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
  - 223 slides, 313 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)
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++
Detailed outline

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

5. Core modern C++
   - Constructors/destructors
   - Static members
   - Allocating objects
   - Advanced OO
   - 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++
1 History and goals
- History
- Why we use it?
C/C++ origins

1967
- Simula

1978
- BCPL
- B
- K and R C
- Classic C

1980
- C with Classes

1985
- Early C++

1989
- ARM C++
- C89

1998
- C++98
- C99

2011
- C++11
- C11

2014
- C++14

2017
- C++17
- C18

2020
- 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
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

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

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

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

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

### Fast
- compiled (unlike Java or C#)
- 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

## 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 for integer declaration
int i;

/* multiline comment
 * in case we need to say more
 */
double d;

/**
 * Best choice : doxygen compatible comments
 * \fn bool isOdd(int i)
 * \brief checks whether i is odd
 * \param i input
 * \return true if i is odd, otherwise false
 */
bool isOdd(int i);
```cpp
1. bool b = true;  // boolean, true or false

2. char c = 'a';  // 8 bits ASCII char
3. char* s = "a C string";  // array of chars ended by \0
4. string t = "a C++ string";  // class provided by the STL

5. char c = -3;  // 8 bits signed integer
6. unsigned char c = 4;  // 8 bits unsigned integer
7. short int s = -444;  // 16 bits signed integer
8. unsigned short s = 444;  // 16 bits unsigned integer
9. short s = -444;  // int is optional
```
Basic types(2)

1. int i = -123456;  // 32 bits signed integer
2. unsigned int i = 1234567;  // 32 bits signed integer
3. long l = 0L;  // 32 or 64 bits (ptr size)
4. unsigned long l = 0UL;  // 32 or 64 bits (ptr size)
5. long long ll = 0LL;  // 64 bits signed integer
6. unsigned long long ll = 0ULL;  // 64 bits unsigned integer
7. float f = 1.23f;  // 32 (23+7+1) bits float
8. double d = 1.23E34;  // 64 (52+11+1) bits float
One needs to include specific header

```cpp
#include <cstdint>

int8_t c = -3;        // 8 bits, replaces char
uint8_t c = 4;        // 8 bits, replaces unsigned char

int16_t s = -444;     // 16 bits, replaced short
uint16_t s = 444;     // 16 bits, replaced unsigned short

int32_t s = -0674;    // 32 bits, replaced int
uint32_t s = 0674;    // 32 bits, replaced unsigned int

int64_t s = -0x1bc;   // 64 bits, replaced long long
uint64_t s = 0x1bc;   // 64 bits, replaced unsigned long long
```
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

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

1. int i = 4;
2. int *pi = &i;
3. int j = *pi + 1;

4. int ai[] = {1,2,3};
5. int *pai = ai;
6. int *paj = pai + 1;
7. int k = *paj + 1;

8. // not compiling
9. int *pak = k;

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

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

// not compiling
int *pak = k;

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

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

// not compiling
int *pak = k;

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

```cpp
int i = 4;
int *pi = &i;
int j = *pi + 1;

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

// not compiling
int *pak = k;

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

Memory layout:

```
0x3040 0x3041 0x3042 0x3043 0x3044 0x3045 0x3046 0x3047 0x3048 0x3049 0x304A
i pi = 0x3040
j = 5 0x3042
pak = 0x3043
k = 3 0x3044
paj = 0x3044
paj = 0x3045
pai = 0x3045
k = 3 0x3046
paj = 0x3046
paj = 0x3047
paj = 0x3047
paj = 0x3048
paj = 0x3048
paj = 0x3049
paj = 0x3049
k = 3 0x304A
k = 3 0x304A
k = 3 0x304A
k = 3 0x304A
k = 3 0x304A
k = 3 0x304A
k = 3 0x304A
k = 3 0x304A
k = 3 0x304A
k = 3 0x304A
```
```c++
int i = 4;
int *pi = &i;
int j = *pi + 1;

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

// not compiling
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;
int *paj = pai + 1;
int k = *paj + 1;

// not compiling
int *pak = k;

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

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

// not compiling
int *pak = k;

