HEP C++ course

D.Chamont, B. Gruber, S. Hageboeck, S. Ponce
sebastien.ponce@cern.ch

CERN

October 15, 2025

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 it is already too long for the time we have
  - 256 slides, 350 pages, 13 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 ?

- full sources at github.com/hsf-training/cpluspluscourse
- latest pdf on GitHub

More courses

The HSF Software Training Center

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

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

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

Outline

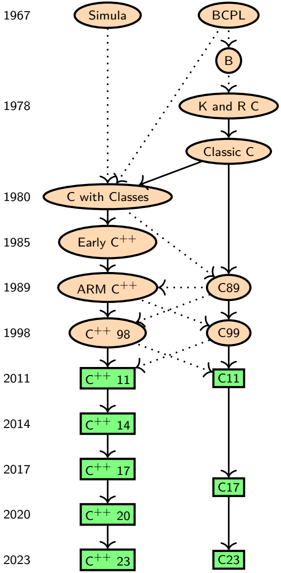
Detailed outline

History and goals

History

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

C/C^{++} origins



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 up to C^{++} 20 (only partially)
- Each slide will be marked with first spec introducing the feature

C++ 11, C++ 14, C++ 17, C++ 20, C++ 23, C++ 26...

Status

- A new C++ specification every 3 years
  - C++ 23 complete since 11th of Feb. 2023
  - work on C++ 26 is ongoing
- Bringing each time a lot of goodies

C++ 11, C++ 14, C++ 17, C++ 20, C++ 23, C++ 26...

Status

- A new C++ specification every 3 years
  - C++ 23 complete since 11th of Feb. 2023
  - work on C++ 26 is ongoing
- 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`

C++	gcc	clang
11	≥4.8	≥3.3
14	≥4.9	≥3.4
17	≥7.3	≥5
20	≥11, 14	≥19
23	>15	>20

Table: Minimum versions of gcc and clang for a given C++ version

Why we use it?

Why is C^{++} our language of choice?

Adapted to large projects

- statically and strongly typed
- object oriented
- widely used (and taught)
- many available libraries

Why is C^{++} our language of choice?

Adapted to large projects

- statically and strongly typed
- object oriented
- widely used (and taught)
- many available libraries

Fast

- compiled to native machine code
  - unlike Java, C#, Python, ...
- allows to go close to hardware when needed

Why is C^{++} our language of choice?

Adapted to large projects

- statically and strongly typed
- object oriented
- widely used (and taught)
- many available libraries

Fast

- compiled to native machine code
  - unlike Java, C#, Python, ...
- allows to go close to hardware when needed

What we get

- the most powerful language
- the most complicated one
- the most error prone?

Language basics

Core syntax and types

Hello World

C++ 98

```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;
}
```

Comments

C++ 98

```cpp
// simple comment until end of line
int i;

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

/**
 * Best choice : doxygen compatible comments
 * \brief checks whether i is odd
 * \param i input
 * \return true if i is odd, otherwise false
 * \see https://www.doxygen.nl/manual/docblocks.html
 */
bool isOdd(int i);
```

Basic types(1)

C++ 98

```cpp
bool b = true;              // boolean, true or false

char c = 'a';               // min 8 bit integer
                            // may be signed or not
                            // can store an ASCII character
signed char c = 4;          // min 8 bit signed integer
unsigned char c = 4;        // min 8 bit unsigned integer

char* s = "a C string";     // array of chars ended by \0
string t = "a C++ string";// class provided by the STL

short int s = -444;         // min 16 bit signed integer
unsigned short s = 444;     // min 16 bit unsigned integer
short s = -444;             // int is optional
```

Basic types(2)

C++ 98

```cpp
int i = -123456;             // min 16, usually 32 bit
unsigned int i = 1234567;    // min 16, usually 32 bit

long l = 0L                  // min 32 bit
unsigned long l = 0UL;       // min 32 bit

long long ll = 0LL;                // min 64 bit
unsigned long long l = 0ULL; // min 64 bit

float f = 1.23f;             // 32 (1+8+23) bit float
double d = 1.23E34;          // 64 (1+11+52) bit float
long double ld = 1.23E34L    // min 64 bit float
```

Portable numeric types

C++ 98

```cpp
#include <cstdint> // defines the following:

std::int8_t c = -3;      // 8 bit signed integer
std::uint8_t c = 4;      // 8 bit unsigned integer

std::int16_t s = -444;   // 16 bit signed integer
std::uint16_t s = 444;   // 16 bit unsigned integer

std::int32_t s = -674;   // 32 bit signed integer
std::uint32_t s = 674;   // 32 bit unsigned integer

std::int64_t s = -1635; // 64 bit signed integer
std::uint64_t s = 1635; // 64 bit unsigned int
```

Integer literals C^{++} 98

```
int i = 1234;            // decimal      (base 10)
int i = 02322;           // octal        (base  8)
int i = 0x4d2;           // hexadecimal  (base 16)
int i = 0X4D2;           // hexadecimal  (base 16)
int i = 0b10011010010;   // binary       (base  2) C++14

int i = 123'456'789;     // digit separators, C++14
int i = 0b100'1101'0010; // digit separators, C++14

42            // int
42u,    42U   // unsigned int
42l,    42L   // long
42ul,   42UL  // unsigned long
42ll,   42LL  // long long
42ull, 42ULL  // unsigned long long
```

Floating-point literals

C++ 98

```
double d = 12.34;
double d = 12.;
double d = .34;
double d = 12e34;            // 12 * 10^34
double d = 12E34;            // 12 * 10^34
double d = 12e-34;           // 12 * 10^-34
double d = 12.34e34;         // 12.34 * 10^34

double d = 123'456.789'101; // digit separators, C++14

double d = 0x4d2.4p3;    // hexfloat, 0x4d2.4 * 2^3
                         // = 1234.25 * 2^3 = 9874

3.14f, 3.14F,  // float
3.14,  3.14,   // double
3.14l, 3.14L,  // long double
```

Arrays and Pointers

Static arrays

C^{++} 98

```cpp
int ai[4] = {1,2,3,4};
int ai[] = {1,2,3,4};  // identical

char ac[3] = {'a','b','c'};    // char array
char ac[4] = "abc";            // valid C string
char ac[4] = {'a','b','c',0}; // same valid string

int i = ai[2];  // i = 3
char c = ac[8]; // at best garbage, may segfault
int i = ai[4];  // also garbage !
```

Pointers

C++ 98

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

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

// compile error
int *pak = k;

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

Pointers

C++ 98

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

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

// compile error
int *pak = k;

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

Memory layout

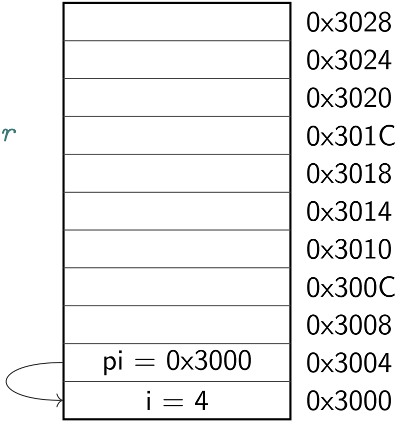
	0x3028
	0x3024
	0x3020
	0x301C
	0x3018
	0x3014
	0x3010
	0x300C
	0x3008
	0x3004
i = 4	0x3000

Pointers

C++ 98

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

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

// compile error
int *pak = k;

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

Memory layout



Pointers

C++ 98

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

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

// compile error
int *pak = k;

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

Memory layout

	0x3028
	0x3024
	0x3020
	0x301C
	0x3018
	0x3014
	0x3010
	0x300C
$j = 5$	0x3008
$pi = 0x3000$	0x3004
$i = 4$	0x3000

Pointers

C^{++} 98

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

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

// compile error
int *pak = k;

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

Memory layout

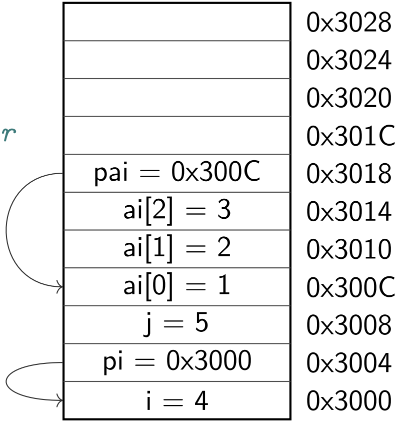
	0x3028
	0x3024
	0x3020
	0x301C
	0x3018
ai[2] = 3	0x3014
ai[1] = 2	0x3010
ai[0] = 1	0x300C
j = 5	0x3008
pi = 0x3000	0x3004
i = 4	0x3000

Pointers

C^{++} 98

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

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

// compile error
int *pak = k;

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

Memory layout



Pointers

C++ 98

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

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

// compile error
int *pak = k;

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

Memory layout

	0x3028
	0x3024
	0x3020
paj = 0x3010	0x301C
pai = 0x300C	0x3018
ai[2] = 3	0x3014
ai[1] = 2	0x3010
ai[0] = 1	0x300C
j = 5	0x3008
pi = 0x3000	0x3004
i = 4	0x3000

Pointers

C^{++} 98

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

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

// compile error
int *pak = k;

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

Memory layout

	0x3028
	0x3024
k = 3	0x3020
paj = 0x3010	0x301C
pai = 0x300C	0x3018
ai[2] = 3	0x3014
ai[1] = 2	0x3010
ai[0] = 1	0x300C
j = 5	0x3008
pi = 0x3000	0x3004
i = 4	0x3000

Pointers

C^{++} 98

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

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

// compile error
int *pak = k;

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

Memory layout

		0x3028
??	pak = 3	0x3024
	k = 3	0x3020
	paj = 0x3010	0x301C
	pai = 0x300C	0x3018
	ai[2] = 3	0x3014
	ai[1] = 2	0x3010
	ai[0] = 1	0x300C
	j = 5	0x3008
	pi = 0x3000	0x3004
	i = 4	0x3000

nullptr

C^{++} 11

A pointer to nothing

- if a pointer doesn't point to anything, set it to `nullptr`
  - useful to e.g. mark the end of a linked data structure
  - or absence of an optional function argument (pointer)
- same as setting it to 0 or `NULL` (before C^{++} 11)
- triggers compilation error when assigned to integer

nullptr

C++ 11

A pointer to nothing

- if a pointer doesn't point to anything, set it to `nullptr`
  - useful to e.g. mark the end of a linked data structure
  - or absence of an optional function argument (pointer)
- same as setting it to 0 or `NULL` (before C++ 11)
- triggers compilation error when assigned to integer

Example code

```
int* ip = nullptr;
int i = NULL;       // compiles, bug?
int i = nullptr;    // ERROR
```

Dynamic arrays using C

C++ 98

```
#include <cstdlib>
#include <cstring>

int *bad;            // pointer to random address
int *ai = nullptr; // better, deterministic, testable

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

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

free(ai); // release memory
```

Good practice: Don't use C's memory management

Use `std::vector` and friends or smart pointers

Manual dynamic arrays using C++

C++ 98

```
#include <cstdlib>
#include <cstring>

// allocate array of 10 ints
int* ai = new int[10];   // uninitialized
int* ai = new int[10]{}; // zero-initialized

delete[] ai; // release array memory

// allocate a single int
int* pi = new int;
int* pi = new int{};
delete pi; // release scalar memory
```

Good practice: Don't use manual memory management

Use `std::vector` and friends or smart pointers

Scopes / namespaces

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

Scope

C++ 98

Definition

Portion of the source code where a given name is valid

Typically :

- simple block of code, within {}
- function, class, namespace
- the global scope, i.e. translation unit (.cpp file + all includes)

Example

```cpp
{ int a;
  { int b;
  } // end of b scope
} // end of a scope
```

Scope and lifetime of variables

C++ 98

Variable life time

- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope

Good practice: Initialisation

- Initialise variables when allocating them!
- This prevents bugs reading uninitialised memory

