HEP C^{++} course

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

CERN

September 12, 2023

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
  - 548 slides, 690 pages, 31 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

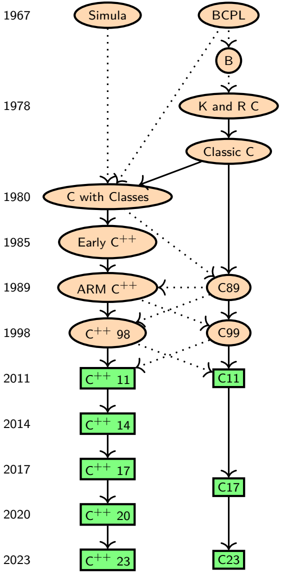
Detailed outline

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, awaiting ISO ballot
  - work on C++ 26 has begun
- 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, awaiting ISO ballot
  - work on C++ 26 has begun
- 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	$\geq$4.8	$\geq$3.3
14	$\geq$4.9	$\geq$3.4
17	$\geq$7.3	$\geq$5
20	>11	>12

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

Why we use it?

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

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

Adapted to large projects

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

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

Adapted to large projects

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

Fast

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

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

Adapted to large projects

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

Fast

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

What we get

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

Language basics

Core syntax and types

2 Language basics

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

Hello World

C^{++} 98

```
#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)


```
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

```
#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

```cpp
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
```

Useful aliases

C^{++} 98

```cpp
#include <cstddef> // (and others) defines:

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

#include <cstdint> // defines:

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

// signed/unsigned integer, can hold any pointer value
std::intptr_t i = reinterpret_cast<intptr_t>(&s);
std::uintptr_t i = reinterpret_cast<uintptr_t>(&s);
```

Extended floating-point types

C++ 23

Extended floating-point types

- *Optional* types, which may be provided
- Custom conversion and promotion rules

```cpp
#include <stdfloat> // may define these:

std::float16_t  = 3.14f16;  // 16 (1+5+10) bit float
std::float32_t  = 3.14f32;  // like float
                            // but different type
std::float64_t  = 3.14f64;  // like double
                            // but different type
std::float128_t = 3.14f128; // 128 (1+15+112) bit float
std::bfloat_t   = 3.14bf16; // 16 (1+8+7) bit float

// also F16, F32, F64, F128 or BF16 suffix possible
```

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

```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
	0x3008
	0x3004
i = 4	0x3000

Pointers


```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

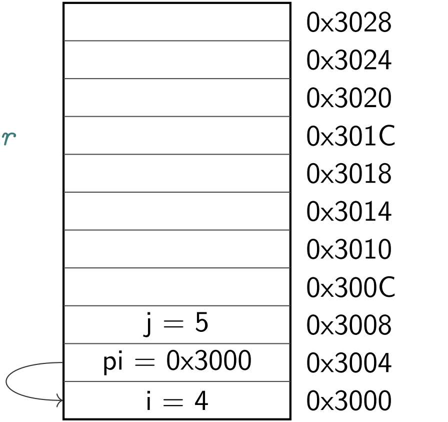
	0x3028
	0x3024
	0x3020
	0x301C
	0x3018
	0x3014
	0x3010
	0x300C
	0x3008
pi = 0x3000	0x3004
i = 4	0x3000

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



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
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
	0x3020
	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
	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

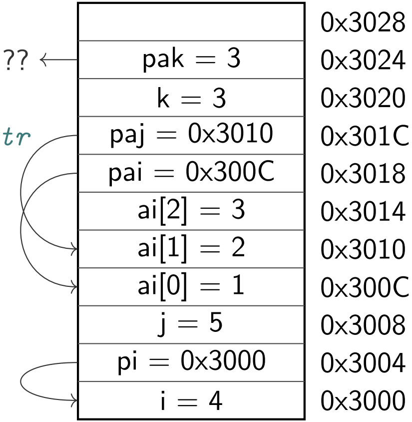
	0x3028
	0x3024
k = 3	0x3020
paj = 0x3010	0x301C
pai = 0x300C	0x3018
ai[2] = 3	0x3014
ai[1] = 2	0x3010
ai[0] = 1	0x300C
j = 5	0x3008
pi = 0x3000	0x3004
i = 4	0x3000

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



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

```cpp
#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

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

```
{ 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

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] = 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

```
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

Easier way to declare nested namespaces

C++ 98

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

C++ 17

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

Unnamed / anonymous namespaces

C++ 98

A namespace without a name!

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

Purpose

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

Supersedes static

```
static int localVar; // equivalent C code
```

Using namespace directives

C++ 98

Avoid "using namespace" directives

- Make all members of a namespace visible in current scope
- Risk of name clashes or ambiguities

```cpp
using namespace std;
cout << "We can print now\n"; // uses std::cout
```

Never use in headers at global scope!

```cpp
#include "PoorlyWritten.h" // using namespace std;
struct array { ... };
array a;  // Error: name clash with std::array
```

What to do instead

- Qualify names: `std::vector`, `std::cout`, …
- Put things that belong together in the same namespace
- Use *using declarations* in local scopes: `using std::cout;`

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
- Inline keyword
- Assertions

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

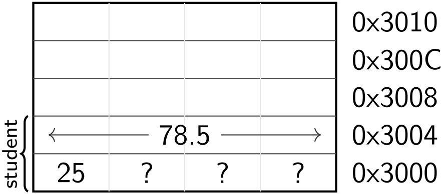
C++ 98

"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

					0x3010
teacher	←	67.0		→	0x300C
teacher	45	?	?	?	0x3008
student	←	78.5		→	0x3004
student	25	?	?	?	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

← 0x3000 →				0x3010
← 67.0 →				0x300C
45	?	?	?	0x3008
← 78.5 →				0x3004
25	?	?	?	0x3000

teacher

student

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
```

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
				0x3000

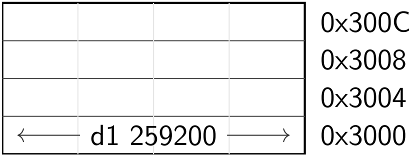
union

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



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

```
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

```
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++ 98

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

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

More sensible example

C++ 98

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

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

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

More sensible example

C^{++} 98

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

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

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

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

typedef and using

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

References

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:

```
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

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

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

```

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

Functions: parameters are passed by value

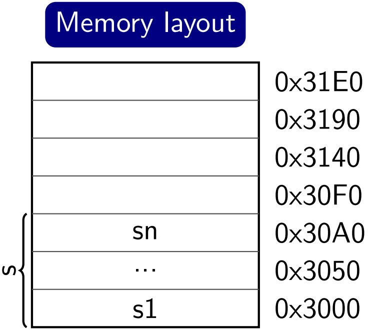

```cpp
struct BigStruct {...};
BigStruct s;

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

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

Memory layout

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

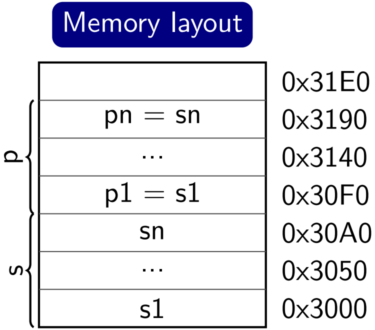
Functions: parameters are passed by value


```
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
```



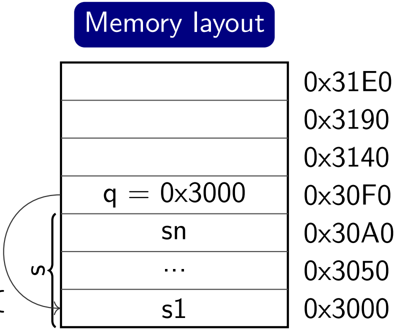
Functions: parameters are passed by value


```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
```



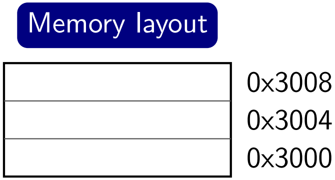
Functions: parameters are passed by value


```
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: 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? 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
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? 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
q = 0x3000	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
q = 0x3000	0x3004
s.a = 2	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 = 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

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

```cpp
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

```cpp
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

C++ 98

Example: don't! Everything in one long function

```
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
- Inline keyword
- Assertions

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
```

Operators(1)

C++ 98

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

```
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)

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
```

Logical Operators

```cpp
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

```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

Avoid - use parentheses

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

Details can be found on cppreference

Control structures

Control structures: if

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

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

```
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

```
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;
}
```

- Initializations and 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

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

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

Practical example

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

Control structures: for loop

C++ 98

for loop syntax

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

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

Practical example

```
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

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

C++ 20

```cpp
for (std::size_t i = 0; auto& 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
   - Inline keyword
   - Assertions

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

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

Usage: myfile.cpp

```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

```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
```

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
```

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`

Inline keyword

Inline keyword

C^{++} 98

Inline functions originally

- Applies to a function to tell the compiler to inline it
  - That is, replace function calls by the function's content (similar to how a macro works)
- Only a hint, compiler can still choose to not inline
- Avoids call overhead at the cost of increasing binary size

Major side effect

- The linker reduces the duplicated functions into one
- An inline function definition can thus live in header files

```
inline int mult(int a, int b) {
  return a * b;
}
```

Inline keyword

C++ 98

Inline functions nowadays

- Compilers can judge far better when to inline or not
  - thus primary purpose is gone
- Putting functions into headers became main purpose
- Many types of functions are marked `inline` by default:
  - function templates
  - `constexpr` functions
  - class member functions

Inline keyword

C++ 17

Inline variables

- Global (or `static` member) variable specified as `inline`
- Same side effect, linker merges all occurrences into one
- Allows to define global variables/constants in headers

```cpp
// global.h
inline int count = 0;
inline const std::string filename = "output.txt";
// a.cpp
#include "global.h"
int f() { return count; }
// b.cpp
#include "global.h"
void g(int i) { count += i; }
```

- Avoid global variables! Global constants are fine.

Assertions

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
- Inline keyword
- **Assertions**

Assertions

C++ 98

Checking invariants in a program

- An invariant is a property that is guaranteed to be true during certain phases of a program, and the program might crash or yield wrong results if it is violated
  - “Here, ‘a’ should always be positive”
- This can be checked using `assert`
- The program will be aborted if the assertion fails

assert in practice - godbolt

```
#include <cassert>
double f(double a) {
  // [...] do stuff with a
  // [...] that should leave it positive
  assert(a > 0.);
  return std::sqrt(a);
}
```

Assertions

C++ 98

Checking invariants in a program

- An invariant is a property that is guaranteed to be true during certain phases of a program, and the program might crash or yield wrong results if it is violated
  - “Here, ‘a’ should always be positive”
- This can be checked using `assert`
- The program will be aborted if the assertion fails

`assert` in practice - godbolt

```
% ./testAssert
Assertion failed: (a > 0.), function f,
                  file testAssert.cpp, line 5.
Abort trap: 6
```

Assertions

C++ 98

Good practice: Assert

- Assertions are mostly for developers and debugging
- Use them to check important invariants of your program
- Prefer handling user-facing errors with helpful error messages/exceptions
- Assertions can impact the speed of a program
  - Assertions are disabled when the macro `NDEBUG` is defined
  - Decide if you want to disable them when you release code

Disabling assertions

Compile a program with NDEBUG defined:

```
g++ -DNDEBUG -O2 -W[...] test.cpp -o test.exe
```

```cpp
double f(double a) {
  assert(a > 0.); // no effect
  return std::sqrt(a);
}
```

Static Assert

C++ 11

Checking invariants at compile time

- To check invariants at compile time, use `static_assert`
- The assertion can be any constant expression (see later)
- The message argument is optional in C++ 17 and later

static_assert

```cpp
double f(UserType a) {
  static_assert(
    std::is_floating_point<UserType>::value,
    "This function expects floating-point types.");
  return std::sqrt(a);
}
```

```
a.cpp: In function 'double f(UserType)':
a.cpp:3:9: error: static assertion failed: This function
                                          expects floating-point types.
    2 | static_assert(
      |   std::is_floating_point<UserType>::value,
      |   ~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Object orientation (OO)

Objects and Classes

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

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

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

this keyword C++ 98

How to know an object's address?

- Sometimes we need to pass a reference to ourself 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
void freeFunc(S & s);
struct S {
  void memberFunc() { // Implicit S* parameter
    freeFunc(*this);  // Pass a reference to ourself
  }
};
S s;
s.memberFunc();        // Passes &s implicitly
```

Method overloading

The rules in C++

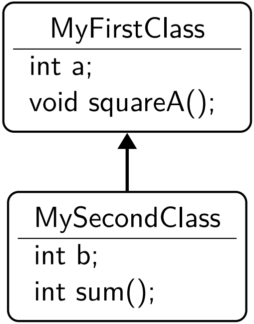
- overloading is authorized and welcome
- signature is part of the method identity
- but not the return type

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

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

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

Inheritance

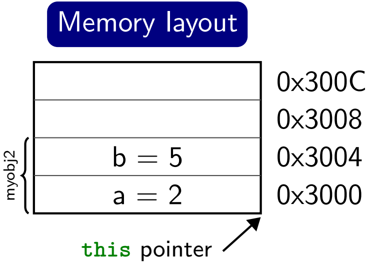
First inheritance


```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


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

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



Managing access to class members

C++ 98

`public` / `private` keywords

`private` allows access only within the class

`public` allows access from anywhere

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

Managing access to class members

public / private keywords

`private` allows access only within the class

`public` allows access from anywhere

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

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

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

Managing access to class members

public / private keywords

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

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

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

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

This breaks MySecondClass !

Managing access to class members(2)

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

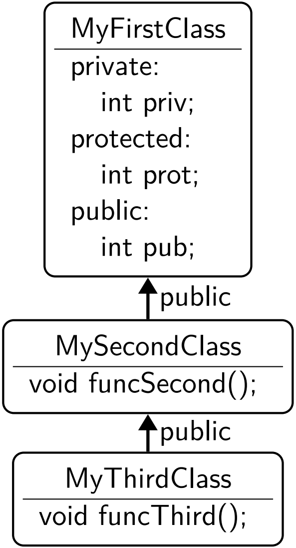
Net result for external code

- only public members of public inheritance are accessible

Net result for code in derived classes

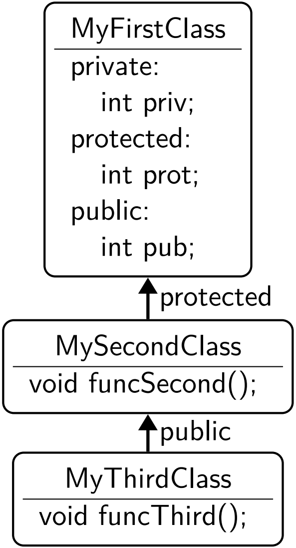
- only public and protected members of public and protected parents are accessible

Managing inheritance privacy - public



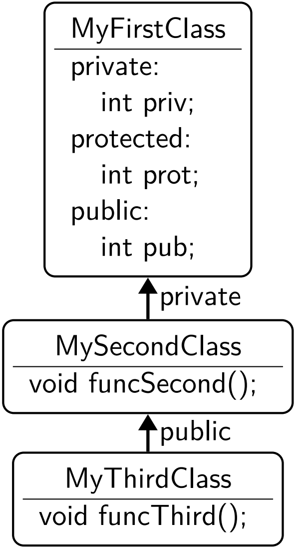

```cpp
void funcSecond() {
  int a = priv;    // Error
  int b = prot;    // OK
  int c = pub;     // OK
}
void funcThird() {
  int a = priv;    // Error
  int b = prot;    // OK
  int c = pub;     // OK
}
void extFunc(MyThirdClass t) {
  int a = t.priv; // Error
  int b = t.prot; // Error
  int c =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

Final class

C++ 11

Idea

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

Practically

```
struct Base final {
  ...
};
struct Derived : Base { // compiler error
  ...
};
```

Constructors/destructors

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

```cpp
class Vector {
public:
  Vector(int n);
  ~Vector();
  void setN(int n, int value);
  int getN(int n);
private:
  int len;
  int* data;
};
Vector::Vector(int n) : len(n) {
  data = new int[n];
}
Vector::~Vector() {
  delete[] data;
}
```

Constructors and inheritance

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

```cpp
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

C++ 98

```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

C^{++} 98

Concept

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

Example - godbolt

```
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

```cpp
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

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

Constructor inheritance

C++ 11

Idea

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

Practically

```
struct Base {
  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}; // also possible: int a = 5;
  Base() = default;
  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 {}

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

A a{1,2};     // A::A(int, int)
A a{1};       // A::A(int)
A a{};        // 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 a;
  float b;
  A();
  A(int);
  A(int, int);
};

A a(1,2);     // A::A(int, int)
A a(1);       // A::A(int)
A a();        // declaration of a function!
A a;          // A::A()
A a = (1,2); // 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

```cpp
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

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

Main characteristics

- allocation on the stack stays valid for the duration of the current scope. It is destroyed when it is popped off the stack.
- memory allocated on the stack is known at compile time and can thus be accessed through a variable.
- the stack is relatively small, it is not a good idea to allocate large arrays, structures or classes
- each thread in a process has its own stack
  - allocations on the stack are thus “thread private”
  - and do not introduce any thread safety issues

Object allocation on the stack

On the stack

- objects are created 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)

```
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
- Type casting
- Operator overloading
- Function objects
- Name Lookups

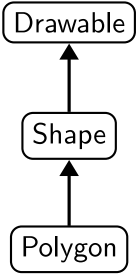
Polymorphism

C++ 98

the concept

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

```
Polygon p;

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

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



Polymorphism

the concept

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

```cpp
Polygon p;

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

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

Memory layout

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

Polygon

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	Drawable.b	0x3004
	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

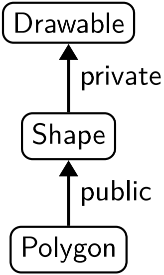
		0x3020
		0x301C
	Polygon.nLines	0x3018
Shape	...	0x3014
Shape	Shape.b	0x3010
Shape	Shape.a	0x300C
Shape	...	0x3008
Shape	Drawable.b	0x3004
Shape	Drawable.a	0x3000

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
}
```



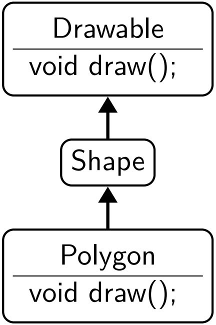
Method overriding

C++ 98

the idea

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

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

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



Virtual methods

C++ 98

the concept

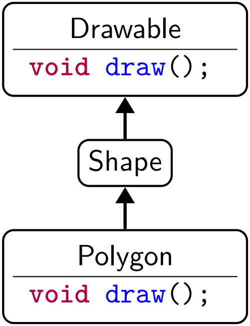
- methods can be declared `virtual`
- for these, the most derived object's implementation is used (i.e. the dynamic type behind a pointer/reference)
- for non-virtual methods, the static type of the variable decides

Virtual methods

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
```

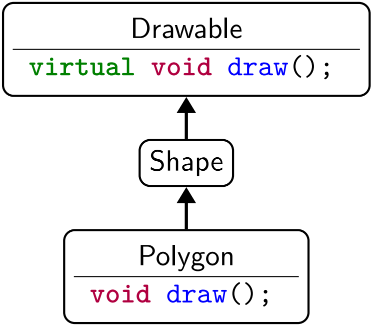


Virtual methods

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

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)

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

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

final keyword

C++ 11

Idea

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

Practically

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

Pure Virtual methods

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

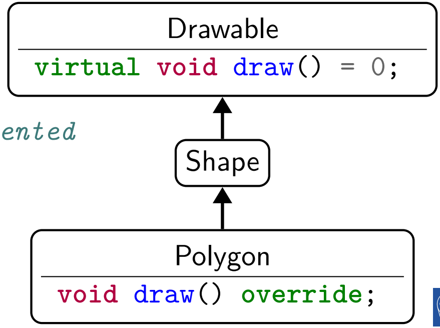
Pure Virtual methods

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

C++ 98

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:

```cpp
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

```cpp
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

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

C++ 98

Exercise: Polymorphism

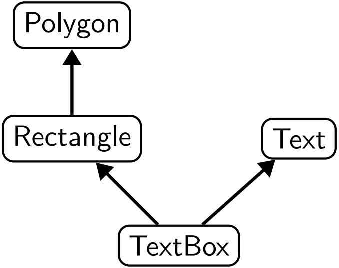
- go to `exercises/polymorphism`
- look at the code
- open trypoly.cpp
- create a Pentagon, call its perimeter method
- create a Hexagon, call its perimeter method
- create a Hexagon, call its parent's perimeter method
- retry with virtual methods

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
}
```

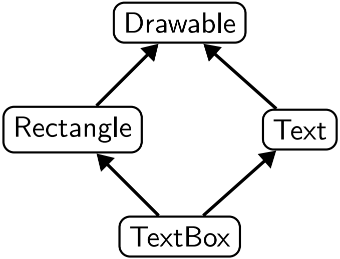
The diamond shape

Definition

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

Problem

- are the members of the grand parent replicated?



