

HEP C++ course

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

CERN

September 30, 2024



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
 - 556 slides, 697 pages, 32 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

- 1 History and goals
- 2 Language basics
- 3 Object orientation (OO)
- 4 Core modern C++
- 5 Expert C++
- 6 Useful tools
- 7 Concurrency
- 8 C++ and python



Detailed outline

- 1 **History and goals**
 - History
 - Why we use it?
- 2 **Language basics**
 - Core syntax and types
 - Arrays and Pointers
 - Scopes / namespaces
 - Class and enum types
 - References
 - Functions
 - Operators
 - Control structures
 - Headers and interfaces
 - Auto keyword
 - Inline keyword
 - Assertions
- 3 **Object orientation (OO)**
 - Objects and Classes
 - Inheritance
 - Constructors/destructors
 - Static members
 - Allocating objects
 - Advanced OO
- 4 **Core modern C++**
 - Type casting
 - Operator overloading
 - Function objects
 - Name Lookups
 - Constness
 - Constant Expressions
 - Exceptions
 - Move semantics
 - Copy elision
 - Templates
 - Lambdas
 - The STL
 - Standard library
 - Ranges
 - RAI and smart pointers
 - Initialization
- 5 **Expert C++**
 - The <=> operator
 - Variadic templates
 - Perfect forwarding
 - SFINAE
 - Concepts
 - Modules
- 6 **Useful tools**
 - C++ editor
 - Version control
 - Code formatting
 - The Compiling Chain
 - Web tools
 - Debugging
 - Sanitizers
 - The Valgrind family
 - Static code analysis
 - Profiling
 - Doxygen
- 7 **Concurrency**
 - Threads and async
 - Mutexes
 - Atomic types
 - Thread-local storage
 - Condition Variables
- 8 **C++ and python**
 - Writing a module
 - Marrying C++ and C
 - The ctypes module
 - The cppy project



History and goals

- 1 History and goals
 - History
 - Why we use it?
- 2 Language basics
- 3 Object orientation (OO)
- 4 Core modern C++
- 5 Expert C++
- 6 Useful tools
- 7 Concurrency
- 8 C++ and python

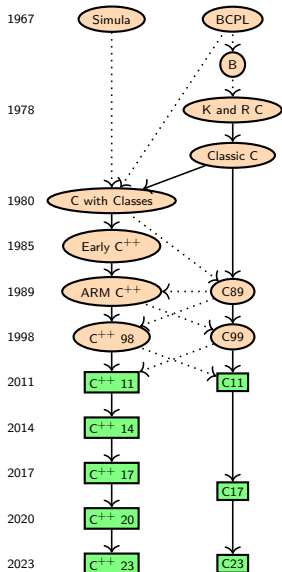


History

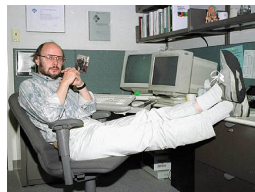
- 1 History and goals
 - History
 - Why we use it?



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	≥4.8	≥3.3
14	≥4.9	≥3.4
17	≥7.3	≥5
20	>11, 14	>12,18
23	>14	>17,18

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



Why is C++ our language of choice?

Adapted to large projects

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

Fast

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

What we get

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



Language basics

- 1 History and goals
- 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
- 3 Object orientation (OO)
- 4 Core modern C++
- 5 Expert C++
- 6 Useful tools
- 7 Concurrency
- 8 C++ and python

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

```
1  #include <iostream>
2
3  // This is a function
4  void print(int i) {
5      std::cout << "Hello, world " << i << std::endl;
6  }
7
8  int main(int argc, char** argv) {
9      int n = 3;
10     for (int i = 0; i < n; i++) {
11         print(i);
12     }
13     return 0;
14 }
```



Comments

C++ 98

```
1 // simple comment until end of line
2 int i;
3
4 /* multiline comment
5  * in case we need to say more
6  */
7 double /* or something in between */ d;
8
9 /**
10  * Best choice : doxygen compatible comments
11  * \brief checks whether i is odd
12  * \param i input
13  * \return true if i is odd, otherwise false
14  * \see https://www.doxygen.nl/manual/docblocks.html
15  */
16 bool isOdd(int i);
```



Basic types(1)

C++ 98

```
1  bool b = true;           // boolean, true or false
2
3  char c = 'a';           // min 8 bit integer
4                          // may be signed or not
5                          // can store an ASCII character
6  signed char c = 4;      // min 8 bit signed integer
7  unsigned char c = 4;    // min 8 bit unsigned integer
8
9  char* s = "a C string"; // array of chars ended by \0
10 string t = "a C++ string"; // class provided by the STL
11
12 short int s = -444;      // min 16 bit signed integer
13 unsigned short s = 444; // min 16 bit unsigned integer
14 short s = -444;         // int is optional
```



Basic types(2)

C++ 98

```
1  int i = -123456;           // min 16, usually 32 bit
2  unsigned int i = 1234567; // min 16, usually 32 bit
3
4  long l = 0L               // min 32 bit
5  unsigned long l = 0UL;    // min 32 bit
6
7  long long ll = 0LL;       // min 64 bit
8  unsigned long long l = 0ULL; // min 64 bit
9
10 float f = 1.23f;          // 32 (1+8+23) bit float
11 double d = 1.23E34;       // 64 (1+11+52) bit float
12 long double ld = 1.23E34L // min 64 bit float
```



Portable numeric types

C++ 98

```
1  #include <cstdint> // defines the following:
2
3  std::int8_t c = -3;    // 8 bit signed integer
4  std::uint8_t c = 4;   // 8 bit unsigned integer
5
6  std::int16_t s = -444; // 16 bit signed integer
7  std::uint16_t s = 444; // 16 bit unsigned integer
8
9  std::int32_t s = -674; // 32 bit signed integer
10 std::uint32_t s = 674; // 32 bit unsigned integer
11
12 std::int64_t s = -1635; // 64 bit signed integer
13 std::uint64_t s = 1635; // 64 bit unsigned int
```



Integer literals

```

1  int i = 1234;           // decimal      (base 10)
2  int i = 02322;        // octal      (base 8)
3  int i = 0x4d2;        // hexadecimal (base 16)
4  int i = 0X4D2;        // hexadecimal (base 16)
5  int i = 0b10011010010; // binary      (base 2) C++14
6
7  int i = 123'456'789;   // digit separators, C++14
8  int i = 0b100'1101'0010; // digit separators, C++14
9
10 42           // int
11 42u, 42U     // unsigned int
12 42l, 42L     // long
13 42ul, 42UL   // unsigned long
14 42ll, 42LL   // long long
15 42ull, 42ULL // unsigned long long

```



Floating-point literals

```

1  double d = 12.34;
2  double d = 12.;
3  double d = .34;
4  double d = 12e34;           // 12 * 1034
5  double d = 12E34;         // 12 * 1034
6  double d = 12e-34;        // 12 * 10-34
7  double d = 12.34e34;      // 12.34 * 1034
8
9  double d = 123'456.789'101; // digit separators, C++14
10
11 double d = 0x4d2.4p3;      // hexfloat, 0x4d2.4 * 23
12                             // = 1234.25 * 23 = 9874
13
14 3.14f, 3.14F, // float
15 3.14, 3.14, // double
16 3.14l, 3.14L, // long double

```



Useful aliases

```
1  #include <cstddef> // (and others) defines:
2
3  // unsigned integer, can hold any variable's size
4  std::size_t s = sizeof(int);
5
6  #include <cstdint> // defines:
7
8  // signed integer, can hold any diff between two pointers
9  std::ptrdiff_t c = &s - &s;
10
11 // signed/unsigned integer, can hold any pointer value
12 std::intptr_t i = reinterpret_cast<intptr_t>(&s);
13 std::uintptr_t i = reinterpret_cast<uintptr_t>(&s);
```



Extended floating-point types

Extended floating-point types

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

```
1  #include <stdfloat> // may define these:
2
3  std::float16_t  = 3.14f16; // 16 (1+5+10) bit float
4  std::float32_t  = 3.14f32; // like float (1+8+23)
5                               // but different type
6  std::float64_t  = 3.14f64; // like double (1+11+52)
7                               // but different type
8  std::float128_t = 3.14f128; // 128 (1+15+112) bit float
9  std::bfloat16_t = 3.14bf16; // 16 (1+8+7) bit float
10
11 // also F16, F32, F64, F128 or BF16 suffix possible
```



Arrays and Pointers

2 Language basics

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



Static arrays

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



Pointers

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



Pointers

```

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

```

Memory layout

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



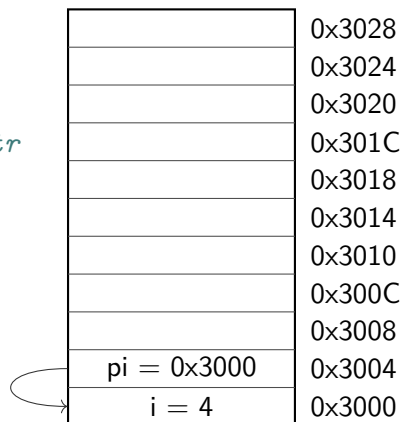
Pointers

```

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

```

Memory layout



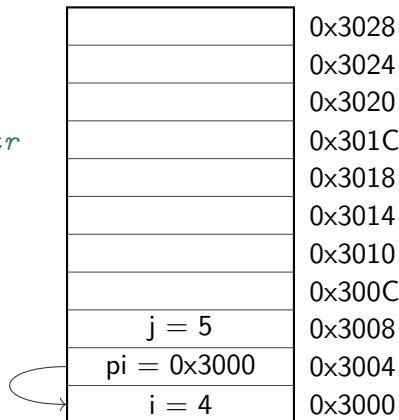
Pointers

```

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

```

Memory layout



Pointers

```

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

```

Memory layout

	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

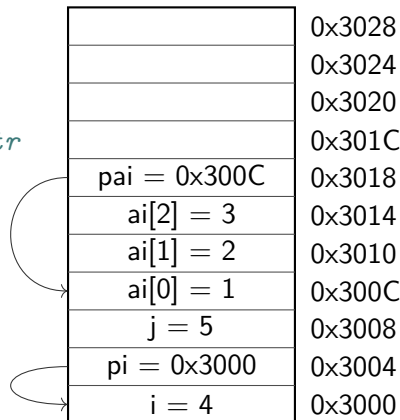
Pointers

```

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

```

Memory layout



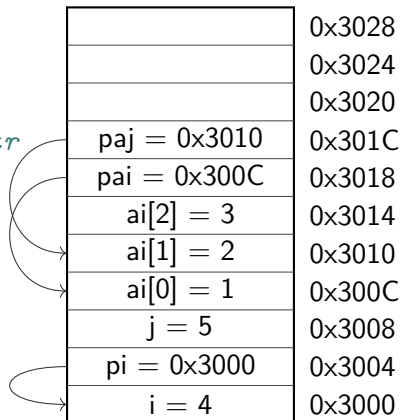
Pointers

```

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

```

Memory layout



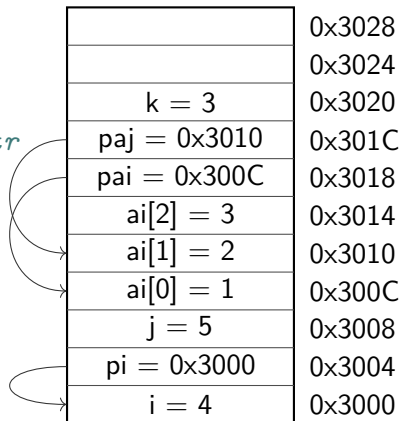
Pointers

```

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

```

Memory layout



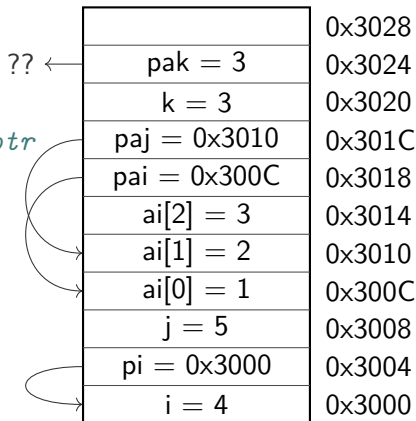
Pointers

```

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

```

Memory layout



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



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

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



Dynamic arrays using C

C++ 98

```
1  #include <cstdlib>
2  #include <cstring>
3
4  int *bad;           // pointer to random address
5  int *ai = nullptr; // better, deterministic, testable
6
7  // allocate array of 10 ints (uninitialized)
8  ai = (int*) malloc(10*sizeof(int));
9  memset(ai, 0, 10*sizeof(int)); // and set them to 0
10
11 ai = (int*) calloc(10, sizeof(int)); // both in one go
12
13 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

```
1  #include <cstdlib>
2  #include <cstring>
3
4  // allocate array of 10 ints
5  int* ai = new int[10]; // uninitialized
6  int* ai = new int[10]{}; // zero-initialized
7
8  delete[] ai; // release array memory
9
10 // allocate a single int
11 int* pi = new int;
12 int* pi = new int{};
13 delete pi; // release scalar memory
```

Good practice: Don't use manual memory management

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

Scopes / namespaces

2 Language basics

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



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

```
1 { int a;  
2   { int b;  
3     } // end of b scope  
4 } // 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

```

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

```

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

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

```

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

```

Memory layout

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

Namespaces

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

```

1  int a;
2  namespace n {
3      int a;    // no clash
4  }
5  namespace p {
6      int a;    // no clash
7      namespace inner {
8          int a; // no clash
9      }
10 }
11 void f() {
12     n::a = 3;
13 }
14 namespace p { // reopen p
15     void f() {
16         p::a = 6;
17         a = 6; //same as above
18         ::a = 1;
19         p::inner::a = 8;
20         inner::a = 8;
21         n::a = 3;
22     }
23 }
24 using namespace p::inner;
25 void g() {
26     a = -1; // err: ambiguous
27 }

```



Nested namespaces

C++ 17

Easier way to declare nested namespaces

C++ 98

```
1 namespace A {  
2     namespace B {  
3         namespace C {  
4             //...  
5         }  
6     }  
7 }
```

C++ 17

```
1 namespace A::B::C {  
2     //...  
3 }
```



Unnamed / anonymous namespaces

C++ 98 Adv

A namespace without a name!

```
1 namespace {  
2     int localVar;  
3 }
```

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

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



Using namespace directives

Avoid "using namespace" directives

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

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

Never use in headers at global scope!

```
1 #include "PoorlyWritten.h" // using namespace std;  
2 struct array { ... };  
3 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



“members” grouped together under one name

```
1  struct Individual {           14  Individual *ptr = &student;
2    unsigned char age;         15  ptr->age = 25;
3    float weight;             16  // same as: (*ptr).age = 25;
4  };
5
6  Individual student;
7  student.age = 25;
8  student.weight = 78.5f;
9
10 Individual teacher = {
11     45, 67.0f
12 };
```



“members” grouped together under one name

```

1  struct Individual {
2      unsigned char age;
3      float weight;
4  };
5
6  Individual student;
7  student.age = 25;
8  student.weight = 78.5f;
9
10 Individual teacher = {
11     45, 67.0f
12 };

```

```

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

```

Memory layout

				0x3010
				0x300C
				0x3008
				0x3004
				0x3000



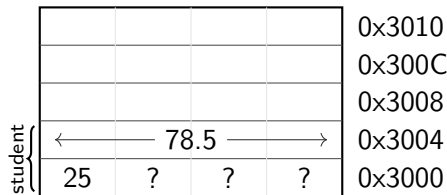
“members” grouped together under one name

```

1  struct Individual {
2      unsigned char age;
3      float weight;
4  };
5
6  Individual student;
7  student.age = 25;
8  student.weight = 78.5f;
9
10 Individual teacher = {
11     45, 67.0f
12 };
14 Individual *ptr = &student;
15 ptr->age = 25;
16 // same as: (*ptr).age = 25;

```

Memory layout



“members” grouped together under one name

```

1  struct Individual {
2      unsigned char age;
3      float weight;
4  };
5
6  Individual student;
7  student.age = 25;
8  student.weight = 78.5f;
9
10 Individual teacher = {
11     45, 67.0f
12 };
14 Individual *ptr = &student;
15 ptr->age = 25;
16 // same as: (*ptr).age = 25;

```

Memory layout

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



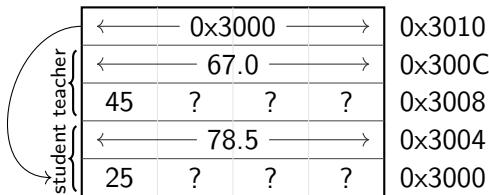
“members” grouped together under one name

