HEP C++ course

Based on the work of
Sébastien Ponce
sebastien.ponce@cern.ch

CERN

March 2022
## Foreword

### What this course is not
- It is not for absolute beginners
- It is not for experts
- It is not complete at all (would need 3 weeks...)
  - although is it already too long for the time we have
  - 234 slides, 330 pages, 10s of exercises...

### How I see it
- **Adaptative**: pick what you want
- **Interactive**: tell me what to skip/insist on
- **Practical**: let’s spend time on real code

### Where to find latest version?
- pdf format at [http://cern.ch/sponce/C++Course](http://cern.ch/sponce/C++Course)
- full sources at [https://github.com/hsf-training/cpluspluscourse](https://github.com/hsf-training/cpluspluscourse)
More courses

The HSF Software Training Center

A set of course modules on more software engineering aspects prepared from within the HEP community

- Unix shell
- Python
- Version control (git, gitlab, github)
- ...

https://hepsoftwarefoundation.org/training/curriculum.html
Outline

1. History and goals
2. Language basics
3. Useful tools
4. Object orientation (OO)
5. Core modern C++
1. History and goals
   - History
   - Why we use it?

2. Language basics
   - Core syntax and types
   - Arrays and Pointers
   - Scopes / namespaces
   - Class and enum types
   - References
   - Functions
   - Operators
   - Control structures

3. Useful tools
   - Headers and interfaces
   - Auto keyword
   - C++ editor
   - Code management
   - Code formatting
   - The Compiling Chain
   - Debugging

4. Object orientation (OO)
   - Objects and Classes
   - Inheritance
   - Constructors/destructors

5. Core modern C++
   - Static members
   - Allocating objects
   - Advanced OO
   - Type casting
   - Operators
   - Functors
   - Constness
   - Exceptions
   - Templates
   - The STL
   - Lambdas
   - pointers and RAII
History and goals

1. History and goals
   - History
   - Why we use it?

2. Language basics

3. Useful tools

4. Object orientation (OO)

5. Core modern C++
History and goals

- History
- Why we use it?
C/C++ origins

- Simula (1967)
- BCPL
- B (1978)
- K and R C
- Classic C
- C with Classes (1980)
- Early C++ (1985)
- ARM C++
- C89
- C++98
- C99
- C++11
- C11
- C++14
- C++17
- C++20
- C inventor: Dennis M. Ritchie
- C++ inventor: Bjarne Stroustrup

- Both C and C++ are born in Bell Labs
- C++ *almost* embeds C
- C and C++ are still under development
- We will discuss all C++ specs but C++20
- Each slide will be marked with first spec introducing the feature
status

- A new C++ specification every 3 years
  - C++20 is ready, officially published by ISO in December 2020
- Bringing each time a lot of goodies
C++11, C++14, C++17, C++20...

**status**

- A new C++ specification every 3 years
  - C++20 is ready, officially published by ISO in December 2020
- Bringing each time a lot of goodies

**How to use C++XX features**

- Use a compatible compiler
- add -std=c++xx to compilation flags
- e.g. -std=c++17

<table>
<thead>
<tr>
<th>C++</th>
<th>gcc</th>
<th>clang</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>≥4.8</td>
<td>≥3.3</td>
</tr>
<tr>
<td>14</td>
<td>≥4.9</td>
<td>≥3.4</td>
</tr>
<tr>
<td>17</td>
<td>≥7.3</td>
<td>≥5</td>
</tr>
<tr>
<td>20</td>
<td>&gt;11</td>
<td>&gt;12</td>
</tr>
</tbody>
</table>

**Table:** Minimum versions of gcc and clang for a given C++ version
1. History and goals
   - History
   - Why we use it?
Why is C++ our language of choice?

Adapted to large projects
- statically and strongly typed
- object oriented
- widely used (and taught)
- many available libraries
Why is C++ our language of choice?

Adapted to large projects
- statically and strongly typed
- object oriented
- widely used (and taught)
- many available libraries

Fast
- compiled (unlike Java, C#, Python, ...)
- allows to go close to hardware when needed
Why is C++ our language of choice?

Adapted to large projects
- statically and strongly typed
- object oriented
- widely used (and taught)
- many available libraries

Fast
- compiled (unlike Java, C#, Python, ...)
- allows to go close to hardware when needed

What we get
- the most powerful language
- the most complicated one
- the most error prone?
Language basics

1 History and goals

2 Language basics
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions

3 Useful tools
- Operators
- Control structures
- Headers and interfaces
- Auto keyword

4 Object orientation (OO)

5 Core modern C++
Core syntax and types

2 Language basics
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- Auto keyword
```cpp
#include <iostream>

// This is a function
void print(int i) {
    std::cout << "Hello, world " << i << std::endl;
}

int main(int argc, char** argv) {
    int n = 3;
    for (int i = 0; i < n; i++) {
        print(i);
    }
    return 0;
}
```
// simple comment until end of line

int i;

/* multiline comment
 * in case we need to say more
 */

double /* or something in between */ d;

/**
 * Best choice: doxygen compatible comments
 * \brief checks whether i is odd
 * \param i input
 * \return true if i is odd, otherwise false
 * \see https://www.doxygenc.nl/manual/docblocks.html
 */

bool isOdd(int i);
Basic types(1)

```cpp
1. bool b = true; // boolean, true or false
2.
3. char c = 'a'; // min 8 bit integer
   // may be signed or not
4. // can store an ASCII character
5. signed char c = 4; // min 8 bit signed integer
6. unsigned char c = 4; // min 8 bit unsigned integer
7.
8. char* s = "a C string"; // array of chars ended by \0
9. string t = "a C++ string"; // class provided by the STL
10.
11. short int s = -444; // min 16 bit signed integer
12. unsigned short s = 444; // min 16 bit unsigned integer
13. short s = -444; // int is optional
14.
```
```cpp
1  int i = -123456;       // min 16, usually 32 bit
2  unsigned int i = 1234567; // min 16, usually 32 bit
3
4  long  l = 0L         // min 32 bit
5  unsigned long l = 0UL;   // min 32 bit
6
7  long long ll = 0LL;    // min 64 bit
8  unsigned long long l = 0ULL; // min 64 bit
9
10 float f = 1.23f;       // 32 (23+8+1) bit float
11 double d = 1.23E34;    // 64 (52+11+1) bit float
12 long double ld = 1.23E34L // min 64 bit float
```
Requires inclusion of a specific header

```cpp
#include <cstdint>

int8_t c = -3;       // 8 bit signed integer
uint8_t c = 4;       // 8 bit unsigned integer
int16_t s = -444;    // 16 bit signed integer
uint16_t s = 444;    // 16 bit unsigned integer
int32_t s = -0674;   // 32 bit signed integer
uint32_t s = 0674;   // 32 bit unsigned integer
int64_t s = -0x1bc;  // 64 bit signed integer
uint64_t s = 0x1bc;  // 64 bit unsigned integer
```
Requires inclusion of headers

```cpp
#include <cstdint> // and many other headers

size_t s = sizeof(int); // unsigned integer
    // can hold any variable's size

#include <cstdint>

ptrdiff_t c = &s - &s; // signed integer, can hold any
    // diff between two pointers

    // int, which can hold any pointer value:
intptr_t i = reinterpret_cast<intptr_t>(&s); // signed
uintptr_t i = reinterpret_cast<uintptr_t>(&s); // unsigned
```
2 Language basics

- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- Auto keyword
Static arrays

```c
1  int ai[4] = {1,2,3,4};
2  int ai[] = {1,2,3,4};  // identical
3
4  char ac[3] = {'a','b','c'};  // char array
5  char ac[4] = "abc";         // valid C string
6  char ac[4] = {'a','b','c',0}; // same valid string
7
8  int i = ai[2];  // i = 3
9  char c = ac[8]; // at best garbage, may segfault
10 int i = ai[4]; // also garbage!
```
```cpp
int i = 4;
int *pi = &i;
int j = *pi + 1;

int ai[] = {1,2,3};
int *pai = ai; // decay to ptr
int *paj = pai + 1;
int k = *paj + 1;

// compile error
int *pak = k;

// seg fault!
int *pak = (int*)k;
int l = *pak;
```
### Pointers

```cpp
1. int i = 4;
2. int *pi = &i;
3. int j = *pi + 1;
4.
5. int ai[] = {1,2,3};
6. int *pai = ai; // decay to ptr
7. int *paj = pai + 1;
8. int k = *paj + 1;
9.
10. // compile error
11. int *pak = k;
12.
13. // seg fault!
14. int *pak = (int*)k;
15. int l = *pak;
```

#### Memory layout

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| i = 4 | 0x3000 |
|       | 0x3004 |
|       | 0x3008 |
|       | 0x3012 |
|       | 0x3016 |
|       | 0x3020 |
|       | 0x3024 |
|       | 0x3028 |
|       | 0x3032 |
|       | 0x3036 |
|       | 0x3040 |
|       | 0x3044 |
|       | 0x3048 |
|       | 0x3052 |
|       | 0x3056 |
|       | 0x3060 |
|       | 0x3064 |

- `i` is stored at 0x3000
- `pi` points to i at 0x3000
- `j` is calculated as `*pi + 1` (5)
- `pai` points to ai at 0x300C
- `paj` points to ai[1] at 0x3010
- `k` is calculated as `*paj + 1` (3)
- `pak` is a pointer to an integer
- `l` is calculated as `*pak`
Pointers

1. `int i = 4;`
2. `int *pi = &i;`
3. `int j = *pi + 1;`
4. `int ai[] = {1, 2, 3};`
5. `int *pai = ai; // decay to ptr`
6. `int *paj = pai + 1;`
7. `int k = *paj + 1;`
8. `int *pak = k;`
9. `// compile error`
10. `int *pak = (int*)k;`
11. `int l = *pak;`

// seg fault!

Memory layout

- `pi = 0x3000`
- `i = 4`
- `0x3000`
- `0x3004`
- `0x3008`
- `0x300C`
- `0x3010`
- `0x3014`
- `0x3018`
- `0x301C`
- `0x3020`
- `0x3024`
- `0x3028`
```cpp
1 int i = 4;
2 int *pi = &i;
3 int j = *pi + 1;
4
5 int ai[] = {1, 2, 3};
6 int *pai = ai; // decay to ptr
7 int *paj = pai + 1;
8 int k = *paj + 1;
9
10 // compile error
11 int *pak = k;
12
13 // seg fault!
14 int *pak = (int*)k;
15 int l = *pak;
```
```cpp
int i = 4;
int *pi = &i;
int j = *pi + 1;

int ai[] = {1, 2, 3};
int *pai = ai; // decay to ptr
int *paj = pai + 1;
int k = *paj + 1;

// compile error
int *pak = k;

// seg fault!
int *pak = (int*)k;
int l = *pak;
```
```cpp
1. int i = 4;
2. int *pi = &i;
3. int j = *pi + 1;
4. 
5. int ai[] = {1, 2, 3};
6. int *pai = ai; // decay to ptr
7. int *paj = pai + 1;
8. int k = *paj + 1;
9. 
10. // compile error
11. int *pak = k;
12. 
13. // seg fault!
14. int *pak = (int*)k;
15. int l = *pak;
```

Memory layout:
```
0x3000: i = 4
0x3004: pi = 0x3000
0x3008: j = 5
0x300C: ai[0] = 1
0x3010: ai[1] = 2
0x3014: ai[2] = 3
0x3018: pai = 0x300C
0x3020: l = 4
0x3024: pak = 3
```
```c
int i = 4;
int *pi = &i;
int j = *pi + 1;

int ai[] = {1, 2, 3};
int *pai = ai; // decay to ptr
int *paj = pai + 1;
int k = *paj + 1;

// compile error
int *pak = k;

// seg fault!
int *pak = (int*)k;
int l = *pak;
```
```cpp
int i = 4;
int *pi = &i;
int j = *pi + 1;

int ai[] = {1, 2, 3};
int *pai = ai; // decay to ptr
int *paj = pai + 1;
int k = *paj + 1;

// compile error
int *pak = k;

// seg fault!
int *pak = (int*)k;
int l = *pak;
```
```cpp
int i = 4;
int *pi = &i;
int j = *pi + 1;

int ai[] = {1,2,3};
int *pai = ai; // decay to ptr
int *paj = pai + 1;
int k = *paj + 1;

// compile error
int *pak = k;

// seg fault!
int *pak = (int*)k;
int l = *pak;
```
Finally a C++ NULL pointer

- if a pointer doesn’t point to anything, set it to `nullptr`
- works like 0 or NULL in standard cases
- triggers compilation error when mapped to integer
Finally a C++ NULL pointer

- if a pointer doesn’t point to anything, set it to `nullptr`
- works like 0 or NULL in standard cases
- triggers compilation error when mapped to integer

Example code

```cpp
1. void* vp = nullptr;
2. int* ip = nullptr;
3. int i = NULL;   // OK -> bug ?
4. int i = nullptr; // ERROR
```
#include <cstdlib>
#include <cstring>

int *bad; // pointer to random address
int *ai = nullptr; // better, deterministic, can be tested

// allocate array of 10 ints (uninitialized)
ai = (int*) malloc(10*sizeof(int));
memset(ai, 0, 10*sizeof(int)); // and set them to 0

ai = (int*) calloc(10, sizeof(int)); // both in one go

free(ai); // release memory

Don’t use C’s memory management
Use std::vector and friends or smart pointers
2 Language basics

- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- Auto keyword
Definition

Portion of the source code where a given name is valid. Typically:
- simple block of code, within {}
- function, class, namespace
- the global scope, i.e. translation unit (.cpp file + all includes)

Example

```cpp
{ int a;
  { int b;
  } // end of b scope
} // end of a scope
```
Scope and lifetime of variables

Variable life time

- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope
- Good practice: initialise variables when allocating them!

```cpp
int a = 1;
{
    int b[4];
    b[0] = a;
}
// Doesn't compile here:
// b[1] = a + 1;
```

Memory layout:

<table>
<thead>
<tr>
<th></th>
<th>0x3000</th>
<th>0x3004</th>
<th>0x3008</th>
<th>0x300C</th>
<th>0x3010</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x3000</td>
</tr>
<tr>
<td>b[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x3004</td>
</tr>
<tr>
<td>b[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x3008</td>
</tr>
<tr>
<td>b[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x300C</td>
</tr>
<tr>
<td>b[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x3010</td>
</tr>
</tbody>
</table>
### Scope and lifetime of variables

**Variable life time**
- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope
- Good practice: initialise variables when allocating them!

```cpp
int a = 1;
{
    int b[4];
    b[0] = a;
}
// Doesn't compile here:
// b[1] = a + 1;
```