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

```c
int i = 4;
int *pi = &i;
int j = *pi + 1;

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

// not compiling
int *pak = k;

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

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

// not compiling
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. Can be tested

// allocate array of 10 ints (not initialized)
ai = (int*) malloc(10*sizeof(int));
// and set them to 0
memset(ai, 0, 10*sizeof(int));

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

// release memory
free(ai);
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
- translation unit (.cpp file + all includes) for global declarations

Example

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

Variable life time

- Variables are 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:
// b[1] = a + 1;
```

Memory layout

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3040</td>
<td>a = 1</td>
</tr>
<tr>
<td>0x3041</td>
<td>0x3044</td>
</tr>
<tr>
<td>0x3042</td>
<td>0x3043</td>
</tr>
<tr>
<td>0x3043</td>
<td></td>
</tr>
<tr>
<td>0x3044</td>
<td></td>
</tr>
</tbody>
</table>
Scope and lifetime of variables

Variable life time

- Variables are 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:
// b[1] = a + 1;
```

Memory layout

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>b[3]</td>
<td>0x3044</td>
</tr>
<tr>
<td>b[2]</td>
<td>0x3043</td>
</tr>
<tr>
<td>b[1]</td>
<td>0x3042</td>
</tr>
<tr>
<td>b[0]</td>
<td>0x3041</td>
</tr>
<tr>
<td>a</td>
<td>0x3040</td>
</tr>
</tbody>
</table>
Variable life time

- Variables are 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:
// b[1] = a + 1;
```

Memory layout:

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>?</td>
<td>0x3040</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
<td>0x3041</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
<td>0x3042</td>
</tr>
<tr>
<td>3</td>
<td>?</td>
<td>0x3043</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0x3044</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```cpp
int a = 1;
{
    int b[4];
    b[0] = a;
}
// Doesn't compile:
// b[1] = a + 1;
```
**Scope and lifetime of variables**

**Variable life time**

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

```cpp
1 int a = 1;
2 {
3    int b[4];
4    b[0] = a;
5 }
6 // Doesn't compile:
7 // b[1] = a + 1;
```

**Memory layout**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0x3041</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x3042</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x3043</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x3044</td>
<td></td>
<td></td>
<td></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 = 2;
}

namespace p {
    void f() {
        p::a = 2;
        a = 2;  // same as above
        ::a = 3;
        p::inner::a = 4;
        inner::a = 4;
        n::a = 5;
    }
}
using namespace p::inner;
void g() {
    a = 3; // error: ambiguous
}
```
Easier way to declare nested namespaces

C++14

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

C++17

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

A namespace without a name!

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

Purpose

- groups a number of declarations
- visible 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
```
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 = 78.5;

Individual teacher = {
    45, 67
};

Individual *studentPtr = &student;
studentPtr->age = 25;
```
“members” grouped together under one name

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

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

Individual teacher = {
    45, 67
};

Individual *studentPtr = &student;
studentPtr->age = 25;
```

Memory layout

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3043</td>
<td></td>
</tr>
<tr>
<td>0x3047</td>
<td></td>
</tr>
<tr>
<td>0x304B</td>
<td></td>
</tr>
<tr>
<td>0x304F</td>
<td></td>
</tr>
<tr>
<td>0x3053</td>
<td>25</td>
</tr>
<tr>
<td>0x3043</td>
<td>45</td>
</tr>
<tr>
<td>0x3047</td>
<td>67</td>
</tr>
<tr>
<td>0x304B</td>
<td>78.5</td>
</tr>
<tr>
<td>0x304F</td>
<td></td>
</tr>
</tbody>
</table>
struct Individual {
    unsigned char age;
    float weight;
};

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

Individual teacher = {
    45, 67
};

Individual *studentPtr = &student;
studentPtr->age = 25;
```
struct Individual {
    unsigned char age;
    float weight;
};

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

Individual teacher = {
    45, 67
};

Individual *studentPtr = &student;
studentPtr->age = 25;
```

Memory layout:

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>67</td>
<td>0x3053</td>
<td>45</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>0x304F</td>
</tr>
<tr>
<td>78.5</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>0x304B</td>
<td>67.0</td>
<td>0x3047</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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
```
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