```

1  struct Individual {
2      unsigned char age;
3      float weight;
4  };
5
6  Individual student;
7  student.age = 25;
8  student.weight = 78.5f;
9
10 Individual teacher = {
11     45, 67.0f
12 };
14 Individual *ptr = &student;
15 ptr->age = 25;
16 // same as: (*ptr).age = 25;

```

Memory layout



union

“members” packed together at same memory location

```
1  union Duration {
2      int seconds;
3      short hours;
4      char days;
5  };
6  Duration d1, d2, d3;
7  d1.seconds = 259200;
8  d2.hours = 72;
9  d3.days = 3;
10 d1.days = 3; // d1.seconds overwritten
11 int a = d1.seconds; // d1.seconds is garbage
```



union

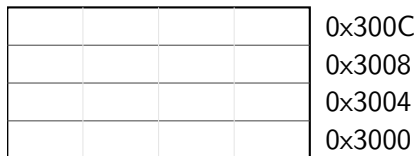
“members” packed together at same memory location

```

1  union Duration {
2      int seconds;
3      short hours;
4      char days;
5  };
6  Duration d1, d2, d3;
7  d1.seconds = 259200;
8  d2.hours = 72;
9  d3.days = 3;
10 d1.days = 3; // d1.seconds overwritten
11 int a = d1.seconds; // d1.seconds is garbage

```

Memory layout



union

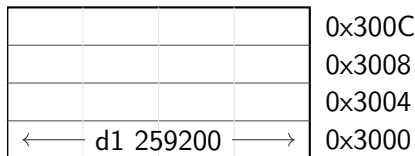
“members” packed together at same memory location

```

1  union Duration {
2      int seconds;
3      short hours;
4      char days;
5  };
6  Duration d1, d2, d3;
7  d1.seconds = 259200;
8  d2.hours = 72;
9  d3.days = 3;
10 d1.days = 3; // d1.seconds overwritten
11 int a = d1.seconds; // d1.seconds is garbage

```

Memory layout



union

C++ 98

“members” packed together at same memory location

```

1  union Duration {
2      int seconds;
3      short hours;
4      char days;
5  };
6  Duration d1, d2, d3;
7  d1.seconds = 259200;
8  d2.hours = 72;
9  d3.days = 3;
10 d1.days = 3; // d1.seconds overwritten
11 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

```

1  union Duration {
2      int seconds;
3      short hours;
4      char days;
5  };
6  Duration d1, d2, d3;
7  d1.seconds = 259200;
8  d2.hours = 72;
9  d3.days = 3;
10 d1.days = 3; // d1.seconds overwritten
11 int a = d1.seconds; // d1.seconds is garbage

```

Memory layout

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



union

“members” packed together at same memory location

```

1  union Duration {
2      int seconds;
3      short hours;
4      char days;
5  };
6  Duration d1, d2, d3;
7  d1.seconds = 259200;
8  d2.hours = 72;
9  d3.days = 3;
10 d1.days = 3; // d1.seconds overwritten
11 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

```

1  union Duration {
2      int seconds;
3      short hours;
4      char days;
5  };
6  Duration d1, d2, d3;
7  d1.seconds = 259200;
8  d2.hours = 72;
9  d3.days = 3;
10 d1.days = 3; // d1.seconds overwritten
11 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`



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

```
1 enum VehicleType {
2
3     BIKE, // 0
4     CAR,  // 1
5     BUS,  // 2
6 };
7 VehicleType t = CAR;
```

```
8 enum VehicleType
9     : int { // C++11
10     BIKE = 3,
11     CAR = 5,
12     BUS = 7,
13 };
14 VehicleType t2 = BUS;
```



Scoped enumeration, aka enum class

C++ 11

Same syntax as enum, with scope

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



Scoped enumeration, aka enum class

Same syntax as enum, with scope

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

Only advantages

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

```
3  enum VType { Bus, Car }; enum Color { Red, Blue };
4  VType t = Bus;
5  if (t == Red) { /* We do enter */ }
6  int a = 5 * Car; // Ok, a = 5
7
8  enum class VT { Bus, Car }; enum class Col { Red, Blue };
9  VT t = VT::Bus;
10 if (t == Col::Red) { /* Compiler error */ }
11 int a = t * 5; // Compiler error
```



More sensible example

C++ 98

```
1  enum class ShapeType {
2      Circle,
3      Rectangle
4  };
5
6  struct Rectangle {
7      float width;
8      float height;
9  };
```



More sensible example

C++ 98

```
1  enum class ShapeType {      10  struct Shape {
2      Circle,                11      ShapeType type;
3      Rectangle              12      union {
4  };                          13          float radius;
5                               14          Rectangle rect;
6  struct Rectangle {         15      };
7      float width;          16  };
8      float height;
9  };
```



More sensible example

C++ 98

```
1  enum class ShapeType {          10  struct Shape {
2      Circle,                    11      ShapeType type;
3      Rectangle                  12      union {
4  };                               13          float radius;
5                                  14          Rectangle rect;
6  struct Rectangle {             15      };
7      float width;               16  };
8      float height;
9  };
17 Shape s;                       20 Shape t;
18 s.type =                        21 t.type =
19     ShapeType::Circle;          22     Shapetype::Rectangle;
20 s.radius = 3.4;                 23 t.rect.width = 3;
21                                24 t.rect.height = 4;
```



typedef and using

C++ 98 / C++ 11

Used to create type aliases

C++ 98

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

C++ 11

```
4 using myint = std::uint64_t;
5 myint count = 17;
6 using position = float[3];
7
8 template <typename T> using myvec = std::vector<T>;
9 myvec<int> myintvec;
```



References

2 Language basics

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



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:

```
1 int i = 2;
2 int &ieref = i; // access to i
3 ieref = 3;     // i is now 3
4
5 // const reference to a member:
6 struct A { int x; int y; } a;
7 const int &x = a.x; // direct read access to A's x
8 x = 4;             // doesn't compile
9 a.x = 4;          // fine
```



Pointers vs References

C++ 98

Specificities of reference

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

Advantages of pointers

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



Pointers vs References

C++ 98

Specificities of reference

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

Advantages of pointers

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

Good practice: References

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



Functions

2 Language basics

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



Functions

```
1 // with return type
2 int square(int a) {
3     return a * a;
4 }
5
6 // multiple parameters
7 int mult(int a,
8         int b) {
9     return a * b;
10 }
11 // no return
12 void log(char* msg) {
13     std::cout << msg;
14 }
15
16 // no parameter
17 void hello() {
18     std::cout << "Hello World";
19 }
```



Functions

```

1  // with return type          11 // no return
2  int square(int a) {         12 void log(char* msg) {
3      return a * a;          13     std::cout << msg;
4  }                            14 }
5                                15
6  // multiple parameters     16 // no parameter
7  int mult(int a,           17 void hello() {
8      int b) {              18     std::cout << "Hello World";
9      return a * b;         19 }
10 }

```

Functions and references to returned values

```

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

```



Function default arguments

```
1 // must be the trailing
2 // argument
3 int add(int a,
4         int b = 2) {
5     return a + b;
6 }
7 // add(1) == 3
8 // add(3,4) == 7
9
11 // multiple default
12 // arguments are possible
13 int add(int a = 2,
14         int b = 2) {
15     return a + b;
16 }
17 // add() == 4
18 // add(3) == 5
```



Functions: parameters are passed by value

C++ 98

```
1  struct BigStruct {...};
2  BigStruct s;
3
4  // parameter by value
5  void printVal(BigStruct p) {
6      ...
7  }
8  printVal(s); // copy
9
10 // parameter by reference
11 void printRef(BigStruct &q) {
12     ...
13 }
14 printRef(s); // no copy
```

Memory layout

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



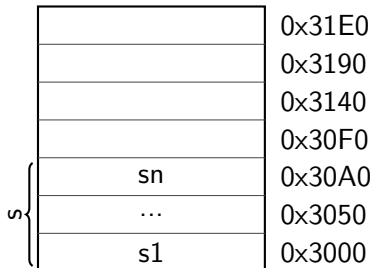
Functions: parameters are passed by value

```

1  struct BigStruct {...};
2  BigStruct s;
3
4  // parameter by value
5  void printVal(BigStruct p) {
6      ...
7  }
8  printVal(s); // copy
9
10 // parameter by reference
11 void printRef(BigStruct &q) {
12     ...
13 }
14 printRef(s); // no copy

```

Memory layout



Functions: parameters are passed by value

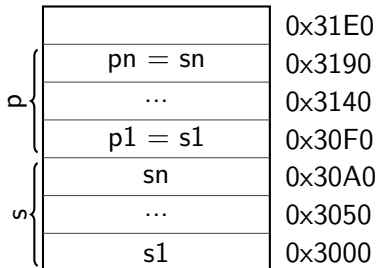
C++ 98

```

1  struct BigStruct {...};
2  BigStruct s;
3
4  // parameter by value
5  void printVal(BigStruct p) {
6      ...
7  }
8  printVal(s); // copy
9
10 // parameter by reference
11 void printRef(BigStruct &q) {
12     ...
13 }
14 printRef(s); // no copy

```

Memory layout



Functions: parameters are passed by value

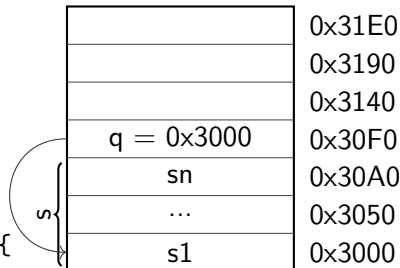
C++ 98

```

1  struct BigStruct {...};
2  BigStruct s;
3
4  // parameter by value
5  void printVal(BigStruct p) {
6      ...
7  }
8  printVal(s); // copy
9
10 // parameter by reference
11 void printRef(BigStruct &q) {
12     ...
13 }
14 printRef(s); // no copy

```

Memory layout

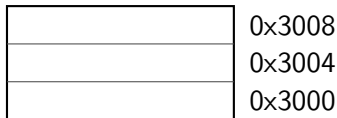


Functions: pass by value or reference?

C++ 98

```
1 struct SmallStruct {int a;};
2 SmallStruct s = {1};
3
4 void changeVal(SmallStruct p) {
5     p.a = 2;
6 }
7 changeVal(s);
8 // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11     q.a = 2;
12 }
13 changeRef(s);
14 // s.a == 2
```

Memory layout



Functions: pass by value or reference?

C++ 98

```

1  struct SmallStruct {int a;};
2  SmallStruct s = {1};
3
4  void changeVal(SmallStruct p) {
5      p.a = 2;
6  }
7  changeVal(s);
8  // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11     q.a = 2;
12 }
13 changeRef(s);
14 // s.a == 2

```

Memory layout

	0x3008
	0x3004
s.a = 1	0x3000



Functions: pass by value or reference?

C++ 98

```

1  struct SmallStruct {int a;};
2  SmallStruct s = {1};
3
4  void changeVal(SmallStruct p) {
5      p.a = 2;
6  }
7  changeVal(s);
8  // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11     q.a = 2;
12 }
13 changeRef(s);
14 // s.a == 2

```

Memory layout

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



Functions: pass by value or reference?

C++ 98

```

1  struct SmallStruct {int a;};
2  SmallStruct s = {1};
3
4  void changeVal(SmallStruct p) {
5      p.a = 2;
6  }
7  changeVal(s);
8  // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11     q.a = 2;
12 }
13 changeRef(s);
14 // s.a == 2

```

Memory layout

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



Functions: pass by value or reference?

C++ 98

```

1  struct SmallStruct {int a;};
2  SmallStruct s = {1};
3
4  void changeVal(SmallStruct p) {
5      p.a = 2;
6  }
7  changeVal(s);
8  // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11     q.a = 2;
12 }
13 changeRef(s);
14 // s.a == 2

```

Memory layout

	0x3008
	0x3004
s.a = 1	0x3000

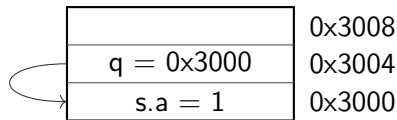


Functions: pass by value or reference?

C++ 98

```
1 struct SmallStruct {int a;};
2 SmallStruct s = {1};
3
4 void changeVal(SmallStruct p) {
5     p.a = 2;
6 }
7 changeVal(s);
8 // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11     q.a = 2;
12 }
13 changeRef(s);
14 // s.a == 2
```

Memory layout



Functions: pass by value or reference?

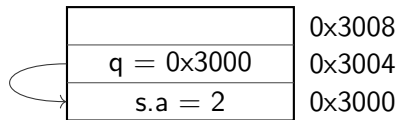
C++ 98

```

1  struct SmallStruct {int a;};
2  SmallStruct s = {1};
3
4  void changeVal(SmallStruct p) {
5      p.a = 2;
6  }
7  changeVal(s);
8  // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11     q.a = 2;
12 }
13 changeRef(s);
14 // s.a == 2

```

Memory layout



Functions: pass by value or reference?

C++ 98

```
1 struct SmallStruct {int a;};
2 SmallStruct s = {1};
3
4 void changeVal(SmallStruct p) {
5     p.a = 2;
6 }
7 changeVal(s);
8 // s.a == 1
9
10 void changeRef(SmallStruct &q) {
11     q.a = 2;
12 }
13 changeRef(s);
14 // 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

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



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

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



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

```
1  /// Count number of dilepton events in data.  
2  /// \param d Dataset to search.  
3  unsigned int countDileptons(Data &d) {  
4      selectEventsWithMuons(d);  
5      selectEventsWithElectrons(d);  
6      return d.size();  
7  }
```



Functions: good practices

C++ 98

Example: don't! Everything in one long function

```
1  unsigned int runJob() { 15      if (...) {
2      // Step 1: data      16          data.erase(...);
3      Data data;          17      }
4      data.resize(123456); 18  }
5      data.fill(...);     19
6                          20      // Step 4: dileptons
7      // Step 2: muons    21      int counter = 0;
8      for (...) {        22      for (...) {
9          if (...) {    23          if (...) {
10             data.erase(...); 24             counter++;
11         }            25         }
12     }                26     }
13     // Step 3: electrons 27
14     for (...) {        28     return counter;
15     }                  29 }
```



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)

C++ 98

Binary and Assignment Operators

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



Operators(1)

C++ 98

Binary and Assignment Operators

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

Increment / Decrement Operators

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



Operators(1)

C++ 98

Binary and Assignment Operators

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

Increment / Decrement Operators

Use wisely

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



Operators(2)

C++ 98

Bitwise and Assignment Operators

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



Operators(2)

C++ 98

Bitwise and Assignment Operators

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

Logical Operators

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



Operators(3)

C++ 98

Comparison Operators

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



Operators(3)

C++ 98

Comparison Operators

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

Precedences

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

Details can be found on [cppreference](#)



Operators(3)

C++ 98

Comparison Operators

```
1  bool a = (3 == 3); // true
2  bool b = (3 != 3); // false
3  bool c = (4 < 4); // false
4  bool d = (4 <= 4); // true
5  bool e = (4 > 4); // false
6  bool f = (4 >= 4); // true
7  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

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

Precedences

Avoid - use parentheses

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

Details can be found on [cppreference](#)



Control structures

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



Control structures: if

if syntax

```
1  if (condition1) {  
2      Statement1; Statement2;  
3  } else if (condition2)  
4      OnlyOneStatement;  
5  else {  
6      Statement3;  
7      Statement4;  
8  }
```

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



Control structures: if

C++ 98

Practical example

```
1  int collatz(int a) {
2      if (a <= 0) {
3          std::cout << "not supported\n";
4          return 0;
5      } else if (a == 1) {
6          return 1;
7      } else if (a%2 == 0) {
8          return collatz(a/2);
9      } else {
10         return collatz(3*a+1);
11     }
12 }
```



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

```
1  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

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

Do not abuse it

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

- Explicit **ifs** are generally easier to read
- Use the ternary operator with short conditions and expressions
- Avoid nesting



Control structures: switch

C++ 98

Syntax