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

Memory layout

	0x3010
	0x300C
	0x3008
	0x3004
$a = 1$	0x3000

Scope and lifetime of variables

C^{++} 98

Variable life time

- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope

Good practice: Initialisation

- Initialise variables when allocating them!
- This prevents bugs reading uninitialised memory

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

Memory layout

b[3] = ?	0x3010
b[2] = ?	0x300C
b[1] = ?	0x3008
b[0] = ?	0x3004
a = 1	0x3000

Scope and lifetime of variables

Variable life time

- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope

Good practice: Initialisation

- Initialise variables when allocating them!
- This prevents bugs reading uninitialised memory

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

Memory layout

b[3] = ?	0x3010
b[2] = ?	0x300C
b[1] = ?	0x3008
b[0] = 1	0x3004
a = 1	0x3000

Scope and lifetime of variables

C^{++} 98

Variable life time

- Variables are (statically) allocated when defined
- Variables are freed at the end of a scope

Good practice: Initialisation

- Initialise variables when allocating them!
- This prevents bugs reading uninitialised memory

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

Memory layout

?	0x3010
?	0x300C
?	0x3008
1	0x3004
a = 1	0x3000

Namespaces

C++ 98

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

```cpp
int a;
namespace n {
  int a;    // no clash
}
namespace p {
  int a;    // no clash
  namespace inner {
    int a; // no clash
  }
}
void f() {
  n::a = 3;
}
namespace p { // reopen p
  void f() {
    p::a = 6;
    a = 6;  //same as above
    ::a = 1;
    p::inner::a = 8;
    inner::a = 8;
    n::a = 3;
  }
}
using namespace p::inner;
void g() {
  a = -1; // err: ambiguous
}
```

Nested namespaces

C++ 17

C++ 98: Old way to declare nested namespaces

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

C++ 17: Nested declaration

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

C++ 17: Namespace alias

```cpp
namespace ABC = A::B::C;
```

Class and enum types

2 Language basics

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

struct

C++ 98

"members" grouped together under one name

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

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

Individual teacher = {
  45, 67.0f
};
```

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

struct

C++ 98

“members” grouped together under one name

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

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

Individual teacher = {
  45, 67.0f
};
```

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

Memory layout

				0x3010
				0x300C
				0x3008
				0x3004
				0x3000

struct

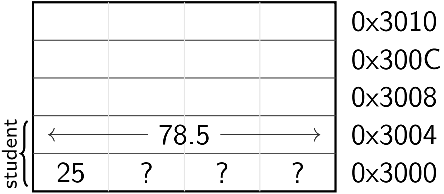
C++ 98

"members" grouped together under one name

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

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

Individual teacher = {
  45, 67.0f
};
```

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

Memory layout



struct

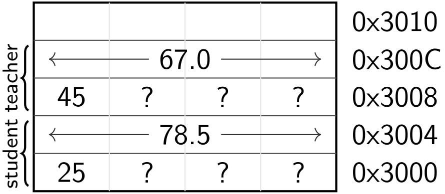
C++ 98

"members" grouped together under one name

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

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

Individual teacher = {
  45, 67.0f
};
```

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

Memory layout



struct

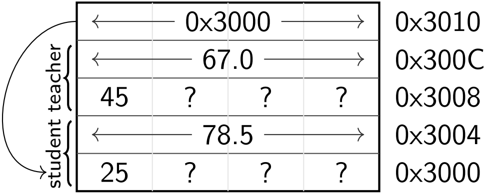
C++ 98

"members" grouped together under one name

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

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

Individual teacher = {
  45, 67.0f
};
```

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

Memory layout



union

C^{++} 98

“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
```

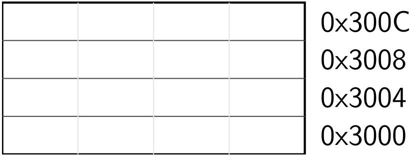
union

C++ 98

"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
```

Memory layout



union

C++ 98

"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
```

Memory layout

				0x300C
				0x3008
				0x3004
← d1 259200 →				0x3000

union

C^{++} 98

"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
```

Memory layout

				0x300C
				0x3008
← d2	72 →	?	?	0x3004
⟵	d1	259200	⟶	0x3000

union

C++ 98

"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
```

Memory layout

				0x300C
d3 3	?	?	?	0x3008
← d2 72 →		?	?	0x3004
← d1 259200 →				0x3000

union

C^{++} 98

“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
```

Memory layout

				0x300C
d3 3	?	?	?	0x3008
← d2	72 →	?	?	0x3004
d1 3	?	?	?	0x3000

union

C++ 98

"members" packed together at same memory location

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

Memory layout

				0x300C
d3 3	?	?	?	0x3008
← d2	72 →	?	?	0x3004
d1 3	?	?	?	0x3000

Good practice: Avoid unions

- Starting with C++ 17: prefer `std::variant`

Enums

C^{++} 98

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

```cpp
enum VehicleType {

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

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

Scoped enumeration, aka enum class

C++ 11

Same syntax as enum, with scope

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

Scoped enumeration, aka enum class

C++ 11

Same syntax as enum, with scope

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

Only advantages

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

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

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

C++ 11

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

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

More sensible example

C^{++} 11

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

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

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

More sensible example

C++ 11

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

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

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

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

```

```cpp
Shape t;
t.type =
  Shapetype::Rectangle;
t.rect.width = 3;
t.rect.height = 4;
```

typedef and using C^{++} 98 / C^{++} 11

Used to create type aliases

C^{++} 98

```cpp
typedef std::uint64_t myint;
myint count = 17;
typedef float position[3];
```

C^{++} 11

```cpp
using myint = std::uint64_t;
myint count = 17;
using position = float[3];

template <typename T> using myvec = std::vector<T>;
myvec<int> myintvec;
```

References

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 C^{++} 98

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

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
a.x = 4;            // fine
```

Pointers vs References

C^{++} 98

Specificities of reference

- Natural syntax
- Cannot be `nullptr`
- Must be assigned when defined, cannot be reassigned
- References to temporary objects must be `const`

Advantages of pointers

- Can be `nullptr`
- Can be initialized after declaration, can be reassigned

Pointers vs References

C^{++} 98

Specificities of reference

- Natural syntax
- Cannot be `nullptr`
- Must be assigned when defined, cannot be reassigned
- References to temporary objects must be `const`

Advantages of pointers

- Can be `nullptr`
- Can be initialized after declaration, can be reassigned

Good practice: References

- Prefer using references instead of pointers
- Mark references `const` to prevent modification

Functions

2 Language basics

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

Functions

C++ 98

```cpp
// with return type
int square(int a) {
  return a * a;
}

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

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

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

Functions

C++ 98

```cpp
// with return type
int square(int a) {
  return a * a;
}

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

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

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

Functions and references to returned values

```cpp
int result = square(2);
int & temp = square(2);        // Not allowed
int const & temp2 = square(2); // OK
```

Function default arguments

C^{++} 98

```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 a = 2,
        int b = 2) {
  return a + b;
}
// add() == 4
// add(3) == 5
```

Functions: parameters are passed by value

C++ 98

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

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

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

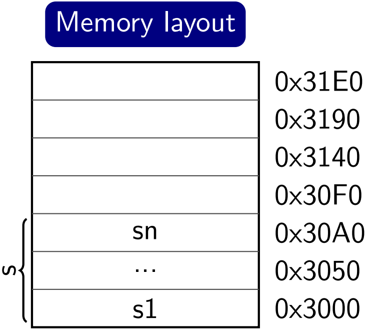
Memory layout

	0x31E0
	0x3190
	0x3140
	0x30F0
	0x30A0
	0x3050
	0x3000

Functions: parameters are passed by value

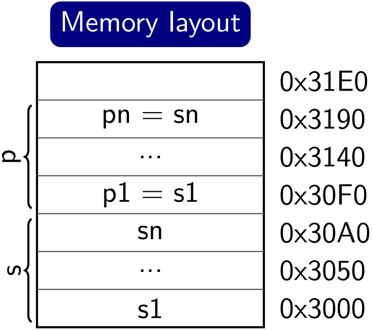
C++ 98

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

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

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



Functions: parameters are passed by value C^{++} 98

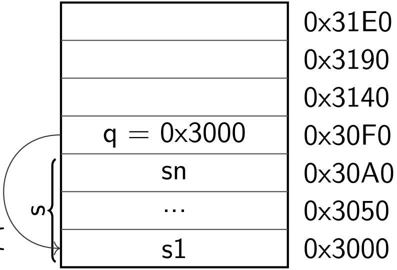
```cpp
struct BigStruct {...};
BigStruct s;

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

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



Functions: parameters are passed by value

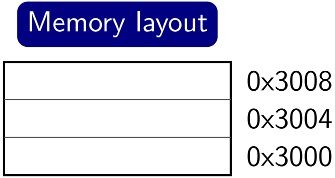

```cpp
struct BigStruct {...};
BigStruct s;

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

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

Memory layout



Functions: pass by value or reference?


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

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

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



Functions: pass by value or reference?


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

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

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

Memory layout

	0x3008
	0x3004
s.a = 1	0x3000

Functions: pass by value or reference?


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

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

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

Memory layout

	0x3008
p.a = 1	0x3004
s.a = 1	0x3000

Functions: pass by value or reference?


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

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

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

Memory layout

	0x3008
p.a = 2	0x3004
s.a = 1	0x3000

Functions: pass by value or reference? C^{++} 98

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

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

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

Memory layout

	0x3008
	0x3004
s.a = 1	0x3000

Functions: pass by value or reference?


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

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

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

Memory layout

	0x3008
q = 0x3000	0x3004
s.a = 1	0x3000

Functions: pass by value or reference?


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

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

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

Memory layout

	0x3008
q = 0x3000	0x3004
s.a = 2	0x3000

Functions: pass by value or reference?


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

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

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

Memory layout

	0x3008
	0x3004
s.a = 2	0x3000

Pass by value, reference or pointer

C++ 98

Different ways to pass arguments to a function

- By default, arguments are passed by value (= copy)
  good for small types, e.g. numbers
- Use references for parameters to avoid copies
  good for large types, e.g. objects
- Use `const` for safety and readability whenever possible

Pass by value, reference or pointer

C++ 98

Different ways to pass arguments to a function

- By default, arguments are passed by value (= copy)
  good for small types, e.g. numbers
- Use references for parameters to avoid copies
  good for large types, e.g. objects
- Use `const` for safety and readability whenever possible

Syntax

```
struct T {...}; T a;
void fVal(T value);              fVal(a);    // by value
void fRef(const T &value); fRef(a);    // by reference
void fPtr(const T *value); fPtr(&a);   // by pointer
void fWrite(T &value);          fWrite(a); // non-const ref
```

Overloading

C++ 98

Overloading

- We can have multiple functions with the same name
  - Must have different parameter lists
  - A different return type alone is not allowed
  - Form a so-called “overload set”
- Default arguments can cause ambiguities

```
int sum(int b);              // 1
int sum(int b, int c);       // 2, ok, overload
// float sum(int b, int c); // disallowed
sum(42); // calls 1
sum(42, 43); // calls 2
int sum(int b, int c, int d = 4); // 3, overload
sum(42, 43, 44); // calls 3
sum(42, 43);       // error: ambiguous, 2 or 3
```

Functions

C++ 98

Exercise: Functions

Familiarise yourself with pass by value / pass by reference.