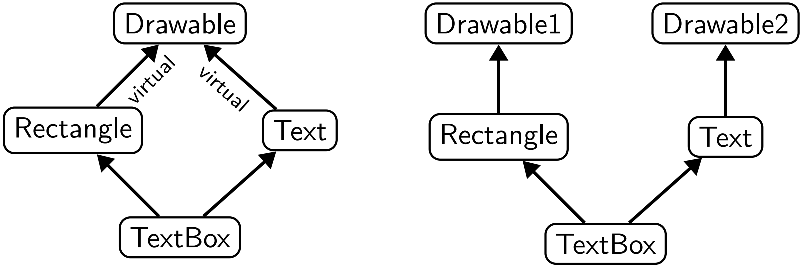
Virtual inheritance

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
- And for rare special cases

Multiple inheritance advice

C++ 98

Good practice: Avoid multiple inheritance

- Except for inheriting from interfaces
- And for rare special cases

Good practice: Absolutely avoid diamond-shaped 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

Type casting

Type casting

C^{++} 98

5 types of casts in C^{++} - These 2 should be used

- `static_cast<Target>(arg)`
  - converts type if the static types allow it
  - including using implicit conversion
  - essentially compile time
- `dynamic_cast<Target>(arg)`
  - checks if object at address of "`arg`" is convertible to `Target`
  - throws `std::bad_cast` or returns `nullptr` if it's not
  - essentially run time

Type casting example

C++ 98

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

A& c = static_cast<A&>(b); // OK. b is also an A
B& d = static_cast<B&>(a); // UB: a is not a B
B& e = static_cast<B&>(c); // OK. c is a B

B& f = dynamic_cast<B&>(c); // OK, c is a B
B& g = dynamic_cast<B&>(a); // throws std::bad_cast
B& foo(A& h) {
  return dynamic_cast<B&>(h);
}
B& i = foo(c); // OK, c is a B

B* j = dynamic_cast<B*>(&a); // nullptr. a not a B.
if (j != nullptr) {
  // Will not reach this
}
```

Type casting

C^{++} 98

5 types of casts in C^{++}- These 3 should be avoided

- `const_cast<Target>(arg)`
  - removes (or adds) constness from a type
  - if you think you need this, rather improve your design!
- `reinterpret_cast<Target>(arg)`
  - changes type irrespective of what ‘arg‘ is
  - almost never a good idea!
- C-style: `(Target)arg`
  - Force-changes type in C-style. No checks. Don’t use.

Casts to avoid

```cpp
void func(A const & a) {
  A& ra = a;                  // Error: not const
  A& ra = const_cast<A&>(a); // Compiles. Bad design!
  // Evil! Don't do this:
  B* b = reinterpret_cast<B*>(&a);
  B* b = (B*)&a;
}
```

Operator overloading

3 Object orientation (OO)

Operator overloading example

C++ 98

```cpp
struct Complex {
  float m_real, m_imaginary;
  Complex(float real, float imaginary);
  Complex operator+(const Complex& other) {
    return Complex(m_real + other.m_real,
                   m_imaginary + other.m_imaginary);
  }
};

Complex c1{2, 3}, c2{4, 5};
Complex c3 = c1 + c2; // (6, 8)
```

Operator overloading

C++ 98

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 anything)

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)

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 !!
```

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 !!
Complex operator+(float a, const Complex& obj) {
  return Complex(a + obj.m_real, obj.m_imaginary);
}
```

Other reason to have non-member operators?

Extending existing classes

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

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

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

```
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)

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
```

Name Lookups

3 Object orientation (OO)

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

Basics of name lookup

C++ 98

Example code

```
std::cout << std::endl;
```

How to find the declaration of a name?

Mainly 2 cases :

- qualified lookup, for names preceded by '::'
  - here cout and endl
  - name is only looked for in given class/namespace/enum class
- unqualified lookup
  - here for std and operator<<
  - name is looked for in a sequence of scopes until found
  - remaining scopes are not examined

Unqualified name lookup and ADL

C++ 98

Ordered list of scopes (simplified)

- file (only for global level usage)
- current namespace/block, enclosing namespaces/blocks, etc...
- current class if any, base classes if any, etc...
- for a call expression (e.g. `f(a, b)` or `a + b`), Argument Dependent Lookup (ADL)

Argument Dependent Lookup (simplified)

To find a function name (including operators), the compiler also examines the arguments. For each argument, it searches:

- class, if any
- direct and indirect base classes, if any
- enclosing namespace

ADL consequences (1)

Use standalone/non-member functions

When a method is not accessing the private part of a class, make it a function in the same namespace

Don't write :

```cpp
namespace MyNS {
  struct A {
    T func(...);
  };
}
```

Prefer :

```cpp
namespace MyNS {
  struct A { ... };
  T func(const A&, ..);
}
```

Advantages :

- minimal change in user code, `func` still feels part of class `A`
- makes sure `func` does not touch internal state of `A`

Notes :

- non-member `func` has to be in same namespace as `A`
- please avoid global namespace

ADL consequences (2)

Customization points and using

Don't write :

```
N::A a,b;
std::swap(a, b);
```

Prefer :

```
N::A a,b;
using std::swap;
swap(a, b);
```

Advantages :

- allows to use `std::swap` by default
- but benefits from any dedicated overload

```
namespace N {
  class A { ... };
  // optimized swap for A
  void swap(A&, A&);
}
```

Hidden Friends

C++ 11

Hidden friends

- Friend functions *defined* in class scope are “hidden friends”
  - Without any declaration outside the class
- They are *not* member functions (no access to `this`)

operator<< as hidden friend

```cpp
class Complex {
  float m_real, m_imag;
public:
  Complex(float, float) { ... }
  friend
  std::ostream & operator<<(std::ostream & os,
                            Complex const & c) {
    return os << c.m_real << " + " << c.m_imag << "i";
  }
};
std::cout << Complex{2.f, 3.f}; // Prints 2 + 3i
```

Hidden Friends and ADL

C++ 11

Advantages of hidden friends

- Are *only* found via ADL, e.g. `std::cout << complex;`
- Compiler needs to consider less functions during ordinary name lookup (faster compilation)
- Avoid accidental implicit conversions

Accidental conversions - two examples in one

```
std::ostream & operator<< // out-of-class definition
  (std::ostream & os, Complex const & c) { ... }
struct Fraction {
  int num, denom;
  operator Complex() const { ... } // case 1: conversion op
};
Complex::Complex(Fraction f); // or case 2: converting ctor
std::cout << Fraction{2,4}; // Prints 0.5 + 0i, would call:
//case 1: operator<<(std::cout, Fraction{2,4}.operator Complex());
//case 2: operator<<(std::cout, Complex(Fraction{2, 3}));
```

Good practice for operator overloading

C++ 98

Good practice: Operator overloading

- Must declare in-class, as member operators:
  - `operator=`, `operator()`, `operator[]`, `operator->`
- Prefer in-class declaration, as member operators:
  - compound assignment operators, e.g. `operator+=`
  - unary operators
    - e.g. `operator++`, `operator--`, `operator!`, unary `operator-` (negate), dereference operator `operator*`
- Prefer definition in-class as hidden friends:
  - binary arithmetic operators, e.g. `operator+`, `operator|`
  - binary logical operators, e.g. `operator<`
  - stream operators, e.g. `operator<<`
- Avoid overloading:
  - Address-of `operator&`
  - Boolean logic operators, e.g. `operator||`
  - Comma operator `operator,`
  - Member dereference operators `operator->*`

Core modern C^{++}

Constness

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

```cpp
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

```
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

Constant Expressions

4 Core modern C++

- Constness
- Constant Expressions
- Exceptions
- Move semantics
- Copy elision
- Templates
- Lambdas
- The STL
- More STL
- Ranges
- RAII and smart pointers
- Initialization

Generalized Constant Expressions C^{++} 14

Reason of being

- use functions to compute constant expressions at compile time

Generalized Constant Expressions C^{++} 14

Reason of being

- use functions to compute constant expressions at compile time

Example

```
constexpr int f(int x) {
  if (x > 1) return x * f(x - 1);
  return 1;
}
constexpr int a = f(5); // computed at compile-time
int b = f(5); // maybe computed at compile-time
int n = ...;  // runtime value
int c = f(n); // computed at runtime
```

Generalized Constant Expressions(2) C++ 14

Notes

- A `constexpr` function *may* be executed at compile time.
  - Arguments must be `constexpr` or literals in order to allow compile time evaluation
  - Function body must be visible to the compiler
- A `constexpr` function can also be used at runtime
- A `constexpr` variable must be initialized at compile time
- Classes can have `constexpr` member functions
- Objects used in constant expressions must be of *literal type*:
  - integral, floating-point, enum, reference, pointer type
  - union (of at least one) or array of literal types
  - class type with a `constexpr` constructor and the destructor is trivial (or `constexpr` since C++20)
- A constexpr function is implicitly `inline` (header files)

Limitations of constexpr functions

C++ 11

C++11

- Non-virtual function with a single return statement

C++14/C++17

- no try-catch, goto or asm statements
- no uninitialized/static/thread_local/non-literal-type variables

C++20

- no coroutines or static/thread_local/non-literal-type variables
- throw and asm statements allowed, but may not be executed
- transient memory allocation (memory allocated at compile-time must be freed again at compile-time)
- virtual functions and uninitialized variables allowed

C++23

- no coroutines, or execution of throw and asm statements
- transient memory allocation
- everything else allowed

Further details on cppreference

Real life example

C++ 11

```cpp
constexpr float toSI(float v, char unit) {
  switch (unit) {
  case 'k': return 1000.0f*v;
  case 'm': return 0.001f*v;
  case 'y': return 0.9144f*v;
  case 'i': return 0.0254f*v;
  ...
  default: return v;
  }
}
constexpr float fromSI(float v, char unit) {
  switch (unit) {
    case 'k': return 0.001f*v;
    case 'y': return 1.093f*v;
  ...
  }
}
```

Real life example(2)

C++ 11

```
class DimLength {
  float m_value;
public:
  constexpr DimLength(float v, char unit):
    m_value(toSI(v, unit)) {
  }
  constexpr float get(char unit) const {
    return fromSI(m_value, unit);
  }
};
constexpr DimLength km(1, 'k');
constexpr float km_y = km.get('y');
constexpr float km_i = km.get('i');
static_assert(km_y == 1093, "expected km == 1093 yards!");
```

Immediate functions C++ 20

Motivation

- Force a function to be executed at compile-time
- Runtime evaluation is forbidden
- Same restrictions as `constexpr` functions
- New keyword: `consteval`

Example

```cpp
consteval int f(int x) {
  if (x > 1) return x * f(x - 1);
  return 1;
}
constexpr int a = f(5); // computed at compile-time
int b = f(5); // computed at compile-time
int n = ...;  // runtime value
int c = f(n); // compilation error
```

constinit

C++ 20

Motivation

- Like a `constexpr` variable, a `constinit` variable guarantees compile-time initialization, but can be modified afterwards
- Only allowed for `static`/global and `thread_local` variables
- Initializer must be a constant expression, but `constexpr` destruction is not required

Example

```cpp
constexpr int f(int x) {
  if (x > 1) return x * f(x - 1);
  return 1;
}
constexpr int a = f(5); // CT init, not modifiable
int b           = f(5); // maybe CT init, modifiable
constinit int c = f(5); // CT init, modifiable
```

Exceptions

Exceptions

C++ 98

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

C++ 98

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

C++ 98

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

```
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

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` or `std::variant`
  - 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

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

Prefer

```cpp
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

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

  - either no exceptions is thrown or they are handled internally
  - checked at compile time
  - allows the compiler to optimize around that knowledge
- a function with `noexcept(expression)` is only `noexcept` when `expression` evaluates to `true` at compile-time

```
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

noexcept operator

C++ 11

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

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

```
template <typename Function>
void g(Function f) noexcept(noexcept(f())) {
  ...
  f();
}
```

Exceptions

Exception Safety Guarantees

A function can offer the following guarantees:

- No guarantee
  - An exception will lead to undefined program state
- Basic exception safety guarantee
  - An exception will leave the program in a defined state
    - At least destructors must be able to run
    - This is similar to the moved from state
  - No resources are leaked
- Strong exception safety guarantee
  - The operation either succeeds or has no effect
  - When an exception occurs, the original state must be restored
  - This is hard to do, possibly expensive, maybe impossible
- No-throw guarantee
  - The function never throws (e.g. is marked `noexcept`)
  - Errors are handled internally or lead to termination

Exceptions

Alternatives to exceptions

- The global variable `errno` (avoid)
- Return an error code (e.g. an `int` or an `enum`)
- Return a `std::error_code` (C++ 11)
- If failing to produce a value is not a hard error, consider returning `std::optional<T>` (C++ 17)
- Return `std::expected<T, E>` (C++ 23), where `T` is the return type on success and `E` is the type on failure
- Terminate the program
  - e.g. by calling `std::terminate()` or `std::abort()`
- Contracts: Specify function pre- and postconditions. Planned for C++ 20, but removed. Will come back in the future

Further resources

- Exceptions and Error Handling - ISO C++ FAQ
- Back to Basics: Exceptions - Klaus Iglberger - CppCon 2020
- The Unexceptional Exceptions - Fedor Pikus - CppCon 2015
- The Dawn of a New Error, (C++ error-handling and exceptions) - Phil Nash - CppCon 2019 (alternatives)
- Exceptions the Other Way Round - Sean Parent - CppNow 2022 (theoretical approach to errors)
- P0709: Zero-overhead deterministic exceptions: Throwing values - Herb Sutter (proposal - distant future)
  - De-fragmenting C++: Making Exceptions and RTTI More Affordable and Usable - Herb Sutter CppCon 2019

Move semantics

Move semantics: the problem

C++ 11

Inefficient code

```
void swap(std::vector<int> &a,
          std::vector<int> &b) {
  std::vector<int> c = a;
  a = b;
  b = c;
}
std::vector<int> v(10000), w(10000);
...
swap(v, w);
```

Move semantics: the problem

C++ 11

Inefficient code

```
void swap(std::vector<int> &a,
          std::vector<int> &b) {
  std::vector<int> c = a;
  a = b;
  b = c;
}
std::vector<int> v(10000), w(10000);
...
swap(v, w);
```

What happens during swap

- one allocation and one release for 10k `int`s
- a copy of 30k `int`s

Move semantics: the problem

C++ 11

Vector's built-in efficient swap

```
std::vector<int> v(10'000), w(10'000);
...
v.swap(w);
```

Move semantics: the problem

C++ 11

Vector's built-in efficient swap

```
std::vector<int> v(10'000), w(10'000);
...
v.swap(w);
```

What happens during swap

- 3 swaps of `int*` (9 copies)
- only some pointers to the underlying storage are swapped

Move semantics: the problem

Another potentially inefficient code

```
T consumeVector(std::vector<int> input) {
  // ... change elements, compute result
  return result;
}

std::vector<int> values{...};
consumeVector(values); // being copied now
```

Move semantics: the problem

C++ 98

Another potentially inefficient code

```
T consumeVector(std::vector<int> input) {
  // ... change elements, compute result
  return result;
}

std::vector<int> values{...};
consumeVector(values); // being copied now
```

Pass by copy

- One unnecessary (large?) allocation and release
- Unnecessary copy of the `ints`

Move semantics: the problem

C++ 98

Another potentially inefficient code

```
T consumeVector(std::vector<int> const & input) {
  std::vector tmp{input};
  // ... change elements, compute result
  return result;
}
std::vector<int> values{...};
consumeVector(values); // maybe copied internally?
```

Move semantics: the problem

C++ 98

Another potentially inefficient code

```
T consumeVector(std::vector<int> const & input) {
  std::vector tmp{input};
  // ... change elements, compute result
  return result;
}
std::vector<int> values{...};
consumeVector(values); // maybe copied internally?
```

Use references to avoid copies?

- Working with pass by reference only moves allocation and copy to a different place

Move semantics: the problem

C++ 98

Another potentially inefficient code

```
T consumeVector(std::vector<int> & input) {
  // ... change elements, compute result
  return result;
}

std::vector<int> values{...};
consumeVector(values); // maybe modified?
somethingElse(values); // safe to use values now???
```

So non-const references?

- Non-`const` references could work, but does the outside know that we're changing the vector?
- Could lead to bugs

Move semantics: the problem

C++ 98

Another potentially inefficient code

```
T consumeVector(std::vector<int> & input) {
  // ... change elements, compute result
  return result;
}

std::vector<int> values{...};
consumeVector(values); // maybe modified?
somethingElse(values); // safe to use values now???
```

The ideal situation

Have a way to express that we move the vector's content

Move semantics C^{++} 11

The idea

- a new type of reference: rvalue reference
  - used for move semantic
  - denoted by `&&`
- 2 new special member functions in every class:

  a move constructor similar to copy constructor

  a move assignment operator similar to assignment operator (now called copy assignment operator)

Move semantics

C++ 11

The idea

- a new type of reference: rvalue reference
  - used for move semantic
  - denoted by &&
- 2 new special member functions in every class:

  a move constructor similar to copy constructor

  a move assignment operator similar to assignment operator (now called copy assignment operator)

Practically

```cpp
T(T const & other); // copy construction
T(       T&& other); // move construction
T& operator=(T const & other); // copy assignment
T& operator=(       T&& other); // move assignment
```

Move semantics

C^{++} 11

A few points

- move constructor and assignment operator are allowed to leave the source object "empty"
  - so do not use the source object afterward
  - leave the source in a valid state (for its destructor)
- if no move semantic is implemented, copies will be performed
- the language and STL understand move semantic
- the compiler moves whenever possible
  - e.g. when passing temporaries or returning from a function

Move semantics

C++ 11

A few points

- move constructor and assignment operator are allowed to leave the source object "empty"
  - so do not use the source object afterward
  - leave the source in a valid state (for its destructor)
- if no move semantic is implemented, copies will be performed
- the language and STL understand move semantic
- the compiler moves whenever possible
  - e.g. when passing temporaries or returning from a function

Practically

```
T f() { T r; return r; } // move r out of f
T v = f(); // move returned (temporary) T into v
void g(T a, T b, T c);
g(f(), T{}, v); // move, move, copy
```

Move semantics

C++ 11

In some cases, you want to force a move

```
void swap(T &a, T &b) {
  T c = a;  // copy construct
  a = b;    // copy assign
  b = c;    // copy assign
}
```

Move semantics

C++ 11

In some cases, you want to force a move

```
void swap(T &a, T &b) {
  T c = a;  // copy construct
  a = b;    // copy assign
  b = c;    // copy assign
}
```

Explicitly request moving

- using the std::move function
- which is basically a cast to an rvalue reference

```
void swap(T &a, T &b) {
  T c = std::move(a);       // move construct
  a = std::move(b);         // move assign
  b = static_cast<T&&>(c); // move assign (don't)
}
```

Move semantics: recommended implementation C++ 11

Use copy and swap idiom

- implement an efficient swap function for your class
  - preferably hidden friend and symmetric
- move constructor
  - consider delegating to default constructor
  - swap `*this` with parameter (source)
- move assignment as `operator=(T source)`
  - parameter passed by value; caller can move or copy into it
  - swap parameter with `*this`
  - end of scope: parameter destroys former content of `*this`
- alternative: move assignment as `operator=(T&& source)`
  - swap parameter with `*this`
  - 1 swap less, separate copy assignment operator needed
  - former content of `*this` destroyed with caller argument
- swap, move constructor/assignment must be `noexcept`

Move semantics: recommended implementation C++ 11

Practically

```
class Movable {
  Movable();
  Movable(const Movable &other);
  Movable(Movable &&other) noexcept :
    Movable() {          // constructor delegation
    swap(*this, other);
  }
  Movable& operator=(Movable other) noexcept { // by value
    swap(*this, other);
    return *this;
  }
  friend void swap(Movable &a, Movable &b) noexcept {...}
};
Movable a, b;
a = b;               // operator= copies b into "other"
a = std::move(b); // operator= moves b into "other"
```

Move semantics: alternative implementation

Practically

```cpp
class Movable {
  Movable();
  Movable(const Movable &other);
  Movable(Movable &&other) noexcept :
    Movable() {          // constructor delegation
    swap(*this, other);
  }
  Movable& operator=(const Movable& other);
  Movable& operator=(Movable&& other) noexcept {
    swap(*this, other);
    return *this;
  }
  friend void swap(Movable &a, Movable &b) noexcept { ... }
};
```

Move Semantic

C++ 11

Exercise: Move semantics

- go to `exercises/move`
- look at the code and run it with callgrind
- understand how inefficient it is
- implement move semantic the easy way in NVector
- run with callgrind and see no improvement
- understand why and fix test.cpp
- see efficiency improvements

prerequisite: be able to use simple templated code

Copy elision

4 Core modern C++

- Constness
- Constant Expressions
- Exceptions
- Move semantics
- Copy elision
- Templates
- Lambdas
- The STL
- More STL
- Ranges
- RAII and smart pointers
- Initialization

Copy elision

C++ 17

Copy elision

```cpp
struct Foo { ... };
Foo f() {
  return Foo(); // return-value opt. (RVO)
}
Foo g() {
  Foo foo;
  return foo; // named return-value opt. (NRVO)
}
int main() {
  // compiler must avoid these copies:
  Foo a = f();
  Foo b = Foo();
  Foo c = Foo(Foo());
  // copy elision allowed, but not guaranteed:
  Foo d = g();
}
```