<table>
<thead>
<tr>
<th>Memory layout</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>b[3] = ?</td>
<td>0x3010</td>
</tr>
<tr>
<td>b[2] = ?</td>
<td>0x300C</td>
</tr>
<tr>
<td>b[1] = ?</td>
<td>0x3008</td>
</tr>
<tr>
<td>b[0] = ?</td>
<td>0x3004</td>
</tr>
<tr>
<td>a = 1</td>
<td>0x3000</td>
</tr>
</tbody>
</table>
Scope and lifetime of variables

Variable life time

- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope
- Good practice: initialise variables when allocating them!

```cpp
int a = 1;
{
    int b[4];
    b[0] = a;
}
// Doesn't compile here:
// b[1] = a + 1;
```

Memory layout:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b[3]</td>
<td>0x3010</td>
</tr>
<tr>
<td>b[2]</td>
<td>0x300C</td>
</tr>
<tr>
<td>b[1]</td>
<td>0x3008</td>
</tr>
<tr>
<td>b[0]</td>
<td>0x3004</td>
</tr>
<tr>
<td>a</td>
<td>0x3000</td>
</tr>
</tbody>
</table>
Scope and lifetime of variables

Variable lifetime

- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope
- Good practice: initialise variables when allocating them!

```cpp
int a = 1;
{
    int b[4];
    b[0] = a;
}
// Doesn't compile here:
// b[1] = a + 1;
```

### Memory layout

<table>
<thead>
<tr>
<th></th>
<th>0x3000</th>
<th>0x3004</th>
<th>0x3008</th>
<th>0x3010</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b[0]</td>
<td>?</td>
<td></td>
<td></td>
<td>0x3010</td>
</tr>
<tr>
<td>b[1]</td>
<td>?</td>
<td></td>
<td></td>
<td>0x300C</td>
</tr>
<tr>
<td>b[2]</td>
<td>?</td>
<td></td>
<td></td>
<td>0x3008</td>
</tr>
<tr>
<td>b[3]</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a = 1</td>
<td>1</td>
<td></td>
<td></td>
<td>0x3004</td>
</tr>
</tbody>
</table>
```
Namespaces

- Namespaces allow to segment your code to avoid name clashes.
- They can be embedded to create hierarchies (separator is '::')

```cpp
int a;
namespace n {
    int a;  // no clash
}
namespace p {
    int a;  // no clash
    namespace inner {
        int a;  // no clash
    }
}
void f() {
    n::a = 3;
}

namespace p {
    void f() {
        p::a = 6;
        a = 6;     // same as above
        ::a = 1;
        p::inner::a = 8;
        inner::a = 8;
        n::a = 3;
    }
}
using namespace p::inner;
void g() {
    a = -1;  // err: ambiguous
}
```
Easier way to declare nested namespaces

```cpp
namespace A {
    namespace B {
        namespace C {
            //...
        }
    }
}
```

```cpp
namespace A::B::C {
    //...
}
```
Unnamed namespaces

A namespace without a name!

```cpp
namespace {
    int localVar;
}
```

Purpose

- groups a number of declarations
- visible only in the current translation unit
- but not reusable outside
- allows much better compiler optimizations and checking
  - e.g. unused function warning
  - context dependent optimizations

Deprecates static

```cpp
static int localVar; // equivalent C code
```
Class and enum types

2 Language basics
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- Auto keyword
“members” grouped together under one name

```cpp
struct Individual {
    unsigned char age;
    float weight;
};

Individual student;
student.age = 25;
student.weight = 78.5f;

Individual teacher = {
    45, 67.0f
};

Individual *ptr = &student;
ptr->age = 25;
// same as: (*ptr).age = 25;
```


```cpp
struct Individual {
    unsigned char age;
    float weight;
};

Individual student;
student.age = 25;
student.weight = 78.5f;

Individual teacher = {
    45, 67.0f
};

Individual *ptr = &student;
ptr->age = 25;
// same as: (*ptr).age = 25;
```

**Memory layout**

<table>
<thead>
<tr>
<th>Address (Hex)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3000</td>
<td></td>
</tr>
<tr>
<td>0x3004</td>
<td></td>
</tr>
<tr>
<td>0x3008</td>
<td></td>
</tr>
<tr>
<td>0x3010</td>
<td></td>
</tr>
<tr>
<td>0x300C</td>
<td></td>
</tr>
</tbody>
</table>
```
```
struct Individual {
  unsigned char age;
  float weight;
};

Individual student;
student.age = 25;
student.weight = 78.5f;

Individual teacher = {
  45, 67.0f
};

Individual *ptr = &student;
ptr->age = 25;
// same as: (*ptr).age = 25;
```

```
Memory layout

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3000</td>
<td>0x3004</td>
<td>0x3008</td>
<td>0x3010</td>
</tr>
<tr>
<td>25</td>
<td>78.5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

"members" grouped together under one name
“members” grouped together under one name

```cpp
struct Individual {
    unsigned char age;
    float weight;
};

Individual student;
student.age = 25;
student.weight = 78.5f;

Individual teacher = {
    45, 67.0f
};

Individual *ptr = &student;
ptr->age = 25;
// same as: (*ptr).age = 25;
```

Memory layout:

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3000</td>
<td>25</td>
</tr>
<tr>
<td>0x3004</td>
<td>78.5</td>
</tr>
<tr>
<td>0x3008</td>
<td>67.0</td>
</tr>
<tr>
<td>0x300C</td>
<td>45</td>
</tr>
<tr>
<td>0x3010</td>
<td></td>
</tr>
</tbody>
</table>
```
struct Individual {
    unsigned char age;
    float weight;
};

Individual student;
student.age = 25;
student.weight = 78.5f;

Individual teacher = {
    45, 67.0f
};

Individual *ptr = &student;
ptr->age = 25;
// same as: (*ptr).age = 25;

“members” grouped together under one name
union Duration {
    int seconds;
    short hours;
    char days;
};

Duration d1, d2, d3;
d1.seconds = 259200;
d2.hours = 72;
d3.days = 3;
d1.days = 3; // d1.seconds overwritten
int a = d1.seconds; // d1.seconds is garbage
union Duration {
    int seconds;
    short hours;
    char days;
};

Duration d1, d2, d3;
d1.seconds = 259200;
d2.hours = 72;
d3.days = 3;
d1.days = 3; // d1.seconds overwritten
int a = d1.seconds; // d1.seconds is garbage
Union

“members” packed together at same memory location

```
union Duration {
    int seconds;
    short hours;
    char days;
};
Duration d1, d2, d3;
d1.seconds = 259200;
d2.hours = 72;
d3.days = 3;
d1.days = 3; // d1.seconds overwritten
int a = d1.seconds; // d1.seconds is garbage
```
“members” packed together at same memory location

```c++
union Duration {
    int seconds;
    short hours;
    char days;
};
Duration d1, d2, d3;
d1.seconds = 259200;
d2.hours = 72;
d3.days = 3;
d1.days = 3; // d1.seconds overwritten
int a = d1.seconds; // d1.seconds is garbage
```
union Duration {
    int seconds;
    short hours;
    char days;
};
Duration d1, d2, d3;
d1.seconds = 259200;
d2.hours = 72;
d3.days = 3;
d1.days = 3; // d1.seconds overwritten
int a = d1.seconds; // d1.seconds is garbage
“members” packed together at same memory location

```cpp
union Duration {
    int seconds;
    short hours;
    char days;
};

Duration d1, d2, d3;
d1.seconds = 259200;
d2.hours = 72;
d3.days = 3;
d1.days = 3; // d1.seconds overwritten
int a = d1.seconds; // d1.seconds is garbage
```
"members" packed together at same memory location

```cpp
union Duration {
    int seconds;
    short hours;
    char days;
};

Duration d1, d2, d3;
d1.seconds = 259200;
d2.hours = 72;
d3.days = 3;
d1.days = 3;  // d1.seconds overwritten
int a = d1.seconds;  // d1.seconds is garbage
```

Starting with C++17: prefer std::variant
Enums

- use to declare a list of related constants (enumerators)
- has an underlying integral type
- enumerator names leak into enclosing scope

```cpp
enum VehicleType {
    BIKE, // 0
    CAR, // 1
    BUS, // 2
};
VehicleType t = CAR;
```

```cpp
enum VehicleType
:
int {
    // C++11
    BIKE = 3,
    CAR = 5,
    BUS = 7,
};
VehicleType t2 = BUS;
```
Scoped enumeration, aka enum class

Same syntax as enum, with scope

```cpp
enum class VehicleType { Bus, Car };
VehicleType t = VehicleType::Car;
```
Scoped enumeration, aka enum class

Same syntax as enum, with scope

```cpp
enum class VehicleType { Bus, Car };  
VehicleType t = VehicleType::Car;
```

Only advantages

- scopes enumerator names, avoids name clashes
- strong typing, no automatic conversion to int

```cpp
enum VType { Bus, Car }; enum Color { Red, Blue };  
VType t = Bus;  
if (t == Red) { /* We do enter */ }  
int a = 5 * Car; // Ok, a = 5
```

```cpp
enum class VT { Bus, Car }; enum class Col { Red, Blue };  
VT t = VT::Bus;  
if (t == Col::Red) { /* Compiler error */ }  
int a = t * 5; // Compiler error
```
enum class ShapeType {
    Circle,
    Rectangle
};

struct Rectangle {
    float width;
    float height;
};

struct Shape {
    ShapeType type;
    union {
        float radius;
        Rectangle rect;
    }
};

Shape s;
s.type = ShapeType::Circle;
s.radius = 3.4;

Shape t;
t.type = ShapeType::Rectangle;
t.rect.width = 3;
t.rect.height = 4;
enum class ShapeType {
    Circle,
    Rectangle
};

struct Rectangle {
    float width;
    float height;
};

struct Shape {
    ShapeType type;
    union {
        float radius;
        Rectangle rect;
    }
};
More sensible example

```cpp
enum class ShapeType {
    Circle,
    Rectangle
};

struct Rectangle {
    float width;
    float height;
};

struct Shape {
    ShapeType type;
    union {
        float radius;
        Rectangle rect;
    }
};

Shape s;
shape.s.type = ShapeType::Circle;
s.radius = 3.4;

Shape t;
t.type = ShapeType::Rectangle;
t.rect.width = 3;
t.rect.height = 4;
```
Used to create type aliases

C++98

typedef uint64_t myint;
myint toto = 17;
typedef int pos[3];

C++11

using myint = uint64_t;
myint toto = 17;
using pos = int[3];

template <typename T> using myvec = std::vector<T>;
myvec<int> titi;
2 Language basics

- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
  - Functions
  - Operators
  - Control structures
  - Headers and interfaces
  - Auto keyword
References

- References allow for direct access to another object
- They can be used as shortcuts / better readability
- They can be declared `const` to allow only read access
- They can be used as function arguments

Example:

```
int i = 2;
int &iref = i; // access to i
iref = 3; // i is now 3

// const reference to a member:
struct A { int x; int y; } a;
const int &x = a.x; // direct read access to A's x
x = 4; // doesn't compile
```
### Specificities of reference

- Natural syntax
- Must be assigned when defined, cannot be `nullptr`
- Cannot be reassigned
- Non-const references to temporary objects are not allowed

### Advantages of pointers

- Can be reassigned to point elsewhere or to `nullptr`
- Clearly indicates that argument may be modified
## Pointers vs References

### Specificities of reference
- natural syntax
- must be assigned when defined, cannot be `nullptr`
- cannot be reassigned
- non-const references to temporary objects are not allowed

### Advantages of pointers
- can be reassigned to point elsewhere or to `nullptr`
- clearly indicates that argument may be modified

### Good practice
- Always use references when you can
- Consider that a reference will be modified
- Use constness when it’s not the case
Functions

2 Language basics

- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References

- Functions
  - Operators
  - Control structures
  - Headers and interfaces
  - Auto keyword
```cpp
// with return type
int square(int a) {
    return a * a;
}

// multiple parameters
int mult(int a, int b) {
    return a * b;
}

// no return
void log(char* msg) {
    std::cout << msg;
}

// no parameter
void hello() {
    std::cout << "Hello World";
}
```
Function default arguments

```cpp
// must be the trailing argument
int add(int a,
    int b = 2) {
    return a + b;
}
// add(1) == 3
// add(3,4) == 7

// multiple default arguments are possible
int add(int a = 2,
    int b = 2) {
    return a + b;
}
// add() == 4
// add(3) == 5
```
struct BigStruct {...};
BigStruct s;

// parameter by value
void printBS(BigStruct p) {
    ...
}
printBS(s); // copy

// parameter by reference
void printBSp(BigStruct &q) {
    ...
}
printBSp(s); // no copy
struct BigStruct {...};

BigStruct s;

// parameter by value
void printBS(BigStruct p) {
   ...
}
printBS(s); // copy

// parameter by reference
void printBSp(BigStruct &q) {
   ...
}
printBSp(s); // no copy
Functions: parameters are passed by value

```c++
1 struct BigStruct {...};
2 BigStruct s;

// parameter by value
4 void printBS(BigStruct p) {
5     ...
6 }
7 printBS(s); // copy

// parameter by reference
10 void printBSp(BigStruct &q) {
11     ...
12 }
13 printBSp(s); // no copy
```

Memory layout:

```
<table>
<thead>
<tr>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3000</td>
</tr>
<tr>
<td>s1</td>
</tr>
<tr>
<td>0x3001</td>
</tr>
<tr>
<td>s</td>
</tr>
<tr>
<td>0x3002</td>
</tr>
<tr>
<td>sn</td>
</tr>
<tr>
<td>0x3003</td>
</tr>
<tr>
<td>p1</td>
</tr>
<tr>
<td>0x3004</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>0x3005</td>
</tr>
<tr>
<td>pn</td>
</tr>
<tr>
<td>0x3006</td>
</tr>
</tbody>
</table>
```
Functions: parameters are passed by value

```cpp
struct BigStruct {...};
BigStruct s;

// parameter by value
void printBS(BigStruct p) {
    ...
}
printBS(s); // copy

// parameter by reference
void printBSp(BigStruct &q) {
    ...
}
printBSp(s); // no copy
```
Functions: pass by value or reference?

```cpp
struct SmallStruct { int a; };
SmallStruct s = {1};

void changeSS(SmallStruct p) {
    p.a = 2;
}
changeSS(s);
// s.a == 1

void changeSS2(SmallStruct &q) {
    q.a = 2;
}
changeSS2(s);
// s.a == 2
```

Memory layout:

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3000</td>
<td></td>
</tr>
<tr>
<td>0x3004</td>
<td></td>
</tr>
<tr>
<td>0x3008</td>
<td></td>
</tr>
</tbody>
</table>
```
```cpp
struct SmallStruct { int a; };
SmallStruct s = {1};

void changeSS(SmallStruct p) {
    p.a = 2;
}
changeSS(s);
// s.a == 1

void changeSS2(SmallStruct &q) {
    q.a = 2;
}
changeSS2(s);
// s.a == 2
```

**Memory layout**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3000</td>
<td>s.a = 1</td>
</tr>
<tr>
<td>0x3004</td>
<td></td>
</tr>
<tr>
<td>0x3008</td>
<td></td>
</tr>
</tbody>
</table>

B. Gruber, S. Hageboeck, S. Ponce
Functions: pass by value or reference?