- Go to `exercises/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

C^{++} 98

Good practice: Write readable functions

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

Example: Good

```cpp
/// Count number of dilepton events in data.
/// \param d Dataset to search.
unsigned int countDileptons(Data &d) {
  selectEventsWithMuons(d);
  selectEventsWithElectrons(d);
  return d.size();
}
```

Functions: good practices

Example: don't! Everything in one long function

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

  // Step 2: muons
  for (....) {
    if (...) {
      data.erase(...);
    }
  }
  // Step 3: electrons
  for (....) {
    if (...) {
      data.erase(...);
    }
  }

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

  return counter;
}
```

Operators

2 Language basics

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

Operators(1)

C++ 98

Binary and Assignment Operators

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

Operators(1)

Binary and 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 Operators

```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)

C++ 98

Binary and Assignment Operators

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

Increment / Decrement Operators

Use wisely

```
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(2)

C++ 98

Bitwise and Assignment Operators

```cpp
unsigned i = 0xee & 0x55;   // 0x44
i |= 0xee;                  // 0xee
i ^= 0x55;                  // 0xbb
unsigned j = ~0xee;         // 0xffffff11
unsigned k = 0x1f << 3;     // 0xf8
unsigned l = 0x1f >> 2;     // 0x7
```

Operators(2)

Bitwise and Assignment Operators

```
unsigned i = 0xee & 0x55;   // 0x44
i |= 0xee;                  // 0xee
i ^= 0x55;                  // 0xbb
unsigned j = ~0xee;         // 0xffffff11
unsigned k = 0x1f << 3;     // 0xf8
unsigned l = 0x1f >> 2;     // 0x7
```

Logical Operators

```
bool a = true;
bool b = false;
bool c = a && b;     // false
bool d = a || b;     // true
bool e = !d;         // false
```

Operators(3)

C++ 98

Comparison Operators

```
bool a = (3 == 3);  // true
bool b = (3 != 3);  // false
bool c = (4 <  4);  // false
bool d = (4 <= 4);  // true
bool e = (4 >  4);  // false
bool f = (4 >= 4);  // true
auto g = (5 <=> 5); // C++20 (later)
```

Operators(3)

C++ 98

Comparison Operators

```cpp
bool a = (3 == 3);  // true
bool b = (3 != 3);  // false
bool c = (4 <  4);  // false
bool d = (4 <= 4);  // true
bool e = (4 >  4);  // false
bool f = (4 >= 4);  // true
auto g = (5 <=> 5); // C++20 (later)
```

Precedences

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

Details can be found on cppreference

Operators(3)

C++ 98

Comparison Operators

```
bool a = (3 == 3);  // true
bool b = (3 != 3);  // false
bool c = (4 <  4);  // false
bool d = (4 <= 4);  // true
bool e = (4 >  4);  // false
bool f = (4 >= 4);  // true
auto g = (5 <=> 5); // C++20 (later)
```

Precedences Avoid

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

Details can be found on cppreference

Operators(3) C^{++} 98

Comparison Operators

```
bool a = (3 == 3);  // true
bool b = (3 != 3);  // false
bool c = (4 <  4);  // false
bool d = (4 <= 4);  // true
bool e = (4 >  4);  // false
bool f = (4 >= 4);  // true
auto g = (5 <=> 5); // C++20 (later)
```

Precedences **Avoid - use parentheses**

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

Details can be found on cppreference

Control structures

Control structures: if

C++ 98

if syntax

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

- The `else` and `else if` clauses are optional
- The `else if` clause can be repeated
- Braces are optional if there is a single statement

Control structures: if

C++ 98

Practical example

```
int collatz(int a) {
  if (a <= 0) {
    std::cout << "not supported\n";
    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

C++ 98

Syntax

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

- If test is `true` `expression1` is returned
- Else, `expression2` is returned

Control structures: conditional operator C++ 98

Syntax

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

- If test is `true` `expression1` is returned
- Else, `expression2` is returned

Practical example

```cpp
const int charge = isLepton ? -1 : 0;
```

Control structures: conditional operator

C++ 98

Syntax

`test ? expression1 : expression2;`

- If test is `true` `expression1` is returned
- Else, `expression2` is returned

Practical example

```cpp
const int charge = isLepton ? -1 : 0;
```

Do not abuse it

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

- Explicit `if`s are generally easier to read
- Use the ternary operator with short conditions and expressions
- Avoid nesting

Control structures: switch

C++ 98

Syntax

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

- The `break` statement is not mandatory but...
- Cases are entry points, not independent pieces
- Execution “falls through” to the next case without a `break`!
- The `default` case may be omitted

Control structures: switch

C++ 98

Syntax

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

- The `break` statement is not mandatory but...
- Cases are entry points, not independent pieces
- Execution "falls through" to the next case without a `break`!
- The `default` case may be omitted

Use break

Avoid `switch` statements with fall-through cases

Control structures: switch

C++ 98

Practical example

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

[[fallthrough]] attribute C^{++} 17

New compiler warning

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

C^{++} 17

```cpp
switch (c) {
  case 'a':
    f();     // Warning emitted
  case 'b': // Warning probably suppressed
  case 'c':
    g();
    [[fallthrough]]; // Warning suppressed
  case 'd':
    h();
}
```

Init-statements for if and switch C^{++} 17

Purpose

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

C^{++} 17

```
if (Value val = GetValue(); condition(val)) {
  f(val); // ok
} else
  g(val); // ok
h(val);   // error, no `val` in scope here
```

Init-statements for if and switch

C++ 17

Purpose

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

C++ 17

```
if (Value val = GetValue(); condition(val)) {
  f(val); // ok
} else
  g(val); // ok
h(val);    // error, no `val` in scope here
```

C++ 98

Don't confuse with a variable declaration as condition:

```
if (Value* val = GetValuePtr())
  f(*val);
```

Control structures: for loop

C++ 98

for loop syntax

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

- Multiple initializations / increments are comma separated
- Initializations can contain declarations
- Braces are optional if loop body is a single statement

Control structures: for loop

C++ 98

for loop syntax

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

- Multiple initializations / increments are comma separated
- Initializations can contain declarations
- Braces are optional if loop body is a single statement

Practical example

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

Control structures: for loop

C^{++} 98

for loop syntax

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

- Multiple initializations / increments are comma separated
- Initializations can contain declarations
- Braces are optional if loop body is a single statement

Practical example

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

Good practice: Don't abuse the for syntax

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

Range-based loops

C++ 11

Reason of being

- Simplifies loops over "ranges" tremendously
- Especially with STL containers and ranges

Syntax

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

Example code

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

Init-statements for range-based loops

C++ 20

Purpose

Allows to limit variable scope in range-based loops

C++ 17

```
std::array data = {"hello", ",", "world"};
std::size_t i = 0;
for (auto& d : data) {
  std::cout << i++ << ' ' << d << '\n';
}
```

C++ 20

```
for (std::size_t i = 0; auto const & d : data) {
  std::cout << i++ << ' ' << d << '\n';
}
```

Control structures: while loop

C++ 98

while loop syntax

```
while(condition) {
  statements;
}

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

- Braces are optional if the body is a single statement

Control structures: while loop

C++ 98

while loop syntax

```
while(condition) {
  statements;
}

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

- Braces are optional if the body is a single statement

Bad example

```
while (n != 1)
  if (0 == n%2) n /= 2;
  else n = 3 * n + 1;
```

Control structures: jump statements C^{++} 98

break	Exits the loop and continues after it
continue	Goes immediately to next loop iteration
return	Exits the current function
goto	Can jump anywhere inside a function, avoid!

Control structures: jump statements

break	Exits the loop and continues after it
continue	Goes immediately to next loop iteration
return	Exits the current function
goto	Can jump anywhere inside a function, avoid!

Bad example

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

Control structures

C++ 11

Exercise: Control structures

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

- Go to `exercises/control`
- Look at `control.cpp`
- Compile it (`make`) and run the program (`./control`)
- Work on the tasks that you find in `README.md`

Headers and interfaces

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

C++ 98

Interface

Set of declarations defining some functionality

- Put in a so-called “header file”
- The implementation exists somewhere else

Header: hello.hpp

```
void printHello();
```

Usage: myfile.cpp

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

Preprocessor

C++ 98

```cpp
// file inclusion
#include "hello.hpp"
// macro constants and function-style macros
#define MY_GOLDEN_NUMBER 1746
#define CHECK_GOLDEN(x) if ((x) != MY_GOLDEN_NUMBER) \
  std::cerr << #x " was not the golden number\n";
// compile time or platform specific configuration
#if defined(USE64BITS) || defined(__GNUG__)
  using myint = std::uint64_t;
#elif
  using myint = std::uint32_t;
#endif
```

Preprocessor

C++ 98

```
// file inclusion
#include "hello.hpp"
// macro constants and function-style macros
#define MY_GOLDEN_NUMBER 1746
#define CHECK_GOLDEN(x) if ((x) != MY_GOLDEN_NUMBER) \
  std::cerr << #x " was not the golden number\n";
// compile time or platform specific configuration
#if defined(USE64BITS) || defined(__GNUG__)
  using myint = std::uint64_t;
#elif
  using myint = std::uint32_t;
#endif
```

Good practice: Use preprocessor only in very restricted cases

- Conditional inclusion of headers
- Customization for specific compilers/platforms

Header include guards

C++ 98

Problem: redefinition by accident

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

Include guards

```cpp
#ifndef MY_HEADER_INCLUDED
#define MY_HEADER_INCLUDED
... // header file content
#endif
```

Pragma once (non-standard)

```cpp
#pragma once
... // header file content
```

Header / source separation

C^{++} 98

Good practice: Headers and source files

- Headers should contain declarations of functions / classes
  - Only create them if interface is used somewhere else
- Might be included/compiled many times
- Good to keep them short
- Minimise *#include* statements
- Put long code in implementation files. Exceptions:
  - Short functions
  - Templates and constexpr functions

Auto keyword

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

C++ 11

Reason of being

- Many type declarations are redundant
- They are often a source for compiler warnings and errors
- Using auto prevents unwanted/unnecessary type conversions

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

Auto keyword

C^{++} 11

Reason of being

- Many type declarations are redundant
- They are often a source for compiler warnings and errors
- Using auto prevents unwanted/unnecessary type conversions

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

Practical usage

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

Loops, references, auto

C++ 98

Exercise: Loops, references, auto

Familiarise yourself with range-based for loops and references

- Go to `exercises/loopsRefsAuto`
- Look at `loopsRefsAuto.cpp`
- Compile it (`make`) and run the program (`./loopsRefsAuto`)
- Work on the tasks that you find in `loopsRefsAuto.cpp`

Object orientation (OO)

Objects and Classes

3 Object orientation (OO)

- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Operator overloading
- Function objects

What are classes and objects

C++ 98

Classes (or "user-defined types")

C structs on steroids

- with inheritance
- with access control
- including methods (aka. member functions)

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

My first class

C++ 98

```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();
```

MyFirstClass
int a;
void squareA();
int sum(int b);

Separating the interface

C++ 98

Header: MyClass.hpp

```cpp
#pragma once
struct MyClass {
  int a;
  void squareA();
};
```

User 1: main.cpp

```cpp
#include "MyClass.hpp"
int main() {
  MyClass mc;
  ...
}
```

Implementation: MyClass.cpp

```cpp
#include "MyClass.hpp"
void MyClass::squareA() {
  a *= a;
}
```

User 2: fun.cpp

```cpp
#include "MyClass.hpp"
void f(MyClass& mc) {
  mc.squareA();
}
```

Implementing methods

C++ 98

Good practice: Implementing methods

- usually in .cpp, outside of class declaration
- using the class name as “namespace”
- short member functions can be in the header
- some functions (templates, constexpr) must be in the header

```cpp
#include "MyFirstClass.hpp"

void MyFirstClass::squareA() {
  a *= a;
}
int MyFirstClass::sum(int b) {
  return a + b;
}
```

Method overloading

C++ 98

The rules in C++

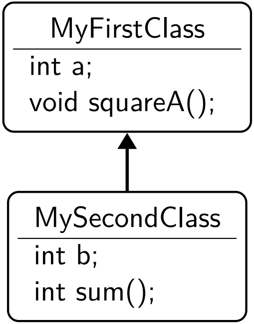
- overloading is authorized and welcome
- signature is part of the method identity
- but not the return type

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

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

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

Inheritance

First inheritance

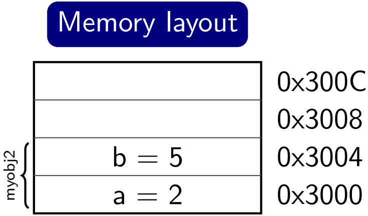

```cpp
struct MyFirstClass {
  int a;
  void squareA() { a *= a; }
};
struct MySecondClass :
  MyFirstClass {
  int b;
  int sum() { return a + b; }
};

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



First inheritance

C^{++} 98