Guaranteed copy elision C++ 17

Allows to write code not allowed in C++ 14 (would not compile)

Guaranteed: Return value optimization (RVO)

```cpp
struct Foo {
  Foo() { ... }
  Foo(const Foo &) = delete;
  Foo(Foo &&) = delete;
  Foo& operator=(const Foo &) = delete;
  Foo& operator=(Foo &&) = delete;
};
Foo f() {
  return Foo();  // ok, no move
}
int main() {
  Foo foo = f(); // ok, no move
}
```

Templates

4 Core modern C++

- Constness
- Constant Expressions
- Exceptions
- Move semantics
- Copy elision
- **Templates**
- Lambdas
- The STL
- More STL
- Ranges
- RAII and smart pointers
- Initialization

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

```cpp
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

C++ 98

Warning

- 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

```
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 C^{++} 98

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) {
  ...
}
```

Nested templates

Nested templates - godbolt

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

template<typename KeyType, typename ValueType> //for class
template<typename OtherKeyType>       //for member function
ValueType Map<KeyType, ValueType>::get
   (const OtherKeyType &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

C++ 98

Specialization

Templates can be specialized for given values of their parameter

```
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;
};
```

Template argument deduction

Template argument deduction

- Template arguments deduced from (function) arguments
- Template arguments can always be specified explicitly
- Only for function templates (Before C++ 14)

```
template <typename T, typename U>
void f(T t, U u) { ... }

f(42, true);          // deduces T=int, U=bool
                      // calls f<int, bool>(42, true);
f<float>(42, true); // sets T=float, deduces U=bool
                      // calls f<float, bool>(42, true);
                      // 42 converted to float before call
```

Template argument deduction

C++ 11

Deduced contexts

- Compiler can even deduce template arguments inside certain expressions (pattern matching)
- See cppreference for details

```cpp
template <typename T>
void f(T* p) { ... }

const int * ip = ...;
f(ip); // deduces T=const int

template <typename T, std::size_t N>
void g(std::array<T*, N> a) { ... }

std::array<int*, 3> aip = ...;
g(aip); // deduces T=int, N=3
```

Template argument deduction

C++ 11

Non-deduced contexts

- Deduction from certain expressions is impossible/forbidden

```cpp
template <typename C>
void f(typename C::value_type v) { ... }
f(std::vector<int>{...}); // cannot deduce C
                          // from a dependent type
template <typename T, std::size_t N>
void g(std::array<T, N * 2> a) { ... }
g(std::array<int, 4>{...}); // deduces T=int,
                            // cannot deduce N from expression
template <typename T>
void h(std::vector<T> v) { ... }
h({1, 2, 3}); // error, braced-initializer list has no type
h(std::vector<int>{1, 2, 3}); // ok, T=int
```

Template argument deduction

C++ 11

Non-deduced contexts

- Deduction from certain expressions is impossible/forbidden

```cpp
template <typename C, typename F>
void reduce(const C& cont, const F& f = std::plus{});
reduce(std::vector<int>{...}); // error: cannot deduce F
// would need: <typename C, typename F = decltype(std::plus{})>

template<typename T>
const T & max(const T &a, const T &b) { ... }
int i = 3;
max(i, 3.14f); // deduces T=int and T=float, error
// either: max<float>(i, 3.14f);
// or:     max(static_cast<float>(i), 3.14f);
```

Template argument deduction

C++ 11

Deduction in partial specializations

- Partial specializations also deduce template arguments
  - with similar rules as for function templates
  - but some more restrictions (cf. cppreference)

```
template <typename T>
struct S { ... }; // primary template

template <typename T>
struct S<T*> { ... }; // specialization 1

template <typename U, std::size_t N>
struct S<std::array<U*, N>> { ... }; // specialization 2

S<int>  s1; // prim. tmpl. (T=int) ok, spec. 1/2 invalid
S<int*> s2; // prim. tmpl. (T=int*) and spec. 1 (T=int) ok
            // spec. 1 is more specialized, will be used
```

Template argument deduction

C^{++} 98

Specialization vs. overloading

- For function templates we can choose between specialization and overloading
- Partial specialization of function templates is forbidden

```cpp
template <typename T>
void f(T t) { ... }


void f(int* t) { ... }


template <typename T>
void f(T* t) { ... }
```

```cpp
template <typename T>
void f(T t) { ... }

template <>
void f<int*>(int* t) { ... }

// part. spec. forbidden:
template <typename T>
void f<T*>(T* t) {...}
```

Template argument deduction

C++ 11

Disadvantages of specialization vs. overloading

- Specialization always needs a primary template
  - Sometimes this does not make sense
- Partial specializations of function templates is forbidden
  - So we need SFINAE workarounds or concepts

Could you express this with specializations?

```cpp
template <typename T>
void f(T* p) { ... }

template <typename T>
void f(std::unique_ptr<T> p) { ... }
```

Good practice: Specialization vs. overloading

Prefer overloading function templates over template specialization

Class Template Argument Deduction (CTAD) C^{++} 17

CTAD

- Deduce the template arguments for a class template
- Based on construction arguments
- Only when no template arguments provided
- Since C^{++} 20: CTAD for aggregates (no constructor needed)

```
template<typename A, typename B, typename C = double>
struct Triple {
  Triple(A a, B b, C c) : a(a), b(b), c(c) {} // C++17
  A a; B b; C c;
};

Triple t{42, true, 3.14}; // Triple<int, bool, double>
Triple<int> t{42, true, 3.14}; // compilation error
Triple<int, bool> t{42, true, 3.14}; // not CTAD
```

Class Template Argument Deduction (CTAD) C++ 17

Deduction guides

- Describe how constructor argument types are mapped to class template arguments

```
template<typename A, typename B>
struct Pair {
 Pair(A a, B b) : a(a), b(b) {}
 A a; B b;
};



Pair p{42, "hello"}; // Pair<int, const char*>
```

Class Template Argument Deduction (CTAD) C++ 17

Deduction guides

- Describe how constructor argument types are mapped to class template arguments

```cpp
template<typename A, typename B>
struct Pair {
  Pair(A a, B b) : a(a), b(b) {}
  A a; B b;
};
template<typename A>
Pair(A, const char*) -> Pair<A, std::string>;

Pair p{42, "hello"}; // Pair<int, std::string>
```

Class Template Argument Deduction (CTAD) C++ 17

Standard library examples

```
std::pair p{1.2, true}; // std::pair<double, bool>
std::tuple t{1.2, true, 32};
                   // std::tuple<double, bool, int>
std::vector v{1, 2, 3}; // std::vector<int>
std::list l{v.begin(), v.end()}; // std::list<int>
std::array a{1, 2, 3}; // std::array<int, 3>

std::mutex m;
std::lock_guard l(m); // std::lock_guard<std::mutex>
```

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
   - Constant Expressions
   - Exceptions
   - Move semantics
   - Copy elision
   - Templates
   - Lambdas
   - The STL
   - More STL
   - Ranges
   - RAII and smart pointers
   - Initialization

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

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

Advantages

- Allows to simplify inner type definition

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

- Used by lambda expressions

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

```
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);
```

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`

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);
```

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 consequence

- 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](...) { ... };
```

Capture list - this

C++ 11

Inside a (non-static) member function, we can capture `this`.

```
[    this](...) { use(*this); };
[       &](...) { use(*this); };
[&, this](...) { use(*this); };
[       =](...) { use(*this); }; // deprecated in C++20
[=, this](...) { use(*this); }; // allowed in C++20
```

Since the captured `this` is a pointer, `*this` refers to the object by reference.

Capture list - this

C++ 11

Inside a (non-static) member function, we can capture `this`.

```
[    this](...) { use(*this); };
[       &](...) { use(*this); };
[&, this](...) { use(*this); };
[       =](...) { use(*this); }; // deprecated in C++20
[=, this](...) { use(*this); }; // allowed in C++20
```

Since the captured `this` is a pointer, `*this` refers to the object by reference.

```
[   *this](...) { use(*this); }; // C++17
[&, *this](...) { use(*this); }; // C++17
[=, *this](...) { use(*this); }; // C++17
```

The object at `*this` is captured by value (the lambda gets a copy).

Details in this blog post.

Generic lambdas

C++ 14

Generic lambdas (aka. polymorphic lambdas)

- The type of lambda parameters may be `auto`.

```
auto add = [](auto a, auto b) { return a + b; };
```

- The generated `operator()` becomes a template function:

```
template <typename T, typename U>
auto operator()(T a, U b) const { return a + b; }
```

- The types of `a` and `b` may be different.

Explicit template parameters (C++ 20)

```
auto add = []<typename T>(T a, T b)
  { return a + b; };
```

- The types of `a` and `b` must be the same.

Higher-order lambdas

C++ 11

Example - godbolt

```cpp
auto build_incrementer = [](int inc) {
  return [inc](int value) { return value + inc; };
};
auto inc1 = build_incrementer(1);
auto inc10 = build_incrementer(10);
int i = 0;
i = inc1(i);   // i = 1
i = inc10(i);  // i = 11
```

How it works

- build_incrementer returns a function object
- this function's behavior depends on a parameter
- note how auto is useful here!

Lambda improvements

C++ 20

Lambda improvements in C++ 20

- Allowed in unevaluated contexts, e.g. within **decltype**, **sizeof**, **typeid**, etc.
- Without captures, are default-constructible and assignable

Examples

```
struct S {
  decltype([](int i) { std::cout << i; }) f;
} s;
s.f(42); // prints "42"

template <typename T>
using CudaPtr = std::unique_ptr<T,
                      decltype([](T* p){ cudaFree(p); })>;

std::set<T, decltype([](T a, T b) { ... })> s2;
```

The STL

4 Core modern C++

- Constness
- Constant Expressions
- Exceptions
- Move semantics
- Copy elision
- Templates
- Lambdas
- The STL
- More STL
- Ranges
- RAII and smart pointers
- Initialization

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

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

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

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>`

```
#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
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

C^{++} 98

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

```
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)

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

Range-based for loops with STL containers

C++ 11

Iterator-based loop (since C++ 98)

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

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

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

Range-based for loops with STL containers 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!

C++ 98

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!

More STL

4 Core modern C++

- Constness
- Constant Expressions
- Exceptions
- Move semantics
- Copy elision
- Templates
- Lambdas
- The STL
- More STL
- Ranges
- RAII and smart pointers
- Initialization

std::string_view cppreference C^{++} 17

Non owning read-only view of a contiguous char sequence

- Doesn't allocate memory
- Similar interface to std::string, but read-only
- Easy to copy, faster for some calls eg. substr (O(1) vs O(n))
- The data pointed to has to outlive the string_view

Some example uses

```cpp
constexpr std::string_view sv {"Some example"};
auto first = sv.substr(0, sv.find_first_of(" "));
std::string some_string {"foo bar"};
std::string_view sv_str(some_string);
char foo[3] = {'f','o','o'};
std::string_view carr(foo, std::size(foo));
```

`std::span<T>` - Concept cppreference

C++ 20

Non owning view of a contiguous sequence of items

- Doesn't allocate memory
- Allows to modify items, but not the container itself
  - no reallocation, push or pop
- Provides container like interface
  - iterators, `size`, `front`/`back`, `subspan`, ...
- Standard contiguous containers convert to it automatically
  - `std::vector`, `std::array`, C arrays, ...
  - not `std::list` or `std::deque` (no contiguous storage)
  - `span` should thus be preferred as function argument
- Simply a pair (pointer, size), so cheap to copy
  - and thus to be passed by value
- `span` can also have a `static extend`
  - meaning the size is determined at compile time
  - and thus only a pointer is stored by `span`

std::span<T> - Usage

C++ 20

Some example - godbolt

```cpp
void print(std::span<int> c) {
  for (auto a : c) { std::cout << a << " "; }
  std::cout << "\n";
}
void times2(std::span<int> c) {
  for(auto &e : c) { e *= 2; }
}
int a[]{23, 45, 67, 89};
print(a);                   // 23 45 67 89

std::vector v{1, 2, 3, 4, 5};
std::span<int> sv = v;
print(sv.subspan(2, 2));  // 3 4

std::array a2{-14, 55, 24};
times2(a2);
print(a2);                  // -28 110 48

std::span<int, 3> sa2 = a2;
std::cout << sizeof(sv) << " " << sizeof(sa2) << "\n";  // 16 8
std::span<int, 3> s2a2 = a; // compilation failure, invalid conversion
```

std::optional cppreference

C^{++} 17

Manages an optionally contained value

- Contextually converts to bool, telling if it contains something
- Has value semantics (Copy, Move, Compare, stack alloc.)
- Useful for the return value of a function that may fail
- Useful in place of pointers where value semantics are intuitive

```
std::optional<Phone> parse_phone(std::string_view in) {
  if (is_valid_phone(in))
    return in;   // equiv. to optional<Phone>{in};
  return {};   // or: return std::nullopt; (empty opt.)
}
if (v) {                         // or: v.is_valid()
  process_phone(v.value()); // or: *v (unchecked)
  v->call(); // calls Phone::call()   (unchecked)
}
v.reset(); assert(!v.has_value()); // or: v = {};
```

std::optional

C++ 17

Code example

```cpp
// address may be given for a person or not
void processPerson(std::string_view name,
                   std::optional<Address> address);
Address addr = wonderland();
processPerson("Alice", std::move(addr));

// Every person has a name, but not always an address
struct Person {
  std::string name;
  std::optional<Address> address;
};
std::vector<Person> ps = ...;
std::sort(ps.begin(), ps.end(),
          [](const Person& a, const Person& b) {
  return a.address < b.address;
}); // sorts by address, persons w/o address at front
```

std::optional

C++ 17

Exercise: Handling optional results

- go to `exercises/optional/optional.cpp`
- modify `mysqrt` so to return an `std::optional<double>`
- guess the consequences on `square`

std::variant cppreference C^{++} 17

A type-safe union

- Allows the variable to hold any of the given types
- `std::get` reads the value of the variant
- and throws `std::bad_variant_access` for bad accesses
- Currently held alternative can be probed

Code example

```cpp
std::variant<int, float, string> opt{100}; // holding int
int ival = std::get<int>(opt); // or std::get<0>(opt)
try {
  float val = std::get<float>(opt) // will throw
} catch (std::bad_variant_access const& ex) {...}
if (std::holds_alternative<float>(opt))
    std::cout << std::get<float>(opt);
if (const float* v = std::get_if<float>(&opt)) { ... }
const int current_alternative = opt.index(); // == 0
```

std::variant and the visitor pattern

C++ 17

std::visit cppreference

- Applies a "visitor" to a given variant
  - A visitor is a callable able to handle all different types
  - Must return `void` or the same type for all overloads

Practically - godbolt

```cpp
struct Visitor {
  auto operator()(int i)   {return "i:"+std::to_string(i);}
  auto operator()(float f) {return "f:"+std::to_string(f);}
  auto operator()(const std::string& s) { return "s:"+s;}
  template <typename T>
  auto operator()(const T& t) { return std::string{"?"}; }
};
void print(std::variant<int, float, std::string, char> v) {
  std::cout << std::visit(Visitor{}, v) << '\n';
}
print(100); print(42.0f); print("example"); print('A');
```

std::variant and the lambda visitor pattern

C++ 17

Idea

- use inheritance to group a set of lambdas

Practically - godbolt

```
template<typename... Ts> // covered in expert part
struct Overload : Ts ... { using Ts::operator()...; };
template<typename... Ts>
Overload(Ts...) -> Overload<Ts...>; // obsolete in C++20
int main(){
  std::variant<int, float, std::string, char> v{ ... };
  auto overloadSet = Overload {
    [](int i) { std::cout << "i32:" << i;},
    [](std::string s) { std::cout << s;},
    [](auto a) { std::cout << "other"; },
  };
  std::visit(overloadSet, v);
}
```

std::variant exercise

C++ 17

Exercise: An alternative to oo polymorphism

- go to `exercises/variant/variant.cpp`
- replace inheritance from a common base class with an `std::variant` on the three kinds of particle.
- two solutions given : with `std::get_if` and with `std::visit`.

std::any cppreference

C++ 17

a type-safe container for single values of any type

- Allows a variable to hold any type (say bye to `void*`)
- `std::any_cast` reads the internal value
- and throws `std::bad_any_cast` for bad accesses
- `std::any_cast` only matches types 1:1, ignoring inheritance

Code example

```cpp
std::any val{100};           // holding int
val = std::string("hello"); // holding string
std::string s = std::any_cast<std::string>(val);
try {
  int val = std::any_cast<int>(val); // will throw
} catch (std::bad_any_cast const& ex) {...}
if (val.type() == typeid(int)) // requires RTTI
  std::cout << std::any_cast<int>(val);
val.reset(); assert(!val.has_value());
```

non-member begin and end

C++ 11

The problem in C++ 98

STL containers and arrays have different syntax for loop

```
std::vector<int> v;
int a[] = {1,2,3};
for(auto it = v.begin(); it != v.end(); it++) {...}
for(int i = 0; i < 3; i++) {...}
```

non-member begin and end C^{++} 11

The problem in C^{++} 98

STL containers and arrays have different syntax for loop

```cpp
std::vector<int> v;
int a[] = {1,2,3};
for(auto it = v.begin(); it != v.end(); it++) {...}
for(int i = 0; i < 3; i++) {...}
```

A new syntax

```cpp
for(auto it = begin(v); it != end(v); it++) {...}
for(auto i = begin(a); i != end(a); i++) {...}
```

non-member begin and end

The problem in C++ 98

STL containers and arrays have different syntax for loop

```cpp
std::vector<int> v;
int a[] = {1,2,3};
for(auto it = v.begin(); it != v.end(); it++) {...}
for(int i = 0; i < 3; i++) {...}
```

A new syntax

```cpp
for(auto it = begin(v); it != end(v); it++) {...}
for(auto i = begin(a); i != end(a); i++) {...}
```

Allowing the best syntax

```cpp
for(auto & element : v) {...}
for(auto & element : a) {...}
```

std::tuple cppreference C++ 11

```cpp
#include <tuple>

std::tuple<int, float, bool> t{42, 3.14f, false};
auto t = std::make_tuple(42, 3.14f, false);
std::tuple t{42, 3.14f, false}; // C++17 CTAD

float& f = std::get<1>(t);    // get 1. member
bool& b = std::get<bool>(t); // get bool member

using Tup = decltype(t);
constexpr auto s = std::tuple_size_v<Tup>; // 3
using E2Type = std::tuple_element<2, Tup>; // bool

auto t2 = std::tuple_cat(t, std::tuple{'A'}, t);
  // <int, float, bool, char, int, float, bool>
bool& b2 = std::get<bool>(t2); // error
```

std::tuple

C++ 11

```cpp
void f(std::tuple<int, float, bool> tuple) {
  int a; float b; bool c;
  std::tie(a, b, c) // tuple<int&, float&, bool&>
    = tuple;
  // Use structured bindings in C++17
}

int& i = ...;
std::tuple{i};                   // tuple<int>
std::make_tuple(i);              // tuple<int>
std::make_tuple(std::ref(i)); // tuple<int&>
std::tuple{std::ref(i)};
  // C++17 CTAD, tuple<std::reference_wrapper<int>>
```

Structured Binding Declarations

C++ 17

- Helps with `std::tuple`, tuple-like, array or aggregate types
- Automatically creates variables and ties them
- CV qualifiers and references allowed

```cpp
std::tuple<int, double, long> tuple = ...;
auto [ a, b, c ] = tuple;

struct S { int i; float f; bool b; } s = ...;
auto&& [ a, b, c ] = std::move(s);

int arr[] = {1, 3, 4};
const auto& [ a, b, c ] = arr;

std::unordered_map<K, V> map = ...;
for (const auto& [key, value] : map) { ... }
```

Structured Binding Declarations

C++ 17

"Tuple-like binding prototcol"

- For non-aggregates, you can opt-in to structured bindings
- The compiler requires these hooks:
  - Specialize `std::tuple_size`
  - Specialize `std::tuple_element`
  - Provide either `T::get<I>()` or `get<I>(T)`

```
class C { ... }; // complex, cannot change
template <>
struct std::tuple_size<C> {
  static constexpr std::size_t value = ...;
};
template <std::size_t I>
struct std::tuple_element<I, C> { using type = ...; };
template <std::size_t I>
auto& get(C& c) { ... }; // -> std::tuple_element<I, C>&
```

compile-time branches

C++ 17

if constexpr

- takes a `constexpr` expression as condition
- evaluates at compile time
- key benefit: the discarded branch can contain invalid code

Example code

```cpp
template <typename T>
auto remove_ptr(T t) {
  if constexpr (std::is_pointer_v<T>) {
    return *t;
  } else {
    return t;
  }
}
int i = ...; int *j = ...;
int r = remove_ptr(i);  // equivalent to i
int q = remove_ptr(j);  // equivalent to *j
```

Pseudo-random number generators (PRNG) C++ 11

Concept

- For generating pseudo-random numbers the STL offers:
  - engines and engine adaptors to generate pseudo random bits
  - distributions to shape these into numbers
  - access to (potential) hardware entropy
- All found in header <random>

Example