```c++
struct SmallStruct { int a; };
SmallStruct s = {1};

void changeSS(SmallStruct p) {
    p.a = 2;
}
changeSS(s);
// s.a == 1

void changeSS2(SmallStruct &q) {
    q.a = 2;
}
changeSS2(s);
// s.a == 2
```

Memory layout:

```
<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3000</td>
<td>p.a = 1</td>
</tr>
<tr>
<td>0x3004</td>
<td>s.a = 1</td>
</tr>
<tr>
<td>0x3008</td>
<td></td>
</tr>
</tbody>
</table>
```
Functions: pass by value or reference?

```c++
struct SmallStruct {int a;};
SmallStruct s = {1};

void changeSS(SmallStruct p) {
    p.a = 2;
}
changeSS(s);
// s.a == 1

void changeSS2(SmallStruct &q) {
    q.a = 2;
}
changeSS2(s);
// s.a == 2
```

Memory layout:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3000</td>
<td>s.a = 1</td>
</tr>
<tr>
<td>0x3004</td>
<td>p.a = 2</td>
</tr>
<tr>
<td>0x3008</td>
<td>s.a = 1</td>
</tr>
</tbody>
</table>

```
struct SmallStruct {int a;};
SmallStruct s = {1};

void changeSS(SmallStruct p) {
    p.a = 2;
}
changeSS(s);
// s.a == 1

void changeSS2(SmallStruct &q) {
    q.a = 2;
}
changeSS2(s);
// s.a == 2
```
struct SmallStruct {int a;};
SmallStruct s = {1};

void changeSS(SmallStruct p) {
    p.a = 2;
}
changeSS(s);
// s.a == 1

void changeSS2(SmallStruct &q) {
    q.a = 2;
}
changeSS2(s);
// s.a == 2
Functions: pass by value or reference?

```cpp
struct SmallStruct { int a; };
SmallStruct s = {1};

void changeSS(SmallStruct p) {
    p.a = 2;
}
changeSS(s);
// s.a == 1

void changeSS2(SmallStruct &q) {
    q.a = 2;
}
changeSS2(s);
// s.a == 2
```

Memory layout:

```
0x3000 0x3004 0x3008
q = 0x3000 0x3004 0x3000
s.a = 2
```
Functions: pass by value or reference?

```c++
struct SmallStruct { int a; };
SmallStruct s = {1};

void changeSS(SmallStruct p) {
    p.a = 2;
}
changeSS(s);
// s.a == 1

void changeSS2(SmallStruct &q) {
    q.a = 2;
}
changeSS2(s);
// s.a == 2
```

Memory layout:

<table>
<thead>
<tr>
<th>0x3000</th>
<th>0x3004</th>
<th>0x3008</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.a = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0000</td>
</tr>
</tbody>
</table>

B. Gruber, S. Hageboeck, S. Ponce

HEP C++ course
Different ways to pass arguments to a function

- by default, arguments are passed by value (= copy) good for small types, e.g. numbers
- prefer references for mandatory parameters to avoid copies
- use pointers for optional parameters to allow `nullptr`
- use `const` for safety and readability whenever possible
Pass by value, reference or pointer

Different ways to pass arguments to a function

- by default, arguments are passed by value (≈ copy)
  good for small types, e.g. numbers
- prefer references for mandatory parameters to avoid copies
- use pointers for optional parameters to allow `nullptr`
- use `const` for safety and readability whenever possible

Syntax

1. ```
   struct T {...}; T a;
```
2. ```
   void f(T value);  f(a);  // by value
```
3. ```
   void fRef(const T &value);  fRef(a);  // by reference
```
4. ```
   void fPtr(const T *value);  fPtr(&a);  // by pointer
```
5. ```
   void fWrite(T &value);  fWrite(a);  // non-const ref
```
Exercise

Familiarise yourself with pass by value / pass by reference.

- go to code/functions
- Look at functions.cpp
- Compile it (make) and run the program (.functions)
- Work on the tasks that you find in functions.cpp
Functions: good practices

Ensure good readability/maintainability:

- Keep functions short
- Do one logical thing (single-responsibility principle)
- Use expressive names
- Document non-trivial functions

Example: Good

```cpp
/// Count number of dilepton events in data.
/// \param d Dataset to search.
unsigned int countDileptons(Data d) {
    selectEventsWithMuons(d);
    selectEventsWithElectrons(d);
    return d.size();
}
```
Example: don’t! Everything in one long function

```cpp
unsigned int runJob() {
    // Step 1: data
    Data data;
    data.resize(123456);
    data.fill(...);

    // Step 2: muons
    for (...) {
        if (...) {
            data.erase(...);
        }
    }

    // Step 3: electrons
    for (...) {
        if (...) {
            data.erase(...);
        }
    }

    // Step 4: dileptons
    int counter = 0;
    for (...) {
        if (...) {
            counter++;
        }
    }

    return counter;
}
```
Operators

2 Language basics

- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces
- Auto keyword
### Binary & Assignment Operators

```c++
int i = 1 + 4 - 2;    // 3
i *= 3;               // 9, short for: i = i * 3;
i /= 2;               // 4
i = 23 % i;           // modulo => 3
```
## Operators(1)

### Binary & Assignment Operators

```cpp
int i = 1 + 4 - 2;  // 3
i *= 3;             // 9, short for: i = i * 3;
i /= 2;             // 4
i = 23 % i;         // modulo => 3
```

### Increment / Decrement

```cpp
int i = 0; i++;   // i = 1
int j = ++i;     // i = 2, j = 2
int k = i++;     // i = 3, k = 2
int l = --i;     // i = 2, l = 2
int m = i--;     // i = 1, m = 2
```
Operators(1)

### Binary & Assignment Operators

1. `int i = 1 + 4 - 2;`  // 3
2. `i *= 3;`  // 9, short for: `i = i * 3;`
3. `i /= 2;`  // 4
4. `i = 23 % i;`  // modulo => 3

### Increment / Decrement

1. `int i = 0; i++;`  // `i = 1`
2. `int j = ++i;`  // `i = 2, j = 2`
3. `int k = i++;`  // `i = 3, k = 2`
4. `int l = --i;`  // `i = 2, l = 2`
5. `int m = i--;`  // `i = 1, m = 2`

Use wisely.
Operators(2)

Bitwise and Assignment Operators

```c
1 int i = 0xee & 0x55; // 0x44
2 i |= 0xee; // 0xee
3 i ^= 0x55; // 0xbb
4 int j = ~0xee; // 0xffffffff11
5 int k = 0x1f << 3; // 0xf8
6 int l = 0x1f >> 2; // 0x7
```
Bitwise and Assignment Operators

```
1 int i = 0xee & 0x55;    // 0x44
2 i |= 0xee;             // 0xee
3 i ^= 0x55;             // 0xbb
4 int j = ~0xee;         // 0xffffffff11
5 int k = 0x1f << 3;     // 0xf8
6 int l = 0x1f >> 2;     // 0x7
```

Boolean Operators

```
1 bool a = true;
2 bool b = false;
3 bool c = a && b;       // false
4 bool d = a || b;       // true
5 bool e = !d;           // false
```
## Comparison Operators

1. `bool a = (3 == 3); // true`
2. `bool b = (3 != 3); // false`
3. `bool c = (4 < 4); // false`
4. `bool d = (4 <= 4); // true`
5. `bool e = (4 > 4); // false`
6. `bool f = (4 >= 4); // true`
Comparison Operators

1. \texttt{bool a = (3 == 3); // true}
2. \texttt{bool b = (3 != 3); // false}
3. \texttt{bool c = (4 < 4); // false}
4. \texttt{bool d = (4 <= 4); // true}
5. \texttt{bool e = (4 > 4); // false}
6. \texttt{bool f = (4 >= 4); // true}

Precedences

\texttt{c &=(1+(++b) | (a--) * 4 \% 5 ^ 7; // ???}

Details can be found on \texttt{cppreference}
## Operators(3)

### Comparison Operators

```cpp
1. bool a = (3 == 3);  // true
2. bool b = (3 != 3);  // false
3. bool c = (4 < 4);   // false
4. bool d = (4 <= 4);  // true
5. bool e = (4 > 4);   // false
6. bool f = (4 >= 4);  // true
```

### Precedences

```
c &= 1+(++b) | (a--) * 4 % 5 ^ 7;  // ???
```

Details can be found on [cppreference](https://en.cppreference.com/w/cpp/language/operator_binary_comparison)

---

**Don’t use**
### Comparison Operators

```cpp
1. bool a = (3 == 3);  // true
2. bool b = (3 != 3);  // false
3. bool c = (4 < 4);   // false
4. bool d = (4 <= 4);  // true
5. bool e = (4 > 4);   // false
6. bool f = (4 >= 4);  // true
```

### Precedences

```cpp
c &= 1+(++b)|(a--)*4%5^7;  // ???
```

Details can be found on [cppreference](https://en.cppreference.com/w/cpp/language/operator_precedence)
2 Language basics

- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators

- Control structures
- Headers and interfaces
- Auto keyword
Control structures: if

if syntax

```cpp
if (condition1) {
    Statement1; Statement2;
} else if (condition2)
    OnlyOneStatement;
else {
    Statement3;
    Statement4;
}
```

- `else` and `else if` clause are optional
- `else if` clause can be repeated
- braces are optional if there is a single statement
Control structures: if

Practical example

```cpp
int collatz(int a) {
    if (a <= 0) {
        std::cout << "not supported";
        return 0;
    } else if (a == 1) {
        return 1;
    } else if (a%2 == 0) {
        return collatz(a/2);
    } else {
        return collatz(3*a+1);
    }
}
```
Control structures: conditional operator

Syntax

```
test ? expression1 : expression2;
- if test is true expression1 is returned
- else expression2 is returned
```
Control structures: conditional operator

Syntax

```
    test ? expression1 : expression2;
```

- if test is `true` expression1 is returned
- else expression2 is returned

Practical example

```
int collatz(int a) {
    return a==1 ? 1 : collatz(a%2==0 ? a/2 : 3*a+1);
}
```
Control structures: conditional operator

Syntax

```c
int collatz(int a) {
    return a==1 ? 1 : collatz(a%2==0 ? a/2 : 3*a+1);
}
```

Practical example

- explicit ifs are easier to read
- use only when obvious and not nested
Control structures: switch

Syntax

```
switch(identifier) {
    case c1 : statements1; break;
    case c2 : statements2; break;
    case c3 : statements3; break;
    ...
    default : instructiond; break;
}
```

- `break` is not mandatory but...
- cases are entry points, not independent pieces
- execution falls through to the next case without a `break`!
- `default` may be omitted
Control structures: switch

Syntax

switch(identifier) {
  case c1 : statements1; break;
  case c2 : statements2; break;
  case c3 : statements3; break;
  ...
  default : instructiond; break;
}

- **break** is not mandatory but...
- cases are entry points, not independent pieces
- execution falls through to the next case without a **break**!
- **default** may be omitted

Use break

Do not try to make use of non breaking cases
**Control structures: switch**

**Practical example**

```cpp
enum class Lang { French, German, English, Other }
...

switch (language) {
    case Lang::French:
        std::cout << "Bonjour";
        break;
    case Lang::German:
        std::cout << "Guten Tag";
        break;
    case Lang::English:
        std::cout << "Good morning";
        break;
    default:
        std::cout << "I do not speak your language";
}
```

B. Gruber, S. Hageboeck, S. Ponce
New compiler warning

Since C++17, compilers are encouraged to warn on fall-through

```cpp
switch (c) {
  case 'a':
    f();    // Warning emitted
  case 'b': // Warning emitted
  case 'c':
    g();
    [[fallthrough]]; // Warning suppressed
  case 'd':
    h();
}
```
Allows to limit variable scope in \textit{if} and \textit{switch} statements

```cpp
if (Value val = GetValue(); condition(val)) {
    f(val);
} else {
    g(val);
}
}

h(val); // compile error
```
init-statements for if and switch

Allows to limit variable scope in `if` and `switch` statements

```cpp
if (Value val = GetValue(); condition(val)) {
    f(val);
} else {
    g(val);
}

h(val); // compile error
```

Don’t confuse with a variable declaration as condition:

```cpp
if (Value* val = GetValuePtr())
    f(*val);
```
Control structures: for loop

for loop syntax

```cpp
for(initializations; condition; increments) {
    statements;
}
```
- initializations and increments are comma separated
- initializations can contain declarations
- braces are optional if loop body is a single statement
Control structures: for loop

for loop syntax

```cpp
for(initializations; condition; increments) {
    statements;
}
```
- initializations and increments are comma separated
- initializations can contain declarations
- braces are optional if loop body is a single statement

Practical example

```cpp
for(int i = 0, j = 0 ; i < 10 ; i++, j = i*i) {
    std::cout << i << '^2 is ' << j << '
';
}
```
Control structures: for loop

for loop syntax

```cpp
for(initializations; condition; increments) {
    statements;
}
```

- initializations and increments are comma separated
- initializations can contain declarations
- braces are optional if loop body is a single statement

Practical example

```cpp
for(int i = 0, j = 0 ; i < 10 ; i++, j = i*i) {
    std::cout << i << '^2 is ' << j << '
';
}
```

Do not abuse the syntax

The `for` loop head should fit in 1-3 lines
Range-based loops

Reason of being

- simplifies loops over “ranges” tremendously
- especially with STL containers

Syntax

```
for ( type iteration_variable : range ) {
    // body using iteration_variable
}
```

Example code

```
int v[4] = {1,2,3,4};
int sum = 0;
for (int a : v) { sum += a; }
```
## Control structures: while loop

### while loop syntax

```
while (condition) {
  statements;
}
```

```
do {
  statements;
} while (condition);
```

- braces are optional if the body is a single statement
Control structures: while loop

while loop syntax

```cpp
while (condition) {
    statements;
}
do {
    statements;
} while (condition);
```

- braces are optional if the body is a single statement

Bad example

```cpp
while (n != 1)
    if (0 == n%2) n /= 2;
else n = 3 * n + 1;
```
Control structures: jump statements

- **break** exits the loop and continues after it
- **continue** goes immediately to next loop iteration
- **return** exists the current function
- **goto** can jump anywhere inside a function, don’t use!
Control structures: jump statements

- **break** exits the loop and continues after it
- **continue** goes immediately to next loop iteration
- **return** exists the current function
- **goto** can jump anywhere inside a function, don’t use!