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

VehicleType t = CAR;

enum VehicleType {
    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

- scoping 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
```
More sensible example

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

struct Rectangle {
    float width;
    float height;
};
```

```cpp
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;
    }
};

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

shape t;
t.type = ShapeType::Rectangle;
t.rect.width = 3;
t.rect.height = 4;
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;
s.type = ShapeType::Circle;
s.radius = 3.4;

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

### C++98

1. 
   ```
   typedef uint64_t myint;
   ```
2. 
   ```
   myint toto = 17;
   ```

### C++11

1. 
   ```
   using myint = uint64_t;
   ```
2. 
   ```
   myint toto = 17;
   ```
3. 
   ```
   template <typename T> using myvec = std::vector<T>;
   ```
4. 
   ```
   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:

```cpp
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
- will never be `nullptr`
- non-const references to temporary objects are not allowed

Advantages of pointers

- can be `nullptr`
- clearly indicates that argument may be modified
Pointers vs References

Specificities of reference
- natural syntax
- will never be `nullptr`
- non-const references to temporary objects are not allowed

Advantages of pointers
- can be `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
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) {
    printf("%s", msg);
}

// no parameter
void hello() {
    printf("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

```cpp
// multiple default arguments are possible
int add(int base = 2,
    int exp = 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
```
functions: parameters are passed by value

```c++
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
```

Memory layout:
```
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x3040</td>
<td>0x3041</td>
<td>0x3042</td>
<td>0x3043</td>
<td>0x3044</td>
<td>0x3045</td>
<td>0x3046</td>
</tr>
<tr>
<td>s</td>
<td>...</td>
<td>sn</td>
<td>0x3042</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p1</td>
<td>...</td>
<td>pn</td>
<td>0x3041</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>...</td>
<td></td>
<td>0x3040</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
```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
```
```cpp
// parameter by value
void printBS(BigStruct p) {
    ...
}
printBS(s); // copy

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

```markdown
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>0x3040</td>
<td></td>
</tr>
<tr>
<td>0x3041</td>
<td></td>
</tr>
<tr>
<td>0x3042</td>
<td></td>
</tr>
</tbody>
</table>

- `s.a = 1` before changing with `changeSS`.
- `s.a == 1` after changing with `changeSS`.
- `s.a = 2` before changing with `changeSS2`.
- `s.a == 2` after changing with `changeSS2`.
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:

```
0x3040
0x3041
0x3042
s.a = 1
```

B. Gruber, S. Hageboeck, S. Ponce
HEP C++ course
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>Memory Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3040</td>
<td>p.a = 1</td>
</tr>
<tr>
<td>0x3041</td>
<td>s.a = 1</td>
</tr>
<tr>
<td>0x3042</td>
<td></td>
</tr>
</tbody>
</table>

B. Gruber, S. Hageboeck, S. Ponce
```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>0x3040</td>
<td>s.a = 1</td>
</tr>
<tr>
<td>0x3041</td>
<td>p.a = 2</td>
</tr>
<tr>
<td>0x3042</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>0x3042</td>
<td>2</td>
</tr>
<tr>
<td>0x3041</td>
<td>1</td>
</tr>
<tr>
<td>0x3040</td>
<td></td>
</tr>
</tbody>
</table>
```

B. Gruber, S. Hageboeck, S. Ponce
HEP C++ course
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
```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>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3040</td>
<td>q</td>
</tr>
<tr>
<td>0x3041</td>
<td>s.a 2</td>
</tr>
<tr>
<td>0x3042</td>
<td></td>
</tr>
</tbody>
</table>
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></th>
<th>0x3042</th>
<th>0x3041</th>
<th>0x3040</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.a</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>p.a</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>q.a</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
```

B. Gruber, S. Hageboeck, S. Ponce

HEP C++ course

44 / 223
Value vs. pointers and references

Different ways to pass arguments to a function

- by default arguments are passed by value (=*/copy, good for small types, e.g. numbers)
- references are preferred to avoid copies
- same for pointers, but less favoured
- use `const` for safety
Value vs. pointers and references

Different ways to pass arguments to a function

- by default arguments are passed by value (＝ copy, good for small types, e.g. numbers)
- references are preferred to avoid copies
- same for pointers, but less favoured
- use `const` for safety

Syntax

```c++
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 copy / 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

Good practices with functions

- Keep functions short
- Do one logical thing
- Use expressive names
- Document the 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();
}
```