```
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

C++ 98

public / private keywords

`private` allows access only within the class

`public` allows access from anywhere

- The default for class is `private`
- The default for struct is `public`

```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

C++ 98

public / private keywords

private allows access only within the class

public allows access from anywhere

- The default for class is private
- The default for struct is public

```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)

C++ 98

Solution is protected keyword

Gives access to classes inheriting from base class

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

```cpp
class MySecondClass :
  public MyFirstClass {
public:
  int sum() {
    return a + b;
  }
private:
  int b;
};
```

Managing inheritance privacy

C++ 98

Inheritance can be public, protected or private

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

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

Managing inheritance privacy

C++ 98

Inheritance can be public, protected or private

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

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

Net result for external code

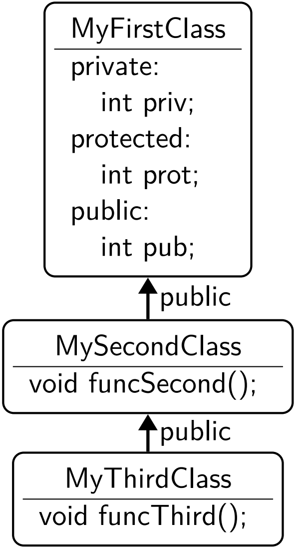
- only public members of public inheritance are accessible

Net result for code in derived classes

- only public and protected members of public and protected parents are accessible

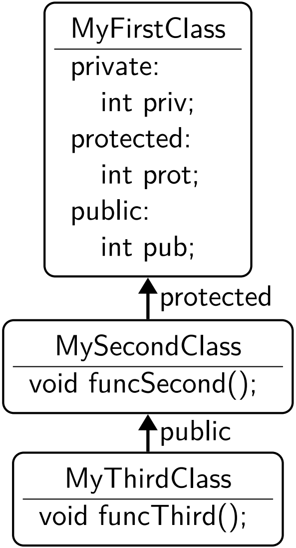
Managing inheritance privacy - public

C^{++} 98



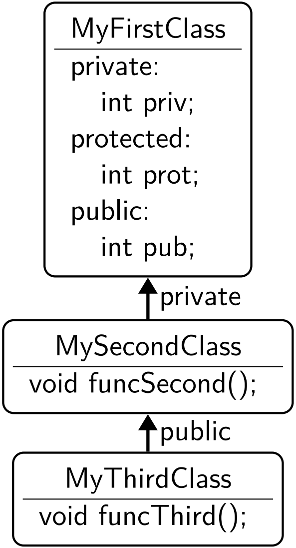

```cpp
void funcSecond() {
  int a = priv;   // Error
  int b = prot;   // OK
  int c = pub;    // OK
}
void funcThird() {
  int a = priv;   // Error
  int b = prot;   // OK
  int c = pub;    // OK
}
void extFunc(MyThirdClass t) {
  int a = t.priv; // Error
  int b = t.prot; // Error
  int c = t.pub;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

Managing inheritance privacy - protected




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

Managing inheritance privacy - private




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

Constructors/destructors

3. Object orientation (OO)
   - Objects and Classes
   - Inheritance
   - Constructors/destructors
   - Static members
   - Allocating objects
   - Advanced OO
   - Operator overloading
   - Function objects

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 C {
public:
  C();
  C(int a);
  ~C();
  ...
protected:
  int a;
};
```

```cpp
// note: special notation for
// initialization of members
C::C() : a(0) {}

C::C(int a) : a(a) {}

C::~C() {}
```

Class constructors and destructors

C++ 98

```
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

C++ 98

```cpp
struct First {
  int a;
  First() {} // leaves a uninitialized
  First(int a) : a(a) {}
};
struct Second : First {
  int b;
  Second();
  Second(int b);
  Second(int a, int b);
};
Second::Second() : First(), b(0) {}
Second::Second(int b) : b(b) {} // First() implicitly
Second::Second(int a, int b) : First(a), b(b) {}
```

Copy constructor

C++ 11

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

C++ 11

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

```
struct C {
  C();
  C(const C &other);
};
```

Copy constructor

C++ 11

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

```
struct C {
  C();
  C(const C &other);
};
```

Good practice: The rule of 3/5 (C++ 98/11) - cppreference

if a class needs a custom destructor, a copy/move constructor or a copy/move assignment operator, it should have all three/five.

Class Constructors and Destructors


```cpp
class Vector {
public:
  Vector(int n);
  Vector(const Vector &other);
  ~Vector();
private:
  int len; int* data;
};
Vector::Vector(int n) : len(n) {
  data = new int[n];
}




Vector::~Vector() { delete[] data; }
```

Class Constructors and Destructors


```cpp
class Vector {
public:
  Vector(int n);
  Vector(const Vector &other);
  ~Vector();
private:
  int len; int* data;
};
Vector::Vector(int n) : len(n) {
  data = new int[n];
}
Vector::Vector(const Vector &other) : len(other.len) {
  data = new int[len];
  std::copy(other.data, other.data + len, data);
}
Vector::~Vector() { delete[] data; }
```

Explicit unary constructor

Concept

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

Example - godbolt

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

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

Explicit unary constructor

C++ 98

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

C++ 11

Idea

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

Details

- without a user-defined constructor, a default one is provided
- any user-defined constructor disables the default one
- but the default one can be requested explicitly
- rule can be more subtle depending on data members

Practically

```
Class() = default; // provide default if possible
Class() = delete;  // disable default constructor
```

Delegating constructor

C++ 11

Idea

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

Practically

```
struct Delegate {
  int m_i;
  explicit Delegate(int i) : m_i(i) {
    ... complex initialization ...
  }
  Delegate() : Delegate(42) {}
};
```

Constructor inheritance

C++ 11

Idea

- avoid having to redeclare parent's constructors
- by stating that we inherit all parent constructors
- derived class can add more constructors

Practically

```
struct Base {
  explicit Base(int a);  // ctor 1
};
struct Derived : Base {
  using Base::Base;
  Derived(int a, int b); // ctor 2
};
Derived d{5};     // calls ctor 1
Derived d{5, 6}; // calls ctor 2
```

Member initialization

C++ 11

Idea

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

Practically

```
struct Base {
  int a{5}; // or: int a = 5;
            Base() = default;
  explicit Base(int _a) : a(_a) {}
};
struct Derived : Base {
  int b{6};
  using Base::Base;
};
Derived d1;     // a = 5, b = 6
Derived d2{7}; // a = 7, b = 6
```

Calling constructors

C++ 11

After object declaration, arguments within {}

```
struct A {
  int i;
  float f;
  A();
  A(int);
  A(int, int);
};

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)
```

Calling constructors the old way

C++ 98

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

```cpp
struct A {
  int i;
  float f;
  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); // A::A(int), comma operator!
A a = {1,2}; // not allowed
```

Constructing arrays and vectors

C++ 11

List of items given within {}

```
int ip[3]{1,2,3};
int* ip = new int[3]{1,2,3}; // not allowed in C++98
std::vector<int> v{1,2,3};   // same
```

Static members

Static members

C++ 98

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

Static.hpp

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

Static members

C++ 98

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

Static.cpp

```cpp
#include "Static.hpp"
int Text::callsToUpper = 0; // required before C++17

std::string Text::upper(std::string lower) {
  callsToUpper++;
  // convert lower to upper case
  // return ...;
}
std::string uppers = Text::upper("my text");
// now Text::callsToUpper is 1
```

Allocating objects

3. Object orientation (OO)
   - Objects and Classes
   - Inheritance
   - Constructors/destructors
   - Static members
   - Allocating objects
   - Advanced OO
   - Operator overloading
   - Function objects

Process memory organization

C++ 98

4 main areas

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

Memory layout

heap
↓ ... ↓
↑ ... ↑
stack
data segment
code segment

The Stack

C^{++} 98

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

C^{++} 98

On the stack

- objects are created on variable definition (constructor called)
- objects are destructed when out of scope (destructor is called)

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

int g() {
  MyFirstClass a; // default constructor called
  ...
}  // destructor called
```

The Heap

C^{++} 98

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
int f() {
  // default constructor called
  MyFirstClass *a = new MyFirstClass;
  delete a; // destructor is called
}
int g() {
  // constructor called
  MyFirstClass *a = new MyFirstClass{3};
} // memory leak !!!
```

Good practice: Prefer smart pointers over new/delete

Prefer smart pointers to manage objects (discussed later)

Array allocation on the heap

Arrays on the heap

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

```cpp
int f() {
  // default constructor called 10 times
  MyFirstClass *a = new MyFirstClass[10];
  ...
  delete[] a; // destructor called 10 times
}
```

Good practice: Prefer containers over new-ed arrays

Prefer containers to manage collections of objects (discussed later)

Advanced OO

3 Object orientation (OO)
- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Operator overloading
- Function objects

this keyword

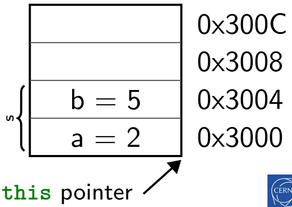
C++ 98

How to know an object's address?

- Sometimes we need the address of the current object
- Or we need to pass our address / a reference to a different entity (for example to implement operators, see later)
- All class methods can use the keyword `this`
  - It returns the address of the current object
  - Its type is `T*` in the methods of a class `T`

```cpp
struct S {
  int a,b;
  // these two are the same:
  int getB() { return b; }       // 5
  int getB() { return this->b; } // 5
  void testAddress() {
    S* addr = this; // 0x3000
  }
} s{2,5};
```

Memory layout



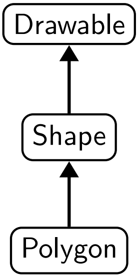
Polymorphism

C++ 98

the concept

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

```cpp
Polygon p;

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

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



Polymorphism

C^{++} 98

the concept

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

```cpp
Polygon p;

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

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

Memory layout

	0x3020
	0x301C
Polygon.nLines	0x3018
...	0x3014
Shape.b	0x3010
Shape.a	0x300C
...	0x3008
Drawable.b	0x3004
Drawable.a	0x3000

Polygon

Polymorphism

the concept

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

```cpp
Polygon p;

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

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

Memory layout

		0x3020
		0x301C
	Polygon.nLines	0x3018
	...	0x3014
	Shape.b	0x3010
	Shape.a	0x300C
Drawable	...	0x3008
Drawable	Drawable.b	0x3004
Drawable	Drawable.a	0x3000

Polymorphism

C^{++} 98

the concept

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

```cpp
Polygon p;

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

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

Memory layout

	0x3020
	0x301C
Polygon.nLines	0x3018
...	0x3014
Shape.b	0x3010
Shape.a	0x300C
...	0x3008
Drawable.b	0x3004
Drawable.a	0x3000

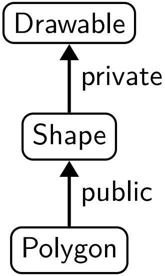
Shape

Inheritance privacy and polymorphism

Only public base classes are visible to outside code

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

```cpp
Polygon p;

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

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



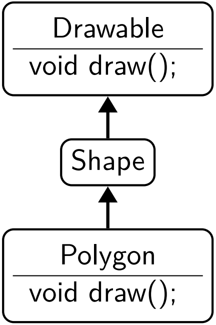
Method overriding

C^{++} 98

the idea

- a method of the parent class can be replaced in a derived class
- but which one is called?