```
1  switch(identifier) {
2      case c1 : statements1; break;
3      case c2 : statements2; break;
4      case c3 : statements3; break;
5      ...
6      default : statementsn; break;
7  }
```

- 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

```
1  switch(identifier) {
2      case c1 : statements1; break;
3      case c2 : statements2; break;
4      case c3 : statements3; break;
5      ...
6      default : statementsn; break;
7  }
```

- 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

```
1  enum class Lang { French, German, English, Other };
2  Lang language = ...;
3  switch (language) {
4      case Lang::French:
5          std::cout << "Bonjour";
6          break;
7      case Lang::German:
8          std::cout << "Guten Tag";
9          break;
10     case Lang::English:
11         std::cout << "Good morning";
12         break;
13     default:
14         std::cout << "I do not speak your language";
15 }
```



[[fallthrough]] attribute

C++ 17

New compiler warning

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

C++ 17

```
1  switch (c) {  
2      case 'a':  
3          f();    // Warning emitted  
4      case 'b': // Warning probably suppressed  
5      case 'c':  
6          g();  
7          [[fallthrough]]; // Warning suppressed  
8      case 'd':  
9          h();  
10 }
```



Init-statements for if and switch

C++ 17

Purpose

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

C++ 17

```
1  if (Value val = GetValue(); condition(val)) {  
2    f(val); // ok  
3  } else  
4    g(val); // ok  
5  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

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

C++ 98

Don't confuse with a variable declaration as condition:

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



Control structures: for loop

C++ 98

for loop syntax

```
1  for(initializations; condition; increments) {  
2      statements;  
3  }
```

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



Control structures: for loop

C++ 98

for loop syntax

- ```
1 for(initializations; condition; increments) {
2 statements;
3 }
```
- Multiple initializations / increments are comma separated
  - Initializations can contain declarations
  - Braces are optional if loop body is a single statement

## Practical example

```
4 for(int i = 0, j = 0 ; i < 10 ; i++, j = i*i) {
5 std::cout << i << "^2 is " << j << '\n';
6 }
```





# Control structures: for loop

C++ 98

## for loop syntax

```
1 for(initializations; condition; increments) {
2 statements;
3 }
```

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

## Practical example

```
4 for(int i = 0, j = 0 ; i < 10 ; i++, j = i*i) {
5 std::cout << i << "^2 is " << j << '\n';
6 }
```

## 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

```
1 for (type iteration_variable : range) {
2 // body using iteration_variable
3 }
```

## Example code

```
4 int v[4] = {1,2,3,4};
5 int sum = 0;
6 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

```
1 std::array data = {"hello", ",", "world"};
2 std::size_t i = 0;
3 for (auto& d : data) {
4 std::cout << i++ << ' ' << d << '\n';
5 }
```

## C++ 20

```
6 for (std::size_t i = 0; auto const & d : data) {
7 std::cout << i++ << ' ' << d << '\n';
8 }
```



# Control structures: while loop

C++ 98

## while loop syntax

```
1 while(condition) {
2 statements;
3 }
4
5 do {
6 statements;
7 } while(condition);
```

- Braces are optional if the body is a single statement



## Control structures: while loop

C++ 98

## while loop syntax

```
1 while(condition) {
2 statements;
3 }
4
5 do {
6 statements;
7 } while(condition);
```

- Braces are optional if the body is a single statement

## Bad example

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

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!

## Bad example

```
1 while (1) {
2 if (n == 1) break;
3 if (0 == n%2) {
4 std::cout << n << '\n';
5 n /= 2;
6 continue;
7 }
8 n = 3 * n + 1;
9 }
```



## 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

```
void printHello();
```

## Usage: myfile.cpp

```
1 #include "hello.hpp"
2 int main() {
3 printHello();
4 }
```



## Preprocessor

C++ 98

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



## Preprocessor

```
1 // file inclusion
2 #include "hello.hpp"
3 // macro constants and function-style macros
4 #define MY_GOLDEN_NUMBER 1746
5 #define CHECK_GOLDEN(x) if ((x) != MY_GOLDEN_NUMBER) \
6 std::cerr << #x " was not the golden number\n";
7 // compile time or platform specific configuration
8 #if defined(USE64BITS) || defined(__GNUG__)
9 using myint = std::uint64_t;
10 #elif
11 using myint = std::uint32_t;
12 #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

```
1 #ifndef MY_HEADER_INCLUDED
2 #define MY_HEADER_INCLUDED
3 ... // header file content
4 #endif
```

## Pragma once (non-standard)

```
1 #pragma once
2 ... // header file content
```



## Good practice: Headers and source files

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



# Auto keyword

## 2 Language basics

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



## 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

```
1 std::vector<int> v;
2 float a = v[3]; // conversion intended?
3 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

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

## Practical usage

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



### 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

## 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



# Inline keyword

## 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

```
1 inline int mult(int a, int b) {
2 return a * b;
3 }
```

## 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 Adv

## 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

```
1 // global.h
2 inline int count = 0;
3 inline const std::string filename = "output.txt";
4 // a.cpp
5 #include "global.h"
6 int f() { return count; }
7 // b.cpp
8 #include "global.h"
9 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**



## 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

```
1 #include <cassert>
2 double f(double a) {
3 // [...] do stuff with a
4 // [...] that should leave it positive
5 assert(a > 0.);
6 return std::sqrt(a);
7 }
```





## 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 - `gdb`

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



### 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
```

```
1 double f(double a) {
2 assert(a > 0.); // no effect
3 return std::sqrt(a);
4 }
```



## Static Assert

## 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`

```

1 double f(UserType a) {
2 static_assert(
3 std::is_floating_point_v<UserType>, // C++17
4 "This function expects floating-point types.");
5 return std::sqrt(a);
6 }

```

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_v<UserType>,
 | ~~~~~^~~~~~

```

# Object orientation (OO)

- 1 History and goals
  - 2 Language basics
  - 3 Object orientation (OO)**
    - Objects and Classes
    - Inheritance
    - Constructors/destructors
    - Static members
    - Allocating objects
    - Advanced OO
    - Type casting
  - 4 Core modern C++
  - 5 Expert C++
  - 6 Useful tools
  - 7 Concurrency
  - 8 C++ and python
- Operator overloading
  - Function objects
  - Name Lookups



# 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

## 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

```
1 struct MyFirstClass {
2 int a;
3 void squareA() {
4 a *= a;
5 }
6 int sum(int b) {
7 return a + b;
8 }
9 };
10
11 MyFirstClass myObj;
12 myObj.a = 2;
13
14 // let's square a
15 myObj.squareA();
```

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



## Separating the interface

## Header: MyClass.hpp

```
1 #pragma once
2 struct MyClass {
3 int a;
4 void squareA();
5 };
```

## User 1: main.cpp

```
1 #include "MyClass.hpp"
2 int main() {
3 MyClass mc;
4 ...
5 }
```

## Implementation: MyClass.cpp

```
1 #include "MyClass.hpp"
2 void MyClass::squareA() {
3 a *= a;
4 }
```

## User 2: fun.cpp

```
1 #include "MyClass.hpp"
2 void f(MyClass& mc) {
3 mc.squareA();
4 }
```





## 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

```
1 #include "MyFirstClass.hpp"
2
3 void MyFirstClass::squareA() {
4 a *= a;
5 }
6 int MyFirstClass::sum(int b) {
7 return a + b;
8 }
```



# Method overloading

C++ 98

## The rules in C++

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

```
1 struct MyFirstClass {
2 int a;
3 int sum(int b);
4 int sum(int b, int c);
5 };
6
7 int MyFirstClass::sum(int b) { return a + b; }
8
9 int MyFirstClass::sum(int b, int c) {
10 return a + b + c;
11 }
```



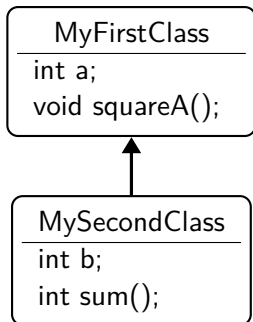
# Inheritance

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



## First inheritance

```
1 struct MyFirstClass {
2 int a;
3 void squareA() { a *= a; }
4 };
5 struct MySecondClass :
6 MyFirstClass {
7 int b;
8 int sum() { return a + b; }
9 };
10
11 MySecondClass myObj2;
12 myObj2.a = 2;
13 myObj2.b = 5;
14 myObj2.squareA();
15 int i = myObj2.sum(); // i = 9
```



## First inheritance

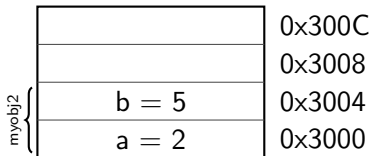
C++ 98

```

1 struct MyFirstClass {
2 int a;
3 void squareA() { a *= a; }
4 };
5 struct MySecondClass :
6 MyFirstClass {
7 int b;
8 int sum() { return a + b; }
9 };
10
11 MySecondClass myObj2;
12 myObj2.a = 2;
13 myObj2.b = 5;
14 myObj2.squareA();
15 int i = myObj2.sum(); // i = 9

```

## Memory layout



# 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`
- The default for struct is `public`

```
1 class MyFirstClass {
2 public:
3 void setA(int x);
4 int getA();
5 void squareA();
6 private:
7 int a;
8 };
9 MyFirstClass obj;
10 obj.a = 5; // error !
11 obj.setA(5); // ok
12 obj.squareA();
13 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`
- The default for struct is `public`

```
1 class MyFirstClass {
2 public:
3 void setA(int x);
4 int getA();
5 void squareA();
6 private:
7 int a;
8 };
9 MyFirstClass obj;
10 obj.a = 5; // error !
11 obj.setA(5); // ok
12 obj.squareA();
13 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

```
1 class MyFirstClass {
2 public:
3 void setA(int a);
4 int getA();
5 void squareA();
6 protected:
7 int a;
8 };

13 class MySecondClass :
14 public MyFirstClass {
15 public:
16 int sum() {
17 return a + b;
18 }
19 private:
20 int b;
21 };
```





# Managing inheritance privacy

C++ 98

## Inheritance can be public, protected or private

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

`public` privacy of inherited members remains unchanged

`protected` inherited public members are seen as protected

`private` all inherited members are seen as private

this is the default for classes if nothing is specified



# Managing inheritance privacy

C++ 98

## Inheritance can be public, protected or private

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

`public` privacy of inherited members remains unchanged

`protected` inherited public members are seen as protected

`private` all inherited members are seen as private

this is the default for classes if nothing is specified

## Net result for external code

- only public members of public inheritance are accessible

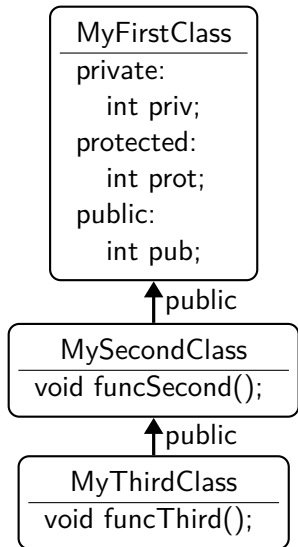
## Net result for code in derived classes

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



## Managing inheritance privacy - public

C++ 98



```

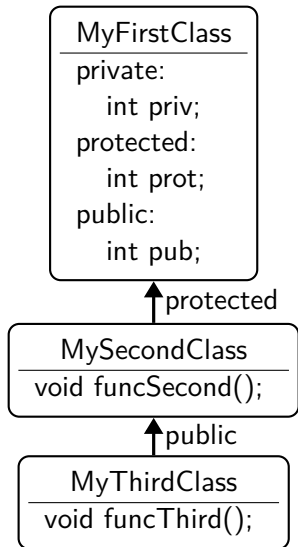
1 void funcSecond() {
2 int a = priv; // Error
3 int b = prot; // OK
4 int c = pub; // OK
5 }
6 void funcThird() {
7 int a = priv; // Error
8 int b = prot; // OK
9 int c = pub; // OK
10 }
11 void extFunc(MyThirdClass t) {
12 int a = t.priv; // Error
13 int b = t.prot; // Error
14 int c = t.pub; // OK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

```



## Managing inheritance privacy - protected

C++ 98



```

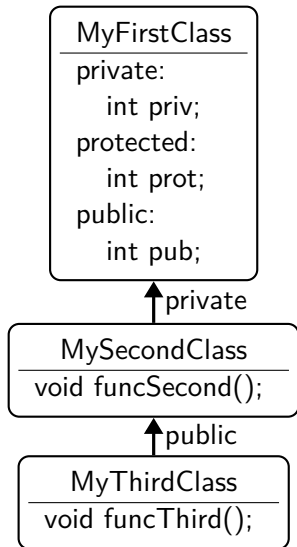
1 void funcSecond() {
2 int a = priv; // Error
3 int b = prot; // OK
4 int c = pub; // OK
5 }
6 void funcThird() {
7 int a = priv; // Error
8 int b = prot; // OK
9 int c = pub; // OK
10 }
11 void extFunc(MyThirdClass t) {
12 int a = t.priv; // Error
13 int b = t.prot; // Error
14 int c = t.pub; // Error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

```



## Managing inheritance privacy - private

C++ 98



```

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

```



## Idea

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

## Practically

```
1 struct Base final {
2 ...
3 };
4 struct Derived : Base { // compiler error
5 ...
6 };
```



# Constructors/destructors

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



# Class constructors and destructors

C++ 98

## 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 ~

```
1 class C {
2 public:
3 C();
4 C(int a);
5 ~C();
6 ...
7 protected:
8 int a;
9 };
10 // note: special notation for
11 // initialization of members
12 C::C() : a(0) {}
13
14 C::C(int a) : a(a) {}
15
16 C::~~C() {}
```





## Class constructors and destructors

```
1 class Vector {
2 public:
3 Vector(int n);
4 ~Vector();
5 void setN(int n, int value);
6 int getN(int n);
7 private:
8 int len;
9 int* data;
10 };
11 Vector::Vector(int n) : len(n) {
12 data = new int[n];
13 }
14 Vector::~~Vector() {
15 delete[] data;
16 }
```



## Constructors and inheritance

```
1 struct First {
2 int a;
3 First() {} // leaves a uninitialized
4 First(int a) : a(a) {}
5 };
6 struct Second : First {
7 int b;
8 Second();
9 Second(int b);
10 Second(int a, int b);
11 };
12 Second::Second() : First(), b(0) {}
13 Second::Second(int b) : b(b) {} // First() implicitly
14 Second::Second(int a, int b) : First(a), b(b) {}
```



## 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

```
1 struct C {
2 C();
3 C(const C &other);
4 };
```



## 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

```
1 struct C {
2 C();
3 C(const C &other);
4 };
```

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

```
1 class Vector {
2 public:
3 Vector(int n);
4 Vector(const Vector &other);
5 ~Vector();
6 private:
7 int len; int* data;
8 };
9 Vector::Vector(int n) : len(n) {
10 data = new int[n];
11 }
12 Vector::Vector(const Vector &other) : len(other.len) {
13 data = new int[len];
14 std::copy(other.data, other.data + len, data);
15 }
16 Vector::~~Vector() { delete[] data; }
```



## Explicit unary constructor

## Concept

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

## Example - godbolt

```
1 void print(const Vector & v) {
2 std::cout << "printing v elements...\n";
3 }
4
5 int main {
6 // calls Vector::Vector(int n) to construct a Vector
7 // then calls print with that Vector
8 print(3);
9 };
```



# Explicit unary constructor

## Concept

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

```
1 class Vector {
2 public:
3 explicit Vector(int n);
4 Vector(const Vector &other);
5 ~Vector();
6 ...
7 };
```





# 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

```
1 Class() = default; // provide default if possible
2 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

```
1 struct Delegate {
2 int m_i;
3 Delegate(int i) : m_i(i) {
4 ... complex initialization ...
5 }
6 Delegate() : Delegate(42) {}
7 };
```



# Constructor inheritance

C++ 11

## Idea

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

## Practically

```
1 struct Base {
2 Base(int a); // ctor 1
3 };
4 struct Derived : Base {
5 using Base::Base;
6 Derived(int a, int b); // ctor 2
7 };
8 Derived d{5}; // calls ctor 1
9 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

```
1 struct Base {
2 int a{5}; // also possible: int a = 5;
3 Base() = default;
4 Base(int _a) : a(_a) {}
5 };
6 struct Derived : Base {
7 int b{6};
8 using Base::Base;
9 };
10 Derived d1; // a = 5, b = 6
11 Derived d2{7}; // a = 7, b = 6
```