```
#include <random>
std::random_device rd;
std::default_random_engine engine{rd()};
std::normal_distribution dist(5.0, 1.5); // μ, σ
double r = dist(engine);
```

Engines

C++ 11

Engines

- Generate random bits (as integers), depending on algorithm
- Can be seeded via constructor or `seed()`
- Algorithms: `linear_congruential_engine`, `mersenne_twister_engine`, `subtract_with_carry_engine`
- Adaptors: `discard_block_engine`, `independent_bits_engine`, `shuffle_order_engine`
- Aliases: `minstd_rand0`, `minstd_rand`, `mt19937`, `mt19937_64`, `ranlux24`, `ranlux48`, `knuth_b` (use these)
- `std::default_random_engine` aliases one of the above

Engine creation with seed

```
std::default_random_engine engine{42}; // or empty
auto i = engine(); // operator(): random integer
```

Determinism and random device

C++ 11

Engines / PRNGs

- Are *deterministic*
- Will produce the same number sequence for the same seed
- May be useful for reproducibility

std::random_device

- Provides *non-deterministic* random numbers
- Uses hardware provided entropy, if available
- May become slow when called too often, e.g. when hardware entropy pool is exhausted
- Use only to seed engine, if *non-deterministic* numbers needed

Non-deterministic engine seeding

```
std::random_device rd;
std::default_random_engine engine{rd()};
```

Distributions

C++ 11

Distributions

- Take random bits from engines and shape them into numbers
- Standard library provides a lot of them
  - Uniform, bernoulli, poisson, normal and sampling distributions
  - See cppreference for the full list

Distributions sharing a non-deterministic engine

```cpp
std::random_device rd;
std::default_random_engine engine{rd()};
std::uniform_int_distribution<int> dist(2,7);//min,max
const int size = dist(engine); // gen. random number
std::vector<int> v(size);
std::normal_distribution<double> norm(5.0, 1.5); //μ,σ
std::generate(begin(v), end(v),
              [&]{ return norm(engine); });
```

C random library

C++ 98

C random library

- The C library (`<cstdlib>`) offers a primitive PRNG:
- `rand()` returns random int between 0 and `RAND_MAX` (incl.)
- `srand(seed)` sets the global seed

Disadvantages

- Seeding is not thread-safe; `rand()` may be.
- Returned numbers have small, implementation-defined range
- No guarantee on quality
- Mapping a random number to a custom range is hard
- E.g.: `rand() % 10` yields biased numbers and is wrong!

Good practice: Strongly avoid the C random library

Use the C++ 11 facilities from the `<random>` header

Ranges

4 Core modern C++

Ranges

C^{++} 20

The range concept

- A range is a sequence of items that you can iterate over
- It must provide iterators to its beginning and end, obtainable via `std::range::begin` and `std::range::end`
- Range concepts are located in `std::ranges` namespace
- All STL containers are ranges

Variations on ranges

There are actually different types of ranges

- `input_range`, `output_range`, `forward_range`, `bidirectional_range`, `random_access_range`, `contiguous_range`
- corresponding to the different iterator categories

Views

C++ 20

The view concept

- A `view` is a non-owning object built from a range or another view after applying some range adaptor(s)
- The time complexity to copy, move, assign is constant
- Range adaptors can be applied/chained with `|` and are lazy
- They live in the `std::views` namespace
- They introduce easy functional programming to the STL

Example - godbolt

```cpp
auto const numbers = std::views::iota(0, 6);
auto even = [](int i) { return 0 == i % 2; };
auto square = [](int i) { return i * i; };
auto results = numbers | std::views::filter(even)
                       | std::views::transform(square);
for (auto a : results) { ... } // 0, 4, 16
```

Many range adaptors are provided by the STL C^{++} 20

Useful subset

all	a view on all elements
filter	applies a filter
transform	applies a transformation
take	takes only first n elements
drop	skips first n elements
join/split	joins ranges or splits into subranges
reverse	reverses view order
elements	for tuple views, extract nth elements of each tuple
keys/values	for pair like views, takes 1st/2nd element of each pair

Remember you can combine them via (|)

Back to laziness

C++ 23

Views are lazy

- Computation is only triggered when iterating
- And can be stopped by e.g. take
- Here, the minimal number of iterations is performed

Example - godbolt

```
// print first 20 prime numbers above 1000000
for (int i: std::views::iota(1000000)
            | std::views::filter(odd)
            | std::views::filter(isPrime)
            | std::views::take(20)) {
  std::cout << i << " ";
}
```

RAII and smart pointers

4 Core modern C++

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

```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);
```

They need clear ownership

```cpp
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

```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);
```

They need clear ownership — Who should release ?

```cpp
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

```
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

```
#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?

```
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

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

Arrays and unique_ptr

C++ 11

Dynamic arrays

- unique_ptr can wrap arrays
- Use T[] as template parameter
- This will enable the subscript operator
- The default constructor is called for each element
- If size known at compile time, prefer std::array
- If size might change, prefer std::vector

```
auto b = std::make_unique<Foo[]>(10);
b[3] = ...;
b[4].someFunction();

// deallocations 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

```
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

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

weak_ptr

C++ 11

weak_ptr: a non-owning observer

- Sometimes want to observe resources without keeping them alive
  - shared_ptr? Resource stays alive
  - Raw pointer? Risk of dangling pointer
- The solution is to construct a weak_ptr from a shared pointer
- To access the resource, convert the weak into a shared_ptr

```cpp
std::shared_ptr<Cache> getSharedCache();
std::weak_ptr<Cache> weakPtr{ getSharedCache() };
// ... shared cache may be invalidated here
if (std::shared_ptr<Cache> cache = weakPtr.lock()) {
  // Cache is alive, we actively extend its lifetime
  return cache->findItem(...);
} else {
  // Cache is nullptr, we need to do something
  weakPtr = recomputeCache(...);
```

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

```
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?

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

Quiz: shared_ptr and weak_ptr in use C^{++} 11

What is the output of this code? - godbolt

```
auto shared = std::make_shared<int>(100);
std::weak_ptr<int> weak{ shared };
print(); // with print as before

auto ptr = weak.lock();
(*ptr)++;          print();

ptr = nullptr;     print();

function(weak);    print();
```

Quiz: shared_ptr and weak_ptr in use

C++ 11

What is the output of this code? - godbolt

```
auto shared = std::make_shared<int>(100);
std::weak_ptr<int> weak{ shared };
print(); // with print as before

auto ptr = weak.lock();
(*ptr)++;         print();

ptr = nullptr;    print();

function(weak);   print();
```

```
Use: 1 value: 100
Use: 2 value: 101
Use: 1 value: 101
Use: 1 (or more) value: ???
```

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.

Initialization

4 Core modern C++

- Constness
- Constant Expressions
- Exceptions
- Move semantics
- Copy elision
- Templates
- Lambdas
- The STL
- More STL
- Ranges
- RAII and smart pointers
- Initialization

Initialization

Initializing scalars

```cpp
int i(42);      // direct initialization
int i{42};      // direct list initialization (C++11)
int i = 42;     // copy initialization
int i = {42};   // copy list initialization (C++11)
```

- All of the above have the same effect: `i == 42`

Narrowing conversions

```cpp
int i(42.3);      // i == 42
int i{42.3};      // compilation error
int i = 42.3;     // i == 42
int i = {42.3};   // compilation error
```

- Braced initialization prevents narrowing conversions

Initialization

Initializing class types with constructors

```
struct C {
  C(int i);
  C(int i, int j);
};

C c(42);     // calls C(42)
C c{42};     // same
C c = 42;    // calls C(42) in C++17, before C(C(42))
C c = {42}; // same

C c(1, 2);     // calls C(1, 2)
C c{1, 2};     // calls C(1, 2)
C c = (1, 2); // calls C(2), comma operator
C c = {1, 2}; // calls C(1, 2) in C++17, before C(C(1, 2))

C c(1.1, 2.2); // calls C(1, 2)
C c{1.1, 2.2}; // compilation error
```

Initialization

std::initializer_list

```cpp
struct C {
  C(int i, int j);
  C(std::initializer_list<int> l);
};

C c(1, 2);     // calls C(1, 2)
C c{1, 2};     // calls C(std::initializer_list<int>{1, 2})
C c = (1, 2); // calls C(2), comma operator, error
C c = {1, 2}; // calls C(std::initializer_list<int>{1, 2})
```

- A constructor with a `std::initializer_list` parameter takes precedence over other overloads.

Beware

```cpp
std::vector<int> v1(4, 5); // {5, 5, 5, 5}
std::vector<int> v2{4, 5}; // {4, 5}
```

Initialization

Type deduction

```
auto i(42);      // int
auto i{42};      // C++11: std::initializer_list<int>
                 // since C++17: int
auto i = 42;     // int
auto i = {42};   // std::initializer_list<int>
```

Initialization

C++ 11

Default initialization

```
T t;
```

- If `T` has a default constructor (including compiler generated), calls it
- If `T` is a fundamental type, `t` is left uninitialized (undefined value)
- If `T` is an array, the same rules are applied to each item

Initialization

C++ 11

Value initialization

```
T t{};      // value initialization
T t = {}; // same
f(T());     // passes value-initialized temporary
f(T{});     // same
T t();      // function declaration
```

- Basically zero-initializes t (or the temporary), unless T has a user defined constructor, which is run in this case
- Details on cppreference

Initialization

C++ 11

Aggregate initialization

```cpp
struct S { int i; double d; };

S s{1, 2.3};                // s.i == 1, s.d == 2.3
S s = {1, 2.3};             // same
S s{.i = 1, .d = 2.3}       // same, designated init. (C++20)
S s = {.i = 1, .d = 2.3} // same
S s{1};          // s.i == 1, s.d == 0.0
S s = {1};       // same
S s{.i = 1}      // same (C++20)
S s = {.i = 1} // same (C++20)
S s{}; // value init, s.i == 0, s.d == 0.0
S s;   // default init, undefined values
```

- An aggregate is a class with no constructors, only public base classes (C++ 17) and members, no virtual functions, no default member initializers (until C++ 14)
- Details on cppreference

Initialization

C++ 11

Good practice: Initialization

- In generic code, for a generic type `T`:
  - `T t;` performs the minimally necessary initialization
  - `T t{};` performs full and deterministic initialization
- Prefer `T t{};` over `T t; memset(&t, 0, sizeof(t));`
- Prefer braces over parentheses to avoid accidental narrowing conversions. E.g. `T t{1, 2};`
  - Unless `T` has a `std::initializer_list` constructor that you do not want to call!
  - Be wary of `std::vector`!
- The STL value initializes when creating new user objects
  - E.g. `vec.resize(vec.size() + 10);`
- Aggregates are very flexible. If your class does not need special initialization, make it an aggregate (rule of zero)

Initialization

Further resources

- The Nightmare of Initialization in C++ - Nicolai Josuttis - CppCon 2018
- Initialization in modern C++ - Timur Doumler - Meeting C++ 2018

Expert C++

The <=> operator

5 Expert C++

- The <=> operator
- Variadic templates
- Perfect forwarding
- SFINAE
- Concepts
- Modules
- Coroutines

The burden of comparison operators

Motivation

- One often needs `operator<` for a user-defined class
  - e.g. when sorting a `std::vector`
  - e.g. when using it as a key for `std::set` or `std::map`
- Many other operators are also desirable for completeness
  - `operator>`, `operator>=`, `operator<=`, ...
  - often implemented reusing e.g. `operator<` and `operator==`
- Should be defined as hidden friend functions
- Much boilerplate code to write. Too much...

The three-way comparison operator C++ 20

Idea

- C++20 introduces `operator<=>`
- named the **three-way comparison operator**.
  - unofficially called **spaceship operator**
- allowing to implement all comparisons in one go

How it works

- it returns *something* which can be compared to 0
  - similar to `std::strcmp`
- lower, greater, or equal to 0 means respectively *lower than*, *greater than* and *equivalent to*
- It is provided by default for all built-in types and many types in the standard library

The three-way comparison operator practically C^{++} 20

Output of operator<=> - godbolt

```cpp
template <typename T>
void three_way_compare( T lhs, T rhs ) {
  auto res = lhs <=> rhs;
  std::cout << lhs << "<=>" << rhs << ": "
            << (res<0) << (res==0) << (res>0)
            << '\n';
}
int main() {
  three_way_compare(1, 2); // 1<=>2: 100
  three_way_compare(2, 2); // 2<=>2: 010
  three_way_compare(2, 1); // 2<=>1: 001
}
```

Different kinds of ordering

C^{++} 20

The return type of operator<=>

- for integers `operator<=>` returns `std::strong_ordering`
- `weak_ordering` and `partial_ordering` also exist

3 types of ordering

strong exactly one test among <0, ==0, and >0 will return `true`

weak like strong but two *equivalent* values may differ

- they are however *equivalent* for ranking
- e.g. rational numbers 2/3 and 4/6

partial like weak but some values are incomparable

- for some values all tests may return `false`
- e.g. compare a floating point to `NaN`

Exercising different orderings

C++ 20

Example - godbolt

```
struct Ratio {
  unsigned n, d ;
  friend std::weak_ordering operator<=>( Ratio const & a,
                                         Ratio const & b ) {
    return (a.n * b.d) <=> (a.d * b.n);
  }
  friend std::ostream & operator<<( std::ostream & os, Ratio const & r ) {
    return (os << r.n << '/' << r.d);
  }
};
int main() {
  // Ratio uses weak_ordering
  three_way_compare(Ratio{1, 2}, Ratio{2, 3}); // 1/2<=>2/3 : 100
  three_way_compare(Ratio{2, 3}, Ratio{4, 6}); // 2/3<=>4/6 : 010
  three_way_compare(Ratio{2, 3}, Ratio{1, 2}); // 2/3<=>1/2 : 001

  // floats use partial_ordering
  three_way_compare(+0., -0.);  // 0<=>-0   : 010
  three_way_compare(0., 1./0.); // 0<=>inf  : 100
  three_way_compare(0., 0./0.); // 0<=>-nan : 000
}
```

Compiler-generated comparison operators C^{++} 20

For a given user-defined class

- defining `operator<=>` allows the compiler to use it when encountering the comparison operators `<`, `<=`, `>` and `>=`
- of course, one can still provide a custom implementation
- the compiler will *NOT* add a default implementation for `operator==` and `operator!=`
  - as those operators mean **equal**, rather than **equivalent**
  - if `operator<=>` does not provide a strong order, it is advised not to define `operator==`

Compiler-generated `operator<=>` C^{++} 20

Default `operator<=>` implementation

- One can ask the compiler to provide a default implementation for `operator<=>` and/or `operator==`
  - declaring them with "`= default`"
  - it will compare the member variables, one by one
- Can be useful e.g. for tuples
- Can be wrong, e.g. for the Ratio class
- If `operator<=>` is defaulted and no `operator==` is defined, then the compiler also provides `operator==`

operator<=> summary

C++ 20

Summary

- Defining `operator<=>` allows you to use `operator<`, `operator>`, `operator<=`, and `operator>=` for free
- The standard library defines a few kinds of orderings
  - strong, weak and partial
- If `operator<=>` does not define a strong ordering, avoid defining `operator==`

Variadic templates

5 Expert C++

- The <=> operator
- Variadic templates
- Perfect forwarding
- SFINAE
- Concepts
- Modules
- Coroutines

Basic variadic template

C++ 11

The idea

- a parameter accepting arbitrarily many arguments (i.e. pack)
- template parameter pack for e.g. types, function parameter packs for values, and expansions, details on cppreference

Recursive example cppinsight

```cpp
template<typename T>
T sum(T v) { return v; }

template<typename T,
         typename... Args>      // temp. param. pack
T sum(T first, Args... args) { // func. param. pack
  return first + sum(args...); // pack expansion
}
int s = sum(1, 2, 3, 8, 7);
```

A couple of remarks

C++ 11

About performance

- do not be afraid of recursion
- everything is at compile time!
- unlike C-style variadic functions
  e.g. `printf(const char* fmt, ...);`

Why it is better than variadic functions

- it's more performant
- type safety is included
- it applies to everything, including user-defined types

Parameter packs

C++ 11

Parameter packs

- only a few operations are supported
  - query the number of elements in a pack with `sizeof...`
  - expand it with `x...`
- can be empty
- can hold different types

```
template<typename... Args>
void f(Args... args) {
  constexpr std::size_t N = sizeof...(Args);
  std::tuple<Args...> t{args...};
}
f(0, 1, 2);         // Args = [int, int, int]
f();                // Args = empty
f(0, true, 3.14); // Args = [int, bool, double]
```

Fold expressions

C++ 17

The idea

- reduces a parameter pack over a binary operator
- details on cppreference

Example

```cpp
template<typename... Args>
auto sum1(Args... args) {
  return (args + ...);      // unary fold over +
}                           // parentheses mandatory
template<typename... Args>
auto sum2(Args... args) {
  return (args + ... + 0); // binary fold over +
}                           // parentheses mandatory
int sum = sum1(); // error
int sum = sum2(); // ok
```

Fold expressions

C++ 17

Example: call a function on all arguments - cppinsight

```cpp
template<typename T>
void print(const T& t) {
  std::cout << t;
}
template<typename... Ts>
void printAll(const Ts&... ts) {
   (print(ts),...); // fold over comma operator
}
```

C++ 11 workaround (don't use anymore)

```cpp
template<typename... Ts>
void printAll(const Ts&... ts) {
   int dummy[]{(print(ts),0)...};
}
```

Variadic class template

C++ 11

The tuple example, simplified - godbolt

```cpp
template <typename... Ts>
struct tuple {};

template <typename T, typename... Ts>
struct tuple<T, Ts...> : tuple<Ts...> {
  tuple(T head, Ts... tail) :
    tuple<Ts...>(tail...), m_head(head) {}
  T m_head;
};

tuple<double, uint64_t, const char*>
  t1(12.2, 42, "big");
```

std::integer_sequence C++ 14

Packs of values

- We can also have non-type template parameter packs
- Useful to pass lists of compile-time constants around
- The standard library includes a few helpers

```
template<typename T, T... Is>
struct integer_sequence { ... };

template<size_t... Is>
using index_sequence =
  integer_sequence<std::size_t, Is...>;

template<size_t N>
using make_index_sequence =
  index_sequence</*0, 1, ..., N-1*/>;
```

std::integer_sequence

C++ 14

Example - make_from_tuple with helper

```cpp
template<typename T,
  typename... Args, std::size_t... Is>
T helper(std::tuple<Args...> args,
  std::index_sequence<Is...>) {
  return T(std::get<Is>(args)...);
}
template<typename T, typename... Args>
T make_from_tuple(std::tuple<Args...> args) {
   return helper<T>(args,
     std::make_index_sequence<sizeof...(Args)>{});
}

struct S { S(int, float, bool) { ... } };
std::tuple t{42, 3.14, false};
S s = make_from_tuple<S>(t);
```

std::integer_sequence

C++ 20

Example - make_from_tuple with lambda

```
template<typename T, typename... Args>
T make_from_tuple(std::tuple<Args...> args) {
   return [&]<std::size_t... Is>(
         std::index_sequence<Is...>){
     return T(std::get<Is>(args)...);
   }(std::make_index_sequence<sizeof...(Args)>{});
}

struct S { S(int, float, bool) { ... } };
std::tuple t{42, 3.14, false};
S s = make_from_tuple<S>(t);
```

Variadic templates

C++ 11

Exercise: Variadic templates

- go to `exercises/variadic`
- you will extend the `tuple` from the last slides
- follow the instructions in the source code

Perfect forwarding

5 Expert C++

- The <=> operator
- Variadic templates
- Perfect forwarding
- SFINAE
- Concepts
- Modules
- Coroutines

The problem

C++ 11

How to write a generic wrapper function?

```
template <typename T>
void wrapper(T arg) {
  // code before
  func(arg);
  // code after
}
```

Example usage :

- emplace_back
- make_unique

Why is it not so simple? C^{++} 11

```cpp
template <typename T>
void wrapper(T arg) {
  func(arg);
}
```

What about references?

- what if `func` takes a reference to avoid copies?
- wrapper would force a copy and we fail to use references

Second try, second failure ? C++ 11

```cpp
template <typename T>
void wrapper(T& arg) {
  func(arg);
}
wrapper(42);
// invalid initialization of
// non-const reference from
// an rvalue
```

and `const T&` won't work when passing something non const

Solution: cover all cases

C++ 11

```cpp
template <typename T>
void wrapper(T& arg) { func(arg); }

template <typename T>
void wrapper(const T& arg) { func(arg); }

template <typename T>
void wrapper(T&& arg) { func(std::move(arg)); }
```

The new problem: scaling to more arguments C++ 11

```cpp
template <typename T1, typename T2>
void wrapper(T1& arg1, T2& arg2)
{ func(arg1, arg2); }

template <typename T1, typename T2>
void wrapper(const T1& arg1, T2& arg2)
{ func(arg1, arg2); }

template <typename T1, typename T2>
void wrapper(T1& arg1, const T2& arg2)
{ func(arg1, arg2); }
...
```

Exploding complexity

- for n arguments, 3^n overloads
- you do not want to try n = 5...

Reference collapsing

C++ 11

Reference to references

- Are formally forbidden, but can occur sometimes

```
template <typename T>
void foo(T t) { T& k = t; } //int& &, error in C++98
int ii = 4;
foo<int&>(ii); // want to pass by reference
```

C++ 11 added rvalue-references

- More combinations possible, like: `int&& &`, or `int&& &&`

Reference collapsing

- Rule: When multiple references are involved, `&` always wins
- `T&& &`, `T& &&`, `T& &` → `T&`
- `T&& &&` → `T&&`

Forwarding references

C++ 11

Rvalue reference in type-deducing context

```
template <typename T>
void f(T&& t) { ... }
```

Forwarding reference

- Next to a template parameter, that can be deduced, `&&` is not an rvalue reference, but a "forwarding reference"
  - aka. "universal reference"
  - So, this applies only to functions
- The template parameter is additionally allowed to be deduced as an lvalue reference, and reference collapsing proceeds:
  - if an lvalue of type `U` is given, `T` is deduced as `U&` and the parameter type `U& &&` collapses to `U&`
  - otherwise (rvalue), `T` is deduced as `U`
    - and forms the parameter type `U&&`

Forwarding references

C++ 11

Examples

```
template <typename T>
void f(T&& t) { ... }

f(4);              // rvalue -> T&& is int&&
double d = 3.14;
f(d);              // lvalue -> T&& is double&
float g() {...}
f(g());            // rvalue -> T&& is float&&
std::string s = "hello";
f(s);              // lvalue -> T&& is std::string&
f(std::move(s)); // rvalue -> T&& is std::string&&
f(std::string{"hello"}); // rvalue -> std::string&&
```

Forwarding references

C++ 20

Careful!