B. Gruber, S. Hageboeck, S. Ponce  
HEP C++ course
Example: don’t! Everything in one long function

```cpp
unsigned int foo() {
    // 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

- 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

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`
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

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

### Increment / Decrement

```
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
```

Use wisely

---

B. Gruber, S. Hageboeck, S. Ponce

HEP C++ course
### 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
```
Operators(2)

Bitwise and Assignment Operators

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

Boolean Operators

```c
bool a = true;
bool b = false;
bool c = a && b;       // false
bool d = a || b;       // true
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. ```
   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/c/language/operator_precedence).
### Comparison Operators

<table>
<thead>
<tr>
<th></th>
<th>Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>bool a = (3 == 3);</code> // true</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><code>bool b = (3 != 3);</code> // false</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>bool c = (4 &lt; 4);</code> // false</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><code>bool d = (4 &lt;= 4);</code> // true</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><code>bool e = (4 &gt; 4);</code> // false</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><code>bool f = (4 &gt;= 4);</code> // true</td>
<td></td>
</tr>
</tbody>
</table>

### Precedences

**Don’t use**

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

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

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

Details can be found on cppreference
Control structures

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) {
    Statements1;
} else if (condition2) {
    Statements2;
} else {
    Statements3;
}
```

- `else` and `else if` part are optional
- `else if` part can be repeated
- Braces are optional if there is a single instruction
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

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

- if test is `true` expression1 is returned
- else expression 2 is returned

Practical example

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

Syntax

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

Try to avoid if statements, use macro definitions instead.

Don’t be greedy.

Use else if when obvious and not nested.

Do not abuse "if 0..." statements.
Control structures: conditional operator

Syntax

test ? expression1 : expression2;

- if test is true expression1 is returned
- else expression 2 is returned

Practical example

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

Do not abuse

explicit ifs are easier to read

to be used only when obvious and not nested
Control structures: switch

Syntax

```cpp
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 carries on with the next case if no `break` is present!
- `default` may be omitted
Control structures: switch

Syntax

```c
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 carries on with the next case if no `break` is present!
- `default` may be omitted

Use break

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

Practical example

```c++
enum class Lang { FRENCH, GERMAN, ENGLISH, OTHER };
...
switch (language) {
    case Lang::FRENCH:
        printf("Bonjour");
        break;
    case Lang::GERMAN:
        printf("Guten Tag");
        break;
    case Lang::ENGLISH:
        printf("Good morning");
        break;
    default:
        printf("I do not speak your language");
}
```
[[fallthrough]] attribute

**C++14**

```cpp
switch (c) {
  case 'a':
    f(); // Warning emitted
  case 'c':
    h();
}
```

**C++17**

```cpp
switch (c) {
  case 'a':
    f();
    [[fallthrough]]; // Warning suppressed
  case 'c':
    h();
}
```
init-statements for if and switch

Allows to simplify if and switch statements

```cpp
1 Value val = GetValue();
2 if (condition(val)) {
3    // on success
4 } else {
5    // on false...
6 }
```

```cpp
1 if (Value val = GetValue(); condition(val)) {
2    // on success
3 } else {
4    // on false...
5 }
```

val is visible only inside the if and else statements
## 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 there is a single instruction
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 there is a single instruction

Practical example

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

Practical example

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

Do not abuse the syntax

The for statement should fit in 1-3 lines
Range based loops

Reason of being
- simplifies loops tremendously
- especially with STL containers

Syntax
```
for ( type iteration_variable : container ) {
   // 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

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

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

- braces are optional if there is a single instruction

Practical example

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

while loop syntax

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

- braces are optional if there is a single instruction

Practical example

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

control commands

- **break** goes out of the loop
- **continue** goes immediately to next iteration
- **return** goes out of current function
Control structures: commands

control commands

- **break** goes out of the loop
- **continue** goes immediately to next iteration
- **return** goes out of current function

Practical example

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

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
- defined in a “header file”
- no implementation defined

Header: hello.hpp
```cpp
void printHello();
```

Usage: myfile.cpp
```cpp
#include "hello.hpp"