## Calling constructors

After object declaration, arguments within {}

```
1 struct A {
2 int a;
3 float b;
4 A();
5 A(int);
6 A(int, int);
7 };
8
9 A a{1,2}; // A::A(int, int)
10 A a{1}; // A::A(int)
11 A a{}; // A::A()
12 A a; // A::A()
13 A a = {1,2}; // A::A(int, int)
```



## Calling constructors the old way

C++ 98

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

```
1 struct A {
2 int a;
3 float b;
4 A();
5 A(int);
6 A(int, int);
7 };
8
9 A a(1,2); // A::A(int, int)
10 A a(1); // A::A(int)
11 A a(); // declaration of a function!
12 A a; // A::A()
13 A a = (1,2); // A::A(int), comma operator!
14 A a = {1,2}; // not allowed
```



## Constructing arrays and vectors

List of items given within {}

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



# Static members

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





# 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

```
1 class Text {
2 public:
3 static std::string upper(std::string);
4 private:
5 static int callsToUpper; // add `inline` in C++17
6 };
```



# Static members

## Concept

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

## Static.cpp

```
1 #include "Static.hpp"
2 int Text::callsToUpper = 0; // required before C++17
3
4 std::string Text::upper(std::string lower) {
5 callsToUpper++;
6 // convert lower to upper case
7 // return ...;
8 }
9 std::string uppers = Text::upper("my text");
10 // now Text::callsToUpper is 1
```



# Allocating objects

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



## 4 main areas

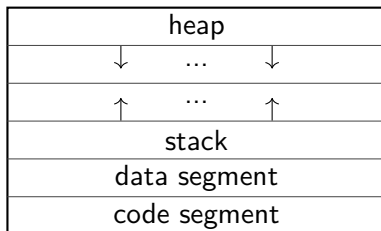
the **code segment** for the machine code of the executable

the **data segment** for global variables

the **heap** for dynamically allocated variables

the **stack** for parameters of functions and local variables

## Memory layout



## 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)

```
1 int f() {
2 MyFirstClass a{3}; // constructor called
3 ...
4 } // destructor called
5
6 int g() {
7 MyFirstClass a; // default constructor called
8 ...
9 } // destructor called
```



## Main characteristics

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



# Object allocation on the heap

C++ 98

## On the heap

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

```
1 int f() {
2 // default constructor called
3 MyFirstClass *a = new MyFirstClass;
4 delete a; // destructor is called
5 }
6 int g() {
7 // constructor called
8 MyFirstClass *a = new MyFirstClass{3};
9 } // 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

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

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



## How to know an object's address?

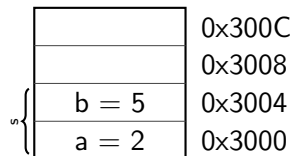
- Sometimes we need the address of the current object
- Or we need to pass our address / a reference to a different entity (for example to implement operators, see later)
- All class methods can use the keyword **this**
  - It returns the address of the current object
  - Its type is  $T^*$  in the methods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

```

1 struct S {
2 int a,b;
3 // these two are the same:
4 int getB() { return b; } // 5
5 int getB() { return this->b; } // 5
6 void testAddress() {
7 S* addr = this; // 0x3000
8 }
9 } s{2,5};

```

## Memory layout



this pointer

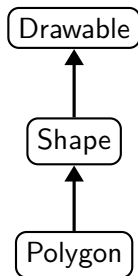


# Polymorphism

## the concept

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

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



# Polymorphism

## the concept

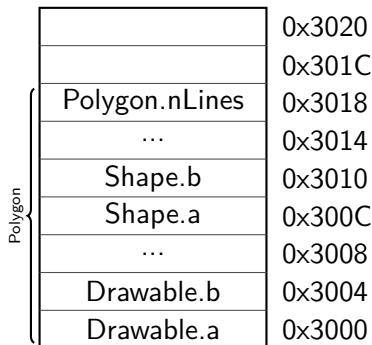
- objects actually have multiple types simultaneously
- and can be used as any of them

```

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

```

## Memory layout



## Polymorphism

## the concept

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

```

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

```

## Memory layout

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

Drawable {



# Polymorphism

## the concept

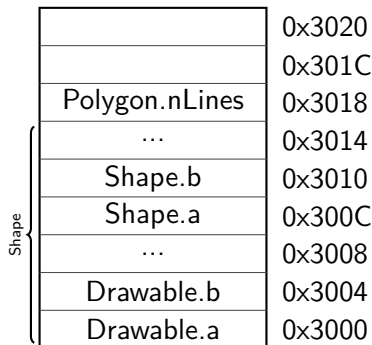
- objects actually have multiple types simultaneously
- and can be used as any of them

```

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

```

## Memory layout

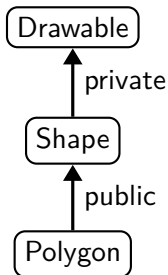


# Inheritance privacy and polymorphism

Only public base classes are visible to outside code

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

```
1 Polygon p;
2
3 int f(Drawable & d) {...}
4 f(p); // Not ok anymore
5
6 try {
7 throw p;
8 } catch (Shape & e) {
9 // ok, will be caught
10 }
```



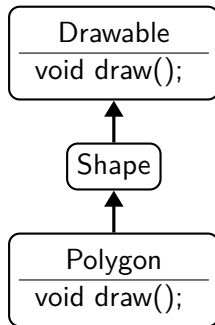


# Method overriding

## the idea

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

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



## 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

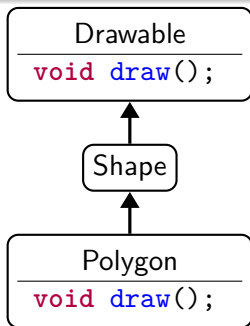


# Virtual methods

## 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

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

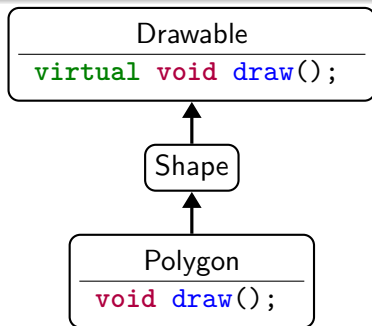


# Virtual methods

## 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

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

```
1 struct Base {
2 virtual void some_func(float);
3 };
4 struct Derived : Base {
5 void some_func(float) override;
6 };
```



# Why was `override` keyword introduced?

To detect the mistake in the following code :

Without `override` (C++ 98)

```
1 struct Base {
2 virtual void some_func(float);
3 };
4 struct Derived : Base {
5 void some_func(double); // oops !
6 };
```

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



## Idea

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

## Practically

```
1 struct Base {
2 virtual void some_func(float);
3 };
4 struct Intermediate : Base {
5 void some_func(float) final;
6 };
7 struct Derived : Intermediate {
8 void some_func(float) override; // error
9 };
```





## Concept

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



## Pure Virtual methods

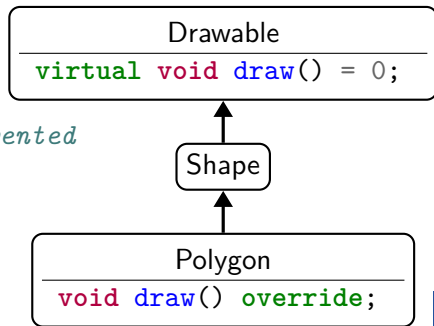
## Concept

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

```

1 // Error : abstract class
2 Shape s;
3
4 // ok, draw has been implemented
5 Polygon p;
6
7 // Shape type still usable
8 Shape & s = p;
9 s.draw();

```



# Polymorphism and destruction

## Owning base pointers

We sometimes need to maintain owning pointers to base classes:

```
1 struct Drawable {
2 virtual void draw() = 0;
3 };
4 Drawable* getImpl();
5
6 Drawable* p = getImpl();
7 p->draw();
8 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:

```
1 struct Drawable {
2 virtual void draw() = 0;
3 };
4 std::unique_ptr<Drawable> getImpl(); // better API
5
6 auto p = getImpl();
7 p->draw();
```

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



# Polymorphism and destruction

## Virtual destructors

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

```
1 struct Drawable {
2 virtual ~Drawable() = default;
3 virtual void draw() = 0;
4 };
5 Drawable* p = getImpl(); // returns derived obj.
6 p->draw();
7 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

## 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

```
1 struct Drawable {
2 virtual ~Drawable()
3 = default;
4 virtual void draw() = 0;
5 }
```

|                          |
|--------------------------|
| Drawable                 |
| virtual void draw() = 0; |



# Overriding overloaded methods

## Concept

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

```
1 struct BaseClass {
2 virtual int foo(std::string);
3 virtual int foo(int);
4 };
5 struct DerivedClass : BaseClass {
6 using BaseClass::foo;
7 int foo(std::string) override;
8 };
9 DerivedClass dc;
10 dc.foo(4); // error if no using
```



## 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

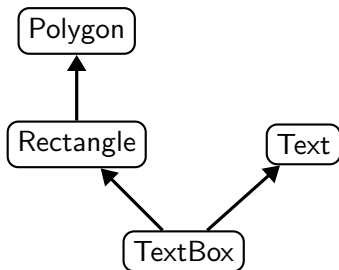




# Multiple Inheritance

## Concept

- one class can inherit from multiple parents



```
1 class TextBox :
2 public Rectangle, Text {
3 // inherits from both
4 // publicly from Rectangle
5 // privately from Text
6 }
```



# The diamond shape

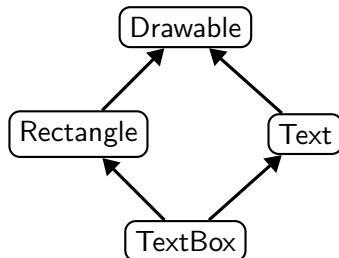
C++ 98

## Definition

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

## Problem

- are the members of the grand parent replicated?

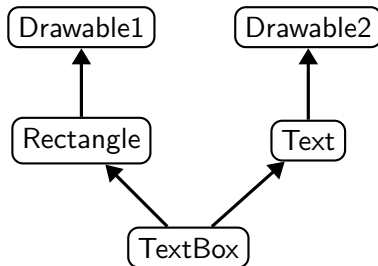
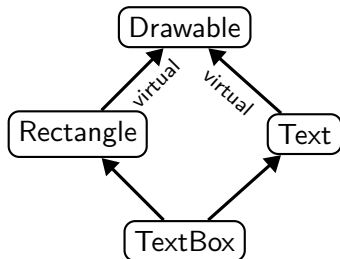


# Virtual inheritance

## Solution

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

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



## Good practice: Avoid multiple inheritance

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

## Absolutely avoid diamond-shape inheritance

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



### Exercise: Virtual inheritance

- go to `exercisecode/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

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



## 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

## Practically - godbolt

```
1 struct A{ virtual ~A()=default; } a;
2 struct B : A {} b;
3
4 A& c = static_cast<A&>(b); // OK. b is also an A
5 B& d = static_cast<B&>(a); // UB: a is not a B
6 B& e = static_cast<B&>(c); // OK. c is a B
7
8 B& f = dynamic_cast<B&>(c); // OK, c is a B
9 B& g = dynamic_cast<B&>(a); // throws std::bad_cast
10 B& foo(A& h) { return dynamic_cast<B&>(h); }
11 B& i = foo(c); // OK, c is a B
12
13 B* j = dynamic_cast<B*>(&a); // nullptr. a not a B.
14 if (j != nullptr) {
15 // Will not reach this
16 }
```





## 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

```
1 void func(A const & a) {
2 A& ra = a; // Error: not const
3 A& ra = const_cast<A&>(a); // Compiles. Bad design!
4 // Evil! Don't do this:
5 B* b = reinterpret_cast<B*>(&a);
6 B* b = (B*)&a;
7 }
```



# Operator overloading

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



## Operator overloading example

C++ 98

```
1 struct Complex {
2 float m_real, m_imaginary;
3 Complex(float real, float imaginary);
4 Complex operator+(const Complex& other) {
5 return Complex(m_real + other.m_real,
6 m_imaginary + other.m_imaginary);
7 }
8 };
9
10 Complex c1{2, 3}, c2{4, 5};
11 Complex c3 = c1 + c2; // (6, 8)
```



## Defining operators for a class

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

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

possible operators: + - \* / % ^ & | ~ ! = < >

+= -= \*= /= %= ^= &= |= << >> >>= <<=

== != <= >= <=> && || ++ -- , ->\* -> () []



# Why have non-member operators?

## Symmetry

```
1 struct Complex {
2 float m_real, m_imaginary;
3 Complex operator+(float other) {
4 return Complex(m_real + other, m_imaginary);
5 }
6 };
7 Complex c1{2.f, 3.f};
8 Complex c2 = c1 + 4.f; // ok
9 Complex c3 = 4.f + c1; // not ok !!
```



# Why have non-member operators?

## Symmetry

```
1 struct Complex {
2 float m_real, m_imaginary;
3 Complex operator+(float other) {
4 return Complex(m_real + other, m_imaginary);
5 }
6 };
7 Complex c1{2.f, 3.f};
8 Complex c2 = c1 + 4.f; // ok
9 Complex c3 = 4.f + c1; // not ok !!
10 Complex operator+(float a, const Complex& obj) {
11 return Complex(a + obj.m_real, obj.m_imaginary);
12 }
```



## Other reason to have non-member operators?

C++ 98

## Extending existing classes

```
1 struct Complex {
2 float m_real, m_imaginary;
3 Complex(float real, float imaginary);
4 };
5
6 std::ostream& operator<<(std::ostream& os,
7 const Complex& obj) {
8 os << "(" << obj.m_real << ", "
9 << obj.m_imaginary << ")";
10 return os;
11 }
12 Complex c1{2.f, 3.f};
13 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

```
1 class Complex {
2 float m_r, m_i;
3 friend Complex operator+(Complex const & a, Complex const & b);
4 public:
5 Complex (float r, float i) : m_r(r), m_i(i) {}
6 };
7 Complex operator+(Complex const & a, Complex const & b) {
8 return Complex{ a.m_r+b.m_r, a.m_i+b.m_i };
9 }
```





## Exercise: Operators

Write a simple class representing a fraction and pass all tests

- go to `exercises/operators`
- look at `operators.cpp`
- inspect `main` and complete the implementation of `class Fraction` step by step
- you can comment out parts of `main` to test in between



# Function objects

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



# 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

```
1 struct Adder {
2 int m_increment;
3 Adder(int increment) : m_increment(increment) {}
4 int operator()(int a) { return a + m_increment; }
5 };
6 Adder inc1{1}, inc10{10};
7 int i = 3;
8 int j = inc1(i); // 4
9 int k = inc10(i); // 13
10 int l = Adder{25}(i); // 28
```



## Function objects

C++ 20

## Function objects as function arguments - godbolt

```
1 int count_if(const auto& range, auto predicate) {
2 int count = 0; // ↑ template (later)
3 for (const auto& e : range)
4 if (predicate(e)) count++;
5 return count;
6 }
7 struct IsBetween {
8 int lower, upper;
9 bool operator()(int value) const {
10 return lower < value && value < upper;
11 }
12 };
13 int arr[]{1, 2, 3, 4, 5, 6, 7};
14 std::cout << count_if(arr, IsBetween{2, 6}); // 3
15 // 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



## Example code

```
1 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
  - in parallel Argument Dependent Lookup (ADL) may happen



# Unqualified name lookup and ADL

## 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...

## Argument Dependent Lookup (simplified)

Only for call expression

- e.g. `f(a, b)` or `a + b`

The compiler also examines arguments one by one and 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 :

```
1 namespace MyNS {
2 struct A {
3 T func(...);
4 };
5 }
```

Prefer :

```
6 namespace MyNS {
7 struct A { ... };
8 T func(const A&, ..);
9 }
```

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





## Customization points and using

Don't write :

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

Prefer :

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

Advantages :

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

```
6 namespace N {
7 class A { ... };
8 // optimized swap for A
9 void swap(A&, A&);
10 }
```