```cpp
template <typename T> struct S {
  S(T&& t) { ... }        // rvalue references
  void f(T&& t) { ... }  // NOT forwarding references
};
```

Half-way correct version

```cpp
template <typename T> struct S {

  template <typename U>
  S(U&& t) { ... } // deducing context -> fwd. ref.
  template <typename U>
  void f(U&& t)    // deducing context -> fwd. ref.
  // ... but now U can be a different type than T
};
```

Forwarding references

C++ 20

Careful!

```
template <typename T> struct S {
  S(T&& t) { ... }          // rvalue references
  void f(T&& t) { ... }  // NOT forwarding references
};
```

Correct version

```
template <typename T> struct S {
  template <typename U, std::enable_if_t<
    std::is_same_v<std::decay_t<U>, T>, int> = 0>
  S(U&& t) { ... } // deducing context -> fwd. ref.
  template <typename U>
  void f(U&& t)     // deducing context -> fwd. ref.
    requires std::same_as<std::decay_t<U>, T> { ... }
};
```

Perfect forwarding

C++ 11

std::remove_reference

- Type trait to remove reference from a type
- If `T` is a reference type, `remove_reference_t<T>` is the type referred to by `T`, otherwise it is `T`.

```cpp
template <typename T>
struct remove_reference      { using type = T; };
template <typename T>
struct remove_reference<T&>  { using type = T; };
template <typename T>
struct remove_reference<T&&> { using type = T; };

template <typename T>
using remove_reference_t =
  typename remove_reference<T>::type; // C++14
```

Perfect forwarding

C++ 11

std::forward

- Forward lvalue-/rvalueness
- Keeps references and maps non-references to rvalue references

```
template<typename T>
T&& forward(remove_reference_t<T>& t) noexcept {  // 1.
  return static_cast<T&&>(t);
}
template<typename T>
T&& forward(remove_reference_t<T>&& t) noexcept { // 2.
  return static_cast<T&&>(t);
}
```

- if T is `int`, selects 2., returns `int&&`
- if T is `int&`, selects 1., returns `int& &&`, i.e. `int&`
- if T is `int&&`, selects 2., returns `int&& &&`, i.e. `int&&`

Perfect forwarding

C++ 11

Example - putting it all together

```
template <typename... T>
void wrapper(T&&... args) {
  func(std::forward<T>(args)...);
}
```

- if we pass an rvalue reference U&& to wrapper
  - T=U, arg is of type U&&
  - func will be called with a U&&
- if we pass an lvalue reference U& to wrapper
  - T=U&, arg is of type U& (reference collapsing)
  - func will be called with a U&
- if we pass a plain U (rvalue) to wrapper
  - T=U, arg is of type U&& (no copy in wrapper)
  - func will be called with a U&&
  - if func takes a plain U, copy happens there, as expected

Real life example

C++ 11

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args) {
  return unique_ptr<T>
    (new T(std::forward<Args>(args)...));
}
```

SFINAE

5 Expert C++

- The <=> operator
- Variadic templates
- Perfect forwarding
- SFINAE
- Concepts
- Modules
- Coroutines

Substitution Failure Is Not An Error (SFINAE) C++ 11

The main idea

- substitution replaces template parameters with the provided arguments (types or values)
- if it leads to invalid code, do not report an error but try other overloads/specializations

Example

```cpp
template <typename T>
void f(typename T::type arg) { ... }
void f(int a) { ... }

f(1); // Calls void f(int)
```

Note: SFINAE is largely superseded by concepts in C++ 20

decltype

C++ 11

The main idea

- gives the type of the result of an expression
- the expression is not evaluated (i.e. unevaluated context)
- at compile time

Example

```cpp
struct A { double x; };
A a;
decltype(a.x) y;         // double
decltype((a.x)) z = y;   // double& (lvalue)
decltype(1 + 2u) i = 4;  // unsigned int

template<typename T, typename U>
auto add(T t, U u) -> decltype(t + u);
// return type depends on template parameters
```

declval

C++ 11

The main idea

- gives you a reference to a “fake” object at compile time
- useful for types that cannot easily be constructed
- use only in unevaluated contexts, e.g. inside `decltype`

```
struct Default {
  int foo() const;
};
class NonDefault {
  private: NonDefault();
  public:  int foo() const;
};
decltype(Default().foo()) n1 = 1;     // int
decltype(NonDefault().foo()) n2 = 2; // error
decltype(std::declval<NonDefault>().foo()) n3 = 3;
```

true_type and false_type

C++ 11

The main idea

- encapsulate a compile-time boolean as type
- can be inherited

Example

```cpp
struct truth : std::true_type { };

constexpr bool test = truth::value; // true
constexpr truth t;
constexpr bool test = t(); // true
constexpr bool test = t;   // true
```

Possible implementation

```cpp
using true_type  = integral_constant<bool, true >;
using false_type = integral_constant<bool, false>;
```

Using SFINAE for introspection

C++ 11

The main idea

- use a primary template, inheriting from `false_type`
- use a template specialization, inheriting from `true_type`
  - which depends on the feature you want to introspect, as part of the context where template argument deduction happens
- let SFINAE choose between the two templates

Example

```cpp
template <typename T, typename = void>
struct hasFoo : std::false_type {}; // primary template
template <typename T>
struct hasFoo<T, decltype(std::declval<T>().foo())>
  : std::true_type {};                 // template special.
struct A{}; struct B{ void foo(); };
static_assert(!hasFoo<A>::value, "A has no foo()");
static_assert(hasFoo<B>::value, "B has foo()");
```

Not so easy actually... C++ 11

Example - godbolt

```cpp
template <typename T, typename = void>
struct hasFoo : std::false_type {};
template <typename T>
struct hasFoo<T, decltype(std::declval<T>().foo())>
  : std::true_type {};

struct A{};
struct B{ void foo(); };
struct C{ int  foo(); };

static_assert(!hasFoo<A>::value, "A has no foo()");
static_assert(hasFoo<B>::value, "B has foo()");
static_assert(!hasFoo<C>::value, "C has no foo()");
static_assert(hasFoo<C,int>::value, "C has foo()");
```

Using `void_t` C++ 17

Concept

- Maps a sequence of given types to `void`
- Introduced in C++ 17, though trivial to implement in C++ 11
- Can be used in specializations to check the validity of an expression

Implementation in header type_traits

```cpp
template <typename...>
using void_t = void;
```

Previous introspection example using void_t C++ 17

Example - godbolt

```cpp
template <typename T, typename = void>
struct hasFoo : std::false_type {};

template <typename T>
struct hasFoo<T,
   std::void_t<decltype(std::declval<T>().foo())>>
: std::true_type {};

struct A{}; struct B{ void foo(); };
struct C{ int foo(); };

static_assert(!hasFoo<A>::value,"unexpected foo()");
static_assert(hasFoo<B>::value, "expected foo()");
static_assert(hasFoo<C>::value, "expected foo()");
```

Standard type traits C^{++} 11

Standard type traits

- Don't implement SFINAE introspection for everything, use the standard library!
- Standard type traits in header `<type_traits>`
- They look like `std::is_*<T>`
- Checks at compile time whether `T` is
  - float, signed, final, abstract, default_constructible, ...
- Result in nested boolean constant `value`
- Since C^{++} 17, variable template versions: `std::is_*_v<T>`, giving the bool directly

```cpp
constexpr bool b = std::is_pointer<int*>::value;
constexpr bool b = std::is_pointer_v<int*>; // C++17
```

SFINAE and the STL

C++ 11/C++ 14

enable_if cppreference

```
template<bool B, typename T=void>
struct enable_if;
```

- If B is true, has a nested alias type to type T
- otherwise, has no such alias

Example

```
template<bool B, typename T=void>
struct enable_if {};
template<typename T>
struct enable_if<true, T> { using type = T; };

template<bool B, typename T = void> // C++14
using enable_if_t = typename enable_if<B, T>::type;
```

Usage example - godbolt

```cpp
template <typename T> constexpr bool is_iterable ... ;

template<typename T,
         typename std::enable_if_t
           <!is_iterable<T>, bool> = true>
void print(T const& t) {
    std::cout << t << "\n";
}
template<typename T,
         typename std::enable_if_t
           <is_iterable<T>, bool> = true>
void print(T const& t) {
    for (auto const& item : t) {
        std::cout << item << "\n";
    }
}
```

Note : using `if constexpr` would be simpler

Back to variadic class templates

C++ 11

The tuple_element trait

```cpp
template <size_t I, typename Tuple>
struct tuple_element;

template <typename T, typename... Ts>
struct tuple_element<0, tuple<T, Ts...>> {
  using type = T;
};

template <size_t I, typename T, typename... Ts>
struct tuple_element<I, tuple<T, Ts...>> {
  using type = typename
    tuple_element<I - 1, tuple<Ts...>>::type;
};
```

Back to variadic class templates C++ 14

The tuple get function

```
template <size_t I, typename... Ts>
std::enable_if_t<I == 0,
  typename tuple_element<0, tuple<Ts...>>::type&>
get(tuple<Ts...>& t) {
  return t.m_head;
}
template <size_t I, typename T, typename... Ts>
std::enable_if_t<I != 0,
  typename tuple_element<I - 1, tuple<Ts...>>::type&>
get(tuple<T, Ts...>& t) {
  return get<I - 1>(static_cast<tuple<Ts...>&>(t));
}
```

with if constexpr and return type deduction C^{++} 17

The tuple get function

```
template <size_t I, typename T, typename... Ts>
auto& get(tuple<T, Ts...>& t) {
  if constexpr(I == 0)
    return t.m_head;
  else
    return get<I - 1>(static_cast<tuple<Ts...>&>(t));
}
```

Good practice: SFINAE vs. if constexpr

- `if constexpr` can replace SFINAE in many places.
- It is usually more readable as well. Use it if you can.

Concepts

Requirements and concepts

Motivation

- Generic programming is made of variable, function and class templates which can be instantiated with different types.
- It is frequent to instantiate them with **unsuited types**, and the resulting compilation errors are cryptic.
- As a last resort, authors provide **documentation**, and practice tricky **template meta-programming**.
- C++20 brings **simpler ways to define constraints** on template parameters.

The world before concepts

C++ 17

C++ 17 work around: SFINAE

- Unsuited arguments can be avoided by inserting fake template arguments, leading to a substitution failure

Practical code - godbolt

```
template
<typename T,
 typename = std::enable_if_t<std::is_floating_point_v<T>>>
bool equal( T e1, T e2 ) {
  return std::abs(e1-e2)<std::numeric_limits<T>::epsilon();
}
... equal(10,5+5) ...
```

The world before concepts

C++ 17

C++ 17 work around: SFINAE

- Unsuited arguments can be avoided by inserting fake template arguments, leading to a substitution failure

Practical code - godbolt

```
template
<typename T,
 typename = std::enable_if_t<std::is_floating_point_v<T>>>
bool equal( T e1, T e2 ) {
  return std::abs(e1-e2)<std::numeric_limits<T>::epsilon();
}
... equal(10,5+5) ...
```

```
<source>:11:12: error: no matching function for call to 'equal(int, int)'
   11    if (equal(10,5+5)) { std::cout << "FAILURE\n"; }
<source>:7:6: note: candidate: 'template<class T, class> bool equal(T, T)'
    7  bool equal( T e1, T e2 )
<source>:7:6: note:   template argument deduction/substitution failed:
In file included from <source>:1:
.../type_traits: In substitution of 'template<bool _Cond, class _Tp>
using enable_if_t = typename std::enable_if::type with bool _Cond = false; _Tp = void':
<source>:6:14:   required from here
.../type_traits:2514:11: error: no type named 'type' in 'struct std::enable_if<false, void>'
 2514      using enable_if_t = typename enable_if<_Cond, _Tp>::type;
```

Basic requirements

C++ 20

A new keyword

- The keyword `requires` lets us define various constraints.

With concepts - godbolt

```
template<typename T>
requires std::is_floating_point_v<T>
bool equal( T e1, T e2 ) {
  return std::abs(e1-e2)<std::numeric_limits<T>::epsilon();
}
... equal(10,5+5) ...
```

Basic requirements

C++ 20

A new keyword

- The keyword `requires` lets us define various constraints.

With concepts - godbolt

```
template<typename T>
requires std::is_floating_point_v<T>
bool equal( T e1, T e2 ) {
  return std::abs(e1-e2)<std::numeric_limits<T>::epsilon();
}
... equal(10,5+5) ...
```

```
<source>:11:12: error: no matching function for call to 'equal(int, int)'
   11    if (equal(10,5+5)) { std::cout << "FAILURE\n"; }
<source>:7:6: note: candidate: 'template<class T, class> bool equal(T, T)'
    7  bool equal( T e1, T e2 )
<source>:7:6: note:   template argument deduction/substitution failed:
<source>:7:6: note: constraints not satisfied
<source>: In substitution of 'template<class T> ... bool equal(T, T) with T = int':
<source>:11:12:   required from here
<source>:7:6:   required by the constraints of 'template<class T> ... bool equal(T, T)'
<source>:6:15: note: the expression 'is_floating_point_v<T> with T = int' evaluated to 'false'
    6  requires std::is_floating_point_v<T>
```

Requirements and overloads

C++ 20

Example of several competing templates

```cpp
template<typename T>
bool equal( T e1, T e2 ) { return (e1==e2); }

template< typename T>
requires std::is_floating_point_v<T>
bool equal( T e1, T e2 )
{return std::abs(e1-e2)<std::numeric_limits<T>::epsilon();}
```

Requirements affect overload resolution

- Overload resolution considers the second function as a better match when the requirements are fulfilled.

Concepts

C++ 20

Definition

- a **concept** gives a name to a given set of requirements
- useful when the same requirements are reused frequently

A new keyword: concept

```
template< typename T>
concept MyFloatingPoint =
  std::is_floating_point_v<T> &&
  std::numeric_limits<T>::epsilon()>0;

template<typename T>
requires MyFloatingPoint<T>
bool equal( T e1, T e2 )
{return std::abs(e1-e2)<std::numeric_limits<T>::epsilon();}
```

Some usages of concepts

C++ 20

Constrained template parameters

- a concept can replace `typename` in a template parameter list

```
template<MyFloatingPoint T>
bool equal( T e1, T e2 ) {
  return std::abs(e1-e2)<std::numeric_limits<T>::epsilon();
}
```

Constrained variables

- a concept can be used together with `auto` to impose requirements on the type of a variable

Example code

```
MyFloatingPoint auto f = 3.14f;
MyFloatingPoint auto d = 3.14;
MyFloatingPoint auto i = 3; // compile error
```

Some usages of concepts

C++ 20

Abbreviated function templates

- function parameters can be constrained as well
  - similar to variables
- the template head becomes obsolete
- the function is still a template though!
- unconstrained `auto` parameters are allowed as well

```
// no template <...> here!
bool equal( MyFloatingPoint auto e1,
            MyFloatingPoint auto e2 ) {
  return std::abs(e1-e2) <
         std::numeric_limits<decltype(e1)>::epsilon();
}

// unconstrained abbreviated function template:
void equal(auto e1, auto e2) { ... }
```

Standard concepts

C++ 20

Never reinvent the wheel

- Writing a bug-proof concept is an expert task
- Prefer the ones provided by the standard library

E.g.: the floating point concept

```cpp
#include <concepts>
bool equal( std::floating_point auto e1,
            std::floating_point auto e2 ) {
  return std::abs(e1-e2) <
         std::numeric_limits<decltype(e1)>::epsilon();
}
```

Checking concepts

C++ 20

Concepts as Boolean operators

- Concepts can be used wherever a `bool` is expected
- they can appear in `if constexpr` conditions
  - since they are evaluated at compile-time

Using concepts with `if constexpr`

```
template<typename T>
bool equal( T e1, T e2 )
{
  if constexpr (std::floating_point<T>) {
    return std::abs(e1-e2)
             < std::numeric_limits<T>::epsilon();
  } else {
    return (e1==e2);
  }
}
```

Advanced requirements overview

C++ 20

requires as an expression

requires can express more than basic requirements. It can

- include other basic requirements
- list expressions that must be valid
- check the return type of some expressions

Practically

```
template<typename T>
concept StreamableAndComparableNumber =
requires( T v1, T v2, std::ostream os ) {
  requires std::integral<T> || std::floating_point<T>;
  os<<v1<<v2;
  { equal(v1,v2) } -> std::convertible_to<bool>;
};
```

Remember: use standard concepts first

Requirements and concepts

C^{++} 20

Summary

- A template can now *require* properties of its parameters
- Compiler error messages clearly state which argument does not fulfill which requirement
- A set of requirements can be gathered in a *concept*
- Overload resolution takes requirements into account
- The standard library provides many ready-to-use concepts
- Writing a new good concept is an expert task

Exercise time

C^{++} 20

Exercise: Concepts

- go to `exercises/concepts`
- you will use concepts to optimize some loop on iterators
- follow the instructions in the source code

Modules

5 Expert C^{++}

- The <=> operator
- Variadic templates
- Perfect forwarding
- SFINAE
- Concepts
- Modules
- Coroutines

Modules

C++ 20

Motivation

- Modules allow for sharing declarations and definitions across translation units
- Header files do the same, but modules have benefits:
  - Better isolation. No cross-talk between multiple pieces of included code, and also not between included code and your code.
  - Better control over public interface of your library. Avoid leaking library dependencies into user code.
  - Faster compilation (import fmt; >1500x faster)

Textual includes

```cpp
#include "math.h"
```

Module importation

```cpp
import math;
```

Writing modules

C++ 20

ratio.cpp

```cpp
export module math;
import <string>;
namespace utils { // ns. and module names orthogonal
  export struct Ratio { float num, denom; };
  export std::string toString(const Ratio& r) { ... }
  void normalize(Ratio& r) { ... } // not exported
}
```

main.cpp

```cpp
import <iostream>;
import math;
int main() {
  auto r = utils::Ratio{1, 3};
  std::cout << utils::toString(r);
}
```

Writing modules

$C^{++}20$

Exporting declarations

- By default, everything inside a module is internal
- Declarations become visible when declared with `export` in the primary module interface
- You can only export entities at namespace scope (including the global scope)
- You cannot export entities with internal linkage (declared static or inside anonymous namespace)
- You cannot export aliases to non-exported types

Writing modules

C++ 20

What to export

```cpp
export module math;

export void f();              // function
export float pi = 3.14f;      // variable
export namespace utils {      // namespace
  enum Status { Yes, No }; // enum (exported)
}
namespace utils {
  export struct Ratio { ... }; // class
  export using R = Ratio;       // alias
  float e = 2.71f;              // (not exported)
}
export { // content of export block
  void h();
}
```

Writing modules

C++ 20

What not to export

```
export module math;

namespace {
  export void f();                  // error
  export float pi = 3.14f;          // error
  export struct Ratio { ... };      // error
}

export static void f();             // error
export static float pi = 3.14f;     // error

class C { ... };
export using D = C;                 // error
```

Visibility and reachability

C++ 20

ratio.cpp

```
export module math;
struct Ratio { float num, denom; };
export Ratio golden() { ... } // golden is visible
```

main.cpp

```
import math;
import <iostream>;
int main() {
  Ratio r = golden(); // error: Ratio not visible
  auto r = golden();  // OK: Ratio is reachable
  std::cout << r.num; // OK: members of reachable
                      // types are visible
  using R = decltype(r);
  R r2{1.f, 3.f};     // OK
}
```

Module structure

C++ 20

```
module;                        // global module fragment (opt.)
#include <cassert>             //   only preprocessor
// ...                         //   statements allowed

export module math;            // module preamble starts with
                               //   exported module name.
import <string>;               //   only imports allowed
// ...                         //

export struct S { ... };       // module purview started
export S f();                  //   at first non-import
std::string h() { ... }

module :private                // private module fragment (opt.)
S f() { ... }                  //   changes here trigger
                               //   no recompilation of BMI
```

Module structure

C++ 20

Pitfall

```cpp
export module math;
#include <header.h>
import <string>;
// ...
```

- Headers included after `export module` add to the module's preamble and purview
- This is usually not what you want

Put includes into global module fragment

```cpp
module;
#include <vector>
export module math;
import <string>;
// ...
```

Module structure

C^{++} 20

Like with header files, we can split interface and implementation:

Interface

```
export module math;

export struct S { ... };
export S f();
```

Implementation

```
module;
#include <cassert>
module math;
import <string>;

S f() { ... }
std::string b() { ... }
```

Submodules and re-exporting

C++ 20

Modules can (re-)export other modules using `export import`:

Module one

```cpp
export module one;

export struct S { ... };
```

Module two

```cpp
export module two;
import <string>;

export std::string b()
  { ... }
```

Module three

```cpp
export module three;
import one;
export import two;

export S f() { ... }
```

Module four

```cpp
export module four;
import three;
// b and f visible
// S and std::string only
//   reachable
```

Module partitions

C++ 20

We can split a module's *interface* into multiple files:

Primary module interface

```
export module math;
export import :structs;

export S f();
```

Module interface partition

```

export module math:structs;

export struct S { ... };
```

Implementation

```
module math;
import <string>;

S f() { ... }
std::string b() { ... }
```

Module implementation units

C++ 20

We can split a module's *implementation* into multiple files:

Interface

```cpp
export module math;
import :foo; // no export
import :bar; // no export
import <string>;

export struct S { ... };
export S f();
std::string b();
```

Implementation unit foo

```cpp
module math:foo;

S f() { ... }
```

Implementation unit bar

```cpp
module math:bar;

std::string b() { ... }
```

Standard library modules

C++ 23

Standard library modules

- Use `import std;` for the entire C++ standard library. Basically everything in namespace `std`.
  - E.g. `memcpy(a, b, n)`, `sqrt(val)`, `uint32_t i` will not compile. You need to prefix `std::`.
- Use `import std.compat;` for the entire C and C++ standard library
- No macros are exported. For these, you still need to include the corresponding headers.

```
import std;           // everything C++
#include <cassert> // for assert()
#include <climits> // for e.g. CHAR_BIT
```

- Definition of further, smaller modules of the standard library is in progress (e.g. fundamentals, containers, networking, etc.)