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

// macros
#define MY_GOLDEN_NUMBER 1746

// compile time decision
#ifdef USE64BITS
    typedef uint64_t myint;
#else
    typedef uint32_t myint;
#endif
```c++
// file inclusion
#include "hello.hpp"

// macros
#define MY_GOLDEN_NUMBER 1746

// compile time decision
#ifdef USE64BITS
    typedef uint64_t myint;
#else
    typedef uint32_t myint;
#endif
```

Use only in very restricted cases

- include of headers
- hardcoded constants before C++11
- portability necessity
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

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

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
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 crash course

# 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
Code formatting

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

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>`)
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 .
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

cpp, gcc -E

g++ -c, gcc -c

ld, gcc, g++

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
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 = no opt., 1-2 = opt., 3 = highly opt. (maybe larger binary), `g` = 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

```
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
```
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’
Debugging

3 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
   - Operators
   - Functors
4 Object orientation (OO)
- Objects and Classes
  - Inheritance
  - Constructors/destructors
  - Static members
  - Allocating objects
  - Advanced OO
  - 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

```cpp
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;
}
```
4 Object orientation (OO)

- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Operators
- Functors
```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;
};

MyFirstClass obj;
obj.a = 5; // error!
obj.setA(5); // ok
int b = obj.getA(); // MySecondClass breaks
```
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;
};
```

```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 (2)

Solution is *protected* keyword

Gives access to classes inheriting from base class

```cpp
#include <iostream>

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;
};
```

B. Gruber, S. Hageboeck, S. Ponce
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 class 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

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

**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

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

MySecondClass
void funcSecond();

MyThirdClass
void funcThird();

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
}
```
Managing inheritance privacy - protected

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

MySecondClass
---
void funcSecond();

MyThirdClass
---
void funcThird();

```cpp
1 void funcSecond() {
2   int a = priv;  // Error
3   int b = prot; // OK
4   int c = pub;  // OK
5 }
6 void funcThird() {
7   int a = priv;  // Error
8   int b = prot; // OK
9   int c = pub;  // OK
10 }
11 void extFunc(MyThirdClass t) {
12   int a = t.priv;  // Error
13   int b = t.prot; // Error
14   int c = t.pub;  // Error
15 }
```

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

MyFirstClass

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

MySecondClass

void funcSecond();

MyThirdClass

void funcThird();

1 void funcSecond() {
  2 int a = priv;   // Error
  3 int b = prot;   // OK
  4 int c = pub;    // OK
  5 }

6 void funcThird() {
  7 int a = priv;   // Error
  8 int b = prot;   // Error
  9 int c = pub;    // Error
 10 }

11 void extFunc(MyThirdClass t) {
  12 int a = t.priv; // Error
  13 int b = t.prot; // Error
  14 int c = t.pub;  // Error
 15 }

B. Gruber, S. Hageboeck, S. Ponce
Final class

Idea
- make sure you cannot inherit from a given class
- by declaring it final

Practically
```cpp
struct Base final {
    ...
};

struct Derived : Base {
    // compiler error
    ...
};
```
Object orientation (OO)
- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- 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) {}
MyFirstClass(int a_):a(a_) {}
~MyFirstClass() {}
```
```cpp
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

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

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

MySecondClass() : MyFirstClass(), b(0) {}
MySecondClass(int b_) : MyFirstClass(), b(b_) {}
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](https://en.cppreference.com/w/cpp/language/rule_of_three)

- 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();

    ...
Explicit unary constructor

Concept

- A constructor with a single non-default parameter can be used by the compiler for an implicit conversion.
- In the code below, 3 is implicitly converted into a `Vector` of size 3, which may be unexpected by the developer.

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

int main {
    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

```cpp
struct BaseClass {
    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;
};

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

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

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
```

B. Gruber, S. Hageboeck, S. Ponce
HEP C++ course
Calling constructors for arrays and vectors

```cpp
int ip[3]{1,2,3};
int* ip = new int[3]{1,2,3};
std::vector<int> v{1,2,3};
```
Calling constructors for arrays and vectors

```
list of items given within {}

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

C++98 nightmare

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
```
Object orientation (OO)
- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
  - Allocating objects
  - Advanced OO
  - 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;
};

int Text::s_nbCallsToUpper = 0;
std::string uppers = Text::upper("my text");
// now Text::s_nbCallsToUpper is 1
```
Allocating objects