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

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



Virtual methods

C^{++} 98

the concept

- methods can be declared `virtual`
- for these, the most derived object's implementation is used (i.e. the dynamic type behind a pointer/reference)
- for non-virtual methods, the static type of the variable decides

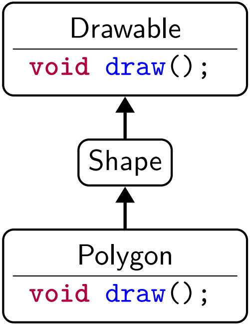
Virtual methods

C++ 98

the concept

- methods can be declared `virtual`
- for these, the most derived object's implementation is used (i.e. the dynamic type behind a pointer/reference)
- for non-virtual methods, the static type of the variable decides

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

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

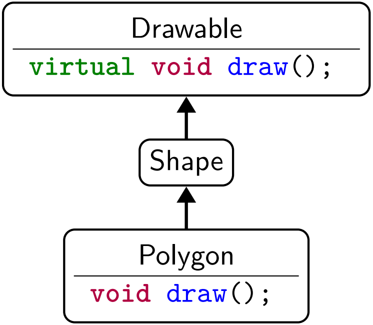


Virtual methods

the concept

- methods can be declared `virtual`
- for these, the most derived object's implementation is used (i.e. the dynamic type behind a pointer/reference)
- for non-virtual methods, the static type of the variable decides

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

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



Virtual methods - implications

C++ 11

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

override keyword

C++ 11

Principle

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

Practically

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

Why was override keyword introduced? C++ 11

To detect the mistake in the following code :

Without override (C++ 98)

```
struct Base {
  virtual void some_func(float);
};
struct Derived : Base {
  void some_func(double); // oops !
};
```

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

Pure Virtual methods

C++ 11

Concept

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

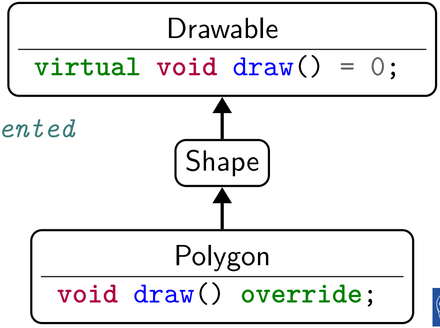
Pure Virtual methods

C++ 11

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();
```



Polymorphism and destruction

Owning base pointers

We sometimes need to maintain owning pointers to base classes:

```
struct Drawable {
    virtual void draw() = 0;
};
Drawable* getImpl();

Drawable* p = getImpl();
p->draw();
delete p;
```

- What happens when p is deleted?
- What if a class deriving from Drawable has a destructor?

Polymorphism and destruction

C++ 98

Owning base pointers

We sometimes need to maintain owning pointers to base classes:

```
struct Drawable {
    virtual void draw() = 0;
};
std::unique_ptr<Drawable> getImpl(); // better API

auto p = getImpl();
p->draw();
```

- What happens when p is deleted?
- What if a class deriving from `Drawable` has a destructor?

Polymorphism and destruction

C++ 11

Virtual destructors

- We can mark a destructor as `virtual`
- This selects the right destructor based on the runtime type

```
struct Drawable {
    virtual ~Drawable() = default;
    virtual void draw() = 0;
};
Drawable* p = getImpl(); // returns derived obj.
p->draw();
delete p; // dynamic dispatch to right destructor
```

Good practice: Virtual destructors

If you expect users to inherit from your class and override methods (i.e. use your class polymorphically), declare its destructor `virtual`

Pure Abstract Class aka Interface

C++ 11

Definition of pure abstract class

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

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

Drawable
virtual void draw() = 0;

Overriding overloaded methods

C++ 98

Concept

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

```cpp
struct BaseClass {
  virtual int foo(std::string);
  virtual int foo(int);
};
struct DerivedClass : BaseClass {
  using BaseClass::foo;
  int foo(std::string) override;
};
DerivedClass dc;
dc.foo(4);        // error if no using
```

Polymorphism

Exercise: Polymorphism

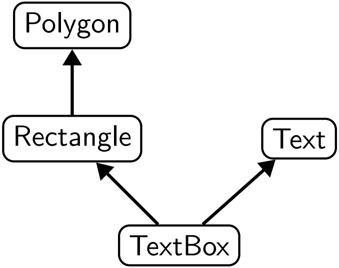
- go to `exercises/polymorphism`
- look at the code
- open trypoly.cpp
- create a Pentagon, call its perimeter method
- create a Hexagon, call its perimeter method
- create a Hexagon, call its parent's perimeter method
- retry with virtual methods

Multiple Inheritance

C++ 98

Concept

- one class can inherit from multiple parents




```cpp
class TextBox :
  public Rectangle, Text {
  // inherits from both
  // publicly from Rectangle
  // privately from Text
}
```

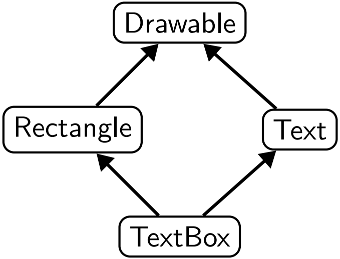
The diamond shape

Definition

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

Problem

- are the members of the grand parent replicated?



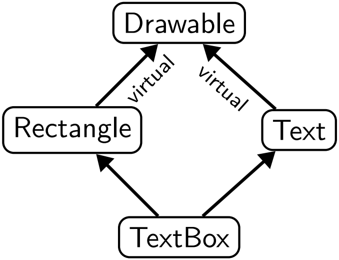
Virtual inheritance

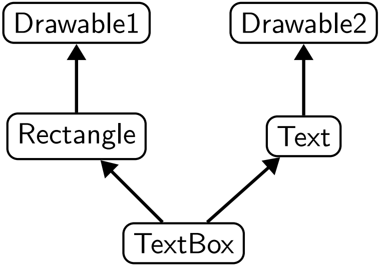
C++ 98

Solution

- inheritance can be `virtual` or not
  - `virtual` inheritance will "share" parents
  - standard inheritance will replicate them
- most derived class will call the virtual base class's constructor

```
class Text : public virtual Drawable {...};
class Rectangle : public virtual Drawable {...};
```





Multiple inheritance advice

C^{++} 98

Good practice: Avoid multiple inheritance

- Except for inheriting from interfaces (=no data members)
- And for rare special cases

Absolutely avoid diamond-shape inheritance

- This is a sign that your architecture is not correct
- In case you are tempted, think twice and change your mind

Virtual inheritance

C++ 98

Exercise: Virtual inheritance

- go to `exercisescode/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

Operator overloading

3 Object orientation (OO)

- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Operator overloading
- Function objects

Operator overloading example

C++ 98

```
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)
```

Operator overloading

Defining operators for 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 operators from below)

Expression	As member	As non-member
@a	(a).operator@()	operator@(a)
a@b	(a).operator@(b)	operator@(a,b)
a=b	(a).operator=(b)	cannot be non-member
a(b...)	(a).operator()(b...)	cannot be non-member
a[b]	(a).operator[](b)	cannot be non-member
a->	(a).operator->()	cannot be non-member
a@	(a).operator@(0)	operator@(a,0)

```
possible operators: + - * / % ^ & | ~ ! = < >
    += -= *= /= %= ^= &= |= << >> >>= <<=
    == != <= >= <=> && || ++ -- , ->* -> () []
```

Why 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 have non-member operators? C++ 98

Symmetry

```cpp
struct Complex {
  float m_real, m_imaginary;
  Complex operator+(float other) {
    return Complex{m_real + other, m_imaginary};
  }
};
Complex c1{2.f, 3.f};
Complex c2 = c1 + 4.f;  // ok
Complex c3 = 4.f + c1;  // not ok !!
Complex operator+(float a, const Complex& obj) {
  return Complex{a + obj.m_real, obj.m_imaginary};
}
```

Other reason to have non-member operators? C++ 98

Extending existing classes

```
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)'
```

Chaining operators

C++ 98

In general, return a reference to the left value

```cpp
struct Complex {
  float m_real, m_imaginary;
  Complex& operator=( const Complex& other ) {
    m_real = other.m_real;
    m_imaginary = other.m_imaginary;
    return *this;
  }
};
Complex c1{2.f, 3.f};
Complex c2, c3;
// right to left associativity
c3 = c2 = c1;
// left to right associativity
std::cout << c1 << c2 << c3 << std::endl;
```

Friend declarations

C++ 98

Concept

- Functions/classes can be declared `friend` within a class scope
- They gain access to all private/protected members
- Useful for operators such as `a + b`
- Don't abuse friends to go around a wrongly designed interface
- Avoid unexpected modifications of class state in a friend!

operator+ as a friend

```cpp
class Complex {
  float m_r, m_i;
  friend Complex operator+(Complex const & a, Complex const & b);
public:
  Complex ( float r, float i ) : m_r(r), m_i(i) {}
};
Complex operator+(Complex const & a, Complex const & b) {
  return Complex{ a.m_r+b.m_r, a.m_i+b.m_i };
}
```

Operators

C++ 98

Exercise: Operators

Write a simple class representing a fraction and pass all tests

- go to `exercises/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

Function objects

3 Object orientation (OO)

- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Operator overloading
- Function objects

Function objects

C++ 98

Concept

- also known as functors (no relation to functors in math)
- a class that implements `operator()`
- allows to use objects in place of functions
- with constructors and data members

```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
```

Function objects

C++ 20

Function objects as function arguments - godbolt

```cpp
int count_if(const auto& range, auto predicate) {
  int count = 0;                // ↑ template (later)
  for (const auto& e : range)
    if (predicate(e)) count++;
  return count;
}
struct IsBetween {
  int lower, upper;
  bool operator()(int value) const {
    return lower < value && value < upper;
  }
};
int arr[]{1, 2, 3, 4, 5, 6, 7};
std::cout << count_if(arr, IsBetween{2, 6}); // 3
// prefer: std::ranges::count_if
```

Core modern C^{++}

1. History and goals
2. Language basics
3. Object orientation (OO)
4. Core modern C^{++}
   - Constness
   - Exceptions
   - Templates
   - Lambdas
   - The STL
   - RAII and smart pointers
5. Useful tools

Constness

4 Core modern C++

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

Constness

C++ 98

The const keyword

- indicates that the element to the left is constant
  - when nothing on the left, applies to the right
- this element won't be modifiable in the future
- this is all checked at compile time

```cpp
int const i = 6;
const int i = 6; // equivalent

// error: i is constant
i = 5;

auto const j = i; // works with auto
```

Constness and pointers

C++ 98

```
int a = 1, b = 2;

int const *i = &a; // pointer to const int
*i = 5; // error, int is const
i = &b; // ok, pointer is not const

int * const j = &a; // const pointer to int
*j = 5; // ok, value can be changed
j = &b; // error, pointer is const

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

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

int const & const l = a; // compile error
```

Member function constness

C++ 98

The const keyword for member functions

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

```
struct Example {
  void foo() const {
    // type of 'this' is 'Example const*'
    data = 0; // Error: member function is const
  }
  void foo() { // ok, overload
    data = 1;  // ok, 'this' is 'Example*'
  }
  int data;
};
Example const e1; e1.foo(); // calls     const foo
Example       e2; e2.foo(); // calls non-const foo
```

Member function constness

C++ 98

Constness is part of the type

- `T const` and `T` are different types
- but: `T` is automatically converted to `T const` when needed

```cpp
void change(int & a);
void read(int const & a);

int a = 0;
int const b = 0;

change(a); // ok
change(b); // error
read(a);   // ok
read(b);   // ok
```

constness

C++ 98

Exercise: Constness

- go to `exercises/constness`
- open constplay.cpp
- try to find out which lines won't compile
- check your guesses by compiling for real

Exceptions

4 Core modern C++

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

Exceptions

Purpose

- 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
try {
  process_data(f);
} catch (const
  std::out_of_range& e) {
  std::cerr << e.what();
}
```

```cpp
void process_data(file &f) {
  ...
  if (i >= buffer.size())
    throw std::out_of_range{
      "buf overflow"};
}
```

Exceptions

Throwing exceptions

- objects of any type can be thrown (even e.g. `int`)

Good practice: Throwing exceptions

- prefer throwing standard exception classes
- throw objects by value

```cpp
#include <stdexcept>
void process_data(file& f) {
  if (!f.open())
    throw std::invalid_argument{"stream is not open"};
  auto header = read_line(f); // may throw an IO error
  if (!header.starts_with("BEGIN"))
    throw std::runtime_error{"invalid file content"};
  std::string body(f.size()); // may throw std::bad_alloc
  ...
}
```

Exceptions

C++ 98

Standard exceptions

- `std::exception`, defined in header `<exception>`
  - Base class of all standard exceptions
  - Get error message: `virtual const char* what() const;`
  - Please derive your own exception classes from this one
- From `<stdexcept>`:
  - `std::runtime_error`, `std::logic_error`, `std::out_of_range`, `std::invalid_argument`, ...
  - Store a string: `throw std::runtime_error{"msg"}`
  - You should use these the most
- `std::bad_alloc`, defined in header `<new>`
  - Thrown by standard allocation functions (e.g. `new`)
  - Signals failure to allocate
  - Carries no message
- ...