## 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&lt;&lt; as hidden friend

```
1 class Complex {
2 float m_real, m_imag;
3 public:
4 Complex(float, float) { ... }
5 friend
6 std::ostream & operator<<(std::ostream & os,
7 Complex const & c) {
8 return os << c.m_real << " + " << c.m_imag << "i";
9 }
10 };
11 std::cout << Complex{2.f, 3.f}; // Prints 2 + 3i
```



# Hidden Friends and ADL

## 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 counterexamples

```
1 std::ostream & operator<< // out-of-class definition
2 (std::ostream & os, Complex const & c) { ... }
3 struct Fraction {
4 int num, denom;
5 operator Complex() const { ... } // conversion op
6 };
7
8 std::cout << Fraction{2,4}; // Prints 0.5 + 0i
9 //calls operator<<(std::cout, Fraction{2,4}.operator Complex());
```



# Hidden Friends and ADL

## 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 counterexamples

```
1 std::ostream & operator<< // out-of-class definition
2 (std::ostream & os, Complex const & c) { ... }
3 struct Fraction {
4 int num, denom;
5 };
6
7 Complex::Complex(Fraction f); // converting ctor
8
9 std::cout << Fraction{2,4}; // Prints 0.5 + 0i
10 //calls operator<<(std::cout, Complex(Fraction{2, 3}));
```



## 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<sup>++</sup>

- 1 History and goals
- 2 Language basics
- 3 Object orientation (OO)
- 4 Core modern C<sup>++</sup>**
  - Constness
  - Constant Expressions
  - Exceptions
  - Move semantics
  - Copy elision
  - Templates
- 5 Lambdas
  - Lambdas
  - The STL
  - Standard library
  - Ranges
  - RAI and smart pointers
  - Initialization
- 6 Expert C<sup>++</sup>
- 7 Useful tools
- 8 Concurrency
- 9 C<sup>++</sup> and python



# Constness

- 4 Core modern C++
  - Constness
  - Constant Expressions
  - Exceptions
  - Move semantics
  - Copy elision
  - Templates
  - Lambdas
  - The STL
  - Standard library
  - Ranges
  - RAI and smart pointers
  - Initialization



## 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

```
1 int const i = 6;
2 const int i = 6; // equivalent
3
4 // error: i is constant
5 i = 5;
6
7 auto const j = i; // works with auto
```





## Constness and pointers

```
1 int a = 1, b = 2;
2
3 int const *i = &a; // pointer to const int
4 *i = 5; // error, int is const
5 i = &b; // ok, pointer is not const
6
7 int * const j = &a; // const pointer to int
8 *j = 5; // ok, value can be changed
9 j = &b; // error, pointer is const
10
11 int const * const k = &a; // const pointer to const int
12 *k = 5; // error, value is const
13 k = &b; // error, pointer is const
14
15 int const &l = a; // reference to const int
16 l = b; // error, reference is const
17
18 int const & const l = a; // compile error
```



## Member function constness

## 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

```

1 struct Example {
2 void foo() const {
3 // type of 'this' is 'Example const*'
4 data = 0; // Error: member function is const
5 }
6 void foo() { // ok, overload
7 data = 1; // ok, 'this' is 'Example*'
8 }
9 int data;
10 };
11 Example const e1; e1.foo(); // calls const foo
12 Example e2; e2.foo(); // calls non-const foo

```



## Member function constness

## Constness is part of the type

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

```
1 void change(int & a);
2 void read(int const & a);
3
4 int a = 0;
5 int const b = 0;
6
7 change(a); // ok
8 change(b); // error
9 read(a); // ok
10 read(b); // ok
```



## 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
  - Standard library
  - Ranges
  - RAI and smart pointers
  - Initialization



# Generalized Constant Expressions

## Reason of being

- use functions to compute constant expressions at compile time



## Generalized Constant Expressions

## Reason of being

- use functions to compute constant expressions at compile time

## Example

```
1 constexpr int f(int x) {
2 if (x > 1) return x * f(x - 1);
3 return 1;
4 }
5 constexpr int a = f(5); // computed at compile-time
6 int b = f(5); // maybe computed at compile-time
7 int n = ...; // runtime value
8 int c = f(n); // computed at runtime
```



## Generalized Constant Expressions(2)

## 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 Adv

## 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

```
1 constexpr float toSI(float v, char unit) {
2 switch (unit) {
3 case 'k': return 1000.0f*v;
4 case 'm': return 0.001f*v;
5 case 'y': return 0.9144f*v;
6 case 'i': return 0.0254f*v;
7 ...
8 default: return v;
9 }
10 }
11 constexpr float fromSI(float v, char unit) {
12 switch (unit) {
13 case 'k': return 0.001f*v;
14 case 'y': return 1.093f*v;
15 ...
16 }
17 }
```



## Real life example(2)

```
1 class DimLength {
2 float m_value;
3 public:
4 constexpr DimLength(float v, char unit):
5 m_value(toSI(v, unit)) {
6 }
7 constexpr float get(char unit) const {
8 return fromSI(m_value, unit);
9 }
10 };
11 constexpr DimLength km(1, 'k');
12 constexpr float km_y = km.get('y');
13 constexpr float km_i = km.get('i');
14 static_assert(km_y == 1093, "expected km == 1093 yards!");
```



## compile-time branches

*if constexpr*

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

## Example code

```
1 template <typename T>
2 auto remove_ptr(T t) {
3 if constexpr (std::is_pointer_v<T>) {
4 return *t;
5 } else {
6 return t;
7 }
8 }
9 int i = ...; int *j = ...;
10 int r = remove_ptr(i); // equivalent to i
11 int q = remove_ptr(j); // equivalent to *j
```



# Immediate functions

## Motivation

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

## Example

```
1 constexpr int f(int x) {
2 if (x > 1) return x * f(x - 1);
3 return 1;
4 }
5 constexpr int a = f(5); // computed at compile-time
6 int b = f(5); // computed at compile-time
7 int n = ...; // runtime value
8 int c = f(n); // compilation error
```



## 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

```
1 constexpr int f(int x) {
2 if (x > 1) return x * f(x - 1);
3 return 1;
4 }
5 constexpr int a = f(5); // CT init, not modifiable
6 int b = f(5); // maybe CT init, modifiable
7 constinit int c = f(5); // CT init, modifiable
```



# Exceptions

- 4 Core modern C++
  - Constness
  - Constant Expressions
  - **Exceptions**
  - Move semantics
  - Copy elision
  - Templates
  - Lambdas
  - The STL
  - Standard library
  - Ranges
  - RAI and smart pointers
  - Initialization



## 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

```
1 try { 7 void process_data(file &f) {
2 process_data(f); 8 ...
3 } catch (const 9 if (i >= buffer.size())
4 std::out_of_range& e) { 10 throw std::out_of_range{
5 std::cerr << e.what(); 11 "buf overflow"};
6 } 12 }
```





## 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

```

1 #include <stdexcept>
2 void process_data(file& f) {
3 if (!f.open())
4 throw std::invalid_argument{"stream is not open"};
5 auto header = read_line(f); // may throw an IO error
6 if (!header.starts_with("BEGIN"))
7 throw std::runtime_error{"invalid file content"};
8 std::string body(f.size()); // may throw std::bad_alloc
9 ...
10 }
```



## 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
- ...



## 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

```
1 try {
2 process_data(f);
3 } catch (const std::invalid_argument& e) {
4 bad_files.push_back(f);
5 } catch (const std::exception& e) {
6 std::cerr << "Failed to process file: " << e.what();
7 }
```

## Good practice: Catching exceptions

- Catch exceptions by const reference



## 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

```
1 try {
2 process_data(f);
3 } catch (const std::bad_alloc& e) {
4 std::cerr << "Insufficient memory for " << f.name();
5 throw; // rethrow
6 }
```



## Catching everything

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

```
1
2 try {
3 callUnknownFramework();
4 } catch(const std::exception& e) {
5 // catches std::exception and all derived types
6 std::cerr << "Exception: " << e.what() << std::endl;
7 } catch(...) {
8 // catches everything else
9 std::cerr << "Unknown exception type" << std::endl;
10 }
```



## 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

```
1 class C { ... };
2 void f() {
3 C c1;
4 throw exception{};
5 // start unwinding
6 C c2; // not run
7 }
8 void g() {
9 C c3; f();
10 }
11 int main() {
12 try {
13 C c4;
14 g();
15 cout << "done"; // not run
16 } catch(const exception&) {
17 // c1, c3 and c4 have been
18 // destructed
19 }
20 }
```



## 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](#)



## 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

```

1 try {
2 for (File const &f : files) {
3 try {
4 process_file(f);
5 }
6 catch (bad_file const & e) {
7 ... // loop continues
8 }
9 }
10 } catch (bad_db const & e) {
11 ... // loop aborted
12 }

1 void process_file(File const & file) {
2 ...
3 if (handle = open_file(file))
4 throw bad_file(file.status());
5 while (!handle) {
6 line = read_line(handle);
7 database.insert(line); // can throw
8 // bad_db
9 }
10 }

```





# Exceptions

C++ 98

## Cost

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

## Avoid

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

## Prefer

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



## noexcept

- 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

- 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

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

```
3 template <typename Function>
4 void g(Function f) noexcept(noexcept(f())) {
5 ...
6 f();
7 }
```



## 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

- 4 Core modern C++
  - Constness
  - Constant Expressions
  - Exceptions
  - **Move semantics**
  - Copy elision
  - Templates
  - Lambdas
  - The STL
  - Standard library
  - Ranges
  - RAI and smart pointers
  - Initialization



## Move semantics: the problem

## Inefficient code

```
1 void swap(std::vector<int> &a,
2 std::vector<int> &b) {
3 std::vector<int> c = a;
4 a = b;
5 b = c;
6 }
7 std::vector<int> v(10000), w(10000);
8 ...
9 swap(v, w);
```





# Move semantics: the problem

## Inefficient code

```
1 void swap(std::vector<int> &a,
2 std::vector<int> &b) {
3 std::vector<int> c = a;
4 a = b;
5 b = c;
6 }
7 std::vector<int> v(10000), w(10000);
8 ...
9 swap(v, w);
```

## What happens during swap

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



# Move semantics: the problem

## Vector's built-in efficient swap

```
1 std::vector<int> v(10'000), w(10'000);
2 ...
3 v.swap(w);
```



# Move semantics: the problem

## Vector's built-in efficient swap

```
1 std::vector<int> v(10'000), w(10'000);
2 ...
3 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

```
1 T consumeVector(std::vector<int> input) {
2 // ... change elements, compute result
3 return result;
4 }
5
6 std::vector<int> values{...};
7 consumeVector(values); // being copied now
```



# Move semantics: the problem

## Another potentially inefficient code

```
1 T consumeVector(std::vector<int> input) {
2 // ... change elements, compute result
3 return result;
4 }
5
6 std::vector<int> values{...};
7 consumeVector(values); // being copied now
```

## Pass by copy

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



## Move semantics: the problem

## Another potentially inefficient code

```
1 T consumeVector(std::vector<int> const & input) {
2 std::vector tmp{input};
3 // ... change elements, compute result
4 return result;
5 }
6 std::vector<int> values{...};
7 consumeVector(values); // maybe copied internally?
```



# Move semantics: the problem

C++ 98 Adv

## Another potentially inefficient code

```
1 T consumeVector(std::vector<int> const & input) {
2 std::vector tmp{input};
3 // ... change elements, compute result
4 return result;
5 }
6 std::vector<int> values{...};
7 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

## Another potentially inefficient code

```
1 T consumeVector(std::vector<int> & input) {
2 // ... change elements, compute result
3 return result;
4 }
5
6 std::vector<int> values{...};
7 consumeVector(values); // maybe modified?
8 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

## Another potentially inefficient code

```
1 T consumeVector(std::vector<int> & input) {
2 // ... change elements, compute result
3 return result;
4 }
5
6 std::vector<int> values{...};
7 consumeVector(values); // maybe modified?
8 somethingElse(values); // safe to use values now???
```

## The ideal situation

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



## 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)



## 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

```
1 T(T const & other); // copy construction
2 T(T&& other); // move construction
3 T& operator=(T const & other); // copy assignment
4 T& operator=(T&& other); // move assignment
```



## 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

## 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

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



# Move semantics

In some cases, you want to force a move

```
1 void swap(T &a, T &b) {
2 T c = a; // copy construct
3 a = b; // copy assign
4 b = c; // copy assign
5 }
```



# Move semantics

In some cases, you want to force a move

```
1 void swap(T &a, T &b) {
2 T c = a; // copy construct
3 a = b; // copy assign
4 b = c; // copy assign
5 }
```

Explicitly request moving

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

```
6 void swap(T &a, T &b) {
7 T c = std::move(a); // move construct
8 a = std::move(b); // move assign
9 b = static_cast<T&&>(c); // move assign (don't)
10 }
```



## Move semantics: recommended implementation C++ 11 Adv

## 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 Adv

## Practically

```

1 class Movable {
2 Movable();
3 Movable(const Movable &other);
4 Movable(Movable &&other) noexcept :
5 Movable() { // constructor delegation
6 swap(*this, other);
7 }
8 Movable& operator=(Movable other) noexcept { // by value
9 swap(*this, other);
10 return *this;
11 }
12 friend void swap(Movable &a, Movable &b) noexcept {...}
13 };
14 Movable a, b;
15 a = b; // operator= copies b into "other"
16 a = std::move(b); // operator= moves b into "other"

```



## Move semantics: alternative implementation

## Practically

```
1 class Movable {
2 Movable();
3 Movable(const Movable &other);
4 Movable(Movable &&other) noexcept :
5 Movable() { // constructor delegation
6 swap(*this, other);
7 }
8 Movable& operator=(const Movable& other);
9 Movable& operator=(Movable&& other) noexcept {
10 swap(*this, other);
11 return *this;
12 }
13 friend void swap(Movable &a, Movable &b) noexcept { ... }
14 };
```



### 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
  - Standard library
  - Ranges
  - RAI and smart pointers
  - Initialization



## Where does it take place ?

```
1 struct Foo { ... };
2 Foo f() {
3 return Foo(); // return-value opt. (RVO)
4 }
5 Foo g() {
6 Foo foo;
7 return foo; // named return-value opt. (NRVO)
8 }
9 int main() {
10 // compiler must avoid these copies:
11 Foo a = f();
12 Foo b = Foo();
13 Foo c = Foo(Foo());
14 // copy elision allowed, but not guaranteed:
15 Foo d = g();
16 }
```



## Guaranteed copy elision

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

## Guaranteed: Return value optimization (RVO)

```
1 struct Foo {
2 Foo() { ... }
3 Foo(const Foo &) = delete;
4 Foo(Foo &&) = delete;
5 Foo& operator=(const Foo &) = delete;
6 Foo& operator=(Foo &&) = delete;
7 };
8 Foo f() {
9 return Foo(); // ok, no move
10 }
11 int main() {
12 Foo foo = f(); // ok, no move
13 }
```



# Templates

- 4 Core modern C++
  - Constness
  - Constant Expressions
  - Exceptions
  - Move semantics
  - Copy elision
  - **Templates**
  - Lambdas
  - The STL
  - Standard library
  - Ranges
  - RAI and smart pointers
  - Initialization



## Concept

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

```
1 template<typename T>
2 const T & max(const T &a, const T &b) {
3 return b < a ? a : b;
4 }
5 template<typename T>
6 struct Vector {
7 int m_len;
8 T* m_data;
9 };
10 template <typename T>
11 std::size_t size = sizeof(T);
```





## 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

1  
2  
3  
4

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

func(3)

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

func(5.2)

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



## Template parameters

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

```
1 template<typename KeyType=int, typename ValueType=KeyType>
2 struct Map {
3 void set(const KeyType &key, ValueType value);
4 ValueType get(const KeyType &key);
5 ...
6 };
7
8 Map<std::string, int> m1;
9 Map<float> m2; // Map<float, float>
10 Map<> m3; // Map<int, int>
11 Map m4; // Map<int, int>, C++17
```



## 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**

```
1 template<typename T>
2 T func(T a); // equivalent to:
3 template<class T>
4 T func(T a);
5
6 template<template<class> class C>
7 C<int> func(C<int> a); // equivalent to:
8 template<template<typename> class C>
9 C<int> func(C<int> a); // equivalent to:
10 template<template<typename> typename C> // C++17
11 C<int> func(C<int> a);
```



## Template implementation

C++ 98