4 Object orientation (OO)
  - Objects and Classes
  - Inheritance
  - Constructors/destructors
  - Static members
  - Allocating objects
  - Advanced OO
  - Operators
  - Functors
Process memory organization

4 main areas

- **the code segment** for the 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:

```
heap

... ... ...

... ... ...

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

int f() {
    // constructor called  
    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 objects are destructed by calling `delete[]`
  - destructor is called for each object of the array

```cpp
1  {
2      // default constructor called 10 times
3      MyFirstClass *a = new MyFirstClass[10];
4      ...
5      delete[] a; // destructor called 10 times
6  }
```
4 Object orientation (OO)

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

- Advanced OO
  - 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
}
```
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
}
```
the problem

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

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

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

```
Drawable
    void draw();

Shape
    void draw();

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
1. Polygon p;
2. p.draw(); // Polygon.draw
3. Shape & s = p;
4. 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:

```
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.
**final keyword**

**Idea**
- make sure you cannot further override a given virtual method
- by declaring it final

**Practically**

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

struct Intermediate : Base {
    void some_func(float) final;
};

struct Derived : Intermediate {
    void some_func(float) override; // error
};
```
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
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();
```

```
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
- the equivalent of an Interface in Java

```cpp
struct Drawable {
    virtual void draw() = 0;
};
```
Concept

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

```cpp
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
```
## 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
Multiple Inheritance

Concept

- one class can inherit from multiple parents

```cpp
1 class TextBox :
2     public Rectangle, Text {
3         // inherits from both
4         // publicly from Rectangle
5         // privately from Text
6     }
```
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
4 Object orientation (OO)

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

```cpp
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@0</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);
}
```

B. Gruber, S. Hageboeck, S. Ponce
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
Functors

Object orientation (OO)
- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- 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
```
Typical usage

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

```cpp
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);
```
Core modern C++

1. History and goals
2. Language basics
3. Useful tools
4. Object orientation (OO)

5. Core modern C++
   - Constness
   - Exceptions
   - Templates
   - The STL
   - Lambdas
   - pointers and RAII
5 Core modern C++
- Constness
- Exceptions
- Templates
- The STL
- Lambdas
- pointers and RAII
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;
*j = 5; // ok, value can be changed
j = &b; // error, pointer is const

// constant pointer to a constant integer
int const * const k = &a;
*k = 5; // error, value is const
k = &b; // error, pointer is const

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

The `const` keyword for class 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 {
        m_member = 0; // Error: function is constant
    }
    int m_member;
};
```
Method constness

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);

4 int a = 0;
5 int const b = 0;

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
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

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

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

Rules and Advice I

- any object can be thrown; best to use those in std::except
  - define your own subclass of std::exception if needed
- an exception will be caught if the thrown object’s type matches a `catch` clause
  - if nothing catches an exception then std::terminate is called
  - throw exceptions by value, catch them by (const) reference
- all objects on the stack between the `throw` and the `catch` are destructed automatically during stack unwinding
  - this should give you a clean release of intermediate resources
  - make sure you are using the RAII idiom in your own classes
Exceptions

Rules and Advice II

- 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`
Exceptions

A more illustrative example

- exceptions are very powerful when there are many calls / lines of 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
1 try {
2   for (const &file : files) {
3     try {
4       process_file(file);
5     } catch (const file_bad &e) {
6       ... // Skips one file
7     }
8   }
9 }
10 } catch (const db_bad &e) {
11   ... // This exception
12   // aborts everything
13 }
```

```cpp
1 int process_file(const fileobj &file) {
2   ...
3   if (fh = open_file(file)) {
4     throw
5     file_bad(file.error.reason());
6   }
7   while (!fh) {
8     line = read_line(fh);
9     add_line_to_db(line); // Can throw
10     // db_bad
11   }
12   return 0;
13 }
```
exceptions have little cost if no exception is thrown
  so they are recommended for exceptional events

however, in a tight, performance-critical loop, where the error
is handled close to the place it happens a more explicit error
code or object might be better
  but profile if you really want to find out!

Don’t

```cpp
// try ... catch in hot loop might
// impact performance
for (const &item: list) {
    try {
        check_item(item); // Can throw
        ... // process item
    } catch (bad_item &e) {
        ... // ignore and keep going
    }
}
```

Do

```cpp
// For this code, explicit error codes
// may be better
for (const &item: list) {
    std::optional<T> err_code = check_item(item);
    if (!err_code) {
        ... // process item
    } else {
        ... // warn and carry on
    }
}
```
The `noexcept` specifier states that a function is guaranteed to not throw an exception. For example:

```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.