### Bad example

```cpp
while (1) {
    if (n == 1) break;
    if (0 == n%2) {
        std::cout << n << ' \n';
        n /= 2;
        continue;
    }
    n = 3 * n + 1;
}
```
Exercise

Familiarise yourself with different kinds of control structures. Re-implement them in different ways.

- Go to code/control
- Look at control.cpp
- Compile it (make) and run the program (./control)
- Work on the tasks that you find in README.md
2 Language basics
- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures

- Headers and interfaces
- Auto keyword
Headers and interfaces

Interface

Set of declarations defining some functionality
- put in a so-called “header file”
- the implementation exists somewhere else

Header: hello.hpp

```cpp
void printHello();
```

Usage: myfile.cpp

```cpp
#include "hello.hpp"

int main() {
    printHello();
}
```
```cpp
// file inclusion
#include "hello.hpp"

// macro constants and function-style macros
#define MY_GOLDEN_NUMBER 1746
#define CHECK_ERROR(x) if ((x) != MY_GOLDEN_NUMBER) \
    std::cerr << #x " was not the golden number\n";

// compile time or platform specific configuration
#if defined(USE64BITS) || defined(__GNUG__) \
    using myint = uint64_t;
#else
    using myint = uint32_t;
#endif
```
```cpp
// file inclusion
#include "hello.hpp"

// macro constants and function-style macros
#define MY_GOLDEN_NUMBER 1746
#define CHECK_ERROR(x) if ((x) != MY_GOLDEN_NUMBER) \
    std::cerr << #x " was not the golden number\n";

// compile time or platform specific configuration
#if defined(USE64BITS) || defined(__GNUG__) \
    using myint = uint64_t;
#else
    using myint = uint32_t;
#endif
```

Use only in very restricted cases

- inclusion of headers
- customization for specific compilers/platforms
Problem: redefinition by accident

- a header may define new names (e.g. types)
- multiple (transitive) inclusions of a header would define those names multiple times, which is a compile error
- solution: guard the content of your headers!

Include guards

```
#ifndef MY_HEADER_INCLUDED
#define MY_HEADER_INCLUDED
...
// content
#endif
```

Pragma once (non-standard)

```
#pragma once
...
// content
```
2 Language basics

- Core syntax and types
- Arrays and Pointers
- Scopes / namespaces
- Class and enum types
- References
- Functions
- Operators
- Control structures
- Headers and interfaces

- Auto keyword
Auto keyword

Reason of being

- many type declarations are redundant
- and lead to compiler errors if you mess up

```
std::vector<int> v;
int a = v[3];
int b = v.size();  // bug? unsigned to signed
```
### Reason of being

- many type declarations are redundant
- and lead to compiler errors if you mess up

```cpp
std::vector<int> v;
int a = v[3];
int b = v.size();  // bug? unsigned to signed
```

### Practical usage

```cpp
std::vector<int> v;
auto a = v[3];
const auto b = v.size();
int sum{0};
for (auto n : v) { sum += n; }
```
Exercise

Familiarise yourself with range-based for loops and references

- go to code/loopsRefsAuto
- Look at loopsRefsAuto.cpp
- Compile it (make) and run the program (.loopsRefsAuto)
- Work on the tasks that you find in loopsRefsAuto.cpp
Useful tools

1. History and goals
2. Language basics
3. Useful tools
   - C++ editor
4. Object orientation (OO)
5. Core modern C++

- Code management
- Code formatting
- The Compiling Chain
- Debugging
3 Useful tools

- C++ editor
- Code management
- Code formatting
- The Compiling Chain
- Debugging
### C++ editors and IDEs

Can dramatically improve your efficiency by
- coloring the code for you to “see” the structure
- helping with indenting and formatting properly
- allowing you to easily navigate in the source tree
- helping with compilation/debugging, profiling, static analysis
- showing you errors and suggestions while typing

- **Visual Studio** heavy, fully fledged IDE for Windows
- **Visual Studio Code** editor, open source, portable, many plugins
- **Eclipse** IDE, open source, portable
- **Emacs** **Vim** editors for experts, extremely powerful.
  They are to IDEs what latex is to PowerPoint
- **CLion, Code::Blocks, Atom, NetBeans, Sublime Text, ...**

Choosing one is mostly a matter of taste
Code management

3 Useful tools

- C++ editor
- Code management
- Code formatting
- The Compiling Chain
- Debugging
Code management tool

Please use one!
- even locally
- even on a single file
- even if you are the only committer

It will soon save your day

A few tools
- **git**  THE mainstream choice. Fast, light, easy to use
- **mercurial**  the alternative to git
- **Bazaar**  another alternative
- **svn**  historical, not distributed - DO NOT USE
- **CVS**  archeological, not distributed - DO NOT USE
# git init myProject
Initialized empty Git repository in myProject/.git/

# vim file.cpp; vim file2.cpp
# git add file.cpp file2.cpp
# git commit -m "Committing first 2 files"
[master (root-commit) c481716] Committing first 2 files
...

# git log --oneline
d725f2e Better STL test
f24a6ce Reworked examples + added stl one
bb54d15 implemented template part
...

# git diff f24a6ce bb54d15
3 Useful tools

- C++ editor
- Code management
- Code formatting
- The Compiling Chain
- Debugging
**.clang-format**

- File describing your formatting preferences
- Should be checked-in at the repository root (project wide)
- `clang-format --style=LLVM -dump-config > .clang-format`
- Adapt style options with help from: [https://clang.llvm.org/docs/ClangFormatStyleOptions.html](https://clang.llvm.org/docs/ClangFormatStyleOptions.html)

**Run clang-format**

- `clang-format --style=LLVM -i <file.cpp>`
- `clang-format -i <file.cpp>` (looks for .clang-format file)
- `git clang-format` (formats local changes)
- `git clang-format <ref>` (formats changes since git <ref>)
- Some editors/IDEs find a .clang-format file and adapt
Exercise Time

- go to any example
- format code with:
  `clang-format --style=GNU -i <file.cpp>`
- inspect changes, try `git diff`
- revert changes using `git checkout -- <file.cpp>`
- go to code directory and create a `.clang-format` file
  `clang-format -style=LLVM -dump-config > .clang-format`
- run `clang-format -i */*.cpp`
- revert changes using `git checkout .`
The Compiling Chain

3 Useful tools

- C++ editor
- Code management
- Code formatting
- The Compiling Chain
- Debugging
The compiling chain

**Source code**

- .cpp, .hpp

**Preprocessor**

- cpp, gcc -E

**Compiler**

- g++ -c, gcc -c

**Linker**

- ld, gcc, g++

**Binary**

- .so, exe

---

**The steps**

- **cpp** the preprocessor handles the # directives (macros, includes) creates “complete” source code (ie. translation unit)

- **g++** the compiler creates machine code from C++ code

- **ld** the linker links several binary files into libraries and executables
Compilers

Available tools

- gcc: the most common and most used 
  free and open source
- clang: drop-in replacement of gcc 
  slightly better error reporting 
  free and open source, based on LLVM
- icc: Intel’s compilers, proprietary but now free 
  optimized for Intel hardware 
  icc being replaced by icx, based on LLVM
- Visual C++ / MSVC: Microsoft’s C++ compiler on Windows

My preferred choice today

- gcc as the de facto standard in HEP
- clang in parallel to catch more bugs
### Useful compiler options (gcc/clang)

#### Get more warnings
- `-Wall` -Wextra get all warnings
- `-Werror` force yourself to look at warnings

#### Optimization
- `-g` add debug symbols
- `-Ox` $0 = \text{no opt.}, 1-2 = \text{opt.}, 3 = \text{highly opt. (maybe larger binary)}, g = \text{opt. for debugging}$

#### Compilation environment
- `-I <path>` where to find header files
- `-L <path>` where to find libraries
- `-l <name>` link with libname.so
- `-E` / `-c` stop after preprocessing / compilation
### Makefiles

#### Why to use them
- an organized way of describing building steps
- avoids a lot of typing

#### Several implementations
- raw Makefiles: suitable for small projects
- cmake: portable, the current best choice
- automake: GNU project solution

```makefile
test : test.cpp libpoly.so
  $(CXX) -Wall -Wextra -o $@ $^
libpoly.so: Polygons.cpp
  $(CXX) -Wall -Wextra -shared -fPIC -o $@ $^
clean:
  rm -f *o *so *~ test test.sol
```
CMake

- a cross-platform meta build system
- generates platform-specific build systems
- see also this basic and detailed talks

Example CMakeLists.txt

```markdown
cmake_minimum_required(VERSION 3.18)
project(hello CXX)

find_package(ZLIB REQUIRED) # for external libs

add_executable(hello main.cpp util.h util.cpp)
target_compile_features(hello PUBLIC cxx_std_17)
target_link_libraries(hello PUBLIC ZLIB::ZLIB)
```
Building a CMake-based project

Start in the directory with the top-level CMakeLists.txt:

1. mkdir build  # will contain all build-related files
2. cd build
3. cmake ..  # configures and generates a build system
4. cmake -DCMAKE_BUILD_TYPE=Release .. # pass arguments
5. ccmaker .  # change configuration using terminal GUI
6. cmake-gui .  # change configuration using Qt GUI
7. cmake --build . -j8  # build project with 8 jobs
8. cmake --build . --target hello  # build only hello
9. sudo cmake --install .  # install project into system
10. cd ..
11. rm -r build  # clean everything
Exercise Time

- go to code/functions
- preprocess functions.cpp (cpp or gcc -E -o output)
- compile functions.o and Structs.o (g++ -c -o output)
- use nm to check symbols in .o files
- look at the Makefile
- try make clean; make
- see linking stage of the final program using g++ -v
  - just add a -v in the Makefile command for functions target
  - run make clean; make
  - look at the collect 2 line, from the end up to “-o functions”
- see library dependencies with ‘ldd functions’
Useful tools

- C++ editor
- Code management
- Code formatting
- The Compiling Chain
- Debugging
Debugging

The problem

- everything compiles fine (no warning)
- but crashes at run time
- no error message, no clue
## Debugging

### The problem
- everything compiles fine (no warning)
- but crashes at run time
- no error message, no clue

### The solution: debuggers
- dedicated program able to stop execution at any time
- and show you where you are and what you have
Debugging

The problem
- everything compiles fine (no warning)
- but crashes at run time
- no error message, no clue

The solution: debuggers
- dedicated program able to stop execution at any time
- and show you where you are and what you have

Existing tools
- **gdb**: THE main player
- **lldb**: the debugger coming with clang, still young
- **idb**: the intel debugger, proprietary

They usually can be integrated into your IDE
gdb crash course

**start gdb**
- `gdb <program>`
- `gdb <program><core file>`
- `gdb --args <program><program arguments>`

**inspect state**
- `bt` prints a backtrace
- `print <var>` prints current content of the variable
- `list` show code around current point
- `up/down` go up or down in call stack

**breakpoints**
- `break <function>` puts a breakpoint on function entry
- `break <file>:<line>` puts a breakpoint on that line
Exercise Time

- go to code/debug
- compile, run, see the crash
- run it in gdb
- inspect backtrace, variables
- find problem and fix bug
- try stepping, breakpoints
- use -Wall -Wextra and see warning
Object orientation (OO)

1 History and goals
2 Language basics
3 Useful tools
4 Object orientation (OO)
   - Objects and Classes
5 Core modern C++

- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Type casting
- Operators
- Functors
Objects and Classes

4 Object orientation (OO)

- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Type casting
- Operators
- Functors
What are classes and objects

Classes (or “user-defined types”)

- C structs on steroids
  - with inheritance
  - with access control
  - including methods

Objects

- instances of classes

A class encapsulates state and behavior of “something”

- shows an interface
- provides its implementation
  - status, properties
  - possible interactions
  - construction and destruction
```cpp
struct MyFirstClass {
    int a;
    void squareA() {
        a *= a;
    }
    int sum(int b) {
        return a + b;
    }
};

MyFirstClass myObj;
myObj.a = 2;

// let's square a
myObj.squareA();
```
Separating the interface

Header : MyFirstClass.hpp

```cpp
#pragma once

struct MyFirstClass {
    int a;
    void squareA();
    int sum(int b);
};
```

Implementation : MyFirstClass.cpp

```cpp
#include "MyFirstClass.hpp"

void MyFirstClass::squareA() {
    a *= a;
}

void MyFirstClass::sum(int b) {
    return a + b;
}
```
Implementing methods

Standard practice
- usually in .cpp, outside of class declaration
- using the class name as namespace
- when reference to the object is needed, use this keyword

```cpp
void MyFirstClass::squareA() {
    a *= a;
}

int MyFirstClass::sum(int b) {
    return a + b;
}
```
this keyword

- This is a hidden parameter to all class methods.
- It points to the current object.
- So it is of type T* in the methods of class T.

```cpp
void ext_func(MyFirstClass& c) {
    ... do something with c ... 
}

int MyFirstClass::some_method(...) {
    ext_func(*this);
}
```
Method overloading

The rules in C++

- overloading is authorized and welcome
- signature is part of the method identity
- but not the return type

```
struct MyFirstClass {
    int a;
    int sum(int b);
    int sum(int b, int c);
};

int MyFirstClass::sum(int b) { return a + b; }

int MyFirstClass::sum(int b, int c) {
    return a + b + c;
}
```
Object orientation (OO)

- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Type casting
- Operators
- Functors
First inheritance

```cpp
struct MyFirstClass {
    int a;
    void squareA() { a *= a; }
};

struct MySecondClass : MyFirstClass {
    int b;
    int sum() { return a + b; }
};

MySecondClass myObj2;
myObj2.a = 2;
myObj2.b = 5;
myObj2.squareA();
int i = myObj2.sum(); // i = 9
```
First inheritance

```cpp
struct MyFirstClass {
    int a;
    void squareA() { a *= a; }
};

struct MySecondClass : MyFirstClass {
    int b;
    int sum() { return a + b; }
};

MySecondClass myObj2;
myObj2.a = 2;
myObj2.b = 5;
myObj2.squareA();
int i = myObj2.sum(); // i = 9
```
Managing access to class members

public / private keywords

- **private** allows access only within the class
- **public** allows access from anywhere

- The default for class is *private*
- A struct is just a class that defaults to *public* access

```cpp
class MyFirstClass {
    public:
    void setA(int x);
    int getA();
    void squareA();

    private:
    int a;
};
```

```cpp
MyFirstClass obj;
obj.a = 5;  // error!
obj.setA(5);  // ok
obj.squareA();
int b = obj.getA();
```