Header units

C++ 20

Purpose

- For easier transition from header-based projects to modules
- Allows to import any header file as if it was a module
- Can be mixed with regular header inclusion

```
#include <set>
import <vector>;

#include "header.hpp"
import "otherheader.hpp";
```

Header units

C++ 20

Effect

- All declarations in the imported header will be exported
- An imported header is not affected by the preprocessor state. Thus, you cannot "configure" the header by defining a macro before importing. This allows the header to be precompiled.
- An imported header may still provide preprocessor macros

```cpp
#define MY_MACRO 1
#include "header.hpp" // may be affected by MY_MACRO
import "otherheader.hpp"; // ignores MY_MACRO
```

Automatic include to import translation C^{++} 20

Importable headers

- A header file which can successfully be either included or imported is called "importable"
  - I.e. the header does not require setup of preprocessor state before inclusion
- All C^{++} standard library headers are importable
- C wrapper headers (<c*>) are not

Include translation

- Allows the compiler to automatically turn includes into imports
- Controlled via compiler flags and the build system
- Basically a standard replacement for precompiled headers

Build system

h.hpp

```
#include <vector>
...
```

a.cpp

```
#include "h.hpp"
...
```

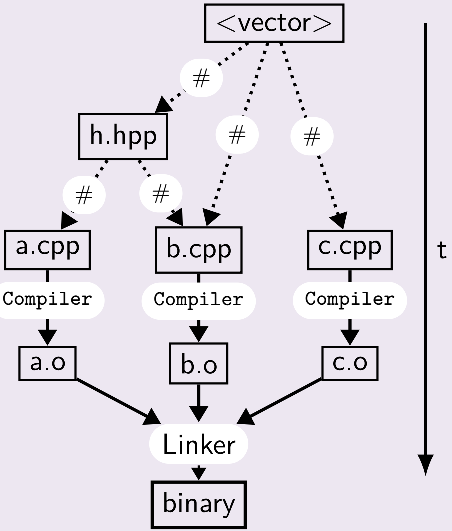
b.cpp

```
#include <vector>
#include "h.hpp"
...
```

c.cpp

```
#include <vector>
...
```

Traditional workflow



Build system

C++ 20

h.cpp

```cpp
export module foo;
import <vector>;
...
```

a.cpp

```cpp
import foo;
...
```

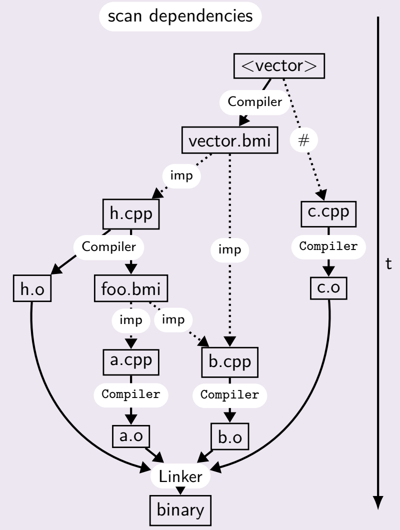
b.cpp

```cpp
import <vector>;
import foo;
...
```

c.cpp

```cpp
#include <vector>
...
```

New workflow



Build system

New challenges

- Resolving a module name to the file name building it
- Translation units no longer independent
  - Extra tool needs to infer dependency graph before building
    - Called a "dependency scanner"
    - Needs to be communicated to the build system
    - Using a new standard dependency file format
  - Parallel and distributed builds need synchronization
- Compilation of module translation units produces a binary module interface (BMI) *and* an object file
  - Need to manage BMIs between multiple compiler invocations
- Tools beside the compiler need to build/read BMIs
  - E.g. static analysis, language servers for auto completion, etc.
- The C^{++} standard specifies very little on how this works
  - We may experience large implementation divergence

Build system

C++ 20

Status

- Dependency scanner since clang 16: `clang-scan-deps`
- Experimental module support in cmake since 3.26, uses `clang-scan-deps`, many fixes in later versions
- MSBuild (Windows) fully handles dependency scanning
- `g++ a.cpp b.cpp ...` "just works", must be one line though
- Experimental extensions to GNU make exist to grow dependency graph on the fly while modules are discovered
- Header units unsupported in all build systems (May 2023)
- C++23: `import std;` supported in MSVC, partially in clang
  - GCC/clang/MSVC also plan support in C++20

Building modules (g++) C++ 20

Case study: g++

- BMI is called CMI (compiled module interfaces)
- By default, g++ caches CMIs in subdirectory `./gcm.cache`
- Uses a module mapper to resolve imports, which specifies a protocol and uses a central server for module maintenance
- Each module needs to be built before it can be imported
  - `g++ -std=c++20 -fmodules-ts -c ratio.cpp -o ratio.o`
  - Generates `ratio.o` and `./gcm.cache/ratio.gcm` (CMI)
- Each header unit needs to be built before it can be imported
  - `g++ -std=c++20 -fmodules-ts -x c++-system-header vector`
  - Generates e.g. `./gcm.cache/usr/include/c++/11/vector.gcm` (CMI)
  - `g++ -std=c++20 -fmodules-ts -x c++-header ratio.h`
  - Generates e.g. `./gcm.cache/,/ratio.h.gcm` (CMI)

Modules

C^{++} 20

Guidance for today

- Start writing importable headers (no macro dependencies)
  - Dual use: `#include` before C^{++} 20 and `import` with C^{++} 20
- Watch progress on module support in your build system

Guidance for tomorrow

- Start writing modules when your users have C^{++} 20
- Maybe import standard headers when C^{++} 20 is available
- Start using `import std;` when available
- Future of header units unclear (implementation difficulties)

Modules

C++ 20

Resources

- A (Short) Tour of C++ Modules - Daniela Engert - CppCon 2021 (very technical)
- Understanding C++ Modules: Part1, Part2 and Part3.
- So, You Want to Use C++ Modules … Cross-Platform? - Daniela Engert - C++ on Sea 2023
- import CMake: 2023 State of C++20 modules in CMake - Bill Hoffman - CppNow 2023

Modules

C^{++} 20

Exercise: Modules

- go to `exercises/modules`
- convert the `Complex.hpp` header into a module named `math`

Exercise: Header units

- go to `exercises/header_units`
- convert all `#include`s into header unit `import`s

Coroutines

5 Expert C++

- The <=> operator
- Variadic templates
- Perfect forwarding
- SFINAE
- Concepts
- Modules
- Coroutines

Why do we need coroutines? C++ 20

The generator use case (one of several)

In python one can write

```python
for i in range(0, 10):
  print(i)
```

What about it in C++?

```cpp
someType range(int first, int last) { ... }
for (auto i : range(0, 10)) {
  std::cout << i << "\n";
}
```

How can we implement range?

Requirements

- someType needs to be some iterable type
- range should execute lazily, only creating values on demand
- meaning range should be suspended and resumed

That's what coroutines are for!

What C++ 20 allows

Interesting part of the implementation (see cppreference)

```
template<std::integral T>
Generator<T> range(T first, const T last) {
  while (first < last) {
    co_yield first++;
  }
}
for (const char i : range(65, 91)) {
  std::cout << i << ' ';
}
std::cout << '\n';
```

The bad news

- class `Generator` is not provided by the STL
- and is ~ 80 lines of boiler plate code...
- may be sorted out in C++ 23
- in the meantime, we'll have to dive into some details!

A few concepts first

C++ 20

Definition of a coroutine

- a function which can be suspended and resumed
  - potentially by another caller (even on a different thread)
- practically a function using one of:

`co_await`	suspends execution
`co_yield`	suspends execution and returns a value
`co_return`	completes execution

Implications

- on suspension, the state of the function needs to be preserved
- the suspended function becomes pure data on the heap
- access to this data is given via the returned value
  - so the returned type must follow some convention
  - it needs to contain a `struct promise_type`

Minimal code

C++ 20

Definition of an empty coroutine

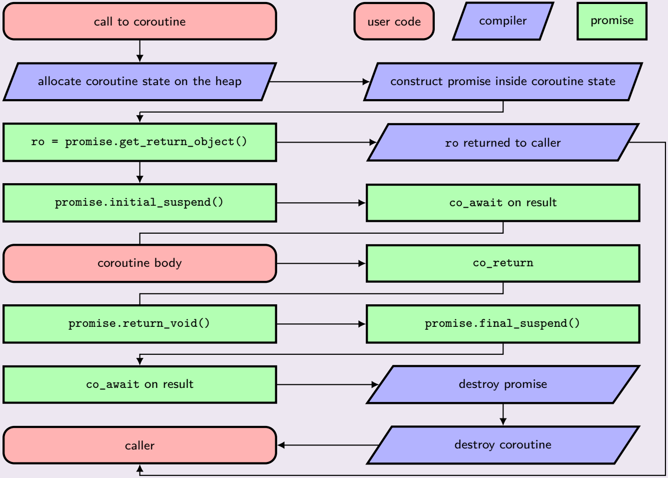
```cpp
#include <coroutine>
struct Task {
  struct promise_type {
    Task get_return_object() { return {}; }
    std::suspend_never initial_suspend() { return {}; }
    std::suspend_never final_suspend() noexcept { return {}; }
    void return_void() {}
    void unhandled_exception() { throw; }
  };
};
Task myCoroutine() { co_return; }
int main() { Task x = myCoroutine(); }
```

promise_type interface

`get_return_object`	builds a Task from the given promise
`initial/final_suspend`	called before/after coroutine starts/ends
`unhandled_exception`	called in case of exception in coroutine
`return_void/value`	returns final value of the coroutine

Minimal sequence of events (nothing suspended) C++ 20

Compiler generated set of calls when calling `myCoroutine`



Credits : original idea for this graph comes from this web site

Suspending a coroutine

C++ 20

Main way to suspend a coroutine: co_wait

- it's an expression taking an "awaitable"/ "awaiter"
  - we'll ignore the distinction for now
- 2 trivial awaiters are provided by the STL: `std::suspend_always` and `std::suspend_never`
- obviously only the first one will trigger a suspension
  - note the other one is used in the minimal code
- more details on awaiters later

Code

```
Task myCoroutine() {
  std::cout << "Step 1 of coroutine\n";
  co_await std::suspend_always{};
  std::cout << "Step 2 of coroutine\n";
  co_await std::suspend_always{};
  std::cout << "final step\n";
}
```

Resuming a coroutine

C++ 20

Principles

- one needs to call `resume()` on the `coroutine_handle`
- BUT user code has no such handle
  - only the coroutine return object (here a `Task` instance)
- so one has to amend the `Task` class
  - and the `promise_type` as it's building the `Task` instance

Code

```cpp
struct Task {
  struct promise_type {
    Task get_return_object() {
      // build Task with the coroutine handle
      return {std::coroutine_handle<promise_type>::from_promise(*this)};
    }
    ... // rest is unchanged
  };
  std::coroutine_handle<promise_type> h_;
  void resume() { h_.resume(); }
};
```

Resuming a coroutine

C++ 20

User code - godbolt

```
Task myCoroutine() {
  std::cout << "Step 1 of coroutine\n";
  co_await std::suspend_always{};
  std::cout << "Step 2 of coroutine\n";
  co_await std::suspend_always{};
  std::cout << "final step\n";
}
int main() {
  auto c = myCoroutine(); // Step 1 runs
  std::cout << "In main, between Step 1 and 2\n";
  c.resume();             // Step 2 runs
  std::cout << "In main, between Step 2 and final step\n";
  c.resume();             // final Step runs
  // c.resume(); // would segfault!
}
```

```
Step 1 of coroutine
In main, between Step 1 and 2
Step 2 of coroutine
In main, between Step 2 and final step
final step
```

A few more details

C++ 20

About suspension of coroutines

- `initial/final_suspend` methods allow to suspend a coroutine at the very start/end (after `co_return`)
- a suspended coroutine is not deallocated
  - i.e. its state will stay on the heap
  - so you can leak coroutines!
- one can directly call `coroutine_handle::destroy` if the execution of a suspended coroutine should be aborted
  - which requires another piece of code in `Task`, e.g.

    ```
    ~Task() { if (h_) h_.destroy(); }
    ```
- the `Task` class should be move only

We need standard `Task` classes in the STL!

- `std::generator` may come in C++ 23

co_yield - returning a value upon suspension C++ 20

What co_yield really is

- an expression like `co_await`
- a shortcut for `co_await promise.yield_value(expr)`
- and we have to provide the `yield_value` method of `promise_type` and the `Task` accessor

Typical code

```cpp
template<typename T> struct Task {
  struct promise_type {
    T value_;
    std::suspend_always yield_value(T&& from) {
      value_ = std::forward<T>(from); // caching the result
      return {};
    }
    ... // rest is unchanged
  };
  T getValue() { return h_.promise().value_; } // accessor for user code
};
```

co_yield - generator behavior

C++ 20

Generator behavior - more boiler plate code

```
template<typename T>
struct Task {
  struct iterator {
    std::coroutine_handle<promise_type> handle;
    auto &operator++() {
      handle.resume();
      if (handle.done()) { handle = nullptr; } // == end iterator
      return *this;
    }
    auto operator++(int) { ++*this; }
    T const& operator*() const { return handle.promise().value_; }
    ...
  };
  iterator begin() { resume(); return {h_}; }
  iterator end() { return {nullptr}; }
  ... // plus error handling is missing
};
```

co_yield - final usage

C^{++} 20

Finally, we can use the generator nicely - godbolt

```
Task range(int first, int last) {
  while (first != last) {
    co_yield first++;
  }
}
for (int i : range(0, 10)) {
  std::cout << i << "";
} // 1 2 3 4 5 6 7 8 9
```

co_yield - final usage

C++ 20

Finally, we can use the generator nicely - godbolt

```cpp
Task range(int first, int last) {
  while (first != last) {
    co_yield first++;
  }
}
for (int i : range(0, 10)) {
  std::cout << i << "";
} // 1 2 3 4 5 6 7 8 9
```

Wait a minute: 0 is missing in the output!

- resume called in `operator++` before 0 is printed
- we should pause in `initial_suspend`

```cpp
struct Task {
  struct promise_type {
    ...
    std::suspend_always initial_suspend() { return {}; }
    ...
  }
};
```

Laziness allows for infinite ranges

C++ 20

Generators do not need to be finite - godbolt

```
Task range(int first) {
  while (true) {
    co_yield first++;
  }
}
for (int i : range(0) | std::views::take(10)) {
  std::cout << i << "\n";
}
```

Real life example

C++ 20

A Fibonacci generator, from cppreference - godbolt

```
Task<long long int> fibonacci(unsigned n) {
  if (n==0) co_return;
  co_yield 0;
  if (n==1) co_return;
  co_yield 1;
  if (n==2) co_return;
  uint64_t a=0;
  uint64_t b=1;
  for (unsigned i = 2; i < n; i++) {
    uint64_t s=a+b;
    co_yield s;
    a=b;
    b=s;
  }
}

int main() {
  for (int i : fibonacci(10)) {
    std::cout << i << "\n";
  }
}
```

Coming back to awaitables/awaiters C^{++} 20

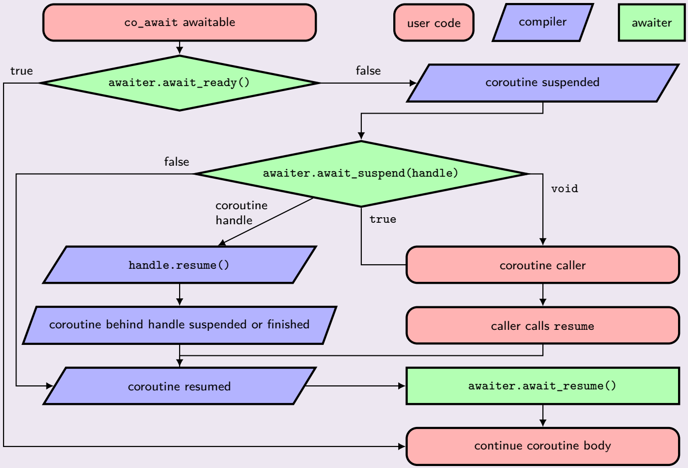
awaitables vs awaiters

- awaitable is the type required by `co_await`/`co_yield`
  - or the expression given has to be convertible to awaitable
  - alternatively the `promise_type` can have an `await_transform` method returning an awaitable
- an awaiter is retrieved from the awaitable by the compiler
  - either by calling operator `co_wait` on it
  - or via direct conversion if no such operator exists
- an awaiter has 3 main methods
  - `await_ready`, `await_suspend` and `await_resume`

How awaiters work

C++ 20

Compiler generated set of calls



awaiter example

C++ 20

Switch to new thread, from cppreference - godbolt

```
auto switch_to_new_thread(std::jthread& out) {
  struct awaiter {
    std::jthread* th;
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) {
      *th = std::jthread([h] { h.resume(); });
      std::cout << "New thread ID: " << th->get_id() << '\n';
    }
    void await_resume() {}
  };
  return awaiter{&out};
}
Task<int> resuming_on_new_thread(std::jthread& out) {
  std::cout << "Started on " << std::this_thread::get_id() << '\n';
  co_await switch_to_new_thread(out);
  std::cout << "Resumed on " << std::this_thread::get_id() << std::endl;
}
int main() {
  std::jthread out;
  resuming_on_new_thread(out);
}
```

awaiter example

C++ 20

Switch to new thread, from cppreference - godbolt

```cpp
auto switch_to_new_thread(std::jthread& out) {
  struct awaiter {
    std::jthread* th;
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) {
      *th = std::jthread([h] { h.resume(); });
      std::cout << "New thread ID: " << th->get_id() << '\n';
    }
    void await_resume() {}
  };
  return awaiter{&out};
}
Task<int> resuming_on_new_thread(std::jthread& out) {
  std::cout << "Started on " << std::this_thread::get_id() << '\n';
  co_await switch_to_new_thread(out);
  std::cout << "Resumed on " << std::this_thread::get_id() << std::endl;
}
int main() {
  std::jthread out;
  resuming_on_new_thread(out);
}
```

Output

```
Started on thread: 140144191063872
New thread ID: 140144191059712
Resumed on thread: 140144191059712
```

Summary on coroutines

C++ 20

Coroutines are usable in C++ 20

- allow to have nice generators
- also very useful for async/IO code
  - another big use case which we did not touch

But library support is poor for now

- standard Task/Generator classes are not provided
- leading to *lots of boiler plate code* for the end user
- `std::generator` is part of C++ 23's draft
- in the meantime, libraries exist, e.g. CppCoro