```
1 template<typename KeyType=int, typename ValueType=KeyType>
2 struct Map {
3 // declaration and inline definition
4 void set(const KeyType &key, ValueType value) {
5 ...
6 }
7 // just declaration
8 ValueType get(const KeyType &key);
9 };
10
11 // out-of-line definition
12 template<typename KeyType, typename ValueType>
13 ValueType Map<KeyType, ValueType>::get
14 (const KeyType &key) {
15 ...
16 }
```



## Nested templates

## Nested templates - godbolt

```

1 template<typename KeyType=int, typename ValueType=KeyType>
2 struct Map {
3 template<typename OtherValueType>
4 void set(const KeyType &key, OtherValueType value) {
5 ...
6 }
7 template<typename OtherKeyType>
8 ValueType get(const OtherKeyType &key);
9 };
10
11 template<typename KeyType, typename ValueType> //for class
12 template<typename OtherKeyType> //for member function
13 ValueType Map<KeyType, ValueType>::get
14 (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

```

1 template<unsigned int N>
2 struct Polygon {
3 float perimeter() {
4 return 2 * N * std::sin(PI / N) * radius;
5 }
6 float radius;
7 };
8
9 Polygon<19> nonadecagon{3.3f};

```



# Template specialization

## Specialization

Templates can be specialized for given values of their parameter

```
1 template<typename F, unsigned int N>
2 struct Polygon { ... }; // primary template
3
4 template<typename F> // partial specialization
5 struct Polygon<F, 6> {
6 F perimeter() { return 6 * radius; }
7 F radius;
8 };
9 template<> // full specialization
10 struct Polygon<int, 6> {
11 int perimeter() { return 6 * radius; }
12 int radius;
13 };
```



## Template argument deduction

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

```

1 template <typename T, typename U>
2 void f(T t, U u) { ... }
3
4 f(42, true); // deduces T=int, U=bool
5 // calls f<int, bool>(42, true);
6 f<float>(42, true); // sets T=float, deduces U=bool
7 // calls f<float, bool>(42, true);
8 // 42 converted to float before call

```





# Template argument deduction

## Deduced contexts

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

```
1 template <typename T>
2 void f(T* p) { ... }
3
4 const int * ip = ...;
5 f(ip); // deduces T=const int
6
7 template <typename T, std::size_t N>
8 void g(std::array<T*, N> a) { ... }
9
10 std::array<int*, 3> aip = ...;
11 g(aip); // deduces T=int, N=3
```



## Template argument deduction

## Non-deduced contexts

- Deduction from certain expressions is impossible/forbidden

```

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

```



## Non-deduced contexts

- Deduction from certain expressions is impossible/forbidden

```

1 template <typename C, typename F>
2 void reduce(const C& cont, const F& f = std::plus{});
3 reduce(std::vector<int>{...}); // error: cannot deduce F
4 // would need: <typename C, typename F = decltype(std::plus{})>
5
6 template<typename T>
7 const T & max(const T &a, const T &b) { ... }
8 int i = 3;
9 max(i, 3.14f); // deduces T=int and T=float, error
10 // either: max<float>(i, 3.14f);
11 // or: max(static_cast<float>(i), 3.14f);

```



# Template argument deduction

## Deduction in partial specializations

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

```

1 template <typename T>
2 struct S { ... }; // primary template
3
4 template <typename T>
5 struct S<T*> { ... }; // specialization 1
6
7 template <typename U, std::size_t N>
8 struct S<std::array<U*, N>> { ... }; // specialization 2
9
10 S<int> s1; // prim. tpl. (T=int) ok, spec. 1/2 invalid
11 S<int*> s2; // prim. tpl. (T=int*) and spec. 1 (T=int) ok
12 // spec. 1 is more specialized, will be used

```



# Template argument deduction

## Specialization vs. overloading

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

```

1 template <typename T>
2 void f(T t) { ... }
3
4
5 void f(int* t) { ... }
6
7
8 template <typename T>
9 void f(T* t) { ... }

```

```

10 template <typename T>
11 void f(T t) { ... }
12
13 template <>
14 void f<int*>(int* t) { ... }
15
16 // part. spec. forbidden:
17 template <typename T>
18 void f<T*>(T* t) {...}

```



## 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?

```

1 template <typename T>
2 void f(T* p) { ... }
3
4 template <typename T>
5 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 Adv

## 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)

## Practically - godbolt

```

1 template<typename A, typename B, typename C = double>
2 struct Triple {
3 Triple(A a, B b, C c) : a(a), b(b), c(c) {}
4 A a; B b; C c;
5 };
6 Triple t{42, true, 3.14}; // Triple<int, bool, double>
7 Triple<int> t2{42, true, 3.14}; // compilation error
8 Triple<int, bool> t3{42, true, 3.14}; // not CTAD

```



## Class Template Argument Deduction (CTAD) C++ 17 Adv

## Deduction guides

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

```
1 template<typename A, typename B>
2 struct Pair {
3 Pair(A a, B b) : a(a), b(b) {}
4 A a; B b;
5 };
6
7
8
9 Pair p{42, "hello"}; // Pair<int, const char*>
```





## Class Template Argument Deduction (CTAD) C++ 17 Adv

## Deduction guides

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

```

1 template<typename A, typename B>
2 struct Pair {
3 Pair(A a, B b) : a(a), b(b) {}
4 A a; B b;
5 };
6 template<typename A>
7 Pair(A, const char*) -> Pair<A, std::string>;
8
9 Pair p{42, "hello"}; // Pair<int, std::string>

```



## Class Template Argument Deduction (CTAD) C++ 17 Adv

## Standard library examples

```
1 std::pair p{1.2, true}; // std::pair<double, bool>
2 std::tuple t{1.2, true, 32};
3 // std::tuple<double, bool, int>
4 std::vector v{1, 2, 3}; // std::vector<int>
5 std::list l{v.begin(), v.end()}; // std::list<int>
6 std::array a{1, 2, 3}; // std::array<int, 3>
7
8 std::mutex m;
9 std::lock_guard l{m}; // std::lock_guard<std::mutex>
```



## 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
  - Standard library
  - Ranges
  - RAI and smart pointers
  - Initialization



# Lambda expressions

C++ 11

## Definition

A lambda expression is a function with no name



## Definition

A lambda expression is a function with no name

## Python example

```
1 data = [1,9,3,8,3,7,4,6,5]
2
3 # without lambdas
4 def isOdd(n):
5 return n%2 == 1
6 print(filter(isOdd, data))
7
8 # with lambdas
9 print(filter(lambda n:n%2==1, data))
```



## Simplified syntax

```
1 auto f = [] (arguments) -> return_type {
2 statements;
3 };
```

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

## Usage example

```
4 int data[] {1,2,3,4,5};
5 auto f = [] (int i) {
6 std::cout << i << " squared is " << i*i << '\n';
7 };
8 for (int i : data) f(i);
```



# Trailing function return type

C++ 11

An alternate way to specify a function's return type

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





## Trailing function return type

C++ 11

An alternate way to specify a function's return type

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

When to use trailing return type

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

```
1 class Equation {
2 using ResultType = double;
3 ResultType evaluate();
4 }
5 Equation::ResultType Equation::evaluate() {...}
6 auto Equation::evaluate() -> ResultType {...}
```



## 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++

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

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

## Error

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



## 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

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



# Default capture is by value

C++ 11

## Code example

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



# Default capture is by value

C++ 11

## Code example

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

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

## Error

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

## Explanation

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



# Capture by reference

C++ 11

## Simple example

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

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



# Capture by reference

C++ 11

## Simple example

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

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

## Mixed case

One can of course mix values and references

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



## Anatomy of a lambda

Lambdas are pure syntactic sugar - [cppinsight](#)

- They are replaced by a functor during compilation

```

1 int sum = 0, off = 1;
2 auto l =
3 [&sum, off]
4
5
6
7 (int x) {
8 sum += x + off;
9 };
10
11
12 l(42);
13 int sum = 0, off = 1;
14 struct __lambda4 {
15 int& sum; int off;
16 __lambda4(int& s, int o)
17 : sum(s), off(o) {}
18
19 auto operator()(int x) const {
20 sum += x + off;
21 }
22 };
23 auto l = __lambda4{sum, off};
24 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

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

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

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





## Capture list - this

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

```

1 [this](...) { use(*this); };
2 [&](...) { use(*this); };
3 [&, this](...) { use(*this); };
4 [=](...) { use(*this); }; // deprecated in C++20
5 [=, this](...) { use(*this); }; // allowed in C++20

```

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

```

1 [*this](...) { use(*this); }; // C++17
2 [&, *this](...) { use(*this); }; // C++17
3 [=, *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 (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 Adv

## Example - godbolt

```
1 auto build_incremter = [](int inc) {
2 return [inc](int value) { return value + inc; };
3 };
4 auto inc1 = build_incremter(1);
5 auto inc10 = build_incremter(10);
6 int i = 0;
7 i = inc1(i); // i = 1
8 i = inc10(i); // i = 11
```

## How it works

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



## Lambda improvements in C++ 20

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

## Example - godbolt

```
1 struct S {
2 decltype([](int i) { std::cout << i; }) f;
3 } s;
4 s.f(42); // prints "42"
5
6 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**
  - Standard library
  - Ranges
  - RAI 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 example - godbolt

```
1 #include <vector>
2 #include <algorithm>
3 #include <functional> // `import std;` in C++23
4 #include <iterator>
5 #include <iostream>
6
7 std::vector<int> in{5, 3, 4}; // initializer list
8 std::vector<int> out(3); // constructor taking size
9 std::transform(in.begin(), in.end(), // input range
10 out.begin(), // start result
11 std::negate{}); // function obj
12 std::copy(out.begin(), out.end(), // -5 -3 -4
13 std::ostream_iterator<int>{std::cout, " "});
```





## 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 [cpreference](#))

- 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

```
1 #include <vector>
2 std::vector<T> v{5, 3, 4}; // 3 Ts, 5, 3, 4
3 std::vector<T> v(100); // 100 default constr. Ts
4 std::vector<T> v(100, 42); // 100 Ts with value 42
5 std::vector<T> v2 = v; // copy
6 std::vector<T> v2 = std::move(v); // move, v is empty
7
8 std::size_t s = v.size();
9 bool empty = v.empty();
10
11 v[2] = 17; // write element 2
12 T& t = v[1000]; // access element 1000, bug!
13 T& t = v.at(1000); // throws std::out_of_range
14 T& f = v.front(); // access first element
15 v.back() = 0; // write to last element
16 T* p = v.data(); // pointer to underlying storage
```



## Containers: std::vector

```
1 std::vector<T> v = ...;
2 auto b = v.begin(); // iterator to first element
3 auto e = v.end(); // iterator to one past last element
4 // all following operations, except reserve, invalidate
5 // all iterators (b and e) and references to elements
6
7 v.resize(100); // size changes, grows: new T{}s appended
8 // shrinks: Ts at end destroyed
9 v.reserve(1000); // size remains, memory increased
10 for (T i = 0; i < 900; i++)
11 v.push_back(i); // add to the end
12 v.insert(v.begin()+3, T{}); // insert after 3rd position
13
14 v.pop_back(); // removes last element
15 v.erase(v.end() - 3); // removes 3rd-last element
16 v.clear(); // removes all elements
```



## Containers: std::unordered\_map

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

```

1 #include <unordered_map>
2 std::unordered_map<std::string, int> m;
3 m["hello"] = 1; // inserts new key, def. constr. value
4 m["hello"] = 2; // finds existing key
5 auto [it, isNewKey] = m.insert({"hello", 0}); // no effect
6 int val = m["world"]; // inserts new key (val == 0)
7 int val = m.at("monde"); // throws std::out_of_range
8
9 if (auto it = m.find("hello"); it != m.end()) // C++17
10 m.erase(it); // remove by iterator (fast)
11 if (m.contains("hello")) // C++20
12 m.erase("hello"); // remove by key, 2. lookup, bad
13 for (auto const& [k, v] : m) // iterate k/v pairs (C++17)
14 std::cout << k << ": " << v << '\n';

```



## std::hash

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

```
1 #include <functional>
2 std::hash<std::string> h;
3 std::cout << h("hello"); // 2762169579135187400
4 std::cout << h("world"); // 8751027807033337960
5
6 class MyClass { int a, b; ... };
7 template<> struct std::hash<MyClass> {
8 std::size_t operator()(MyClass const& c) {
9 std::hash<int> h;
10 return h(c.a) ^ h(c.b); // xor to combine hashes
11 }
12 };
```



## 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

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



## algorithms

- implementation of an algorithm working on data
- with a well defined behavior (defined complexity)
- irrespective of
  - the data handled
  - the container where the data live
  - the iterator used to go through data (almost)
- examples
  - for\_each, find, find\_if, count, count\_if, search
  - copy, swap, transform, replace, fill, generate
  - remove, remove\_if
  - unique, reverse, rotate, shuffle, partition
  - sort, partial\_sort, merge, make\_heap, min, max
  - lexicographical\_compare, iota, reduce, partial\_sum
- see also [105 STL Algorithms in Less Than an Hour](#) and the [algorithms library](#) on cppreference



## functors / function objects

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





## Functors / function objects

## Example

```
1 struct Incrementer {
2 int m_inc;
3 Incrementer(int inc) : m_inc(inc) {}
4
5 int operator()(int value) const {
6 return value + m_inc;
7 }
8 };
9 std::vector<int> v{1, 2, 3};
10 const auto inc = 42;
11 std::transform(v.begin(), v.end(), v.begin(),
12 Incrementer{inc});
```



# Prefer lambdas over functors

C++ 11

## With lambdas

```
1 std::vector<int> v{1, 2, 3};
2 const auto inc = 42;
3 std::transform(begin(v), end(v), begin(v),
4 [inc](int value) {
5 return value + inc;
6 });
```



# Prefer lambdas over functors

C++ 11

## With lambdas

```
1 std::vector<int> v{1, 2, 3};
2 const auto inc = 42;
3 std::transform(begin(v), end(v), begin(v),
4 [inc](int value) {
5 return value + inc;
6 });
```

## Good practice: Use STL algorithms with lambdas

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



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

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



## Range-based for loops with STL containers

C++ 11

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

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

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

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



## Range-based for loops with STL containers

C++ 11

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

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

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

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

## STL way (since C++ 98)

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



## More examples

C++ 17

```
1 std::list<int> l = ...;
2
3 // Finds the first element in a list between 1 and 10.
4 const auto it = std::find_if(l.begin(), l.end(),
5 [](int i) { return i >= 1 && i <= 10; });
6 if (it != l.end()) {
7 int element = *it; ...
8 }
9
10 // Computes sin(x)/(x + DBL_MIN) for elements of a range.
11 std::vector<double> r(l.size());
12 std::transform(l.begin(), l.end(), r.begin(),
13 [](auto x) { return std::sin(x)/(x + DBL_MIN); });
14
15 // reduce/fold (using addition)
16 const auto sum = std::reduce(v.begin(), v.end());
```



## More examples

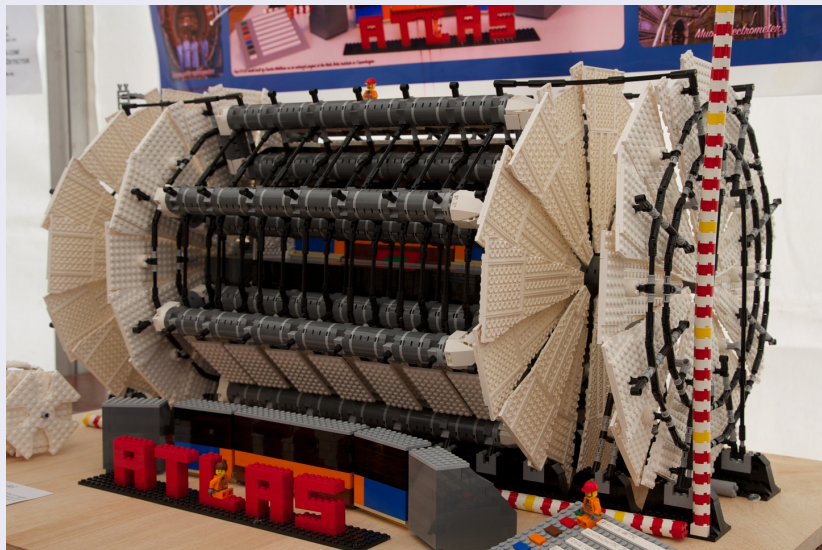
```
1 std::vector<int> v = ...;
2
3 // remove duplicates
4 std::sort(v.begin(), v.end());
5 auto newEndIt = std::unique(v.begin(), v.end());
6 v.erase(newEndIt, v.end());
7
8 // remove by predicate
9 auto p = [](int i) { return i > 42; };
10 auto newEndIt = std::remove_if(v.begin(), v.end(), p);
11 v.erase(newEndIt, v.end());
12
13 // remove by predicate (C++20)
14 std::erase_if(v, p);
```