The `noexcept(expression)` specifier evaluates the expression and, if `true`, the function it applied to guarantees it won’t throw:

```cpp
int safe_if_long_is_8_bytes() noexcept(sizeof(long)==8);
```

Use `noexcept` on leaf functions where you can be sure of the behaviour.

Since C++11 destructors are `noexcept` - never throw from them.

B. Gruber, S. Hageboeck, S. Ponce

HEP C++ course
the **noexcept** (expression) operator performs a compile-time check to know whether an expression can throw exceptions

- it returns a bool, which is *true* if no exceptions will be emitted

```cpp
constexpr bool callCannotThrow = noexcept(f());
if constexpr (callCannotThrow) { ... }
```
5 Core modern C++

- Constness
- Exceptions
- Templates
- The STL
- Lambdas
- pointers and RAII
**Concept**

- The C++ way to write reusable code
- aka macros on steroids
- Applicable to functions and objects

```cpp
1 template<typename T>
2 const T & max(const T &a, const T &b) {
3     return a > b ? a : b;
4 }

5
6 template<typename T>
7 struct Vector {
8     int m_len;
9     T* m_data;
10    }
```
**Warning**

These are really like macros

- they are compiled n times
- 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

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

- can be a class,
- you can have several
- last ones can have a default value

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

Map<std::string, int> m1;
Map<float> m2;   // Map<float, float>
Map<> m3;        // Map<int, int>
```
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) {
    ...
}
template parameters can also be a value

- 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
template<typename F, unsigned int N> struct Polygon {
    Polygon(F radius) : m_radius(radius) {}
    F perimeter() {return 2*N*sin(PI/N)*m_radius;}
    F m_radius;
};

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
5 Core modern C++

- Constness
- Exceptions
- Templates
- The STL
- Lambdas
- pointers and RAII
The Standard Template Library

What it is

- A library of standard templates
- Everything you need, or ever dreamed of
  - strings, containers, iterators
  - algorithms, functions, sorters
  - functors, allocators
  - ...
- Portable
- Reusable
- Efficient
What it is

- A library of standard templates
- Everything you need, or ever dreamed of
  - strings, containers, iterators
  - algorithms, functions, sorters
  - functors, allocators
  - ...

- Portable
- Reusable
- Efficient

Just 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

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

containers

- a structure containing data
- with a given way of handling it
- irrespective of
  - the data itself (templated)
  - the memory allocation of the structure (templated)
  - the algorithms that may use the structure

examples

- string
- tuple, 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
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
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
#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(), vr.begin(), std::multiplies{}); // function

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

#include <vector>
#include <algorithm>

std::vector<int> vi, vr(3);
vi.push_back(5); vi.push_back(3); vi.push_back(4);

std::transform(vi.begin(), vi.end(), vi.begin(), vr.begin(), std::multiplies<int>());

for(std::vector<int>::iterator it = vr.begin(); it != vr.end(); it++) {
    std::cout << *it << ' ';
}

STL and functors

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

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

You may find the STL quite difficult to use.

- template syntax is really tough
- it is hard to debug (compilers spit out long error novels)

However, this has improved a lot with C++11
And may again in C++20 with concepts
Loops and auto keyword with the STL

### Old way

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

### New way

```cpp
std::vector<int> v = ...;
int sum = 0;
for (auto a : v) { sum += a; }
```
Loops and auto keyword with the STL

**Old way**

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

**New way**

```cpp
std::vector<int> v = ...;
int sum = 0;
for (auto a : v) { sum += a; }
```
Loops and auto keyword with the STL

Old way

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

New way

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

STL way (C++17)

```c++
std::vector<int> v = ...;
int sum = std::reduce(v.begin(), v.end(), 0);
```
Lambdas

5 Core modern C++
- Constness
- Exceptions
- Templates
- The STL
- Lambdas
- pointers and RAII
A new way to specify function’s return type