Exceptions

Catching exceptions

- a catch clause catches exceptions of the same or derived type
- multiple catch clauses will be matched in order
- if no catch clause matches, the exception propagates
- if the exception is never caught, `std::terminate` is called

```cpp
try {
  process_data(f);
} catch (const std::invalid_argument& e) {
  bad_files.push_back(f);
} catch (const std::exception& e) {
  std::cerr << "Failed to process file: " << e.what();
}
```

Good practice: Catching exceptions

- Catch exceptions by const reference

Exceptions

C++ 98

Rethrowing exceptions

- a caught exception can be rethrown inside the catch handler
- useful when we want to act on an error, but cannot handle and want to propagate it

```cpp
try {
  process_data(f);
} catch (const std::bad_alloc& e) {
  std::cerr << "Insufficient memory for " << f.name();
  throw; // rethrow
}
```

Exceptions

Catching everything

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

```cpp

try {
  callUnknownFramework();
} catch(const std::exception& e) {
  // catches std::exception and all derived types
  std::cerr << "Exception: " << e.what() << std::endl;
} catch(...) {
  // catches everything else
  std::cerr << "Unknown exception type" << std::endl;
}
```

Exceptions

C++ 98

Stack unwinding

- all objects on the stack between a `throw` and the matching `catch` are destructed automatically
- this should cleanly release intermediate resources
- make sure you are using the RAII idiom for your own classes

```cpp
class C { ... };
void f() {
  C c1;
  throw exception{};
    // start unwinding
  C c2; // not run
}
void g() {
  C c3; f();
}
```

```cpp
int main() {
  try {
    C c4;
    g();
    cout << "done"; // not run
  } catch(const exception&) {
    // c1, c3 and c4 have been
    // destructed
  }
}
```

Exceptions

C++ 17

Good practice: Exceptions

- 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
  - use `assert` and tests
- *don't* use exceptions to provide alternative/skip return values
  - you can use `std::optional`, `std::variant` or `std::expected` (C++ 23)
  - avoid using the global C-style `errno`
- never throw in destructors
- see also the C++ core guidelines and the ISO C++ FAQ

Exceptions

A more illustrative example

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

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

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

Exceptions

C++ 98

Cost

- exceptions have little cost if no exception is thrown
  - they are recommended to report *exceptional* errors
- for performance, when error raising and handling are close, or errors occur often, prefer error codes or a dedicated class
- when in doubt about which error strategy is better, profile!

Avoid

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

Prefer

```
for (string const &num: nums) {
  optional<int> i = convert(num);
  if (i) {
    process(*i);
  } else {
    ... // log and continue
  }
}
```

noexcept specifier

C++ 11

noexcept

- a function with the `noexcept` specifier states that it guarantees to not throw an exception

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

  - either no exceptions is thrown or they are handled internally
  - if one is thrown, ‘std::terminate‘ is called
  - allows the compiler to optimize around that knowledge
- a function with `noexcept(expression)` is only `noexcept` when `expression` evaluates to `true` at compile-time

```cpp
int safe_if_8B() noexcept(sizeof(long)==8);
```

Good practice: `noexcept`

- Use `noexcept` on leaf functions where you know the behavior
- C++11 destructors are `noexcept` - never throw from them

Templates

4 Core modern C++

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

Templates

C++ 17

Concept

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

```
template<typename T>
const T & max(const T &a, const T &b) {
  return b < a ? a : b;
}
template<typename T>
struct Vector {
  int m_len;
  T* m_data;
};
template <typename T>
std::size_t size = sizeof(T);
```

Templates

Notes on templates

- they are compiled for each instantiation
- they need to be defined before used
  - so all template code must typically be in headers
  - or declared to be available externally (`extern template`)
- this may lead to longer compilation times and bigger binaries

```cpp
template<typename T>
T func(T a) {
  return a;
}
```

func(3)

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

func(5.2)

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

Templates

Template parameters

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

```cpp
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>
Map m4;           // Map<int, int>, C++17
```

Template parameters

typename vs. class keyword

- for declaring a template type parameter, the `typename` and `class` keyword are semantically equivalent
- template template parameters require C++ 17 for `typename`

```cpp
template<typename T>
T func(T a); // equivalent to:
template<class T>
T func(T a);

template<template<class> class C>
C<int> func(C<int> a); // equivalent to:
template<template<typename> class C>
C<int> func(C<int> a); // equivalent to:
template<template<typename> typename C> // C++17
C<int> func(C<int> a);
```

Template implementation

C++ 98

```cpp
template<typename KeyType=int, typename ValueType=KeyType>
struct Map {
  // declaration and inline definition
  void set(const KeyType &key, ValueType value) {
    ...
  }
  // just declaration
  ValueType get(const KeyType &key);
};

// out-of-line definition
template<typename KeyType, typename ValueType>
ValueType Map<KeyType, ValueType>::get
   (const KeyType &key) {
  ...
}
```

Non-type template parameter C++ 98 / C++ 17 / C++ 20

template parameters can also be values

- integral types, pointer, enums in C++ 98
- auto in C++ 17
- literal types (includes floating points) in C++ 20

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

Polygon<19> nonadecagon{3.3f};
```

Template specialization

Specialization

Templates can be specialized for given values of their parameter

```cpp
template<typename F, unsigned int N>
struct Polygon { ... }; // primary template

template<typename F> // partial specialization
struct Polygon<F, 6> {
  F perimeter() { return 6 * radius; }
  F radius;
};
template<>              // full specialization
struct Polygon<int, 6> {
  int perimeter() { return 6 * radius; }
  int radius;
};
```

The full power of templates

Exercise: Templates

- go to `exercises/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

Lambdas

4 Core modern C++

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

Lambda expressions

C++ 11

Definition

A lambda expression is a function with no name

Lambda expressions

C++ 11

Definition

A lambda expression is a function with no name

Python example

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

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

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

C++ Lambdas

C++ 11

Simplified syntax

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

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

Usage example

```cpp
int data[]{1,2,3,4,5};
auto f = [](int i) {
  std::cout << i << " squared is " << i*i << '\n';
};
for (int i : data) f(i);
```

Trailing function return type

C++ 11

An alternate way to specify a function's return type

```
int f(float a);                    // classic
auto f(float a) -> int;            // trailing
auto f(float a) { return 42; }     // deduced, C++14
```

Trailing function return type

C++ 11

An alternate way to specify a function's return type

```cpp
int f(float a);                     // classic
auto f(float a) -> int;             // trailing
auto f(float a) { return 42; } // deduced, C++14
```

When to use trailing return type

- Only way to specify return type for lambdas
- Allows to simplify inner type definition

```cpp
class Equation {
  using ResultType = double;
  ResultType evaluate();
}
Equation::ResultType Equation::evaluate() {...}
auto Equation::evaluate() -> ResultType {...}
```

Capturing variables

C++ 11

Adaptable lambdas

- Adapt lambda's behaviour by accessing variables outside of it
- This is called "capture"

Capturing variables

C++ 11

Adaptable lambdas

- Adapt lambda's behaviour by accessing variables outside of it
- This is called "capture"

First attempt in C++

```cpp
int increment = 3;
int data[]{1,9,3,8,3,7,4,6,5};
auto f = [](int x) { return x+increment; };
for(int& i : data) i = f(i);
```

Capturing variables

C++ 11

Adaptable lambdas

- Adapt lambda's behaviour by accessing variables outside of it
- This is called "capture"

First attempt in C++

```cpp
int increment = 3;
int data[]{1,9,3,8,3,7,4,6,5};
auto f = [](int x) { return x+increment; };
for(int& i : data) i = f(i);
```

Error

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

Capturing variables

C++ 11

The capture list

- local variables outside the lambda must be explicitly captured
  - unlike in Python, Java, C#, Rust, ...
- captured variables are listed within initial []

Capturing variables

C++ 11

The capture list

- local variables outside the lambda must be explicitly captured
  - unlike in Python, Java, C#, Rust, ...
- captured variables are listed within initial []

Example

```cpp
int increment = 3;
int data[]{1,9,3,8,3,7,4,6,5};
auto f = [increment](int x) { return x+increment; };
for(int& i : data) i = f(i);
```

Default capture is by value

C++ 11

Code example

```
int sum = 0;
int data[]{1,9,3,8,3,7,4,6,5};
auto f = [sum](int x) { sum += x; };
for (int i : data) f(i);
```

Default capture is by value

C++ 11

Code example

```
int sum = 0;
int data[]{1,9,3,8,3,7,4,6,5};
auto f = [sum](int x) { sum += x; };
for (int i : data) f(i);
```

Error

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

Default capture is by value

C++ 11

Code example

```
int sum = 0;
int data[]{1,9,3,8,3,7,4,6,5};
auto f = [sum](int x) { sum += x; };
for (int i : data) f(i);
```

Error

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

Explanation

- By default, variables are captured by value
- The lambda's `operator()` is `const inline`

Capture by reference

C++ 11

Simple example

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

```
int sum = 0;
int data[]{1,9,3,8,3,7,4,6,5};
auto f = [&sum](int x) { sum += x; };
for (int i : data) f(i);
```

Capture by reference

C++ 11

Simple example

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

```
int sum = 0;
int data[]{1,9,3,8,3,7,4,6,5};
auto f = [&sum](int x) { sum += x; };
for (int i : data) f(i);
```

Mixed case

One can of course mix values and references

```
int sum = 0, off = 1;
int data[]{1,9,3,8,3,7,4,6,5};
auto f = [&sum, off](int x) { sum += x + off; };
for (int i : data) f(i);
```

Capture by value vs. by reference

C++ 11

See the difference between val and ref

```
int data[]{1,9,3,8,3,7,4,6,5};
int increment = 3;
auto val = [ inc](int x) { return x+inc; };
auto ref = [&inc](int x) { return x+inc; };

increment = 4;

for(int& i : data) i = val(i); // increments by 3
for(int& i : data) i = ref(i); // increments by 4
```

Init capture

C++ 14

Capture with an initializer

In C++ 14, can declare captures with initializers

```
auto f = [inc = 1+2](int x) { return x+inc; };
auto g = [inc = getInc()](int x) { return x+inc; };
for(int& i : data) i = f(i); // increments by 3
for(int& i : data) i = g(i); // unknown increment
```

Anatomy of a lambda

C++ 11

Lambdas are pure syntactic sugar - cppinsight

- They are replaced by a functor during compilation

```cpp
int sum = 0, off = 1;
auto l =
[&sum, off]



(int x) {
  sum += x + off;
};


l(42);
```

```cpp
int sum = 0, off = 1;
struct __lambda4 {
  int& sum; int off;
  __lambda4(int& s, int o)
  : sum(s), off(o) {}

  auto operator()(int x) const {
    sum += x + off;
  }
};
auto l = __lambda4{sum, off};
l(42);
```

Some nice consequences

- Lambda expressions create ordinary objects
- They can be copied, moved, or inherited from

Capture list

C++ 11

all by value

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

Capture list

C++ 11

all by value

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

all by reference

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

Capture list

C++ 11

all by value

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

all by reference

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

mix

```
[&,  b](...) { ... };
[=, &b](...) { ... };
```

The STL

4 Core modern C++

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

The Standard Template Library

C++ 98

What it is

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

The Standard Template Library

What it is

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

Use it

and adapt it to your needs, thanks to templates

STL in practice

C++ 14

STL example - godbolt