This breaks MySecondClass!
Managing access to class members

**public / private keywords**

- private allows access only within the class
- public allows access from anywhere

- The default for class is private
- A struct is just a class that defaults to public access

```cpp
class MyFirstClass {
  public:
    void setA(int x);
    int getA();
    void squareA();
  private:
    int a;
};
```

```cpp
MyFirstClass obj;
obj.a = 5; // error!
obj.setA(5); // ok
obj.squareA();
int b = obj.getA();
```
Managing access to class members

public / private keywords

- private allows access only within the class
- public allows access from anywhere

The default for class is private

A struct is just a class that defaults to public access

```cpp
class MyFirstClass {
public:
  void setA(int x);
  int getA();
  void squareA();
private:
  int a;
};

MyFirstClass obj;
obj.a = 5;  // error !
obj.setA(5);  // ok
obj.squareA();
int b = obj.getA();

This breaks MySecondClass !
```
Managing access to class members (2)

Solution is *protected* keyword

Gives access to classes inheriting from base class

```cpp
class MyFirstClass {
public:
    void setA(int a);
    int getA();
    void squareA();
protected:
    int a;
};

class MySecondClass : public MyFirstClass {
public:
    int sum() {
        return a + b;
    }
private:
    int b;
};
```
Inheritance can be public, protected or private

It influences the privacy of inherited members for external code. The code of the class itself is not affected

- **public**: privacy of inherited members remains unchanged
- **protected**: inherited public members are seen as protected
- **private**: all inherited members are seen as private
  - this is the default for class if nothing is specified
Managing inheritance privacy

Inheritance can be public, protected or private

It influences the privacy of inherited members for external code. The code of the class itself is not affected

- **public**: privacy of inherited members remains unchanged
- **protected**: inherited public members are seen as protected
- **private**: all inherited members are seen as private
  this is the default for classes if nothing is specified

Net result for external code

- only public members of public inheritance are accessible

Net result for grand child code

- only public and protected members of public and protected parents are accessible
Managing inheritance privacy - public

```cpp
void funcSecond() {
    int a = priv; // Error
    int b = prot; // OK
    int c = pub; // OK
}

void funcThird() {
    int a = priv; // Error
    int b = prot; // OK
    int c = pub; // OK
}

void extFunc(MyThirdClass t) {
    int a = t.priv; // Error
    int b = t.prot; // Error
    int c = t.pub; // OK
}
```

B. Gruber, S. Hageboeck, S. Ponce
Managing inheritance privacy - protected

MyFirstClass
- private:
  - int priv;
- protected:
  - int prot;
- public:
  - int pub;

MySecondClass
- protected:
  - int prot;

MyThirdClass
- public:
  - int pub;

```cpp
void funcSecond() {
    int a = priv;   // Error
    int b = prot;   // OK
    int c = pub;    // OK
}

void funcThird() {
    int a = priv;   // Error
    int b = prot;   // OK
    int c = pub;    // OK
}

void extFunc(MyThirdClass t) {
    int a = t.priv;  // Error
    int b = t.prot;  // Error
    int c = t.pub;   // Error
}
```

C++98
Managing inheritance privacy - private

MyFirstClass

private:
  int priv;
protected:
  int prot;
public:
  int pub;

MySecondClass

void funcSecond();

void funcThird();

MyThirdClass

void funcSecond() {
  int a = priv; // Error
  int b = prot; // OK
  int c = pub; // OK
}

void funcThird() {
  int a = priv; // Error
  int b = prot; // Error
  int c = pub; // Error
}

void extFunc(MyThirdClass t) {
  int a = t.priv; // Error
  int b = t.prot; // Error
  int c = t.pub; // Error
}

B. Gruber, S. Hageboeck, S. Ponce
Constructors/destructors

4. Object orientation (OO)
   - Objects and Classes
   - Inheritance
   - Constructors/destructors
   - Static members
   - Allocating objects
   - Advanced OO
   - Type casting
   - Operators
   - Functors
Class Constructors and Destructors

Concept
- special functions called when building/destroying an object
- a class can have several constructors, but only one destructor
- the constructors have the same name as the class
- same for the destructor with a leading ~

```cpp
class MyFirstClass {
  public:
    MyFirstClass();
    MyFirstClass(int a);
    ~MyFirstClass();
    ...

  protected:
    int a;
};

// note special notation for initialization of members
MyFirstClass() : a(0) {} // initialization of members
MyFirstClass(int a_):a(a_) {}
~MyFirstClass() {}
```
class Vector {
public:
    Vector(int n);
    ~Vector();
    void setN(int n, int value);
    int getN(int n);
private:
    int len;
    int* data;
};
Vector::Vector(int n) : len(n) {
    data = new int[n];
}
Vector::~Vector() {
    delete[] data;
}
Constructors and inheritance

struct MyFirstClass {
    int a;
    MyFirstClass();
    MyFirstClass(int a);
};

struct MySecondClass : MyFirstClass {
    int b;
    MySecondClass();
    MySecondClass(int b);
    MySecondClass(int a, int b);
};

MySecondClass::MySecondClass() : MyFirstClass(), b(0) {}
MySecondClass::MySecondClass(int b_) : MyFirstClass(), b(b_) {}
MySecondClass::MySecondClass(int a_, int b_) : MyFirstClass(a_), b(b_) {}
Copy constructor

Concept

- special constructor called for replicating an object
- takes a single parameter of type `const &` to class
- provided by the compiler if not declared by the user
- in order to forbid copy, use `= delete` (see next slides)
  - or private copy constructor with no implementation in C++98
Copy constructor

Concept
- special constructor called for replicating an object
- takes a single parameter of type `const &` to class
- provided by the compiler if not declared by the user
- in order to forbid copy, use `= delete` (see next slides)
- or private copy constructor with no implementation in C++98

```cpp
struct MySecondClass : MyFirstClass {
    MySecondClass();
    MySecondClass(const MySecondClass &other);
};
```
Copy constructor

Concept

• special constructor called for replicating an object
• takes a single parameter of type const & to class
• provided by the compiler if not declared by the user
• in order to forbid copy, use = delete (see next slides)
  • or private copy constructor with no implementation in C++98

```cpp
struct MySecondClass : MyFirstClass {
    MySecondClass();
    MySecondClass(const MySecondClass &other);
};
```

The rule of 3/5/0 (C++98/C++11 and newer) - cppreference

• if a class has a destructor, a copy/move constructor or a
  copy/move assignment operator, it should have all three/five.
  strive for having none.
class Vector {
    public:
    Vector(int n);
    Vector(const Vector &other);
    ~Vector();
    ... 
};
Vector::Vector(int n) : len(n) {
    data = new int[n];
}
Vector::Vector(const Vector &other) : len(other.len) {
    data = new int[len];
    std::copy(other.data, other.data + len, data);
}
Vector::~Vector() { delete[] data; }

Explicit unary constructor

Concept

- A constructor with a single non-default parameter can be used by the compiler for an implicit conversion.

```cpp
void print( const Vector & v )
    std::cout<<"printing v elements...\n";
}

int main {
    // calls Vector::Vector(int n) to construct a Vector
    // then calls print with that Vector
    print(3);
};
```
Explicit unary constructor

Concept

- The keyword `explicit` forbids such implicit conversions.
- It is recommended to use it systematically, except in special cases.

```cpp
class Vector {
public:
    explicit Vector(int n);
    Vector(const Vector &other);
    ~Vector();
    ...};
```
Defaulted Constructor

Idea
- avoid empty default constructors like `ClassName() {}`
- declare them as `= default`

Details
- when no user defined constructor, a default is provided
- any user-defined constructor disables the default one
- but they can be enforced
- rule can be more subtle depending on members

Practically
1. `ClassName() = default;`  // provide/force default
2. `ClassName() = delete;`  // do not provide default
Delegating constructor

Idea

- avoid replication of code in several constructors
- by delegating to another constructor, in the initializer list

Practically

```cpp
struct Delegate {
    int m_i;
    Delegate() { ... complex initialization ... }
    Delegate(int i) : Delegate(), m_i(i) {}
};
```
Constructor inheritance

Idea

- avoid having to re-declare parent’s constructors
- by stating that we inherit all parent constructors

Practically

```cpp
struct BaseClass {
    BaseClass(int value);
};

struct DerivedClass : BaseClass {
    using BaseClass::BaseClass;
};

DerivedClass a{5};
```
Member initialization

Idea

- avoid redefining same default value for members n times
- by defining it once at member declaration time

Practically

```c++
struct BaseClass {
    int a{5}; // also possible: int a = 5;
    BaseClass() = default;
    BaseClass(int _a) : a(_a) {}
};

struct DerivedClass : BaseClass {
    int b{6};
    using BaseClass::BaseClass;
};

DerivedClass d{7}; // a = 7, b = 6
```
Calling constructors

After object declaration, arguments within {}:

```cpp
struct A {
    int a;
    float b;
    A();
    A(int);
    A(int, int);
};

struct B {
    int a;
    float b;
    // no constructor declared!
};

A a{1,2};    // A::A(int, int)
A a{1};      // A::A(int)
A a{};       // A::A()
A a;         // A::A()
A a = {1,2}; // A::A(int, int)
B b = {1, 2.3}; // aggregate initialization
```
Calling constructors the old way

Arguments are given within (), aka C++98 nightmare

```
struct A {
    int a;
    float b;
    A();
    A(int);
    A(int, int);
};

struct B {
    int a;
    float b;
    // no constructor declared!
};

A a(1, 2);  // A::A(int, int)
A a(1);     // A::A(int)
A a();      // declaration of a function!
A a;        // A::A()
A a = {1, 2}; // not allowed
B b = {1, 2.3}; // OK
```
### Calling constructors for arrays and vectors

#### C++11

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td><code>int ip[3]{1,2,3};</code></td>
</tr>
<tr>
<td>11</td>
<td><code>int* ip = new int[3]{1,2,3};</code></td>
</tr>
<tr>
<td>12</td>
<td><code>std::vector&lt;int&gt; v{1,2,3};</code></td>
</tr>
</tbody>
</table>
Calling constructors for arrays and vectors

### C++11

```cpp
type ip[3]{1,2,3};
int* ip = new int[3]{1,2,3};
std::vector<int> v{1,2,3};
```

### C++98 nightmare

```cpp
int ip[3]{1,2,3};     // OK
int* ip = new int[3]{1,2,3}; // not allowed
std::vector<int> v{1,2,3}; // not allowed
```
4. **Object orientation (OO)**
   - Objects and Classes
   - Inheritance
   - Constructors/destructors
   - **Static members**
     - Allocating objects
     - Advanced OO
     - Type casting
     - Operators
     - Functors
Static members

**Concept**

- members attached to a class rather than to an object
- usable with or without an instance of the class
- identified by the `static` keyword

```cpp
class Text {
public:
    static std::string upper(std::string) {...}
private:
    static int s_nbCallsToUpper; // add `inline` in C++17
};
```

```cpp
int Text::s_nbCallsToUpper = 0; // required before C++17
std::string uppers = Text::upper("my text");
// now Text::s_nbCallsToUpper is 1
```
Object orientation (OO)

- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members

Allocating objects
- Advanced OO
- Type casting
- Operators
- Functors
Process memory organization

4 main areas

- **the code segment** for the machine code of the executable
- **the data segment** for global variables
- **the heap** for dynamically allocated variables
- **the stack** for parameters of functions and local variables

Memory layout:

```
      ↓   ...   ↓
        ↓

      ↑   ...   ↑

      ↓
      ↓

stack

      ↓
      ↓

data segment

      ↓
      ↓

code segment
```
Main characteristics

- Allocation on the stack stays valid for the duration of the current scope. It is destroyed when it is popped off the stack.

- Memory allocated on the stack is known at compile time and can thus be accessed through a variable.

- The stack is relatively small, it is not a good idea to allocate large arrays, structures or classes.

- Each thread in a process has its own stack
  - Allocations on the stack are thus “thread private”
  - And do not introduce any thread safety issues
Object allocation on the stack

On the stack

- objects are created when declared (constructor called)
- objects are destructed when out of scope (destructor is called)

```cpp
int f() {
    MyFirstClass a{3}; // constructor called
    ...
}
// destructor called

{
    MyFirstClass a; // default constructor called
    ...
}
// destructor called
```
Main characteristics

- Allocated memory stays allocated until it is specifically deallocated
  - beware memory leaks
- Dynamically allocated memory must be accessed through pointers
- large arrays, structures, or classes should be allocated here
- there is a single, shared heap per process
  - allows to share data between threads
  - introduces race conditions and thread safety issues!
Object allocation on the heap

On the heap

- objects are created by calling `new` (constructor is called)
- objects are destructed by calling `delete` (destructor is called)

```cpp
1 {
  // default constructor called
  MyFirstClass *a = new MyFirstClass;
  ...
  delete a; // destructor is called
}

int f() {
  // constructor called
  // memory leak !!!
  MyFirstClass *a = new MyFirstClass(3);
  ...
} // memory leak !!!
```
Arrays on the heap

- arrays of objects are created by calling `new []`
default constructor is called for each object of the array
- arrays of object are destructed by calling `delete []`
destructor is called for each object of the array

```cpp
1  {
  // default constructor called 10 times
  MyFirstClass *a = new MyFirstClass[10];
  ...
  delete[] a; // destructor called 10 times
}
```
4 Object orientation (OO)

- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Type casting
- Operators
- Functors
**Polymorphism**

*the concept*

- objects actually have multiple types simultaneously
- and can be used as any of them

```cpp
Polygon p;

int f(Drawable & d) {...}
f(p);  //ok

try {
    throw p;
} catch (Shape & e) {
    // will be caught
}
```
**Polymorphism**

The concept:
- Objects actually have multiple types simultaneously.
- And can be used as any of them.

```cpp
Polygon p;

int f(Drawable & d) {...}
f(p);  // ok

try {
    throw p;
} catch (Shape & e) {
    // will be caught
}
```
Polymorphism

The concept

- Objects actually have multiple types simultaneously
- And can be used as any of them

```cpp
Polygon p;

int f(Drawable & d) {...}
f(p); // ok

try {
    throw p;
} catch (Shape & e) {
    // will be caught
}
```

Memory layout:

```
+---------------------+------+
| Drawable.a           | 0x3000 |
| Drawable.b           | 0x3004 |
| Shape.a              | 0x300C |
| Shape.b              | 0x3010 |
| Polygon.nLines       | 0x3018 |
| ...                  | 0x3014 |
```