```cpp
ReturnType fn_name(ArgType1, ArgType2); //old
auto fn_name(ArgType1, ArgType2) -> ReturnType;
```

## Advantages
- Allows to simplify inner type definition
- C++14: ReturnType is not mandatory, compiler can deduce it
Function return type

A new way to specify function’s return type

```cpp
ReturnType fn_name(ArgType1, ArgType2);  // old
auto fn_name(ArgType1, ArgType2) -> ReturnType;
```

Advantages

- Allows to simplify inner type definition

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

- C++14: ReturnType is not mandatory, compiler can deduce it
- will be used for lambdas
Definition

A lambda 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))
```
Definition

A lambda is a function with no name.

Python example

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

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

6 # with lambdas
7 print(filter(lambda n:n%2==1, data))
```
Simplified syntax

```cpp
[] (args) -> type {
    code;
}
```

The type specification is optional

Usage example

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

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

First attempt in C++

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

```
error: 'increment' is not captured
      [](int x) { return x+increment; });
```
Variable capture

- external variables need to 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;
});
```
Variable capture

- external variables need to 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; });
```
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

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

Code example

```
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
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; });
```
### 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

\[=]\(\ldots\) \{ \ldots \};
Capture list

**all by value**

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

**all by reference**

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

**all by value**

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

**all by reference**

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

**mix**

```
[& b] (...) { ... }
[=, &b] (...) { ... }
```
**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!
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 << " ";
```
To make the STL usable in C++

```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
pointers and RAII

5 Core modern C++
- Constness
- Exceptions
- Templates
- The STL
- Lambdas
- pointers and RAII
Pointers: why they are error prone?

They need initialization

```c++
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?**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>char *s;</td>
<td>They need initialization</td>
</tr>
<tr>
<td>2</td>
<td>try {</td>
<td>Seg Fault</td>
</tr>
<tr>
<td>3</td>
<td>callThatThrows();</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s = (char*) malloc(...);</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>strncpy(s, ...);</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>} catch (...) { ... }</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>bar(s);</td>
<td></td>
</tr>
</tbody>
</table>

```cpp
define callThatThrows():
    return 0;

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

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

They need to be released

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

Seg Fault
Pointers: why they are error prone?

They need initialization

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

They need to be released

```c++
char *s = (char*) malloc(...);
strncpy(s, ...);
if (0 != strcmp(s, ...)) return;
foo(s);
free(s);
```
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, ...);
```

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);
```
Pointers: why they are error prone?

**They need initialization**

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

*Seg Fault*

**They need to be released**

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

*Memory leak*

**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);
```

*Who should release?*
This problem exists for any resource

For example with a file

```cpp
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

File class

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;
};
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 !
}

Usage of File class
## std::unique_ptr

**an RAII pointer**

- wraps a regular pointer
- has move only semantic
  - the pointer is only owned once
- in `<memory>` header

### Usage

```cpp
Foo *p = new Foo{}; // allocation
std::unique_ptr<Foo> uptr(p);
std::cout << uptr.get() << " points to " << uptr->someMember << '
';
void f(std::unique_ptr<Foo> ptr);
f(std::move(uptr)); // transfer of ownership
// deallocation when exiting f
std::cout << uptr.get() << '
'; // 0
```
std::unique_ptr

an RAII pointer

- wraps a regular pointer
- has move only semantic
  - the pointer is only owned once
- in `<memory>` header

Usage

```cpp
Foo *p = new Foo{}; // allocation
std::unique_ptr<Foo> uptr(p);
std::cout << uptr.get() << " points to "
    << uptr->someMember << '\n';
void f(std::unique_ptr<Foo> ptr);
f(std::move(uptr)); // transfer of ownership
// deallocation when exiting f
std::cout << uptr.get() << '\n'; // 0
```
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?
Quiz

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,
// calling constructor with one argument
auto a = std::make_unique<Foo>(memberValue);
std::cout << a.get() << " points to "
    << a->someMember << std::endl;

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

// deallocations
```
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 for observer functions (or references)
  - remember that unique_ptr is move only
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 for observer functions (or references)
  - 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()};
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 : 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 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.
This is the end

Questions?

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