```cpp
#include <vector>
#include <algorithm>
#include <functional>    // `import std;` in C++23
#include <iterator>
#include <iostream>

std::vector<int> in{5, 3, 4};    // initializer list
std::vector<int> out(3);  // constructor taking size
std::transform(in.begin(), in.end(),  // input range
               out.begin(),           // start result
               std::negate{});        // function obj
std::copy(out.begin(), out.end(),     // -5 -3 -4
  std::ostream_iterator<int>{std::cout, " "});
```

STL's concepts

C^{++} 98

containers

- data structures for managing a range of elements, irrespective of:
  - the data itself (templated)
  - the memory allocation of the structure (templated)
  - the algorithms that may use the structure (iterators)

Examples (→ string and container library on cppreference)

- string, string_view (C^{++} 17)
- list, forward_list (C^{++} 11), vector, deque, array (C^{++} 11)
- [multi]map, [multi]set (C^{++} 23: flat_[multi]map, flat_[multi]set)
- unordered_[multi]map (C^{++} 11), unordered_[multi]set (C^{++} 11)
- stack, queue, priority_queue
- span (C^{++} 20)
- non-containers: bitset, pair, tuple (C^{++} 11), optional (C^{++} 17), variant (C^{++} 17), any (C^{++} 17), expected (C^{++} 23)

Containers: std::vector

C++ 11

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

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

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

Containers: std::vector

C++ 11

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

v.resize(100); // size changes, grows: new T{}s appended
               //             shrinks: Ts at end destroyed
v.reserve(1000); // size remains, memory increased
for (T i = 0; i < 900; i++)
  v.push_back(i); // add to the end
v.insert(v.begin()+3, T{}); // insert after 3rd position

v.pop_back();          // removes last element
v.erase(v.end() - 3); // removes 3rd-last element
v.clear();             // removes all elements
```

Containers: std::unordered_map

C++ 11

Conceptually a container of `std::pair<Key const, Value>`

```cpp
#include <unordered_map>
std::unordered_map<std::string, int> m;
m["hello"] = 1;  // inserts new key, def. constr. value
m["hello"] = 2;  // finds existing key
auto [it, isNewKey] = m.insert({"hello", 0}); // no effect
//     ^ C++17: "Structured binding"
int val = m["world"];     // inserts new key (val == 0)
int val = m.at("monde"); // throws std::out_of_range

if (auto it = m.find("hello"); it != m.end()) // C++17
    m.erase(it);          // remove by iterator (fast)
if (m.contains("hello")) // C++20
    m.erase("hello"); // remove by key, 2. lookup, bad
for (auto const& [k, v] : m) // iterate k/v pairs (C++17)
    std::cout << k << ": " << v << '\n';
```

std::hash

C++ 11

- The standard utility to create hash codes
- Used by `std::unordered_map` and others
- Can be customized for your types via template specialization

```cpp
#include <functional>
std::hash<std::string> h;
std::cout << h("hello"); // 2762169579135187400
std::cout << h("world"); // 8751027807033337960

class MyClass { int a, b; ... };
template<> struct std::hash<MyClass> {
  std::size_t operator()(MyClass const& c) {
    std::hash<int> h;
    return h(c.a) ^ h(c.b); // xor to combine hashes
  }
};
```

STL's concepts

C++ 11

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
  - std::reverse_iterator, std::back_insert_iterator, ...

Iterator example - godbolt

```cpp
std::vector<int> const v = {1,2,3,4,5,6,7,8,9};
auto const end = v.rend() - 3; // arithmetic
for (auto it = v.rbegin();
     it != end;       // compare positions
     it += 2)         // jump 2 positions
  std::cout << *it; // dereference, prints: 975
```

STL's concepts

algorithms

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

STL's concepts

C++ 98

functors / function objects

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

Functors / function objects

C++ 11

Example

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

  int operator()(int value) const {
    return value + m_inc;
  }
};
std::vector<int> v{1, 2, 3};
const auto inc = 42;
std::transform(v.begin(), v.end(), v.begin(),
               Incrementer{inc});
```

Prefer lambdas over functors

C++ 11

With lambdas

```
std::vector<int> v{1, 2, 3};
const auto inc = 42;
std::transform(begin(v), end(v), begin(v),
               [inc](int value) {
                 return value + inc;
               });
```

Prefer lambdas over functors

C++ 11

With lambdas

```
std::vector<int> v{1, 2, 3};
const auto inc = 42;
std::transform(begin(v), end(v), begin(v),
               [inc](int value) {
                 return value + inc;
               });
```

Good practice: Use STL algorithms with lambdas

- Prefer lambdas over functors when using the STL
- Avoid binders like `std::bind2nd`, `std::ptr_fun`, etc.

Range-based for loops with STL containers

C++ 11

Iterator-based loop (since C++ 98)

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

Range-based for loops with STL containers

C++ 11

Iterator-based loop (since C++ 98)

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

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

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

Range-based for loops with STL containers C^{++} 11

Iterator-based loop (since C^{++} 98)

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

Range-based for loop (since C^{++} 11)

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

STL way (since C^{++} 98)

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

More examples

C++ 17

```cpp
std::list<int> l = ...;

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

// Computes sin(x)/(x + DBL_MIN) for elements of a range.
std::vector<double> r(l.size());
std::transform(l.begin(), l.end(), r.begin(),
  [](auto x) { return std::sin(x)/(x + DBL_MIN); });

// reduce/fold (using addition)
const auto sum = std::reduce(v.begin(), v.end());
```

More examples

C++ 11

```cpp
std::vector<int> v = ...;

// remove duplicates
std::sort(v.begin(), v.end());
auto newEndIt = std::unique(v.begin(), v.end());
v.erase(newEndIt, v.end());

// remove by predicate
auto p = [](int i) { return i > 42; };
auto newEndIt = std::remove_if(v.begin(), v.end(), p);
v.erase(newEndIt, v.end());

// remove by predicate (C++20)
std::erase_if(v, p);
```

Welcome to lego programming!

Using the STL

Exercise: STL

- go to `exercises/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

Using the STL

C++ 98

Be brave and persistent!

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

RAII and smart pointers

4 Core modern C++

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

Pointers: why are they error prone?

C++ 98

They need initialization

```
char *s;
try {
  foo(); // may throw
  s = new char[100];
  read_line(s);
} catch (...) { ... }
process_line(s);
```

Pointers: why are they error prone?

C^{++} 98

They need initialization **Seg Fault**

```
char *s;
try {
  foo(); // may throw
  s = new char[100];
  read_line(s);
} catch (...) { ... }
process_line(s);
```

Pointers: why are they error prone?

C++ 98

They need initialization

Seg Fault

```cpp
char *s;
try {
  foo(); // may throw
  s = new char[100];
```

They need to be released

```cpp
char *s = new char[100];
read_line(s);
if (s[0] == '#') return;
process_line(s);
delete[] s;
```

Pointers: why are they error prone? C++ 98

They need initialization — Seg Fault

```cpp
char *s;
try {
  foo(); // may throw
  s = new char[100];
```

They need to be released — Memory leak

```cpp
char *s = new char[100];
read_line(s);
if (s[0] == '#') return;
process_line(s);
delete[] s;
```

Pointers: why are they error prone?

They need initialization — Seg Fault

```
char *s;
try {
  foo(); // may throw
  s = new char[100];
```

They need to be released — Memory leak

```
char *s = new char[100];
read_line(s);
```

They need clear ownership

```
char *s = new char[100];
read_line(s);
vec.push_back(s);
set.add(s);
std::thread t1{func1, vec};
std::thread t2{func2, set};
```

Pointers: why are they error prone? C++ 98

They need initialization — **Seg Fault**

```
char *s;
try {
  foo(); // may throw
  s = new char[100];
```

They need to be released — **Memory leak**

```
char *s = new char[100];
read_line(s);
```

They need clear ownership — **Who should release ?**

```
char *s = new char[100];
read_line(s);
vec.push_back(s);
set.add(s);
std::thread t1{func1, vec};
std::thread t2{func2, set};
```

This problem exists for any resource

C++ 11

For example with a file

```
std::FILE *handle = std::fopen(path, "w+");
if (nullptr == handle) { throw ... }
std::vector v(100, 42);
write(handle, v);
if (std::fputs("end", handle) == EOF) {
  return;
}
std::fclose(handle);
```

Which problems do you spot in the above snippet?

Resource Acquisition Is Initialization (RAII) C++ 98

Practically

Use variable construction/destruction and scope semantics:

- wrap the resource inside a class
- acquire resource in constructor
- release resource in destructor
- create an instance on the stack
  - automatically destructed when leaving the scope
  - including in case of exception
- use move semantics to pass the resource around

RAII in practice

C++ 98

An RAII File class

```
class File {
public:
  // constructor: acquire resource
  File(const char* filename)
    : m_handle(std::fopen(filename, "w+")) {
    // abort constructor on error
    if (m_handle == nullptr) { throw ... }
  }
  // destructor: release resource
  ~File() { std::fclose(m_handle); }
  void write (const char* str) {
    ...
  }
private:
  std::FILE* m_handle; // wrapped resource
};
```

RAII usage

C++ 98

Usage of File class

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

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

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

Good practice: Use `std::fstream` for file handling

The standard library provides `std::fstream` to handle files, use it!

std::unique_ptr

C++ 11

A RAII pointer

- wraps and behaves like a regular pointer
- get underlying pointer using `get()`
- when destroyed, deletes the object pointed to
- has move-only semantic
  - the pointer has unique ownership
  - copying will result in a compile error

std::unique_ptr

C++ 11

A RAII pointer

- wraps and behaves like a regular pointer
- get underlying pointer using get()
- when destroyed, deletes the object pointed to
- has move-only semantic
  - the pointer has unique ownership
  - copying will result in a compile error

```cpp
#include <memory>
void f(std::unique_ptr<Foo> ptr) {
  ptr->bar();
} // deallocation when f exits

std::unique_ptr<Foo> p{ new Foo{} }; // allocation
f(std::move(p)); // transfer ownership
assert(p.get() == nullptr);
```

Quiz

C++ 11

What do you expect?

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

Quiz

C^{++} 11

What do you expect?

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

Compilation Error - godbolt

```
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 C++ 14

std::make_unique

- allocates and constructs an object with arguments and wraps it with `std::unique_ptr` in one step
- no `new` or `delete` calls anymore!
- no memory leaks if used consistently

std::make_unique

C++ 14

std::make_unique

- allocates and constructs an object with arguments and wraps it with std::unique_ptr in one step
- no new or delete calls anymore!
- no memory leaks if used consistently

std::make_unique usage

```
{
  // calls new File("logfile.txt") internally
  auto f = std::make_unique<File>("logfile.txt");
  f->write("hello logfile!");
} // deallocation at end of scope
```

RAII or raw pointers

C++ 11

When to use what?

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

RAII or raw pointers

C++ 11

When to use what?

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

A question of ownership

```cpp
std::unique_ptr<T> produce();
void observe(const T&);
void modifyRef(T&);
void modifyPtr(T*);
void consume(std::unique_ptr<T>);
std::unique_ptr<T> pt{produce()}; // Receive ownership
observe(*pt);                     // Keep ownership
modifyRef(*pt);                   // Keep ownership
modifyPtr(pt.get());              // Keep ownership
consume(std::move(pt));           // Transfer ownership
```

std::unique_ptr usage summary

C++ 11

Good practice: std::unique_ptr

- std::unique_ptr is about lifetime management
  - use it to tie the lifetime of an object to a unique RAII owner
  - use raw pointers/references to refer to another object without owning it or managing its lifetime
- use std::make_unique for creation
- strive for having no new/delete in your code
- for dynamic arrays, std::vector may be more useful

std::shared_ptr

C++ 11

std::shared_ptr : a reference-counting pointer

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

std::make_shared : creates a std::shared_ptr

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

Quiz: std::shared_ptr in use

C++ 11

What is the output of this code? - godbolt

```
auto shared = std::make_shared<int>(100);
auto print = [shared](){
  std::cout << "Use: " << shared.use_count() << " "
            << "value: " << *shared << "\n";
};
print();
{
  auto ptr{ shared };
  (*ptr)++;
  print();
}
print();
```

Quiz: std::shared_ptr in use

C++ 11

What is the output of this code? - godbolt

```cpp
auto shared = std::make_shared<int>(100);
auto print = [shared](){
  std::cout << "Use: " << shared.use_count() << " "
            << "value: " << *shared << "\n";
};
print();
{
  auto ptr{ shared };
  (*ptr)++;
  print();
}
print();
```

```
Use: 2 value: 100
Use: 3 value: 101
Use: 2 value: 101
```

Quiz: std::shared_ptr in use

C^{++} 11

What is the output of this code?

```
auto shared = std::make_shared<int>(100);
auto print = [&shared](){
  std::cout << "Use: " << shared.use_count() << " "
            << "value: " << *shared << "\n";
};
print();
{
  auto ptr{ shared };
  (*ptr)++;
  print();
}
print();
```

```
Use: 1 value: 100
Use: 2 value: 101
Use: 1 value: 101
```

Rule of zero

C^{++} 11

Good practice: Single responsibility principle (SRP)

Every class should have only one responsibility.