B. Gruber, S. Hageboeck, S. Ponce
Polymorphism

the concept

- objects actually have multiple types simultaneously
- and can be used as any of them

```
1 Polygon p;
2
3 int f(Drawable & d) {...}
4 f(p);    //ok
5
6 try {
7    throw p;
8 } catch (Shape & e) {
9    // will be caught
10 }
```
Inheritance privacy and polymorphism

Only public inheritance is visible to code outside the class

- private and protected are not
- this may restrict usage of polymorphism

```cpp
Polygon p;

int f(Drawable & d) {...}
f(p); // Not ok anymore

try {
    throw p;
} catch (Shape & e) {
    // ok, will be caught
}
```
Method overriding

The problem

- a given method of the parent can be overridden in a child
- but which one is called?

```
1  Polygon p;
2  p.draw(); // ?
3
4  Shape & s = p;
5  s.draw(); // ?
```
Virtual methods

**the concept**
- methods can be declared `virtual`
- for these, the most derived object is always considered
- for others, the type of the variable decides
Virtual methods

The concept

- Methods can be declared `virtual`.
- For these, the most derived object is always considered.
- For others, the type of the variable decides.

```cpp
1 Polygon p;
2 p.draw(); // Polygon.draw
3
4 Shape & s = p;
5 s.draw(); // Drawable.draw
```

```
Drawable
void draw();

Shape

Polygon
void draw();
```
Virtual methods

the concept

- methods can be declared `virtual`
- for these, the most derived object is always considered
- for others, the type of the variable decides

```cpp
define Polygon p;
define p.draw(); // Polygon.draw

Shape & s = p;
define s.draw(); // Polygon.draw
```

**Diagram:**
```
Drawable
virtual void draw();
```
```
Shape
```
```
Polygon
void draw();
```
Virtual methods - implications

**Mechanics**
- Virtual methods are dispatched at run time
  - While non-virtual methods are bound at compile time
- They also imply extra storage and an extra indirection
  - Practically, the object stores a pointer to the correct method
  - In a so-called "virtual table" ("vtable")

**Consequences**
- Virtual methods are "slower" than standard ones
- And they can rarely be inlined
- Templates are an alternative for performance-critical cases
**Principle**

- when overriding a virtual method
- the `override` keyword should be used
- the `virtual` keyword is then optional

**Practically**

```cpp
struct Base {
    virtual void some_func(float);
};

struct Derived : Base {
    void some_func(float) override;
};
```
Why was override keyword introduced?

To detect the mistake in the following code:

```cpp
// Without override (C++98)
struct Base {
    virtual void some_func(float);
};
struct Derived : Base {
    void some_func(double); // oops!
};
```

- with `override`, you would get a compiler error
- if you forget `override` when you should have it, you get a compiler warning
Pure Virtual methods

Concept

- unimplemented methods that must be overridden
- marked by = 0 in the declaration
- makes their class abstract
- only non-abstract classes can be instantiated

```
// Error: abstract class
Shape s;

// ok, draw has been implemented
Polygon p;

// Shape type still usable
Shape & s = p;
s.draw();
```
**Pure Virtual methods**

**Concept**
- unimplemented methods that must be overridden
- marked by \( = 0 \) in the declaration
- makes their class abstract
- only non-abstract classes can be instantiated

```cpp
// Error: abstract class
Shape s;

// ok, draw has been implemented
Polygon p;

// Shape type still usable
Shape & s = p;
s.draw();
```

Diagram:
```
Drawable
|-- virtual void draw() = 0;

Shape
|-- Polygon

void draw() override;
```
**Pure Abstract Class aka Interface**

**Definition of pure abstract class**

- a class that has
  - no data members
  - all its methods pure virtual
  - a `virtual` destructor
- the equivalent of an Interface in Java

```cpp
struct Drawable {
    ~Drawable() = default;
    virtual void draw() = 0;
};
```
Overriding overloaded methods

Concept

- overriding an overloaded method will hide the others
- unless you inherit them using `using`

```c++
struct BaseClass {
    int foo(std::string);
    int foo(int);
};

struct DerivedClass : BaseClass {
    using BaseClass::foo;
    int foo(std::string);
};

DerivedClass dc;
dc.foo(4); // error if no using
```
Polymorphism

Exercise Time

- go to code/polymorphism
- look at the code
- open trypoly.cpp
- create a Pentagon, call its perimeter method
- create a Hexagon, call its perimeter method
- create a Hexagon, call its parent’s perimeter method
- retry with virtual methods

B. Gruber, S. Hageboeck, S. Ponce

HEP C++ course
**Concept**

- One class can inherit from multiple parents

```cpp
class TextBox :
public Rectangle, Text {
    // Inherits from both
    // Publicly from Rectangle
    // Privately from Text
}
```
The diamond shape

**Definition**
- situation when one class inherits several times from a given grand parent

**Problem**
- are the members of the grand parent replicated?
Virtual inheritance

Solution

- inheritance can be *virtual* or not
- *virtual* inheritance will “share” parents
- standard inheritance will replicate them

```cpp
1 class Text : public virtual Drawable {...};
2 class Rectangle : public virtual Drawable {...};
```
Multiple inheritance advice

Do not use multiple inheritance

- Except for inheriting from interfaces
- and for rare special cases
Multiple inheritance advice

Do not use multiple inheritance
- Except for inheriting from interfaces
- and for rare special cases

Do not use diamond shapes
- This is a sign that your architecture is not correct
- In case you are tempted, think twice and change your mind
Virtual inheritance

Exercise Time

- go to code/virtual_inheritance
- look at the code
- open trymultiherit.cpp
- create a TextBox and call draw
- Fix the code to call both draws by using types
- retry with virtual inheritance
Virtual inheritance

Good practice
if you write a class and expect users to inherit from it, declare its destructor `virtual`

Warning
in case of virtual inheritance it is the most derived class that calls the virtual base class’s constructor
Object orientation (OO)

- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Type casting
- Operators
- Functors
Type casting

5 types of casts in C++

- `static_cast<Target>(arg)`: Convert type if the static types allow it
- `dynamic_cast<Target>(arg)`: Check if object at address of "arg" is compatible with the type Target. Throw `std::bad_cast` if it’s not.

```cpp
struct A{ virtual ~A(){} } a;
struct B : A {} b;

A& c = static_cast<A&>(b); // OK. b is also an A
B& d = static_cast<B&>(a); // UB: a is not a B
B& e = static_cast<B&>(c); // OK. c is a B
B& f = dynamic_cast<B&>(c); // OK, c is a B
B& g = dynamic_cast<B&>(a); // Exception: not a B
```
Type casting

5 types of casts in C++

- `static_cast<Target>(arg)`: Convert type if the static types allow it.
- `dynamic_cast<Target>(arg)`: Check if object at address of “arg” is compatible with the type Target. Return `nullptr` if it’s not.

```c++
1 B* d = dynamic_cast<B*>(&a); // nullptr. a not a B.
2 if (d != nullptr) {
3   // Will not reach this
4 }
5
6 if (auto bPtr = dynamic_cast<B*>(&c)) {
7   // OK, we will get here
8 }
```
Type casting

5 types of casts in C++

- **const_cast**: Remove constness from a type. If you think you need this, first try to improve the design!

- **reinterpret_cast<Target>(arg)**: Change type irrespective of what ‘arg’ is. *Almost never* a good idea!

- C-style: `(Target)arg`: Force-change type in C-style. No checks. Don’t use this.

Casts to avoid

```c
void func(A const & a) {
    A& ra = a; // Error: not const
    A& ra = const_cast<A&>(a); // Compiles. Bad design!
    // Evil! Don't do this:
    B* b = reinterpret_cast<B*>(&a);
    B* b = (B*)&a;
}
```
4 Object orientation (OO)
- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Type casting
- Operators
- Functors
struct Complex {
    float m_real, m_imaginary;
    Complex(float real, float imaginary);
    Complex operator+(const Complex& other) {
        return Complex(m_real + other.m_real,
                      m_imaginary + other.m_imaginary);
    }
};

Complex c1{2, 3}, c2{4, 5};
Complex c3 = c1 + c2; // (6, 8)
Defining operators of a class

- implemented as a regular method
  - either inside the class, as a member function
  - or outside the class (not all)
- with a special name (replace @ by anything)

<table>
<thead>
<tr>
<th>Expression</th>
<th>As member</th>
<th>As non-member</th>
</tr>
</thead>
<tbody>
<tr>
<td>@a</td>
<td>(a).operator@()</td>
<td>operator@(@a)</td>
</tr>
<tr>
<td>a@b</td>
<td>(a).operator@(@b)</td>
<td>operator@(@a,@b)</td>
</tr>
<tr>
<td>a=b</td>
<td>(a).operator=(@b)</td>
<td>cannot be non-member</td>
</tr>
<tr>
<td>a(b...)</td>
<td>(a).operator()(b...)</td>
<td>cannot be non-member</td>
</tr>
<tr>
<td>a[b]</td>
<td>(a).operator<a href="@b"></a></td>
<td>cannot be non-member</td>
</tr>
<tr>
<td>a-&gt;</td>
<td>(a).operator-&gt;(@)</td>
<td>cannot be non-member</td>
</tr>
<tr>
<td>a@</td>
<td>(a).operator@(@0)</td>
<td>operator@(@a,0)</td>
</tr>
</tbody>
</table>
Why to have non-member operators?

Symmetry

```cpp
struct Complex {
    float m_real, m_imaginary;
    Complex operator+(float other) {
        return Complex(m_real + other, m_imaginary);
    }
};

Complex c1{2.f, 3.f};
Complex c2 = c1 + 4.f;  // ok
Complex c3 = 4.f + c1;  // not ok !!
Why to have non-member operators?

Symmetry

```cpp
struct Complex {
    float m_real, m_imaginary;
    Complex operator+(float other) {
        return Complex(m_real + other, m_imaginary);
    }
};
Complex c1{2.f, 3.f};
Complex c2 = c1 + 4.f; // ok
Complex c3 = 4.f + c1; // not ok !!
Complex operator+(float a, const Complex& obj) {
    return Complex(a + obj.m_real, obj.m_imaginary);
}
```
Other reason to have non-member operators?

Extending existing classes

```cpp
struct Complex {
    float m_real, m_imaginary;
    Complex(float real, float imaginary);
};

std::ostream& operator<<(std::ostream& os, const Complex& obj) {
    os << "\(" << obj.m_real << ", \)" << obj.m_imaginary << "\)";
    return os;
}
Complex c1{2.f, 3.f};
std::cout << c1 << std::endl;  // Prints '(2, 3)'
```
Exercise

Write a simple class representing a fraction and pass all tests

- go to code/operators
- look at operators.cpp
- inspect main and complete the implementation of class Fraction step by step
- you can comment out parts of main to test in between
4 Object orientation (OO)

- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Type casting
- Operators
- Functors
Concept

- a class that implements `operator()`
- allows to use objects in place of functions
- and as objects have constructors, allow to construct functions

```cpp
struct Adder {
    int m_increment;
    Adder(int increment) : m_increment(increment) {}
    int operator()(int a) { return a + m_increment; }
};

Adder inc1{1}, inc10{10};
int i = 3;
int j = inc1(i);  // 4
int k = inc10(i); // 13
int l = Adder{25}(i); // 28
```
**Functors**

### Typical usage

- pass a function to another one
- or to an STL algorithm

```c++
struct BinaryFunction {
  virtual double operator()(double a, double b) = 0;
};
double binary_op(double a, double b, BinaryFunction &func) {
  return func(a, b);
}
struct Add : BinaryFunction {
  double operator()(double a, double b) override {
    return a+b;
  }
};
Add addfunc;
double c = binary_op(a, b, addfunc);
```
5 Core modern C++

- Constness
- Exceptions
- Templates
- The STL
- Lambdas
- pointers and RAII
## Constness

The `const` keyword

- indicate that the element to the left is constant
- this element won’t be modifiable in the future
- this is all checked at compile time

```cpp
// standard syntax
int const i = 6;

// error : i is constant
i = 5;

// also ok, when nothing on the left,
// const applies to the element on the right
const int j = 6;
```
// pointer to a constant integer
int a = 1, b = 2;
int const *i = &a;
i = &b; // ok, pointer is not const

// constant pointer to an integer
int * const j = &a;
i = &b; // ok, value can be changed
j = &b; // error, pointer is const

// constant pointer to a constant integer
int const * const k = &a;
i = &b; // error, pointer is const

// const reference
int const & l = a;
l = b; // error, reference is const

int const & const l = a; // compile error
The `const` keyword for member functions

- indicate that the function does not modify the object
- in other words, `this` is a pointer to a constant object

```cpp
struct Example {
    void foo() const {
        // type of 'this' is 'Example const*'
        m_member = 0; // Error: member function is const
    }
    int m_member;
};
```
Constness is part of the type

- T const and T are different types
- however, T is automatically cast to T const when needed

```cpp
1 void func(int & a);
2 void funcConst(int const & a);
3
4 int a = 0;
5 int const b = 0;
6
7 func(a); // ok
8 func(b); // error
9 funcConst(a); // ok
10 funcConst(b); // ok
```
**Exercise Time**

- go to code/constness
- open constplay.cpp
- try to find out which lines won’t compile
- check your guesses by compiling for real
5 Core modern C++
- Constness
- Exceptions
- Templates
- The STL
- Lambdas
- pointers and RAII
Exceptions

The concept

• to handle *exceptional* events that happen rarely
• and cleanly jump to a place where the error can be handled

In practice

• add an exception handling block with **try ... catch**
  • when exceptions are possible *and can be handled*
• throw an exception using **throw**
  • when a function cannot proceed or recover internally

```
#include <stdexcept>
...
try {
  process_stream_data(s);
} catch (const range_error& e) {
  cerr << e.what() << endl;
}
```

```
void process_stream_data(stream &s) {
  ...
  if (data_location >= buffer.length()) {
    throw range_error("buf overflow");
  }
  ...
}
```
Exceptions

Rules and behavior

- objects of any type can be thrown
  - prefer standard exception types from the `<stdexcept>` header
  - define your own subclass of `std::exception` if needed
- an exception will be caught if the type in the catch clause matches or is a base class of the thrown object’s static type
  - if no one catches an exception then `std::terminate` is called
- you can have multiple catch clauses, will be matched in order
- all objects on the stack between the `throw` and the `catch` are destructed automatically during stack unwinding
  - this should cleanly release intermediate resources
  - make sure you are using the RAII idiom for your own classes
Advice

- throw exceptions by value, catch them by (const) reference
- use exceptions for *unlikely* runtime errors outside the program’s control
  - bad inputs, files unexpectedly not found, DB connection, . . .
- *don’t* use exceptions for logic errors in your code
  - consider assert and tests
- *don’t* use exceptions to provide alternative return values (or to skip them)
  - you can use std::optional or std::variant
  - avoid using the global C-style errno
- See also the C++ core guidelines and the ISO C++ FAQ
A more illustrative example

- exceptions are very powerful when there is much code between the error and where the error is handled
- they can also rather cleanly handle different types of errors
- **try/catch** statements can also be nested

```cpp
void process_file(File const & file) {
    if (handle = open_file(file)) {
        throw bad_file(file.status());
    }
    while (!handle) {
        line = read_line(handle);
        database.insert(line); /* can throw */
    }
}
```

```cpp
try {
    for (File const & f : files) {
        try {
            process_file(f);
        }
        catch (bad_file const & e) {
            ... // loop continues
        }
    }
    catch (bad_db const & e) {
        ... // loop aborted
    }
} catch (bad_db const & e) {
    ... // loop aborted
}
```
Exceptions

Catching everything

- sometimes we need to catch all possible exceptions
- e.g. in main, a thread, a destructor, interfacing with C, ...