Useful tools

C^{++} editor

6 Useful tools

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

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

6 Useful tools

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

clang-format

Exercise: clang-format

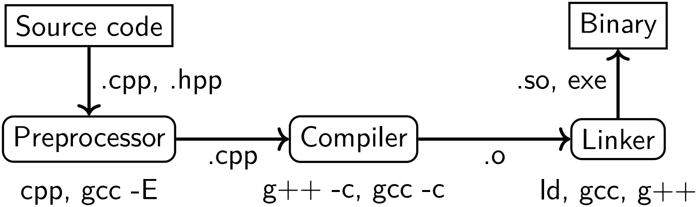
- Go to any example
- Format code with:
  `clang-format --style=GNU -i <file.cpp>`
- Inspect changes, try `git diff .`
- Revert changes using `git checkout -- <file.cpp>` or `git checkout .`
- Go to exercises directory and create a .clang-format file
  `clang-format -style=LLVM -dump-config > .clang-format`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

The Compiling Chain

6 Useful tools

- C++ editor
- Version control
- Code formatting
- The Compiling Chain
- Web tools
- Debugging
- Sanitizers
- The Valgrind family
- Static code analysis
- Profiling
- Doxygen

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; slightly better error reporting; 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_minimum_required(VERSION 3.18)
project(hello CXX)

find_package(ZLIB REQUIRED) # for external libs

add_executable(hello main.cpp util.h util.cpp)
target_compile_features(hello PUBLIC cxx_std_17)
target_link_libraries(hello PUBLIC ZLIB::ZLIB)
```

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

6 Useful tools
- C++ editor
- Version control
- Code formatting
- The Compiling Chain
- Web tools
- Debugging
- Sanitizers
- The Valgrind family
- Static code analysis
- Profiling
- Doxygen

Godbolt / Compiler Explorer

Concept

An online generic compiler with immediate feedback. Allows:

- compiling online any code against any version of any compiler
- inspecting the assembly generated
- use of external libraries (over 50 available !)
- running the code produced
- using tools, e.g. ldd, include-what-you-use, ...
- sharing small pieces of code via permanent short links

Typical usage

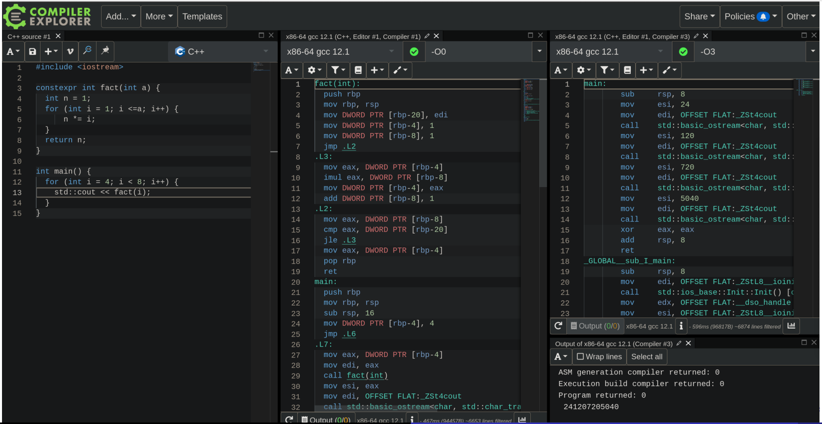
- check small pieces of code on different compilers
- check some new C^{++} functionality and its support
- optimize small pieces of code
- NOT relevant for large codes

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

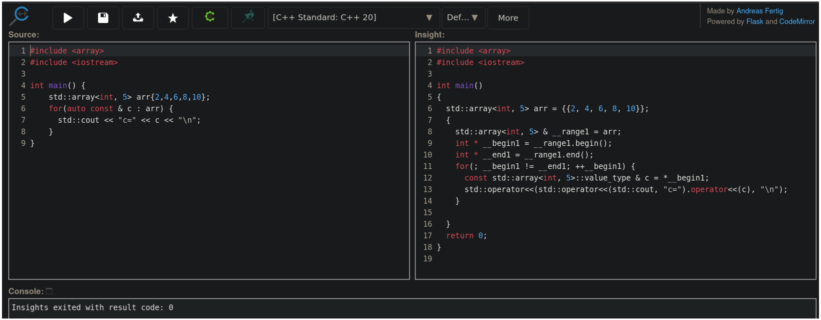
- understand how things work behind the C++ syntax
- debug some non working pieces of code

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

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 ...

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

Sanitizers

Address Sanitizer (ASan)

ASan introduction

- Compiler instrumentation
- Program stops on invalid memory access, e.g.
    - Invalid read/write on heap and stack
    - Double free/delete, use after free
    - Buffer overflow on stack (few tools can do this)
    - Only linux: memory leaks

Address Sanitizer (ASan)

ASan introduction

- Compiler instrumentation
- Program stops on invalid memory access, e.g.
  - Invalid read/write on heap and stack
  - Double free/delete, use after free
  - Buffer overflow on stack (few tools can do this)
  - Only linux: memory leaks

Usage (gcc/clang syntax)

- Compile with
  `-fsanitize=address -fno-omit-frame-pointer -g`
- With clang, add optionally:
  `-fsanitize-address-use-after-return=always`
  `-fsanitize-address-use-after-scope`
- Link with `-fsanitize=address`
- Run the program

Address Sanitizer (ASan)

How it works

- Compiler adds run-time checks (∼2x slow down)
- `IsPoisoned(address)` looks up state of address in asan's "shadow memory"
- Shadow memory: memory where 1 shadow byte tracks state of 8 application bytes (state = accessible, poisoned, ...)
- Functions that deal with memory (`new()` / `delete()` / strings / ...) update entries in shadow memory when called

asan instrumentation (mock code)

```
int i = *address;
```

Address Sanitizer (ASan)

How it works

- Compiler adds run-time checks (∼2x slow down)
- `IsPoisoned(address)` looks up state of address in asan's "shadow memory"
- Shadow memory: memory where 1 shadow byte tracks state of 8 application bytes (state = accessible, poisoned, ...)
- Functions that deal with memory (`new()` / `delete()` / strings / ...) update entries in shadow memory when called

asan instrumentation (mock code)

```
if (IsPoisoned(address)) {
  ReportError(address, kAccessSize, kIsWrite);
}
int i = *address;
```

ASan red zones

- If adjacent data blocks are owned by the process, the operating system will allow an access

Illegal access (not detected without ASan)

```cpp
void foo() {
  char a[8];
  char b[8];
  a[8] = '1';
}
```

Memory layout

				0x300C
				0x3008
				0x3004
a[0-7]	b[0-7]			0x3000

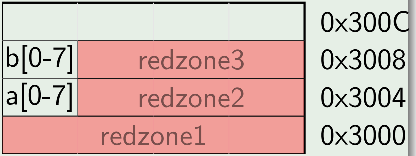
ASan red zones

- If adjacent data blocks are owned by the process, the operating system will allow an access
- ASan surrounds blocks of memory by poisoned red zones
- Program stops when accessing a red zone

Illegal access (not detected without ASan)

```
  void foo() {
+   char redzone1[32];
    char a[8];
+   char redzone2[24];
    char b[8];
+   char redzone3[24];
+   // <poison redzones>
    a[8] = '1';
+   // <unpoison redzones>
  }
```

Memory layout



```
==34015==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffee93ed968 a
WRITE of size 1 at 0x7ffee93ed968 thread T0
    #0 0x106812df3 in foo() asan.cpp:4
    #1 0x106812ed8 in main asan.cpp:9
    #2 0x7fff6d3923d4 in start (libdyld.dylib:x86_64+0x163d4)

Address 0x7ffee93ed968 is located in stack of thread T0 at offset 40 in frame
    #0 0x106812cdf in foo() asan.cpp:1

  This frame has 2 object(s):
    [32, 40) 'a' (line 2) <== Memory access at offset 40 overflows this variable
    [64, 72) 'b' (line 3)
Shadow bytes around the buggy address:
=>0x1fffdd27db20: 00 00 00 00 00 00 00 00 f1 f1 f1 f1 00[f2]f2 f2
  0x1fffdd27db30: 00 f3 f3 f3 00 00 00 00 00 00 00 00 00 00 00 00
  0x1fffdd27db40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1fffdd27db50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
```

Finding memory leaks with ASan

- On linux, ASan can display memory leaks
- Start executable with
  `ASAN_OPTIONS=detect_leaks=1 ./myProgram`

```
==113262==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 32 byte(s) in 1 object(s) allocated from:
  #0 0x7f2671201647 in operator new(unsigned long) /build/dkonst/WORK/build/contri
  #1 0x4033c7 in memoryLeak[abi:cxx11]() /afs/cern.ch/user/s/shageboe/asan.cpp:33
  #2 0x403633 in main /afs/cern.ch/user/s/shageboe/asan.cpp:40
  #3 0x7f2670a15492 in __libc_start_main (/lib64/libc.so.6+0x23492)

Indirect leak of 22 byte(s) in 1 object(s) allocated from:
  #0 0x7f2671201647 in operator new(unsigned long) /build/dkonst/WORK/build/contri
  #1 0x403846 in void std::__cxx11::basic_string<char, std::char_traits<char>, std:
  #2 0x4033f4 in std::__cxx11::basic_string<char, std::char_traits<char>, std::allo
  #3 0x4033f4 in memoryLeak[abi:cxx11]() /afs/cern.ch/user/s/shageboe/asan.cpp:33
  #4 0x403633 in main /afs/cern.ch/user/s/shageboe/asan.cpp:40
  #5 0x7f2670a15492 in __libc_start_main (/lib64/libc.so.6+0x23492)

SUMMARY: AddressSanitizer: 54 byte(s) leaked in 2 allocation(s).
```

Address sanitizer (ASan)

Wrap up

- If a program crashes, run it with asan
- Should be part of every C++ continuous integration system
- It will also find bugs that by luck didn't crash the program
- It doesn't generate false positives

More info

- https://github.com/google/sanitizers/wiki/AddressSanitizer
- Compile with asan, and start executable using
  `ASAN_OPTIONS=help=1 <executable>`

Address sanitizer (ASan)

Exercise: address sanitizer

- Go to `exercises/asan`
- Compile and run the program `./asan`
- There are two bugs and one memory leak. Use asan to trace them down.

Thread sanitizer (TSan)

TSan

- Thread sanitizer detects many data races in MT programs
- Recompile your program with e.g.
  `clang++ -fsanitize=thread -g -O1 datarace.cpp`

```
% ./a.out
WARNING: ThreadSanitizer: data race (pid=19219)
Write of size 4 at 0x7fcf47b21bc0 by thread T1:
  #0 Thread1 datarace.c:4 (exe+0x00000000a360)

Previous write of size 4 at 0x7fcf47b21bc0 by main thread:
  #0 main datarace.c:10 (exe+0x00000000a3b4)

Thread T1 (running) created at:
  #0 pthread_create tsan_interceptors.cc:705 (exe+0x00000000c790)
  #1 main datarace.c:9 (exe+0x00000000a3a4)
```

https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual

Undefined Behaviour Sanitizer (UBSan)

UBSan

- Tracks uninitialised memory, broken arithmetic, wrong array indexing and other undefined behaviour
- Recompile your program with e.g.
  `clang++ -fsanitize=undefined -g -O1 ub.cpp`

```
% ./a.out
up.cpp:3:5: runtime error: signed integer overflow:
            2147483647 + 1 cannot be represented in type 'int'
```

https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

The Valgrind family

6 Useful tools

- C^{++} editor
- Version control
- Code formatting
- The Compiling Chain
- Web tools
- Debugging
- Sanitizers
- The Valgrind family
- Static code analysis
- Profiling
- Doxygen

The valgrind family

Valgrind fundamentals

- valgrind is a framework for different tools
- a processor simulator allowing checks in between instructions
- slow (10-50 times slower than normal execution)
- easy to use : "valgrind <your executable>"
  - no recompilation
  - better with -g -O0, but not strictly needed
- it is free and open source

The valgrind family

Valgrind fundamentals

- valgrind is a framework for different tools
- a processor simulator allowing checks in between instructions
- slow (10-50 times slower than normal execution)
- easy to use : "valgrind <your executable>"
  - no recompilation
  - better with -g -O0, but not strictly needed
- it is free and open source

Main tools

memcheck	a memory checker (default tool) and leak detector
callgrind	a call graph builder
helgrind	a race condition detector

memcheck

- keeps track of all memory allocations and deallocations
- is able to detect accesses to unallocated memory
- and even tell you when it was deallocated if it was
- or what is the closest array in case of overflow
- is able to list still allocated memory when program exits (memory leaks detection)

valgrind

Exercise: valgrind

- go to `exercises/valgrind`
- compile, run, it should work
- run with valgrind, see the problem
- fix the problem

- go back to the `exercises/debug` exercise
- check it with valgrind
- analyze the issue, see that the variance was biaised
- fix the issue

memcheck

Exercise: memcheck

- go to `exercises/memcheck`
- compile, run, it should work
- run with valgrind, see LEAK summary
- run with --leak-check=full to get details
- analyze and correct it

callgrind and kcachegrind

callgrind

- keeps track of all function calls
- and time spent in each function
- build statistics on calls, CPU usages and more
- outputs flat statistics file, quite unreadable

kcachegrind

- a gui exploiting statistics built by callgrind
- able to browse graphically the program calls
- able to “graph” CPU usage on the program structure

callgrind

Exercise: callgrind

- go to `exercises/callgrind`
- compile, run, it will be slow
- change nb iterations to 20
- run with valgrind --tool=callgrind
- look at output with kcachegrind
- change fibo call to fibo2
- observe the change in kcachegrind

helgrind

- keeps track of all pthreads activity
- in particular keeps track of all mutexes
- builds a graph of dependencies of the different actions
- works on the resulting graph to detect:
  - possible dead locks
  - possible data races

helgrind

- keeps track of all pthreads activity
- in particular keeps track of all mutexes
- builds a graph of dependencies of the different actions
- works on the resulting graph to detect:
  - possible dead locks
  - possible data races

Note the "possible". It finds future problems!

helgrind

Exercise: helgrind

- go to `exercises/helgrind`
- compile, run
- check it with valgrind. You may see strange behavior or it will be perfectly fine
- check it with valgrind --tool=helgrind
- understand issue and fix

Static code analysis

Static analysis

The problem

- all the tools discussed so far work on binaries
- they analyze the code being run
- so there is a coverage problem (e.g. for error cases)

Static analysis

The problem

- all the tools discussed so far work on binaries
- they analyze the code being run
- so there is a coverage problem (e.g. for error cases)

A (partial) solution : analyzing the source code

- build a graph of dependencies of the calls
- use graph tools to detect potential memory corruptions, memory leaks or missing initializations

Static analysis

The problem

- all the tools discussed so far work on binaries
- they analyze the code being run
- so there is a coverage problem (e.g. for error cases)

A (partial) solution : analyzing the source code

- build a graph of dependencies of the calls
- use graph tools to detect potential memory corruptions, memory leaks or missing initializations

Existing tools

- Coverity: proprietary tool, the most complete
- cppcheck: free and opensource, but less complete
- clang-tidy: clang-based “linter”, includes clang static analyzer

cppcheck

Exercise: cppcheck

- go to `exercises/cppcheck`
- compile, run, see that it works
- use valgrind: no issue
- use cppcheck, see the problem
- analyze the issue, and fix it
- bonus: understand why valgrind did not complain
  and how the standard deviation could be biased
  hint : use gdb and check addresses of v and diffs

clang-tidy

Documentation and supported checks

- https://clang.llvm.org/extra/clang-tidy/

Run clang-tidy

- `clang-tidy <file.cpp> -checks=...`
- `clang-tidy <file.cpp>` (checks from .clang-tidy file)
- `clang-tidy <file.cpp> --fix` (applies fixes)

Compilation flags

- clang-tidy needs to know exactly how your program is built
- `clang-tidy ... -- <all compiler flags>`

.clang-tidy file

- describes which checks to run
- usually checked in at repository root

Automatically collecting compilation flags

- clang-tidy looks for a file called compile_commands.json
- contains the exact build flags for each .cpp file
- generate with CMake:
  `cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ...`
- for Makefiles try ▸ Bear
- allows to run clang-tidy in bulk on all files:
  - `run-clang-tidy -checks ...`
  - `run-clang-tidy` (checks from .clang-tidy)
  - `run-clang-tidy -fix` (applies fixes)

clang-tidy

Exercise: clang-tidy

- go to any example which compiles (e.g. `exercises/cppcheck`)
- `mkdir build && cd build`
- `cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ..`
- `clang-tidy <../file.cpp> -checks=*`
- inspect output
- run with `--fix` flag
- revert changes using `git checkout <../file.cpp>`

Profiling

6 Useful tools

- C^{++} editor
- Version control
- Code formatting
- The Compiling Chain
- Web tools
- Debugging
- Sanitizers
- The Valgrind family
- Static code analysis
- Profiling
- Doxygen

Profiling

Conceptually

- take a measurement of a performance aspect of a program
  - where in my code is most of the time spent?
  - is my program compute or memory bound?
  - does my program make good use of the cache?
  - is my program using all cores most of the time?
  - how often are threads blocked and why?
  - which API calls are made and in which order?
  - ...
- the goal is to find performance bottlenecks
- is usually done on a compiled program, not on source code

perf, VTune and uProf

perf

- perf is a powerful command line profiling tool for linux
- compile with `-g -fno-omit-frame-pointer`
- `perf stat -d <prg>` gathers performance statistics while running `<prg>`
- `perf record -g <prg>` starts profiling `<prg>`
- `perf report` displays a report from the last profile
- More information in this wiki, this website or this talk.

Intel VTune and AMD uProf

- Graphical profilers from CPU vendors with rich features
- Needs vendor's CPU for full experience
- More information on Intel's website and AMD's website

Doxygen

6 Useful tools
- C++ editor
- Version control
- Code formatting
- The Compiling Chain
- Web tools
- Debugging
- Sanitizers
- The Valgrind family
- Static code analysis
- Profiling
- Doxygen

Doxygen

Doxygen

- Generates documentation from source code comments
- Output formats: HTML, LaTeX, XML, ...
  - May be input for further generators, e.g. Sphinx
- Doxygen uses a config file, usually called Doxyfile
- Run `doxygen -g` to generate an initial Doxyfile
- Edit Doxyfile (e.g. output format, source code location, etc.)
- Run `doxygen` to (re-)generate documentation
- View e.g. HTML documentation using a standard web browser
- More on the doxygen website

Doxygen

Comment blocks

```cpp
/**
 * text/directives ...
 */
some_cpp_entity

/*!
 * text/directives ...
 */
some_cpp_entity
///
/// text/directives ...
///
some_cpp_entity

//!
//! text/directives ...
//!
some_cpp_entity
```

- More details available here

Comment blocks for members

```cpp
int a_class_member; ///< trailing documentation
```

Doxygen

Comment directives

```cpp
/**
 * Long description here. May include HTML etc.
 *
 * \brief Checks whether i is odd
 * \tparam Integral The integral type of the input
 * \param i Input value
 * \return True if i is odd, otherwise false
 * \throw std::out_of_range if i is larger than 100
 * \see isEven
 */
template <typename Integral>
bool isOdd(Integral i);
```

- All directives can also start with @ instead of \
- A list of all commands can be found here

Concurrency

Threads and async

7 Concurrency
- Threads and async
- Mutexes
- Atomic types
- Thread-local storage
- Condition Variables

Basic concurrency

C++ 11/C++ 20

Threading

- C++ 11 added `std::thread` in `<thread>` header
- takes a function as argument of its constructor
- must be detached or joined before the main thread terminates
- C++ 20: `std::jthread` automatically joins at destruction

Example code

```
void foo() {...}
void bar() {...}
int main() {
  std::thread t1{foo};
  std::thread t2{bar};
  for (auto t: {&t1,&t2}) t->join();
  return 0;
}
```

Basic concurrency

C++ 11/C++ 20

Threading

- C++ 11 added `std::thread` in `<thread>` header
- takes a function as argument of its constructor
- must be detached or joined before the main thread terminates
- C++ 20: `std::jthread` automatically joins at destruction

Example with jthread (C++ 20)

```cpp
void foo() {...}
void bar() {...}
int main() {
  std::jthread t1{foo};
  std::jthread t2{bar};
  return 0;
}
```

The thread constructor

C++ 11

Can take a function and its arguments

```
void function(int j, double j) {...};
std::thread t1{function, 1, 2.0};
```

The thread constructor

C++ 11

Can take a function and its arguments

```cpp
void function(int j, double j) {...};
std::thread t1{function, 1, 2.0};
```

Can take any function-like object

```cpp
struct AdderFunctor {
  AdderFunctor(int i): m_i(i) {}
  int operator() (int j) const { return m_i+j; }
  int m_i;
};
std::thread t2{AdderFunctor{2}, 5};
int a;
std::thread t3{[](int i) { return i+2; }, a};
std::thread t4{[a]       { return a+2; }};
```

Basic asynchronicity

C++ 11

Concept

- separation of the specification of what should be done and the retrieval of the results
- "start working on this, and let me check if it's done"

Basic asynchronicity

C++ 11

Concept

- separation of the specification of what should be done and the retrieval of the results
- "start working on this, and let me check if it's done"

Practically

- `std::async` function launches an asynchronous task
- `std::future<T>` allows to retrieve the result

Basic asynchronicity

C++ 11

Concept

- separation of the specification of what should be done and the retrieval of the results
- “start working on this, and let me check if it’s done”

Practically

- `std::async` function launches an asynchronous task
- `std::future<T>` allows to retrieve the result

Example code