Good practice: Rule of zero

- If your class has any special member functions (except ctor.)
  - Your class probably deals with a resource, use RAII
  - Your class should only deal with this resource (SRP)
  - Apply rule of 3/5: write/default/delete all special members
- Otherwise: do not declare any special members (rule of zero)
  - A constructor is fine, if you need some setup
  - If your class holds a resource as data member:
    wrap it in a smart pointer, container, or any other RAII class

smart pointers

Exercise: Smart pointers

- go to `exercises/smartPointers`
- compile and run the program. It doesn't generate any output.
- Run with valgrind if possible to check for leaks

```
$ valgrind --leak-check=full --track-origins=yes ./smartPointers
```

- In the *essentials course*, go through `problem1()` and `problem2()` and fix the leaks using smart pointers.
- In the *advanced course*, go through `problem1()` to `problem4()` and fix the leaks using smart pointers.
- `problem4()` is the most difficult. Skip if not enough time.

Useful tools

C^{++} editor

5 Useful tools

- C^{++} editor
- Version control
- Code formatting
- The Compiling Chain
- Web tools
- 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

Version control

5 Useful tools

- C++ editor
- Version control
- Code formatting
- The Compiling Chain
- Web tools
- Debugging

Version control

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
- Subversion: Historical, not distributed - don't use
- CVS: Archeological, not distributed - don't 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

5 Useful tools

- C^{++} editor
- Version control
- Code formatting
- The Compiling Chain
- Web tools
- Debugging

clang-format

.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>)
- Most editors/IDEs can find a .clang-format file and adapt

clang-format

Exercise: clang-format

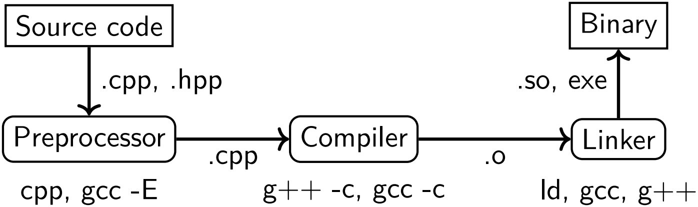
- Go to any example
- Format code with:
  `clang-format --style=GNU -i <file.cpp>`
- Inspect changes, try `git diff .`
- Revert changes using `git checkout -- <file.cpp>` or `git checkout .`
- Go to exercises directory and create a .clang-format file
  `clang-format -style=LLVM -dump-config > .clang-format`
- Run `clang-format -i <any_exercise>/*.cpp`
- Revert changes using `git checkout <any_exercise>`

The Compiling Chain

5 Useful tools

- C^{++} editor
- Version control
- Code formatting
- The Compiling Chain
- Web tools
- Debugging

The compiling chain



The steps

cpp the preprocessor
handles the # directives (macros, includes)
creates "complete" source code (ie. translation unit)

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

ld the linker
links several binary files into libraries and executables

Compilers

Available tools

- gcc — the most common and most used
  free and open source
- clang — drop-in replacement of gcc
  free and open source, based on LLVM
- icc icx — 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 (also: -g3 or -ggdb)

-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

How to inspect object files?

Listing symbols : nm

- gives list of symbols in a file
  - these are functions and constants
  - with their internal (mangled/encoded) naming
- also gives type and location in the file for each symbol
  - 'U' type means undefined
    - so a function used but not defined
    - linking will be needed to resolve it
- use -C option to demangle on the fly

```
> nm -C Struct.o
                 U strlen
                 U _Unwind_Resume
000000000000008a T SlowToCopy::SlowToCopy(SlowToCopy const&)
0000000000000000 T SlowToCopy::SlowToCopy()
0000000000000064 T SlowToCopy::SlowToCopy(std::__cxx11::basic_st
```

How to inspect libraries/executables?

Listing dependencies : `ldd`

- gives (recursive) list of libraries required by the given argument
  - and if/where they are found in the current context
- use `-r` to list missing symbols (mangled)

```
> ldd -r trypoly
    linux-vdso.so.1 (0x00007f3938085000)
    libpoly.so => not found
    libstdc++.so.6 => /lib/x86_64-linux-gnu/libstdc++.so.6 (0x00
    [...]
    undefined symbol: _ZNK7Hexagon16computePerimeterEv    (./try
    undefined symbol: _ZNK7Polygon16computePerimeterEv    (./try
    undefined symbol: _ZN7HexagonC1Ef    (./trypoly.sol)
    undefined symbol: _ZN8PentagonC1Ef    (./trypoly.sol)
```

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
```

CMake

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

Example CMakeLists.txt

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

find_package(ZLIB REQUIRED) # for external libs

add_executable(hello main.cpp util.h util.cpp)
set(CMAKE_CXX_STANDARD 20)
target_link_libraries(hello PUBLIC ZLIB::ZLIB)
```

CMake - Building

Building a CMake-based project

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

```
mkdir build # will contain all build-related files
cd build
cmake ..     # configures and generates a build system
cmake -DCMAKE_BUILD_TYPE=Release .. # pass arguments
ccmake .     # change configuration using terminal GUI
cmake-gui . # change configuration using Qt GUI
cmake --build . -j8     # build project with 8 jobs
cmake --build . --target hello  # build only hello
sudo cmake --install . # install project into system
cd ..
rm -r build # clean everything
```

Compiler chain

Exercise: Compiler chain

- go to `exercises/polymorphism`
- preprocess Polygons.cpp (g++ -E -o output)
- compile Polygons.o and trypoly.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 trypoly target
  - run make clean; make
  - look at the collect 2 line, from the end up to "-o trypoly"
- see library dependencies of ‘trypoly‘ using ‘ldd‘

Web tools

5 Useful tools

- C++ editor
- Version control
- Code formatting
- The Compiling Chain
- Web tools
- Debugging

Godbolt / Compiler Explorer

Concept

An online generic compiler with immediate feedback. Allows:

- trying various compilers in the browser
- inspecting the assembly generated
- use of external libraries (over 50 available !)
- running the produced code
- sharing small pieces of code via permanent short links

Typical usage

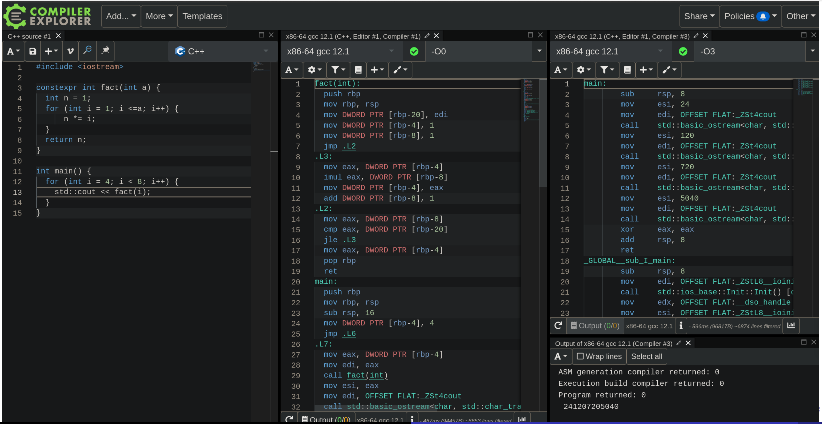
- check small pieces of code on different compilers
- check some new C^{++} functionality and its support
- optimize small pieces of code
- NOT relevant for large codes

Godbolt by example

Check effect of optimization flags

https://godbolt.org/z/Pb8WsWjEx

- Check generated code with -O0, -O1, -O2, -O3
- See how it gets shorter and simpler



cppinsights

Concept

Reveals the actual code behind C^{++} syntactic sugar

- lambdas
- range-based loops
- templates
- initializations
- auto
- ...

Typical usage

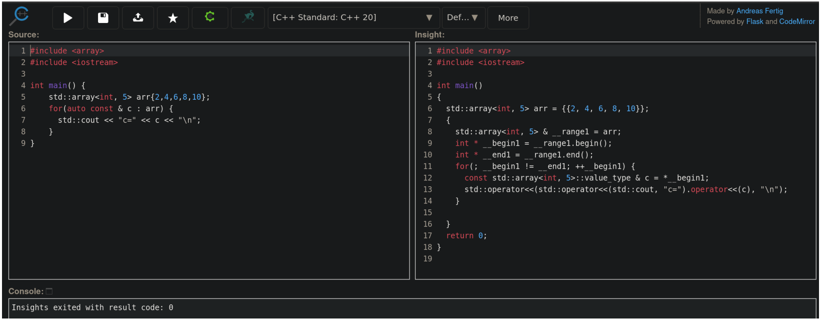
- understand how things work behind the C^{++} syntax
- debug some non working pieces of code

cppinsights by example

Check how range-based loop work

https://cppinsights.io/s/b886aa76

- See how they map to regular iterators
- And how operators are converted to function calls



Debugging

5 Useful tools
- C++ editor
- Version control
- Code formatting
- The Compiling Chain
- Web tools
- **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/LLVM
- gdb-oneapi the Intel OneAPI debugger

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

gdb

Exercise: gdb

- go to `exercises/debug`
- compile, run, see the crash
- run it in gdb (or lldb on newer MacOS)
- inspect backtrace, variables
- find problem and fix bug
- try stepping, breakpoints

Debugging UIs

User interfaces for debuggers

- offer convenience on top of command line
- windows for variables, breakpoints, call stack, active threads, watch variables in-code, disassembly, run to cursor ...

Native gdb Try "tui enable" for a simple built-in UI

VSCode Built-in support for gdb

CodeLLDB VS Code plugin for LLDB

GDB dashboard Poplar terminal UI for gdb

GEF Modern terminal UI for gdb

- some editors and most IDEs have good debugger integration

This is the end

Questions ?

https://github.com/hsf-training/cpluspluscourse/raw/download/talk/C++Course_full.pdf
https://github.com/hsf-training/cpluspluscourse

Index of Good Practices

Index of Exercises

Index of Godbolt code examples

Books

- A Tour of C++, Third Edition
  Bjarne Stroustrup, Addison-Wesley, Sep 2022
  ISBN-13: 978-0136816485
- Effective Modern C++
  Scott Meyers, O'Reilly Media, Nov 2014
  ISBN-13: 978-1-491-90399-5
- C++ Templates - The Complete Guide, 2nd Edition
  David Vandevoorde, Nicolai M. Josuttis, and Douglas Gregor
  ISBN-13: 978-0-321-71412-1
- C++ Best Practices, 2nd Edition
  Jason Turner
  https://leanpub.com/cppbestpractices
- Clean Architecture
  Robert C. Martin, Pearson, Sep 2017
  ISBN-13: 978-0-13-449416-6
- The Art of UNIX Programming
  Eric S. Raymond, Addison-Wesley, Sep 2002
  ISBN-13: 978-0131429017
- Introduction to Algorithms, 4th Edition
  T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Apr 2022
  ISBN-13: 978-0262046305

Conferences

- CppCon — cppcon.org — CppCon
- C^{++} Now — cppnow.org — BoostCon
- Code::Dive — codedive.pl — codediveconference
- ACCU Conference — accu.org — ACCUConf
- Meeting C^{++} — meetingcpp.com — MeetingCPP
- See link below for more information
  `https://isocpp.org/wiki/faq/conferences-worldwide`