```cpp
try {
    callUnknownFramework();
} catch(const std::exception& e) {
    // catches std::exception and all derived types
    std::cerr << "Exception: " << e.what() << std::endl;
} catch(...) {
    // catches everything else
    std::cerr << "Unknown exception type" << std::endl;
}
```
exceptions have little cost if no exception is thrown
- they are recommended to report *exceptional* errors
- for performance, when error raising and handling are close, or errors occur often, prefer error codes or a dedicated class
- when in doubt about which error strategy is better, profile!

```cpp
Avoid
for (string const &num: nums) {
    try {
        int i = convert(num); // can throw
        process(i);
    } catch (not_an_int const &e) {
        ... // log and continue
    }
}

Prefer
for (string const &num: nums) {
    optional<int> i = convert(num);
    if (i) {
        process(*i);
    } else {
        ... // log and continue
    }
}
```
a function with the **noexcept** specifier states that it guarantees to not throw an exception

```cpp
int f() noexcept;
```

- either no exceptions will be thrown or they are handled internally
- checked at compile time, so it allows the compiler to optimise around that knowledge

a function with `noexcept(expression)` is only **noexcept** when expression evaluates to `true` at compile-time

```cpp
int safe_if_8B_long() noexcept(sizeof(long)==8);
```

- Use **noexcept** on leaf functions where you know the behaviour
- C++11 destructors are **noexcept** - never throw from them
**noexcept operator**

- The `noexcept(expression)` operator checks at compile-time whether an expression can throw exceptions.
- It returns a `bool`, which is `true` if no exceptions can be thrown.

```cpp
constexpr bool callCannotThrow = noexcept(f());
if constexpr (callCannotThrow) { ... }
```

```cpp
template <typename Function>
void g(Function f) noexcept(noexcept(f())) {
  ...
  f();
}
```
Core modern C++

- Constness
- Exceptions
- Templates
- The STL
- Lambdas
- Pointers and RAII
Templates

Concept

- The C++ way to write reusable code
  - like macros, but fully integrated into the type system
- Applicable to functions, classes and variables

```cpp
template<typename T>
const T & max(const T &a, const T &b) {
    return a > b ? a : b;
}

template<typename T>
struct Vector {
    int m_len;
    T* m_data;
};

template <typename T>
std::size_t size = sizeof(T);
```
Templates

Warning
- They are compiled for each instantiation
- They need to be defined before used
  - So all templated code has to be in headers
- This may lead to longer compilation times and bigger libraries

```cpp
1 template<typename T>
2 T func(T a) {
3     return a;
4 }
```

```cpp
int func(int a) {
    return a;
}
```

```cpp
double func(double a) {
    return a;
}
```
**Templates**

**Template parameters**
- can be types, values or other templates
- you can have several
- default values allowed starting at the last parameter

```cpp
1 template<typename KeyType=int, typename ValueType=KeyType>
2 struct Map {
3     void set(const KeyType &key, ValueType value);
4     ValueType get(const KeyType &key);
5 }
6
7 Map<std::string, int> m1;
8 Map<float> m2; // Map<float, float>
9 Map<> m3;      // Map<int, int>
```
Templates implementation

```cpp
template<typename KeyType=int, typename ValueType=KeyType>
struct Map {
    void set(const KeyType &key, ValueType value);
    ValueType get(const KeyType &key);
};

template<typename KeyType, typename ValueType>
void Map<KeyType, ValueType>::set
    (const KeyType &key, ValueType value) {
    ...
}

template<typename KeyType, typename ValueType>
ValueType Map<KeyType, ValueType>::get
    (const KeyType &key) {
    ...
}
```

B. Gruber, S. Hageboeck, S. Ponce

HEP C++ course
template parameters can also be values

- integral types, pointer, enums in C++98
- `auto` in C++17
- floats and literal types in C++20

```cpp
template<unsigned int N>
struct Polygon {
    Polygon(float radius);
    float perimeter() {return 2*N*sin(PI/N)*m_radius;}
    float m_radius;
};
```
Specialization

templates can be specialized for given values of their parameter

```cpp
// Specialization of Polygon for N=6

template<typename F>
struct Polygon<F, 6> {
    Polygon(F radius) : m_radius(radius) {}
    F perimeter() {return 6*m_radius;}
    F m_radius;
};
```
The full power of templates

Exercise Time

- go to code/templates
- look at the OrderedVector code
- compile and run playwithsort.cpp. See the ordering
- modify playwithsort.cpp and reuse OrderedVector with Complex
- improve OrderedVector to template the ordering
- test reverse ordering of strings (from the last letter)
- test order based on Manhattan distance with complex type
- check the implementation of Complex
- try ordering complex of complex
Core modern C++
- Constness
- Exceptions
- Templates
- The STL
- Lambdas
- pointers and RAII
What it is

- A library of standard templates
- Has almost everything you need
  - strings, containers, iterators
  - algorithms, functions, sorters
  - functors, allocators
  - ...
- Portable
- Reusable
- Efficient
The Standard Template Library

What it is

- A library of standard templates
- Has almost everything you need
  - strings, containers, iterators
  - algorithms, functions, sorters
  - functors, allocators
  - ...
- Portable
- Reusable
- Efficient

Use it

and adapt it to your needs, thanks to templates
```cpp
#include <vector>
#include <algorithm>

std::vector<int> vi{5, 3, 4}; // initializer list
std::vector<int> vr(3); // constructor taking int

std::transform(vi.begin(), vi.end(), // range1
               vi.begin(), // start range2
               vr.begin(), // start result
               std::multiplies{}); // function objects

for(auto n : vr) {
    std::cout << n << ' ';
}
```
STL’s concepts

containers

- data structures for managing a range of elements
- irrespective of
  - the data itself (templated)
  - the memory allocation of the structure (templated)
  - the algorithms that may use the structure
- examples
  - string, string_view (C++17)
  - list, forward_list (C++11), vector, deque, array (C++11)
  - map, set, multimap, multiset
  - unordered_map (C++11), unordered_set (C++11)
  - stack, queue, priority_queue
  - span (C++20)
- non-containers: pair, tuple (C++11), optional (C++17), variant (C++17), any (C++17)
- see also the string and container library on cppreference
Containers: std::vector

```cpp
#include <vector>

std::vector<T> v{5, 3, 4}; // 3 Ts, 5, 3, 4
std::vector<T> v(100);    // 100 default constr. Ts
std::vector<T> v(100, 42); // 100 Ts with value 42
std::vector<T> v2 = v;    // copy
std::vector<T> v2 = std::move(v); // move, v is empty

std::size_t s = v.size();
bool empty = v.empty();

v[2] = 17;   // write element 2
T& t = v[1000]; // access element 1000, bug!
T& t = v.at(1000); // throws std::out_of_range
T& f = v.front();  // access first element
v.back() = 0;    // write to last element
T* v.data();    // pointer to underlying storage
```
Containers: std::vector

```cpp
std::vector<T> v = ...;
auto b = v.begin(); // iterator to first element
auto e = v.end();   // iterator to one past last element
// all following operations, except reserve, invalidate
// all iterators (b and e) and references to elements

v.resize(100);    // size changes, grows: new T{}s appended
    // shrinks: Ts at end destroyed
v.reserve(1000);  // size remains, memory increased
for (T i = 0; i < 900; i++)
    v.push_back(i); // add to the end
v.insert(v.begin()+3, T{}); // insert after 3rd position
v.pop_back();       // removes last element
v.erase(v.end() - 3); // removes 3rd-last element
v.clear();          // removes all elements
```
Stl’s concepts

Iterators

- Generalization of pointers
- Allow iteration over some data
- Irrespective of
  - The container used (templated)
  - The data itself (container is templated)
  - The consumer of the data (templated algorithm)

Examples
- Iterator
- Reverse_iterator
- Const_iterator
STL’s concepts

**algorithms**

- implementation of an algorithm working on data
- with a well defined behavior (defined complexity)
- irrespective of:
  - the data handled
  - the container where the data live
  - the iterator used to go through data (almost)
- examples
  - for_each, find, find_if, count, count_if, search
  - copy, swap, transform, replace, fill, generate
  - remove, remove_if
  - unique, reverse, rotate, shuffle, partition
  - sort, partial_sort, merge, make_heap, min, max
  - lexicographical_compare, iota, reduce, partial_sum
- see also 105 STL Algorithms in Less Than an Hour and the algorithms library on cppreference
functors / function objects

- generic utility functions
- as structs with `operator()`
- mostly useful to be passed to STL algorithms
- implemented independently of
  - the data handled (templated)
  - the context (algorithm) calling it
- examples
  - plus, minus, multiplies, divides, modulus, negate
  - equal_to, less, greater, less_equal, ...
  - logical_and, logical_or, logical_not
  - bit_and, bit_or, bit_xor, bit_not
  - identity, not_fn
  - bind, bind_front
- see also documentation on [cppreference](http://cppreference.com)
Functors / function objects

Example

```cpp
struct Incrementer {
    int m_inc;
    Incrementer(int inc) : m_inc(inc) {}

    int operator()(int value) const {
        return value + m_inc;
    }
};

std::vector<int> v;
v.push_back(5); v.push_back(3); ...
std::transform(v.begin(), v.end(), v.begin(),
               Incrementer{42});
```
#include <vector>

#include <algorithm>

std::vector<int> vi{5, 3, 4}; // initializer list
std::vector<int> vr(3); // constructor taking int

std::transform(vi.begin(), vi.end(), vi.begin(), // range1
                vr.begin(), // start range2
                vr.begin(), // start result
                std::multiplies{}); // function objects

for(auto n : vr) {
    std::cout << n << ' ';
}

Range-based for loops with STL containers

Iterator-based loop (since C++98)

```
std::vector<int> v = ...;
int sum = 0;
for (std::vector<int>::iterator it = v.begin(); it != v.end(); it++)
    sum += *it;
```
Range-based for loops with STL containers

### Iterator-based loop (since C++98)

```cpp
std::vector<int> v = ...;
int sum = 0;
for (std::vector<int>::iterator it = v.begin(); it != v.end(); it++)
    sum += *it;
```

### Range-based for loop (since C++11)

```cpp
std::vector<int> v = ...;
int sum = 0;
for (auto a : v) { sum += a; }
```
Range-based for loops with STL containers

### Iterator-based loop (since C++98)

```cpp
std::vector<int> v = ...;
int sum = 0;
for (std::vector<int>::iterator it = v.begin(); it != v.end(); it++)
    sum += *it;
```

### Range-based for loop (since C++11)

```cpp
std::vector<int> v = ...;
int sum = 0;
for (auto a : v) { sum += a; }
```

### STL way (since C++98)

```cpp
std::vector<int> v = ...;
int sum = std::accumulate(v.begin(), v.end(), 0);
// std::reduce(v.begin(), v.end(), 0); // C++17
```
STL and functors

```cpp
// Finds the first element in a list between 1 and 10.
list<int> l = ...;
...

list<int>::iterator it =
    find_if(l.begin(), l.end(),
        compose2(logical_and<bool>(),
            bind2nd(greater_equal<int>(), 1),
            bind2nd(less_equal<int>(), 10)));

// Computes \( \sin(x)/(x + \text{DBL\_MIN}) \) for elements of a range.
transform(first, last, first,
        compose2(divides<double>(), // non-standard
            ptr_fun(sin),
            bind2nd(plus<double>(), DBL_MIN)));
```

Deprecation warning

Binders and function adaptors were removed in C++17 or C++20

B. Gruber, S. Hageboeck, S. Ponce
// Finds the first element in a list between 1 and 10.
std::list<int> l = ...;
...
const auto it =
    std::find_if(l.begin(), l.end(),
        [](int i) { return i >= 1 && i <= 10; });

// Computes sin(x)/(x + DBL_MIN) for elements of a range.
std::transform(first, last, first,
    [](auto x) { return sin(x)/(x + DBL_MIN); });
Welcome to lego programming!

B. Gruber, S. Hageboeck, S. Ponce
Using the STL

Exercise Time

- go to code/stl
- look at the non STL code in randomize.nostl.cpp
  - it creates a vector of ints at regular intervals
  - it randomizes them
  - it computes differences between consecutive ints
  - and the mean and variance of it
- open randomize.cpp and complete the “translation” to STL
- see how easy it is to reuse the code with complex numbers
Be brave and persistent!

- you may find the STL quite difficult to use
- template syntax is really tough
- it is hard to get right, compilers spit out long error novels
  - but, compilers are getting better with error messages
- C++20 will help with concepts and ranges
- the STL is extremely powerful and flexible
- it will be worth your time!
5 Core modern C++

- Constness
- Exceptions
- Templates
- The STL
- Lambdas
- pointers and RAII
An alternate way to specify a function's return type

```cpp
ReturnType func(Arg1 a, Arg2 b);  // classic
auto func(Arg1 a, Arg2 b) -> ReturnType;
```
Trailing function return type

An alternate way to specify a function’s return type

```cpp
ReturnType func(Arg1 a, Arg2 b);  // classic
auto func(Arg1 a, Arg2 b) -> ReturnType;
```

Advantages

- Allows to simplify inner type definition

```cpp
class Class {
    using ReturnType = int;
    ReturnType func();
}
Class::ReturnType Class::func() {...}
auto Class::func() -> ReturnType {...}
```

- C++14: ReturnType not required, compiler can deduce it
- Used by lambda expressions
Definition

A lambda expression is a function with no name.
**Definition**

A lambda expression is a function with no name.

**Python example**

```python
data = [1,9,3,8,3,7,4,6,5]

# without lambdas
def isOdd(n):
    return n%2 == 1
print(filter(isOdd, data))

# with lambdas
print(filter(lambda n:n%2==1, data))
```
**C++11**

### Simplified syntax

```cpp
class auto lambda = [] (arguments) -> return_type {  
    statements;
};
```

- The return type specification is optional
- `lambda` is an instance of a functor type, which is generated by the compiler

### Usage example