```cpp
int f() {...}
std::future<int> res = std::async(f);
std::cout << res.get() << "\n";
```

Mixing the two

C++ 11

Is async running concurrently ?

- it depends!
- you can control this with a launch policy argument

  std::launch::async start executing immediately on a separate thread (may be a new thread, a thread pool, ...)

  std::launch::deferred causes lazy execution in current thread

  - execution starts when `get()` is called on the returned future
- default is not specified, but tries to use existing concurrency!

Mixing the two

C++ 11

Is async running concurrently ?

- it depends!
- you can control this with a launch policy argument

  std::launch::async start executing immediately on a separate thread (may be a new thread, a thread pool, ...)

  std::launch::deferred causes lazy execution in current thread

  - execution starts when `get()` is called on the returned future
- default is not specified, but tries to use existing concurrency!

Usage

```
int f() {...}
auto res1 = std::async(std::launch::async, f);
auto res2 = std::async(std::launch::deferred, f);
```

Fine-grained control on asynchronous execution C++ 11

`std::packaged_task` template

- creates an asynchronous version of any function-like object
  - identical arguments
  - returns a `std::future`
- provides access to the returned future
- associated with threads, gives full control on execution

Fine-grained control on asynchronous execution C++ 11

std::packaged_task template

- creates an asynchronous version of any function-like object
  - identical arguments
  - returns a std::future
- provides access to the returned future
- associated with threads, gives full control on execution

Usage

```cpp
int f() { return 42; }
std::packaged_task<int()> task{f};
auto future = task.get_future();
task();
std::cout << future.get() << std::endl;
```

Mutexes

7 Concurrency
- Threads and async
- Mutexes
- Atomic types
- Thread-local storage
- Condition Variables

Races

C++ 11

Example code - godbolt

```cpp
int a = 0;
void inc() { a++; };
void inc100() {
  for (int i=0; i < 100; i++) inc();
};
int main() {
  std::thread t1{inc100};
  std::thread t2{inc100};
  for (auto t: {&t1,&t2}) t->join();
  std::cout << a << "\n";
}
```

Races

C++ 11

Example code - godbolt

```cpp
int a = 0;
void inc() { a++; };
void inc100() {
  for (int i=0; i < 100; i++) inc();
};
int main() {
  std::thread t1{inc100};
  std::thread t2{inc100};
  for (auto t: {&t1,&t2}) t->join();
  std::cout << a << "\n";
}
```

What do you expect? (Exercise exercises/race)

Races

C++ 11

Example code - godbolt

```cpp
int a = 0;
void inc() { a++; };
void inc100() {
  for (int i=0; i < 100; i++) inc();
};
int main() {
  std::thread t1{inc100};
  std::thread t2{inc100};
  for (auto t: {&t1,&t2}) t->join();
  std::cout << a << "\n";
}
```

What do you expect? (Exercise exercises/race)

Anything between 100 and 200 !!!

Atomicity

C++ 11

Definition (wikipedia)

- an operation (or set of operations) is atomic if it appears to the rest of the system to occur instantaneously

Practically

- an operation that won't be interrupted by other concurrent operations
- an operation that will have a stable environment during execution

Atomicity

Definition (wikipedia)

- an operation (or set of operations) is atomic if it appears to the rest of the system to occur instantaneously

Practically

- an operation that won't be interrupted by other concurrent operations
- an operation that will have a stable environment during execution

Is ++ operator atomic ?

Atomicity

C++ 11

Definition (wikipedia)

- an operation (or set of operations) is atomic if it appears to the rest of the system to occur instantaneously

Practically

- an operation that won't be interrupted by other concurrent operations
- an operation that will have a stable environment during execution

Is ++ operator atomic ?

Usually not. It behaves like :

```
eax = a          // memory to register copy
increase eax     // increase (atomic CPU instruction)
a = eax          // copy back to memory
```

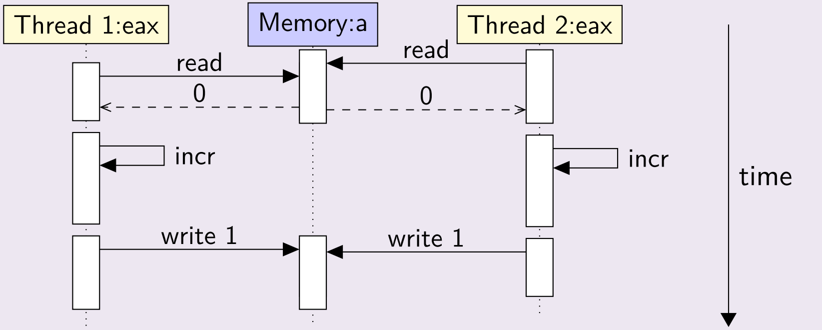
Timing

C++ 11

Code

```
eax = a          // memory to register copy
increase eax     // increase (atomic CPU instruction)
a = eax          // copy back to memory
```

For 2 threads



Mutexes and Locks

C++ 17

Concept

- Use locks to serialize access to a non-atomic piece of code

Mutexes and Locks

C++ 17

Concept

- Use locks to serialize access to a non-atomic piece of code

The objects

std::mutex	in the `mutex` header. **Mut**ual **ex**clusion
std::scoped_lock	RAII to lock and unlock automatically
std::unique_lock	same, but can be released/relocked explicitly

Mutexes and Locks

C++ 17

Concept

- Use locks to serialize access to a non-atomic piece of code

The objects

std::mutex	in the `mutex` header. **Mut**ual **ex**clusion
std::scoped_lock	RAII to lock and unlock automatically
std::unique_lock	same, but can be released/relocked explicitly

Practically - godbolt

```cpp
int a = 0;
std::mutex m;
void inc() {
  std::scoped_lock lock{m};
  a++;
}
```

Mutexes and Locks

C++ 17

Good practice: Locking

- Generally, use `std::scoped_lock`. Before C++ 17 use `std::lock_guard`.
- Hold as short as possible, consider wrapping critical section in block statement { }
- Only if manual control needed, use `std::unique_lock`

```cpp
void function(...) {
  // uncritical work ...
  {
    std::scoped_lock myLocks{mutex1, mutex2, ...};
    // critical section
  }
  // uncritical work ...
}
```

Mutexes and Locks

Exercise: Mutexes and Locks

- Go to `exercises/race`
- Look at the code and try it
  See that it has a race condition
- Use a mutex to fix the issue
- See the difference in execution time

Dead locks

C^{++} 11

Scenario

- 2 mutexes, 2 threads
- locking order different in the 2 threads

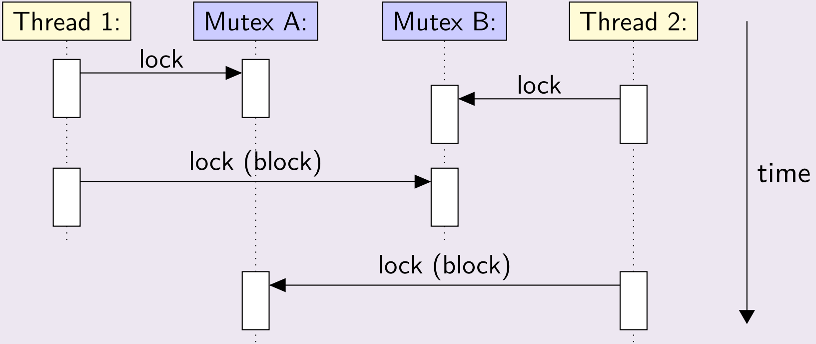
Dead locks

C++ 11

Scenario

- 2 mutexes, 2 threads
- locking order different in the 2 threads

Sequence diagram



How to avoid dead locks C^{++} 11

Possible solutions

- C^{++} 17: `std::scoped_lock lock{m1, m2};` comes with deadlock-avoidance algorithm
- Never take several locks
  - Or add master lock protecting the locking phase
- Respect a strict locking order across all threads
- Do not use locks
  - Use other techniques, e.g. lock-free queues

Shared mutex / locks

C++ 17

Sharing a mutex

- Normal `std::mutex` objects cannot be shared
- `std::shared_mutex` to the rescue, but can be slower
- It locks *either* in exclusive *or* in shared mode; never both

```
Data data; std::shared_mutex mutex;
auto reader = [&](){
  std::shared_lock lock{mutex};
  read(data); // Many can read
};
std::thread r1{reader}, r2{reader}, ...;

std::thread writer([&](){
  std::scoped_lock lock{mutex}; // exclusive
  modify(data); // Only one can write
});
```

Atomic types

7 Concurrency
- Threads and async
- Mutexes
- Atomic types
- Thread-local storage
- Condition Variables

Atomic types in C++ C++ 11

std::atomic template

- Any trivially copyable type can be made atomic in C++
- Most useful for integral types
- May internally use locks for custom types

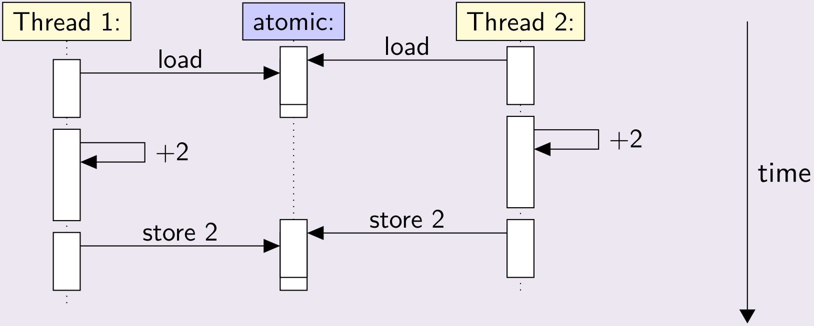
```
std::atomic<int> a{0};
std::thread t1([&](){ a++; });
std::thread t2([&](){ a++; });
a += 2;
t1.join(); t2.join();
assert( a == 4 ); // Guaranteed to succeed
```

Warning: Expressions using an atomic type are *not* atomic!

```cpp
std::atomic<int> a{0};
std::thread t1([&]{ a = a + 2; });
std::thread t2([&]{ a = a + 2; });
```

- Atomic load; value+2; atomic store

Sequence diagram

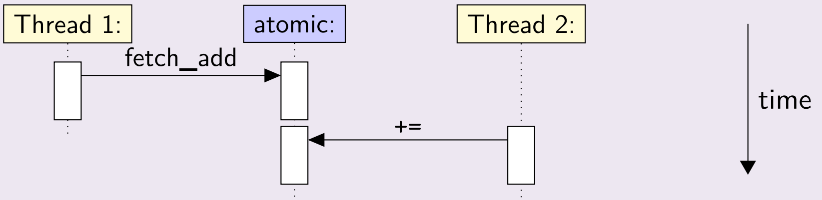


Solution: Use atomic member functions

- The member functions of `std::atomic` are thread safe
- `fetch_add` and `operator+=`: atomic { load; add; store }
- But don't confuse "`a += 2`" and "`a = a + 2`"

```
std::atomic<int> a{0};
std::thread t1([&]{ a.fetch_add(2); });
std::thread t2([&]{ a += 2; });
```

Sequence diagram



Atomic references

C++ 20

std::atomic_ref template

- Wraps a T& and makes access to it atomic
- Like std::atomic<T>, but does not contain the T

```
int a{0};
std::thread t1([&]{ std::atomic_ref<int> r{a}; r++;});
std::thread t2([&]{ std::atomic_ref{a}++; });
t1.join(); t2.join();
a += 2; // non-atomic (fine, threads joined before)
assert( a == 4 ); // Guaranteed to succeed
```

Don't mix concurrent atomic and non-atomic access

```
std::thread t3([&]{ std::atomic_ref{a}++; });
a += 2; // data race
t3.join();
```

Atomic types in C++

Exercise: Atomics

- Go to `exercises/atomic`
- You'll find a program with the same race condition as in `race`
- Fix it using `std::atomic`

Thread-local storage

7 Concurrency
- Threads and async
- Mutexes
- Atomic types
- Thread-local storage
- Condition Variables

Thread-local storage

C++ 20

thread_local keyword

- A variable can be declared thread-local
- Then every thread will have its own copy
- Most useful for "working memory" in each thread
- Note: Can also be declared directly on the stack of a thread
  - and will be faster, thus should be preferred when possible

```cpp
thread_local int a{0};
std::jthread t1([&] { a++; });
std::jthread t2([&] { a++; });
a += 2;
t1.join(); t2.join();
assert( a == 2 ); // Guaranteed to succeed
```

Condition Variables

7 Concurrency
- Threads and async
- Mutexes
- Atomic types
- Thread-local storage
- Condition Variables

Condition variables

C++ 11

Communicating between threads

- Take the case where threads are waiting for other thread(s)
- `std::condition_variable`
  - from `<condition_variable>` header
- Allows for a thread to sleep (= conserve CPU time) until a given condition is satisfied

Condition variables

C++ 11

Communicating between threads

- Take the case where threads are waiting for other thread(s)
- `std::condition_variable`
  - from `<condition_variable>` header
- Allows for a thread to sleep (= conserve CPU time) until a given condition is satisfied

Usage

- Use RAII-style locks to protect shared data
- `wait()` will block until the condition is met
  - you can have several waiters sharing the same mutex
- `notify_one()` will wake up one waiter
- `notify_all()` will wake up all waiters

Using condition variables: notify C^{++} 17

Producer side: providing data to waiting threads

- Protect data with a mutex, and use condition variable to notify consumers
- Optimal use: don't hold lock while notifying
  - waiting threads would be blocked

```cpp
std::mutex mutex;
std::condition_variable cond;
Data data;
std::thread producer([&](){
  {
    std::scoped_lock lock{mutex};
    data = produceData(); // may take long ...
  }
  cond.notify_all();
});
```

Using condition variables: wait C++ 11

Mechanics of wait

- Many threads are waiting for shared data
- Pass a `unique_lock` and a predicate for wakeup to `wait()`
- `wait()` sends threads to sleep while predicate is `false`
- `wait()` will only lock when necessary; unlocked while sleeping
- Threads might wake up spuriously, but `wait()` returns only when lock available *and* predicate `true`

Naïve waiting

```cpp
auto processData = [&](){
  std::unique_lock<std::mutex> lock{mutex};
  cond.wait(lock, [&](){ return data.isReady(); });
  process(data);
};
```

Using condition variables: wait

C++ 11

Waiting / waking up

- `notify_all()` is called, threads wake up
- Threads try to lock mutex, and evaluate predicate
- One thread succeeds to acquire mutex, starts data processing
- Problem: Thread holds mutex now, other threads are blocked!

Naïve waiting

```
auto processData = [&](){
  std::unique_lock<std::mutex> lock{mutex};
  cond.wait(lock, [&](){ return data.isReady(); });
  process(data);
};
```

Using condition variables: correct wait

C++ 11

Waiting / waking up

- Solution: Put locking and waiting in a scope
- Threads will one-by-one wake up, acquire lock, evaluate predicate, release lock

Correct waiting

```
auto processData = [&](){
  {
    std::unique_lock<std::mutex> lock{mutex};
    cond.wait(lock, [&](){ return data.isReady(); });
  }
  process(data);
};
std::thread t1{processData}, t2{processData}, ...;
for (auto t : {&producer, &t1, &t2, ...}) t->join();
```

Condition variables

Exercise: Condition variables

- Go to `exercises/condition_variable`
- Look at the code and run it
  See that it has a race condition
- Fix the race condition in the usage of the condition variable
- Try to make threads process data in parallel

C^{++} and python

Writing a module

8 C++ and python

- Writing a module
- Marrying C++ and C
- The ctypes module
- The cppyy project

How to build a python 3 module around C++ code

C++ code : mandel.hpp

```
int mandel(Complex const & a);
```

Basic Module(1): wrap your method

mandelModule.cpp - see exercises/python exercise

```cpp
#include <Python.h>
#include "mandel.hpp"
PyObject * mandel_wrapper(PyObject * self,
                          PyObject * args) {
  // Parse Input
  float r, i;
  if (!PyArg_ParseTuple(args, "ff", &r, &i))
    return nullptr;
  // Call C++ function
  int result = mandel(Complex(r, i));
  // Build returned objects
  return PyLong_FromLong(result);
}
```

Basic Module(2): create the python module

mandelModule.cpp - see exercises/python exercise

```cpp
// declare the modules' methods
PyMethodDef mandelMethods[] = {
    {"mandel", mandel_wrapper, METH_VARARGS,
    "computes nb of iterations for mandelbrot set"},
    {nullptr, nullptr, 0, nullptr}
};
// declare the module
struct PyModuleDef mandelModule = {
  PyModuleDef_HEAD_INIT,
  "mandel", nullptr, -1, mandelMethods
};
PyMODINIT_FUNC PyInit_mandel() {
  return PyModule_Create(&mandelModule);
}
```

Basic Module(3): use it

First compile the module

- as a regular shared library
- with '-I \$(PYTHON_INCLUDE)'

mandel.py - see exercises/python exercise

```
from mandel import mandel
v = mandel(0.7, 1.2)
```

Marrying C++ and C

A question of mangling

Mangling

the act of converting the name of variable or function to a symbol name in the binary code

C versus C++ symbol names

- C uses bare function name
- C++ allows overloading of functions by taking the signature into account
- so C++ mangling has to contain signature

C mangling

Source : file.c

```c
float sum(float a, float b);
int square(int a);
// won't compile : conflicting types for 'square'
// float square(float a);
```

Binary symbols : file.o

```
# nm file.o
000000000000001a T square
0000000000000000 T sum
```

C++ mangling

Source : file.cpp

```cpp
float sum(float a, float b);
int square(int a);
// ok, signature is different
float square(float a);
```

Binary symbols : file.o

```
# nm file.o
0000000000000000 T _Z3sumff
000000000000002a T _Z6squaref
000000000000001a T _Z6squarei
```

Forcing C mangling in C^{++}

extern “C”

These functions will use C mangling :

```
extern "C" {
  float sum(float a, float b);
  int square(int a);
}
```

Forcing C mangling in C^{++}

extern "C"

These functions will use C mangling :

```
extern "C" {
  float sum(float a, float b);
  int square(int a);
}
```

You can now call these C^{++} functions from C code

Forcing C mangling in C++

extern "C"

These functions will use C mangling :

```
extern "C" {
  float sum(float a, float b);
  int square(int a);
}
```

You can now call these C++ functions from C code

Limitations

- no C++ types should go out
- no exceptions either (use `noexcept` here)
- member functions cannot be used
  - they need to be wrapped one by one

The ctypes module

8 C++ and python
- Writing a module
- Marrying C++ and C
- The ctypes module
- The cppyy project

The ctypes python module

From the documentation

- provides C compatible data types
- allows calling functions in DLLs or shared libraries
- can be used to wrap these libraries in pure Python

ctypes: usage example

C^{++} code : mandel.hpp

```
int mandel(Complex const & a);
```

"C" code : mandel_cwrapper.hpp

```
extern "C" {
  int mandel(float r, float i) {
    return mandel(Complex(r, i));
  };
}
```

calling the mandel library

```
from ctypes import *
libmandel = CDLL('libmandelc.so')
v = libmandel.mandel(c_float(0.3), c_float(1.2))
```

Marrying C++ and python

Exercise: C++ and python

- go to `exercises/python`
- look at the original python code mandel.py
- time it ('time python3 mandel.py')
- look at the code in mandel.hpp/cpp
- look at the python module mandel_module.cpp
- compile and modify mandel.py to use it
- see the gain in time
- look at the C wrapper in mandel_cwrapper.cpp
- modify mandel.py to use libmandelc directly with ctypes

Note : you may have to add '.' to LD_LIBRARY_PATH and PYTHONPATH

The cppyy project

8 C^{++} and python

- Writing a module
- Marrying C^{++} and C
- The ctypes module
- The cppyy project

Automatic Python-C++ bindings

The cppyy project

- originated from the ROOT project
- still young, version 1.0 from Mid 2018
- but very active, current version 2.1.0
- extremely powerful for interfacing C++ and python

How it works

- uses Just in Time compilation through cling
  - an interactive C++ interpreter

cppyy crash course(1)

Shamelessly copied from the cppyy documentation

```
>>> import cppyy
>>> cppyy.include('zlib.h')        # bring in C++ definitions
>>> cppyy.load_library('libz')     # load linker symbols
>>> cppyy.gbl.zlibVersion()        # use a zlib API
'1.2.11'
```

```
>>> import cppyy
>>> cppyy.cppdef("""
... class MyClass {
... public:
...     MyClass(int i) : m_data(i) {}
...     virtual ~MyClass() {}
...     virtual int add_int(int i) { return m_data + i; }
...     int m_data;
... };""")
True
```

cppyy crash course(1)

```
>>> from cppyy.gbl import MyClass
>>> m = MyClass(42)
>>> cppyy.cppdef("""
... void say_hello(MyClass* m) {
...     std::cout << "Hello, the number is: " << m->m_data << std::endl;
... }""")
True
>>> MyClass.say_hello = cppyy.gbl.say_hello
>>> m.say_hello()
Hello, the number is: 42
>>> m.m_data = 13
>>> m.say_hello()
Hello, the number is: 13
```

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