when calling a function? VIII

```
int square(int p) {      // <- Program counter
    int y;
    y = p * p;
    return y;
}

int main() {
    int x = 2;
    int i = square(x);
    cout << i << endl;
}
```



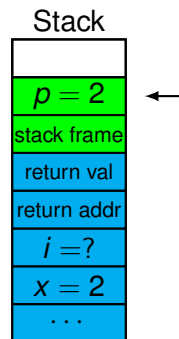
- Enter the function with the program counter
- Allocate “stack frame”, indicating that everything that follows is local to the function

What happens when calling a function? IX

```
int square(int p) {           // ← Program counter
    int y;
    y = p * p;
    return y;
}

int main() {
    int x = 2;
    int i = square(x);
    cout << i << endl;
}
```

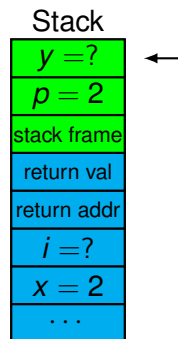
- Allocate the local variable `p` and copy its value from `x`



What happens when calling a function? X

```
int square(int p) {  
    int y;           // ← Program counter  
    y = p * p;  
    return y;  
}  
  
int main() {  
    int x = 2;  
    int i = square(x);  
    cout << i << endl;  
}
```

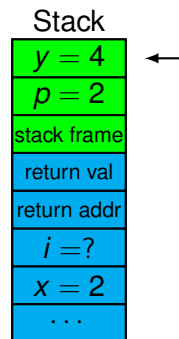
- Allocate the local variable `p` and copy its value from `x`
- Allocate local variables as soon as they are defined
- Allocate `y` with unknown value, since it was not initialized



What happens when calling a function? XI

```
int square(int p) {  
    int y;  
    y = p * p;           // ← Program counter  
    return y;  
}  
  
int main() {  
    int x = 2;  
    int i = square(x);  
    cout << i << endl;  
}
```

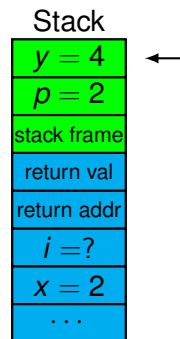
- Compute the square and assign the result to `y`



What happens when calling a function? XII

```
int square(int p) {  
    int y;  
    y = p * p;  
    return y;  
} // <- Program counter  
  
int main() {  
    int x = 2;  
    int i = square(x);  
    cout << i << endl;  
}
```

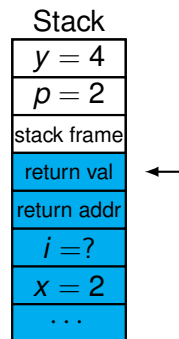
- Compute the square and assign the result to `y`
- Write the result to the placeholder variable for the return value



What happens when calling a function? XIII

```
int square(int p) {  
    int y;  
    y = p * p;  
    return y;           // <- Program counter  
}  
  
int main() {  
    int x = 2;  
    int i = square(x);  
    cout << i << endl;  
}
```

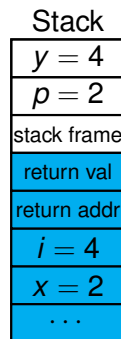
- Designate the memory as free until the stack frame
- This does *not* imply, that the memory is deleted!



What happens when calling a function? XIII

```
int square(int p) {  
    int y;  
    y = p * p;  
    return y;  
}  
  
int main() {  
    int x = 2;  
    int i = square(x);  
    cout << i << endl;  
}
```

// ← Program counter

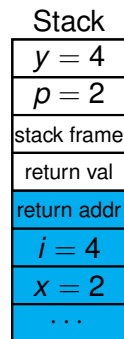


- Assign the return value to the designated variable `i`
- If there were no assignment, the return value will be lost

What happens when calling a function? XIV

```
int square(int p) {  
    int y;  
    y = p * p;  
    return y;  
}  
  
int main() {  
    int x = 2;  
    int i = square(x); // <- Program counter  
    cout << i << endl;  
}
```

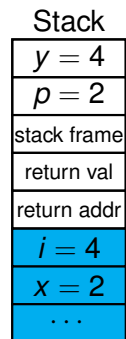
- CPU jumps back to the stored return address
- CPU will continue program execution from there



What happens when calling a function? XV

```
int square(int p) {  
    int y;  
    y = p * p;  
    return y;  
}  
  
int main() {  
    int x = 2;  
    int i = square(x); // <- Program counter  
    cout << i << endl;  
}
```

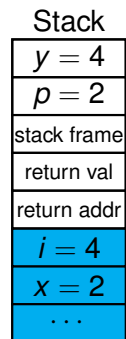
- Remove the return address from stack



What happens when calling a function? XV

```
int square(int p) {  
    int y;  
    y = p * p;  
    return y;  
}  
  
int main() {  
    int x = 2;  
    int i = square(x);  
    cout << i << endl;    // <- Program counter  
}
```

- Execute the print-statement



What happens when calling a function? XVI

```
int square(int p) {  
    int y;  
    y = p * p;  
    return y;  
}
```

```
int main() {  
    int x = 2;  
    int i = square(x);  
    cout << i << endl;  
}
```

// <- Program counter

- Reached the end of the program, hence free (but not delete) the remaining memory

Stack

y = 4
p = 2
stack frame
return val
return addr
i = 4
x = 2
...



What happens when calling a function? Comments

- Conclusion from previous slides: Calling a function is costly
- Natural question:
Should I avoid using “small” function for speed?

What happens when calling a function? Comments

- Conclusion from previous slides: Calling a function is costly
- Natural question:
Should I avoid using “small” function for speed?
- Answer: NO!
- Functions help to structure your code!

What happens when calling a function? Comments

- Conclusion from previous slides: Calling a function is costly
- Natural question:
Should I avoid using “small” function for speed?
- Answer: NO!
- Functions help to structure your code!
- For small functions, you may use the `inline` keyword, e.g.

```
inline int square(int p) {  
    return ( p * p );  
}
```

- Keyword `inline` gives a hint on the compiler to copy & paste the function body instead of performing a full-blown function call

Outline

- 1 About C++
- 2 Resources
- 3 Hello World

- 4 Comments and Code Blocks
- 5 Types and Operations
- 6 Loops and Conditionals
- 7 Types and Operations II: Containers
- 8 Functions

Section 10

Outlook on other C++ courses

The Intermediate Course will contain:

- aggregate types (better arrays, look-up tables)

Section 10

Outlook on other C++ courses

The Intermediate Course will contain:

- aggregate types (better arrays, look-up tables)
- Object Oriented programming

Section 10

Outlook on other C++ courses

The Intermediate Course will contain:

- aggregate types (better arrays, look-up tables)
- Object Oriented programming
- glimpses on the Standard Template Library

Section 10

Outlook on other C++ courses

The Intermediate Course will contain:

- aggregate types (better arrays, look-up tables)
- Object Oriented programming
- glimpses on the Standard Template Library
- handling exceptions

Section 10

Outlook on other C++ courses

The Intermediate Course will contain:

- aggregate types (better arrays, look-up tables)
- Object Oriented programming
- glimpses on the Standard Template Library
- handling exceptions
- basics on Software Design

Section 10

Outlook on other C++ courses

The Intermediate Course will contain:

- aggregate types (better arrays, look-up tables)
- Object Oriented programming
- glimpses on the Standard Template Library
- handling exceptions
- basics on Software Design
- and a realistic programming challenge

The Future

Future C/C⁺⁺/Fortran courses include:

- parallelization with MPI and OpenMPI (biennial change between Frankfurt and Mainz)
- Profiling / Debugging