```cpp
std::vector<int> data{1, 2, 3, 4, 5};
std::for_each(begin(data), end(data), [](int i) {
std::cout << "The square of " << i
    << " is " << i*i << std::endl;
});
```
Capturing variables

Python code

```python
1 increment = 3
2 data = [1,9,3,8,3,7,4,6,5]
3 map(lambda x : x + increment, data)
```
Capturing variables

Python code

```python
increment = 3
data = [1,9,3,8,3,7,4,6,5]
map(lambda x : x + increment, data)
```

First attempt in C++

```c++
int increment = 3;
std::vector<int> data{1,9,3,8,3,7,4,6,5};
transform(begin(data), end(data), begin(data),
[](int x) { return x+increment; });
```
Capturing variables

Python code

```python
increment = 3
data = [1,9,3,8,3,7,4,6,5]
map(lambda x : x + increment, data)
```

First attempt in C++

```cpp
int increment = 3;
std::vector<int> data{1,9,3,8,3,7,4,6,5};
transform(begin(data), end(data), begin(data), 
    [](int x) { return x+increment; });
```

Error

```cpp
error: 'increment' is not captured
    [](int x) { return x+increment; });
```
Capturing variables

The capture list

- local variables outside the lambda must be explicitly captured
- captured variables are listed within initial []

Example

```cpp
int increment = 3;
std::vector<int> data{1,9,3,8,3,7,4,6,5};
transform(begin(data), end(data), begin(data),
[increment](int x) {
    return x + increment;
});
```

B. Gruber, S. Hageboeck, S. Ponce

HEP C++ course
Capturing variables

The capture list
- Local variables outside the lambda must be explicitly captured
- Captured variables are listed within initial `[]`

Example
```cpp
int increment = 3;
std::vector<int> data{1,9,3,8,3,7,4,6,5};
transform(begin(data), end(data), begin(data),
[increment](int x) {
    return x+increment;
});
```
Default capture is by value

Code example

```cpp
int sum = 0;
std::vector<int> data{1, 9, 3, 8, 3, 7, 4, 6, 5};
for_each(begin(data), end(data),
    [sum](int x) { sum += x; });
```

Error

`error: assignment of read-only variable 'sum'
[sum](int x) { sum += x; });`

Explanation

By default, variables are captured by value, and the lambda's `operator()` is `const`.
Default capture is by value

Code example

```cpp
int sum = 0;
std::vector<int> data{1, 9, 3, 8, 3, 7, 4, 6, 5};
for_each(begin(data), end(data),
    [sum](int x) { sum += x; });
```

Error

```
error: assignment of read-only variable 'sum'
    [sum](int x) { sum += x; });
```
Default capture is by value

Code example

```cpp
int sum = 0;
std::vector<int> data{1, 9, 3, 8, 3, 7, 4, 6, 5};
for_each(begin(data), end(data),
    [sum](int x) { sum += x; });
```

Error

```
error: assignment of read-only variable 'sum'
[sum](int x) { sum += x; });
```

Explanation

By default, variables are captured by value, and the lambda’s `operator()` is `const`.
Capture by reference

Simple example

In order to capture by reference, add '& ' before the variable

```cpp
int sum = 0;
std::vector<int> data{1,9,3,8,3,7,4,6,5};
for_each(begin(data), end(data),
    [&sum](int x) { sum += x; });
```
Capture by reference

Simple example

In order to capture by reference, add '& before the variable

```cpp
int sum = 0;
std::vector<int> data{1,9,3,8,3,7,4,6,5};
for_each(begin(data), end(data),
    [&sum](int x) { sum += x; });
```

Mixed case

One can of course mix values and references

```cpp
int sum = 0, offset = 1;
std::vector<int> data{1,9,3,8,3,7,4,6,5};
for_each(begin(data), end(data),
    [&sum, offset](int x) {
        sum += x + offset;
    });
```
Capture list

all by value

```cpp
[=](...) { ... };
```
Capture list

all by value

```cpp
[=](...) { ... };
```

all by reference

```cpp
[&](...) { ... };
```
Capture list

all by value

```cpp
[=] (...) { ... };
```

all by reference

```cpp
[&] (...) { ... };
```

mix

```cpp
[&, b] (...) { ... };
[=, &b] (...) { ... };
```
Anatomy of a lambda

```cpp
int sum = 0, off = 1;
auto l = [
    &sum, off
](int x) {
    sum += x + off;
};
l(42);
```

See also result on cppinsights.io.
Higher-order lambdas

Example

```cpp
auto build_incrementer = [](int inc) {
    return [inc](int value) { return value + inc; };
};

auto inc1 = build_incrementer(1);
auto inc10 = build_incrementer(10);
int i = 0;
i = inc1(i);  // i = 1
i = inc10(i); // i = 11
```

How it works

- `build_incrementer` returns a function object
- This function's behavior depends on a parameter
- Note how `auto` is useful here!
Prefere lambdas over functors

Before lambdas

```cpp
struct Incrementer {
    int m_inc;
    Incrementer(int inc) : m_inc(inc) {}
    int operator()(int value) {
        return value + m_inc;
    }
};

std::vector<int> v{1, 2, 3};
std::transform(begin(v), end(v), begin(v),
               Incrementer(1));
for (auto a : v) std::cout << a << " ";
```
Prefer lambdas over functors

With lambdas

```cpp
std::vector<int> v{1, 2, 3};
std::transform(begin(v), end(v), begin(v),
    [](int value) {
        return value + 1;
    });
for (auto a : v) std::cout << a << " ";
```
With lambdas

```cpp
std::vector<int> v{1, 2, 3};
std::transform(begin(v), end(v), begin(v),
    [](int value) {
        return value + 1;
    });
for (auto a : v) std::cout << a << " ";
```

Conclusion

Use the STL with lambdas!
Lambdas

Exercise Time

- go to code/lambdas
- look at the code (it’s the solution to the stl exercise)
- use lambdas to simplify it
Core modern C++
- Constness
- Exceptions
- Templates
- The STL
- Lambdas
- pointers and RAIi
Pointers: why they are error prone?

They need initialization

```cpp
1 char *s;
2 try {
3     callThatThrows();
4     s = (char*) malloc(...);
5     strncpy(s, ...);
6 } catch (...) { ... }
7 bar(s);
```
Pointers: why they are error prone?

They need initialization

```cpp
char *s;
try {
    callThatThrows();
    s = (char*) malloc(...);
    strncpy(s, ...);
} catch (...) { ... }
bar(s);
```

This code shows how errors can occur with pointers. The `char *s;` line initializes the pointer `s`. However, it doesn't initialize the memory pointed to by `s` with `malloc(...)`, which can lead to a Seg Fault if the memory is not allocated properly.

They need to be released

```cpp
char *s = (char*) malloc(...);
strncpy(s, ...);
if (0 != strncmp(s, ...)) return;
foo(s);
free(s);
```

Here, the memory allocated by `malloc(...)` is not released by `free(s)` when it's no longer needed. This can lead to a memory leak if the memory is not freed appropriately.

They need clear ownership

```cpp
char *s = (char*) malloc(...);
strncpy(s, ...);
someVector.push_back(s);
someSet.add(s);
std::thread t1(vecConsumer, someVector);
std::thread t2(setConsumer, someSet);
```

In this code, ownership of the pointers `s` is not clearly defined. The memory allocated with `malloc(...)` is not explicitly released by the threads `t1` and `t2`. This can lead to issues if the threads finish before the memory is freed.
Pointers: why they are error prone?

They need initialization

```cpp
char *s;
try {
    callThatThrows();
    s = (char*) malloc(...);
```

They need to be released

```cpp
char *s = (char*) malloc(...);
strncpy(s, ...);
if (0 != strncmp(s, ...)) return;
foo(s);
free(s);
```
Points: why they are error prone?

They need initialization

```cpp
char *s;
try {
    callThatThrows();
    s = (char*) malloc(...);
}
```

**Seg Fault**

They need to be released

```cpp
char *s = (char*) malloc(...);
strncpy(s, ...);
if (0 != strncmp(s, ...)) return;
foo(s);
free(s);
```

**Memory leak**
Pointers: why they are error prone?

They need initialization

```cpp
1 char *s;
2 try {
3   callThatThrows();
4   s = (char*) malloc(...);
```

They need to be released

```cpp
1 char *s = (char*) malloc(...);
2 strncpy(s, ...);
```

They need clear ownership

```cpp
1 char *s = (char*) malloc(...);
2 strncpy(s, ...);
3 someVector.push_back(s);
4 someSet.add(s);
5 std::thread t1(vecConsumer, someVector);
6 std::thread t2(setConsumer, someSet);
```
Pointers: why they are error prone?

They need initialization

```c++
char *s;
try {
    callThatThrows();
    s = (char*) malloc(...);
}
```

---

Seg Fault

They need to be released

```c++
char *s = (char*) malloc(...);
strncpy(s, ...);
```

---

Memory leak

They need clear ownership

```c++
char *s = (char*) malloc(...);
strncpy(s, ...);
someVector.push_back(s);
someSet.add(s);
std::thread t1(vecConsumer, someVector);
std::thread t2(setConsumer, someSet);
```
This problem exists for any resource

For example with a file

```c++
try {
    FILE *handle = std::fopen(path, "w+");  
    if (nullptr == handle) { throw ... } 
    if (std::fputs(str, handle) == EOF) {
        throw ...  
    }
    fclose(handle); 
} catch (...) { ... }
```
Resource Acquisition Is Initialization (RAII)

Practically

Use object semantic to acquire/release resources

- wrap the resource inside an object
- acquire resource in constructor
- release resource in destructor
- create this object on the stack so that it is automatically destructed when leaving the scope, including in case of exception
- use move semantics to pass the resource around
RAII in practice

```cpp
class File {
    public:
        File(const char* filename) :
            m_file_handle(std::fopen(filename, "w+")) {
            if (m_file_handle == NULL) { throw ... } }
        ~File() { std::fclose(m_file_handle); } 
        void write (const char* str) {
            if (std::fputs(str, m_file_handle) == EOF) { 
                throw ... 
            }
        }
    private:
        FILE* m_file_handle;
};
```
### Usage of File class

```cpp
void log_function() {
    // file opening, aka resource acquisition
    File logfile("logfile.txt") ;

    // file usage
    logfile.write("hello logfile!");

    // file is automatically closed by the call to
    // its destructor, even in case of exception!
}
```

- on real projects, use `std::fstream` to handle files
an RAII pointer

- wraps a regular pointer
- has move only semantic
  - the pointer has unique ownership
  - copying will result in a compile error
- in `<memory>` header
std::unique_ptr

**an RAII pointer**

- wraps a regular pointer
- has move only semantic
  - the pointer has unique ownership
  - copying will result in a compile error
- in `<memory>` header

**Usage**

```cpp
std::unique_ptr<Foo> p{ new Foo{} }; // allocation
std::cout << p.get() << " points to "
    << p->someMember << '\n';
void f(std::unique_ptr<Foo> ptr);
f(std::move(p)); // transfer ownership
// deallocation when exiting f
assert(p.get() == nullptr);
```
Quiz

```cpp
Foo *p = new Foo{};  // allocation
std::unique_ptr<Foo> uptr(p);
void f(std::unique_ptr<Foo> ptr);
f(uptr);  // transfer of ownership
```

What do you expect?
Foo *p = new Foo{};  // allocation
std::unique_ptr<Foo> uptr(p);
void f(std::unique_ptr<Foo> ptr);
f(uptr);  // transfer of ownership

What do you expect?

Compilation Error

test.cpp:15:5: error: call to deleted constructor of 'std::unique_ptr<Foo>'
f(uptr);
^^^^
/usr/include/c++/4.9/bits/unique_ptr.h:356:7: note: 'unique_ptr' has been explicitly marked deleted here
unique_ptr(const unique_ptr&) = delete;
std::make_unique

- directly allocates a unique_ptr
- no `new` or `delete` calls anymore!
**std::make_unique**

- directly allocates a unique_ptr
- no new or delete calls anymore!

### make_unique usage

```cpp
// allocation of one Foo object,
// calls new Foo(arg1, arg2) internally
auto a = std::make_unique<Foo>(arg1, arg2);
std::cout << a.get() << " points to "
    << a->someMember << 'n';

// allocation of an array of Foos
// calls default constructor
auto b = std::make_unique<Foo[]>(10);

// deallocations at end of scope
```

B. Gruber, S. Hageboeck, S. Ponce

HEP C++ course
### When to use what?

- Always use RAII for all resources, in particular allocations.
- You thus never have to release / deallocate yourself.
- Use raw pointers as non-owning, re-bindable observers.
- Remember that `unique_ptr` is move only.

```cpp
// A question of ownership
unique_ptr<T> producer();
void observer(const T&);
void modifier(T&);
void consumer(unique_ptr<T>);

// Receive ownership
observer(*pt);

// Keep ownership
modifier(*pt);

// Transfer ownership
consumer(std::move(pt));
```
RAII or raw pointers

When to use what?

- Always use RAII for all resources, in particular allocations
- You thus never have to release / deallocate yourself
- Use raw pointers as non-owning, re-bindable observers
- Remember that `unique_ptr` is move only

A question of ownership

```cpp
unique_ptr<T> producer();
void observer(const T&);
void modifier(T&);
void consumer(unique_ptr<T>);
unique_ptr<T> pt{producer()}; // Receive ownership
observer(*pt); // Keep ownership
modifier(*pt); // Keep ownership
consumer(std::move(pt)); // Transfer ownership
```
unique_ptr usage summary

It’s about lifetime management

- Use unique_ptr in functions taking part in lifetime management
- Otherwise use raw pointers or references
**shared_ptr, make_shared**

**shared_ptr**: a reference counting pointer
- wraps a regular pointer similar to `unique_ptr`
- has move and copy semantic
- uses reference counting internally
  - "Would the last person out, please turn off the lights?"
- reference counting is thread-safe, therefore a bit costly

**make_shared**: creates a shared_ptr

```cpp
{
    auto sp = std::make_shared<Foo>(); // #ref = 1
    vector.push_back(sp); // #ref = 2
    set.insert(sp); // #ref = 3
} // #ref 2
```
Exercise Time

- go to code/smartPointers
- compile and run the program. It doesn’t generate any output.
- Run with valgrind to check for leaks
  ```
  $ valgrind --leak-check=full --track-origins=yes ./smartPointers
  ```
- Go through `problem1()` to `problem3()` and fix the leaks using smart pointers.
- `problem4()` is the most difficult. Skip if not enough time.
Questions?

https://github.com/hsf-training/cpluspluscourse
http://cern.ch/sponce/C++Course