# Welcome to lego programming!

C++ 98



## 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



## 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!



# Standard library

- 4 Core modern C++
  - Constness
  - Constant Expressions
  - Exceptions
  - Move semantics
  - Copy elision
  - Templates
  - Lambdas
  - The STL
  - **Standard library**
  - Ranges
  - RAI and smart pointers
  - Initialization



## 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

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



## 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 extent
  - meaning the size is determined at compile time
  - and thus only a pointer is stored by `span`



## std::span&lt;T&gt; - Usage

## Some example - godbolt

```

1 void print(std::span<const int> c) {
2 for (auto a : c) { std::cout << a << " "; }
3 std::cout << "\n";
4 }
5 void times2(std::span<int> c) {
6 for(auto &e : c) { e *= 2; }
7 }
8 int a[]{23, 45, 67, 89};
9 print(a); // 23 45 67 89
10
11 std::vector v{1, 2, 3, 4, 5};
12 std::span<const int> sv = v;
13 print(sv.subspan(2, 2)); // 3 4
14
15 std::array a2{-14, 55, 24};
16 times2(a2);
17 print(a2); // -28 110 48
18
19 std::span<const int, 3> sa2 = a2;
20 std::cout << sizeof(sv) << " " << sizeof(sa2) << "\n"; // 16 8
21 std::span<const int, 3> s2a2 = a; // compilation failure, invalid conversi

```



std::optional `ppreference`

## 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

```
1 std::optional<Phone> parse_phone(std::string_view in) {
2 if (is_valid_phone(in))
3 return in; // equiv. to optional<Phone>{in};
4 return {}; // or: return std::nullopt; (empty opt.)
5 }
6 if (v) { // or: v.has_value()
7 process_phone(v.value()); // or: *v (unchecked)
8 v->call(); // calls Phone::call() (unchecked)
9 }
10 v.reset(); assert(!v.has_value()); // or: v = {};
```





## Code example

```
1 // address may be given for a person or not
2 void processPerson(std::string_view name,
3 std::optional<Address> address);
4 Address addr = wonderland();
5 processPerson("Alice", std::move(addr));
6
7 // Every person has a name, but not always an address
8 struct Person {
9 std::string name;
10 std::optional<Address> address;
11 };
12 std::vector<Person> ps = ...;
13 std::sort(ps.begin(), ps.end(),
14 [](const Person& a, const Person& b) {
15 return a.address < b.address;
16 }); // sorts by address, persons w/o address at front
```



### 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::expected **cp**reference

Manages either of 2 values (expected or not)

- templated by the 2 value types
- Useful for the return value of a function that may fail
  - and would then return another type (error type)

Practically - godbolt

```
1 enum class Error {...};
2 std::expected<int, Error> parse(std::string_view in) {
3 if (is_valid(in)) return convert_to_int(in);
4 return std::unexpected(Error::...);
5 }
6 auto v = parse(...);
7 if (v) { // or v.has_value()
8 std::cout << *v; // (unchecked)
9 foo(v.value()); // may throw bad_expected_access
10 } else
11 log(v.error());
```



## 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

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



## std::variant and the visitor pattern

std::visit [cpreference](#)

- 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](#)

```

1 struct Visitor {
2 auto operator()(int i) {return "i:"+std::to_string(i);}
3 auto operator()(float f) {return "f:"+std::to_string(f);}
4 auto operator()(const std::string& s) { return "s:"+s;}
5 template <typename T>
6 auto operator()(const T& t) { return std::string{"?"}; }
7 };
8 void print(std::variant<int, float, std::string, char> v) {
9 std::cout << std::visit(Visitor{}, v) << '\n';
10 }
11 print(100); print(42.0f); print("example"); print('A');
```



## std::variant and the lambda visitor pattern

C++ 17 Adv

## Idea

- use inheritance to group a set of lambdas

## Practically - godbolt

```
1 template<typename... Ts> // covered in expert part
2 struct Overload : Ts ... { using Ts::operator()...; };
3 template<typename... Ts>
4 Overload(Ts...) -> Overload<Ts...>; // obsolete in C++20
5 int main(){
6 std::variant<int, float, std::string, char> v{ ... };
7 auto overloadSet = Overload {
8 [](int i) { std::cout << "i32:" << i;},
9 [](std::string s) { std::cout << s;},
10 [](auto a) { std::cout << "other"; },
11 };
12 std::visit(overloadSet, v);
13 }
```



### 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`.



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

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





## The problem in C++ 98

STL containers and arrays have different syntax for loop

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



## non-member begin and end

## The problem in C++ 98

STL containers and arrays have different syntax for loop

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

## A new syntax

```
5 for(auto it = begin(v); it != end(v); it++) {...}
6 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

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

## A new syntax

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

## Allowing the best syntax

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



## std::tuple cppreference

## C++ 11 Adv

```
1 #include <tuple>
2
3 std::tuple<int, float, bool> t{42, 3.14f, false};
4 auto t = std::make_tuple(42, 3.14f, false);
5 std::tuple t{42, 3.14f, false}; // C++17 CTAD
6
7 float& f = std::get<1>(t); // get 1. member
8 bool& b = std::get<bool>(t); // get bool member
9
10 using Tup = decltype(t);
11 constexpr auto s = std::tuple_size_v<Tup>; // 3
12 using E2Type = std::tuple_element<2, Tup>; // bool
13
14 auto t2 = std::tuple_cat(t, std::tuple{'A'}, t);
15 // <int, float, bool, char, int, float, bool>
16 bool& b2 = std::get<bool>(t2); // error
```



## std::tuple

## C++ 11 Adv

```
1 void f(std::tuple<int, float, bool> tuple) {
2 int a; float b; bool c;
3 std::tie(a, b, c) // tuple<int&, float&, bool&
4 = tuple;
5 // Use structured bindings in C++17
6 }
7
8 int& i = ...;
9 std::tuple{i}; // tuple<int>
10 std::make_tuple(i); // tuple<int>
11 std::make_tuple(std::ref(i)); // tuple<int&>
12 std::tuple{std::ref(i)};
13 // C++17 CTAD, tuple<std::reference_wrapper<int>>
```



## Structured Binding Declarations

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

Practically - `godbolt`

```

1 std::tuple<int, double, long> tuple = ...;
2 auto [a, b, c] = tuple;
3
4 struct S { int i; float f; bool b; } s = ...;
5 auto&& [a, b, c] = std::move(s);
6
7 int arr[] = {1, 3, 4};
8 const auto& [a, b, c] = arr;
9
10 std::unordered_map<K, V> map = ...;
11 for (const auto& [key, value] : map) { ... }

```



# Structured Binding Declarations

## “Tuple-like binding protocol”

- 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)`

```
1 class C { ... }; // complex, cannot change
2 template <>
3 struct std::tuple_size<C> {
4 static constexpr std::size_t value = ...;
5 };
6 template <std::size_t I>
7 struct std::tuple_element<I, C> { using type = ...; };
8 template <std::size_t I>
9 auto& get(C& c) { ... }; // -> std::tuple_element<I, C>&
```



`std::chrono` library present in `<chrono>` header

## Clocks

- consists of a starting point (or epoch) and a tick rate
  - E.g. January 1, 1970 and every second
- C++ defines several clock type
  - `system_clock` system time, aka wall clock time, or C time
  - `steady_clock` monotonic but unrelated to wall clock time
  - `high_resolution_clock` clock with the smallest tick period

## `time_point` and `duration`

- provide easy manipulation of times and duration
- clock dependent
- `duration_cast` allows conversions between duration types
  - available helper types : nanoseconds, microseconds, milliseconds, seconds, minutes, hours, ...





How to measure the time spent in some code - `godbolt`

```
1 #include <chrono>
2
3 std::chrono::high_resolution_clock clock;
4 auto start = clock::now();
5 ... // code to be timed
6 std::chrono::duration<float> ticks = clock.now() - start;
7
8 auto millis = std::chrono::duration_cast
9 <std::chrono::milliseconds>(ticks);
10 std::cout << "it took " << ticks.count() << " ticks"
11 << ", that is " << millis.count() << " ms\n";
```



How to measure the time spent in some code - `godbolt`

```
1 #include <chrono>
2
3 std::chrono::high_resolution_clock clock;
4 auto start = clock::now();
5 ... // code to be timed
6 std::chrono::duration<float> ticks = clock.now() - start;
7
8 auto millis = std::chrono::duration_cast
9 <std::chrono::milliseconds>(ticks);
10 std::cout << "it took " << ticks.count() << " ticks"
11 << ", that is " << millis.count() << " ms\n";
```

## Warning

- this does not measure the amount of CPU used !
- neither the time spent on a CPU (think suspended threads)



## Pseudo-random number generators (PRNG)

C++ 11 Adv

## 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

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



## 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

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



## 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

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



## 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

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



## 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++
  - Constness
  - Constant Expressions
  - Exceptions
  - Move semantics
  - Copy elision
  - Templates
  - Lambdas
  - The STL
  - Standard library
  - **Ranges**
  - RAI and smart pointers
  - Initialization





## 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



## 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`

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



## Many range adaptors are provided by the STL C++ 20 Adv

## 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 (|)



## 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`

```
1 // print first 20 prime numbers above 1000000
2 for (int i: std::views::iota(1000000)
3 | std::views::filter(odd)
4 | std::views::filter(isPrime)
5 | std::views::take(20)) {
6 std::cout << i << " ";
7 }
```



# RAII and smart pointers

- 4 Core modern C++
  - Constness
  - Constant Expressions
  - Exceptions
  - Move semantics
  - Copy elision
  - Templates
  - Lambdas
  - The STL
  - Standard library
  - Ranges
  - **RAII and smart pointers**
  - Initialization



# Pointers: why are they error prone?

C++ 98

## They need initialization

```
1 char *s;
2 try {
3 foo(); // may throw
4 s = new char[100];
5 read_line(s);
6 } catch (...) { ... }
7 process_line(s);
```



## Pointers: why are they error prone?

C++ 98

They need initialization

Seg Fault

```
1 char *s;
2 try {
3 foo(); // may throw
4 s = new char[100];
5 read_line(s);
6 } catch (...) { ... }
7 process_line(s);
```



## Pointers: why are they error prone?

C++ 98

## They need initialization

Seg Fault

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

## They need to be released

```
1 char *s = new char[100];
2 read_line(s);
3 if (s[0] == '#') return;
4 process_line(s);
5 delete[] s;
```





## Pointers: why are they error prone?

C++ 98

They need initialization

Seg Fault

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

They need to be released

Memory leak

```
1 char *s = new char[100];
2 read_line(s);
3 if (s[0] == '#') return;
4 process_line(s);
5 delete[] s;
```



## Pointers: why are they error prone?

## They need initialization

## Seg Fault

```

1 char *s;
2 try {
3 foo(); // may throw
4 s = new char[100];

```

## They need to be released

## Memory leak

```

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

```

## They need clear ownership

```

1 char *s = new char[100];
2 read_line(s);
3 vec.push_back(s);
4 set.add(s);
5 std::thread t1{func1, vec};
6 std::thread t2{func2, set};

```



## Pointers: why are they error prone?

C++ 98

## They need initialization

Seg Fault

```

1 char *s;
2 try {
3 foo(); // may throw
4 s = new char[100];

```

## They need to be released

Memory leak

```

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

```

## They need clear ownership

Who should release ?

```

1 char *s = new char[100];
2 read_line(s);
3 vec.push_back(s);
4 set.add(s);
5 std::thread t1{func1, vec};
6 std::thread t2{func2, set};

```



# This problem exists for any resource

For example with a file

```
1 std::FILE *handle = std::fopen(path, "w+");
2 if (nullptr == handle) { throw ... }
3 std::vector v(100, 42);
4 write(handle, v);
5 if (std::fputs("end", handle) == EOF) {
6 return;
7 }
8 std::fclose(handle);
```

Which problems do you spot in the above snippet?



## 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



## An RAI File class

```
1 class File {
2 public:
3 // constructor: acquire resource
4 File(const char* filename)
5 : m_handle(std::fopen(filename, "w+")) {
6 // abort constructor on error
7 if (m_handle == nullptr) { throw ... }
8 }
9 // destructor: release resource
10 ~File() { std::fclose(m_handle); }
11 void write (const char* str) {
12 ...
13 }
14 private:
15 std::FILE* m_handle; // wrapped resource
16};
```



## Usage of File class

```
1 void log_function() {
2 // file opening, aka resource acquisition
3 File logfile("logfile.txt");
4
5 // file usage
6 logfile.write("hello logfile!"); // may throw
7
8 // file is automatically closed by the call to
9 // its destructor, even in case of exception!
10 }
```

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

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



## 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





## 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

```
1 #include <memory>
2 void f(std::unique_ptr<Foo> ptr) {
3 ptr->bar();
4 } // deallocation when f exits
5
6 std::unique_ptr<Foo> p{ new Foo{} }; // allocation
7 f(std::move(p)); // transfer ownership
8 assert(p.get() == nullptr);
```



What do you expect?

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



## What do you expect?

```

1 void f(std::unique_ptr<Foo> ptr);
2 std::unique_ptr<Foo> uptr(new Foo{});
3 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

## 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

```
1 {
2 // calls new File("logfile.txt") internally
3 auto f = std::make_unique<File>("logfile.txt");
4 f->write("hello logfile!");
5 }
```

*// deallocation at end of scope*



## Arrays and unique\_ptr

## 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

```
1 auto b = std::make_unique<Foo[]>(10);
2 b[3] = ...;
3 b[4].someFunction();
4
5 // deallocations at end of scope
```



## 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

```
1 std::unique_ptr<T> produce();
2 void observe(const T&);
3 void modifyRef(T&);
4 void modifyPtr(T*);
5 void consume(std::unique_ptr<T>);
6 std::unique_ptr<T> pt{produce()}; // Receive ownership
7 observe(*pt); // Keep ownership
8 modifyRef(*pt); // Keep ownership
9 modifyPtr(pt.get()); // Keep ownership
10 consume(std::move(pt)); // Transfer ownership
```





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** : 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`

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



## 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

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



Quiz: `std::shared_ptr` in use

C++ 11

What is the output of this code? - `godbolt`

```

1 auto shared = std::make_shared<int>(100);
2 auto print = [shared]() {
3 std::cout << "Use: " << shared.use_count() << " "
4 << "value: " << *shared << "\n";
5 };
6 print();
7 {
8 auto ptr{ shared };
9 (*ptr)++;
10 print();
11 }
12 print();

```



Quiz: `std::shared_ptr` in use

What is the output of this code? - `godbolt`

```

1 auto shared = std::make_shared<int>(100);
2 auto print = [shared]() {
3 std::cout << "Use: " << shared.use_count() << " "
4 << "value: " << *shared << "\n";
5 };
6 print();
7 {
8 auto ptr{ shared };
9 (*ptr)++;
10 print();
11 }
12 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?

```

1 auto shared = std::make_shared<int>(100);
2 auto print = [&shared]() {
3 std::cout << "Use: " << shared.use_count() << " "
4 << "value: " << *shared << "\n";
5 };
6 print();
7 {
8 auto ptr{ shared };
9 (*ptr)++;
10 print();
11 }
12 print();

```

```

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

```



## Quiz: shared\_ptr and weak\_ptr in use

What is the output of this code? - [godbolt](#)

```
1 auto shared = std::make_shared<int>(100);
2 std::weak_ptr<int> weak{ shared };
3 print(); // with print as before
4
5 auto ptr = weak.lock();
6 (*ptr)++; print();
7
8 ptr = nullptr; print();
9
10 function(weak); print();
```



## Quiz: shared\_ptr and weak\_ptr in use

What is the output of this code? - [godbolt](#)

```
1 auto shared = std::make_shared<int>(100);
2 std::weak_ptr<int> weak{ shared };
3 print(); // with print as before
4
5 auto ptr = weak.lock();
6 (*ptr)++; print();
7
8 ptr = nullptr; print();
9
10 function(weak); print();
```

Use: 1 value: 100

Use: 2 value: 101

Use: 1 value: 101

Use: 1 (or more) value: ???





### 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 RAI
  - 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 RAI class



## 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
  - Standard library
  - Ranges
  - RAI and smart pointers
  - Initialization



# Initialization

## Initializing scalars

```

1 int i(42); // direct initialization
2 int i{42}; // direct list initialization (C++11)
3 int i = 42; // copy initialization
4 int i = {42}; // copy list initialization (C++11)

```

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

## Narrowing conversions

```

1 int i(42.3); // i == 42
2 int i{42.3}; // compilation error
3 int i = 42.3; // i == 42
4 int i = {42.3}; // compilation error

```

- Braced initialization prevents narrowing conversions



# Initialization

## Initializing class types with constructors

```

1 struct C {
2 C(int i);
3 C(int i, int j);
4 };
5
6 C c(42); // calls C(42)
7 C c{42}; // same
8 C c = 42; // calls C(42) in C++17, before C(C(42))
9 C c = {42}; // same
10
11 C c(1, 2); // calls C(1, 2)
12 C c{1, 2}; // calls C(1, 2)
13 C c = (1, 2); // calls C(2), comma operator
14 C c = {1, 2}; // calls C(1, 2) in C++17, before C(C(1, 2))
15
16 C c(1.1, 2.2); // calls C(1, 2)
17 C c{1.1, 2.2}; // compilation error

```



# Initialization

## std::initializer\_list

```

1 struct C {
2 C(int i, int j);
3 C(std::initializer_list<int> l);
4 };
5
6 C c(1, 2); // calls C(1, 2)
7 C c{1, 2}; // calls C(std::initializer_list<int>{1, 2})
8 C c = (1, 2); // calls C(2), comma operator, error
9 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

```

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

```



# Initialization

## Type deduction

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



## 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





## Value initialization

```
1 T t{}; // value initialization
2 T t = {}; // same
3 f(T()); // passes value-initialized temporary
4 f(T{}); // same
5 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](#)



## Aggregate initialization

```

1 struct S { int i; double d; };
2
3 S s{1, 2.3}; // s.i == 1, s.d == 2.3
4 S s = {1, 2.3}; // same
5 S s{.i = 1, .d = 2.3} // same, designated init. (C++20)
6 S s = {.i = 1, .d = 2.3} // same
7 S s{1}; // s.i == 1, s.d == 0.0
8 S s = {1}; // same
9 S s{.i = 1} // same (C++20)
10 S s = {.i = 1} // same (C++20)
11 S s{}; // value init, s.i == 0, s.d == 0.0
12 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 [cpreference](#)



## 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++

- 1 History and goals
- 2 Language basics
- 3 Object orientation (OO)
- 4 Core modern C++
- 5 Expert C++
  - Variadic templates
  - Perfect forwarding
  - SFINAE
  - Concepts
  - Modules
  - Coroutines
- 6 Useful tools
- 7 Concurrency
- 8 C++ and python



# The $\Leftrightarrow$ operator

- 5 Expert C++
  - The  $\Leftrightarrow$  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

## 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 Adv

## Output of operator&lt;=&gt; - godbolt

```
1 template <typename T>
2 void three_way_compare(T lhs, T rhs) {
3 auto res = lhs <=> rhs;
4 std::cout << lhs << "<=>" << rhs << ": "
5 << (res<0) << (res==0) << (res>0)
6 << '\n';
7 }
8 int main() {
9 three_way_compare(1, 2); // 1<=>2: 100
10 three_way_compare(2, 2); // 2<=>2: 010
11 three_way_compare(2, 1); // 2<=>1: 001
12 }
```



# Different kinds of ordering

## 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

## Example - godbolt

```

1 struct Ratio {
2 unsigned n, d ;
3 friend std::weak_ordering operator<=>(Ratio const & a,
4 Ratio const & b) {
5 return (a.n * b.d) <=> (a.d * b.n);
6 }
7 friend std::ostream & operator<<(std::ostream & os, Ratio const & r) {
8 return (os << r.n << '/' << r.d);
9 }
10 };
11 int main() {
12 // Ratio uses weak_ordering
13 three_way_compare(Ratio{1, 2}, Ratio{2, 3}); // 1/2<=>2/3 : 100
14 three_way_compare(Ratio{2, 3}, Ratio{4, 6}); // 2/3<=>4/6 : 010
15 three_way_compare(Ratio{2, 3}, Ratio{1, 2}); // 2/3<=>1/2 : 001
16
17 // floats use partial_ordering
18 three_way_compare(+0., -0.); // 0<=>-0 : 010
19 three_way_compare(0., 1./0.); // 0<=>inf : 100
20 three_way_compare(0., 0./0.); // 0<=>-nan : 000
21 }

```



### 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==`



## Default operator&lt;=&gt; 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==`



## 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  $\langle = \rangle$  operator
  - **Variadic templates**
  - Perfect forwarding
  - SFINAE
  - Concepts
  - Modules
  - Coroutines



# Basic variadic template

## 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 [cpreference](#)

## Recursive example [cppinsight](#)

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





# A couple of remarks

## 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

- 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

```
1 template<typename... Args>
2 void f(Args... args) {
3 constexpr std::size_t N = sizeof...(Args);
4 std::tuple<Args...> t{args...};
5 }
6 f(0, 1, 2); // Args = [int, int, int]
7 f(); // Args = empty
8 f(0, true, 3.14); // Args = [int, bool, double]
```



# Fold expressions

## The idea

- reduces a parameter pack over a binary operator
- details on [cppreference](#)

## Example

```
1 template<typename... Args>
2 auto sum1(Args... args) {
3 return (args + ...); // unary fold over +
4 } // parentheses mandatory
5 template<typename... Args>
6 auto sum2(Args... args) {
7 return (args + ... + 0); // binary fold over +
8 } // parentheses mandatory
9 int sum = sum1(); // error
10 int sum = sum2(); // ok
```



# Fold expressions

C++ 17 Adv

Example: call a function on all arguments - [cppinsight](#)

```
1 template<typename T>
2 void print(const T& t) {
3 std::cout << t;
4 }
5 template<typename... Ts>
6 void printAll(const Ts&... ts) {
7 (print(ts),...); // fold over comma operator
8 }
```

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

```
1 template<typename... Ts>
2 void printAll(const Ts&... ts) {
3 int dummy[] {(print(ts),0)...};
4 }
```

## Variadic class template

## The tuple example, simplified - godbolt

```
1 template <typename... Ts>
2 struct tuple {};
3
4 template <typename T, typename... Ts>
5 struct tuple<T, Ts...> : tuple<Ts...> {
6 tuple(T head, Ts... tail) :
7 tuple<Ts...>(tail...), m_head(head) {}
8 T m_head;
9 };
10
11 tuple<double, uint64_t, const char*>
12 t1(12.2, 42, "big");
```



## 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

```
1 template<typename T, T... Is>
2 struct integer_sequence { ... };
3
4 template<size_t... Is>
5 using index_sequence =
6 integer_sequence<std::size_t, Is...>;
7
8 template<size_t N>
9 using make_index_sequence =
10 index_sequence< /*0, 1, ..., N-1*/>;
```



## std::integer\_sequence

## Example - make\_from\_tuple with helper

```
1 template<typename T,
2 typename... Args, std::size_t... Is>
3 T helper(std::tuple<Args...> args,
4 std::index_sequence<Is...>) {
5 return T(std::get<Is>(args)...);
6 }
7 template<typename T, typename... Args>
8 T make_from_tuple(std::tuple<Args...> args) {
9 return helper<T>(args,
10 std::make_index_sequence<sizeof...(Args)>{});
11 }
12
13 struct S { S(int, float, bool) { ... } };
14 std::tuple t{42, 3.14, false};
15 S s = make_from_tuple<S>(t);
```



## std::integer\_sequence

## Example - make\_from\_tuple with lambda

```
1 template<typename T, typename... Args>
2 T make_from_tuple(std::tuple<Args...> args) {
3 return [&<std::size_t... Is>(
4 std::index_sequence<Is...>){
5 return T(std::get<Is>(args)...);
6 }<std::make_index_sequence<sizeof...(Args)>{}>{});
7 }
8
9 struct S { S(int, float, bool) { ... } };
10 std::tuple t{42, 3.14, false};
11 S s = make_from_tuple<S>(t);
```





## 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



How to write a generic wrapper function?

```
1 template <typename T>
2 void wrapper(T arg) {
3 // code before
4 func(arg);
5 // code after
6 }
```

Example usage :

- `emplace_back`
- `make_unique`



# Why is it not so simple?

```
1 template <typename T>
2 void wrapper(T arg) {
3 func(arg);
4 }
```

## 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 ?

```
1 template <typename T>
2 void wrapper(T& arg) {
3 func(arg);
4 }
5 wrapper(42);
6 // invalid initialization of
7 // non-const reference from
8 // an rvalue
```

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



## Solution: cover all cases

```
1 template <typename T>
2 void wrapper(T& arg) { func(arg); }
3
4 template <typename T>
5 void wrapper(const T& arg) { func(arg); }
6
7 template <typename T>
8 void wrapper(T&& arg) { func(std::move(arg)); }
```



## The new problem: scaling to more arguments

C++ 11 Adv

```
1 template <typename T1, typename T2>
2 void wrapper(T1& arg1, T2& arg2)
3 { func(arg1, arg2); }
4
5 template <typename T1, typename T2>
6 void wrapper(const T1& arg1, T2& arg2)
7 { func(arg1, arg2); }
8
9 template <typename T1, typename T2>
10 void wrapper(T1& arg1, const T2& arg2)
11 { func(arg1, arg2); }
12 ...
```

## Exploding complexity

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



## Reference collapsing

## Reference to references

- Are formally forbidden, but can occur sometimes

```

1 template <typename T>
2 void foo(T t) { T& k = t; } //int& &, error in C++98
3 int ii = 4;
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& &`  $\rightarrow$  `T&`
- `T&& &&`  $\rightarrow$  `T&&`





## Rvalue reference in type-deducing context

```
1 template <typename T>
2 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

## Examples

```
1 template <typename T>
2 void f(T&& t) { ... }
3
4 f(4); // rvalue -> T is int
5 double d = 3.14;
6 f(d); // lvalue -> T is double
7 float g() {...}
8 f(g()); // rvalue -> T is float
9 std::string s = "hello";
10 f(s); // lvalue -> T is std::string
11 f(std::move(s)); // rvalue -> T is std::string
12 f(std::string{"hello"}); // rvalue -> std::string
```



## Forwarding references

## Careful!

```
1 template <typename T> struct S {
2 S(T&& t) { ... } // rvalue references
3 void f(T&& t) { ... } // NOT forwarding references
4 };
```

## Half-way correct version

```
1 template <typename T> struct S {
2
3 template <typename U>
4 S(U&& t) { ... } // deducing context -> fwd. ref.
5 template <typename U>
6 void f(U&& t) // deducing context -> fwd. ref.
7 // ... but now U can be a different type than T
8 };
```



## Forwarding references

## Careful!

```

1 template <typename T> struct S {
2 S(T&& t) { ... } // rvalue references
3 void f(T&& t) { ... } // NOT forwarding references
4 };

```

## Correct version

```

1 template <typename T> struct S {
2 template <typename U, std::enable_if_t<
3 std::is_same_v<std::decay_t<U>, T>, int> = 0>
4 S(U&& t) { ... } // deducing context -> fwd. ref.
5 template <typename U>
6 void f(U&& t) // deducing context -> fwd. ref.
7 requires std::same_as<std::decay_t<U>, T> { ... }
8 };

```

# Perfect forwarding

## 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.

```
1 template <typename T>
2 struct remove_reference { using type = T; };
3 template <typename T>
4 struct remove_reference<T&> { using type = T; };
5 template <typename T>
6 struct remove_reference<T&&> { using type = T; };
7
8 template <typename T>
9 using remove_reference_t =
10 typename remove_reference<T>::type; // C++14
```



## Perfect forwarding

`std::forward`

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

```

1 template<typename T>
2 T&& forward(remove_reference_t<T>& t) noexcept { // 1.
3 return static_cast<T&&>(t);
4 }
5 template<typename T>
6 T&& forward(remove_reference_t<T>&& t) noexcept { // 2.
7 return static_cast<T&&>(t);
8 }

```

- 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

## Example - putting it all together

```
1 template <typename... T>
2 void wrapper(T&&... args) {
3 func(std::forward<T>(args)...);
4 }
```

- 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

```
1 template<typename T, typename... Args>
2 unique_ptr<T> make_unique(Args&&... args) {
3 return unique_ptr<T>
4 (new T(std::forward<Args>(args)...));
5 }
```





# SFINAE

## 5 Expert C++

- The  $\langle = \rangle$  operator
- Variadic templates
- Perfect forwarding
- **SFINAE**
- Concepts
- Modules
- Coroutines



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

## 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

```
1 template <typename T>
2 void f(typename T::type arg) { ... }
3 void f(int a) { ... }
4
5 f(1); // Calls void f(int)
```

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



## 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

```
1 struct A { double x; };
2 A a;
3 decltype(a.x) y; // double
4 decltype((a.x)) z = y; // double& (lvalue)
5 decltype(1 + 2u) i = 4; // unsigned int
6
7 template<typename T, typename U>
8 auto add(T t, U u) -> decltype(t + u);
9 // return type depends on template parameters
```



## The main idea

- function to create 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`

```
1 struct Default {
2 int foo() const;
3 };
4 class NonDefault {
5 private: NonDefault();
6 public: int foo() const;
7 };
8 decltype(Default().foo()) n1 = 1; // int
9 decltype(NonDefault().foo()) n2 = 2; // error
10 decltype(std::declval<NonDefault>().foo()) n3 = 3;
```



## true\_type and false\_type

## The main idea

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

## Example

```
1 struct truth : std::true_type { };
2
3 constexpr bool test = truth::value; // true
4 constexpr truth t;
5 constexpr bool test = t(); // true
6 constexpr bool test = t; // true
```

## Possible implementation

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



## Using SFINAE for introspection

## 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

```

1 template <typename T, typename = void>
2 struct hasFoo : std::false_type {}; // primary template
3 template <typename T>
4 struct hasFoo<T, decltype(std::declval<T>().foo())>
5 : std::true_type {}; // template special.
6 struct A{}; struct B{ void foo(); };
7 static_assert(!hasFoo<A>::value, "A has no foo()");
8 static_assert(hasFoo::value, "B has foo()");

```



## Example - godbolt

```
1 template <typename T, typename = void>
2 struct hasFoo : std::false_type {};
3 template <typename T>
4 struct hasFoo<T, decltype(std::declval<T>().foo())>
5 : std::true_type {};
6
7 struct A{};
8 struct B{ void foo(); };
9 struct C{ int foo(); };
10
11 static_assert(!hasFoo<A>::value, "A has no foo()");
12 static_assert(hasFoo::value, "B has foo()");
13 static_assert(!hasFoo<C>::value, "C has no foo()");
14 static_assert(hasFoo<C,int>::value, "C has foo()");
```



## 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

```
1 template <typename...>
2 using void_t = void;
```





Previous introspection example using `void_t` C++ 17 Adv

## Example - godbolt

```
1 template <typename T, typename = void>
2 struct hasFoo : std::false_type {};
3
4 template <typename T>
5 struct hasFoo<T,
6 std::void_t<decltype(std::declval<T>().foo())>>
7 : std::true_type {};
8
9 struct A{}; struct B{ void foo(); };
10 struct C{ int foo(); };
11
12 static_assert(!hasFoo<A>::value, "unexpected foo()");
13 static_assert(hasFoo::value, "expected foo()");
14 static_assert(hasFoo<C>::value, "expected foo()");
```



## 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

```
1 constexpr bool b = std::is_pointer<int*>::value;
2 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

```
1 template<bool B, typename T=void>
2 struct enable_if {};
3 template<typename T>
4 struct enable_if<true, T> { using type = T; };
5
6 template<bool B, typename T = void> // C++14
7 using enable_if_t = typename enable_if<B, T>::type;
```



## Usage example - godbolt

```
1 template <typename T> constexpr bool is_iterable ... ;
2
3 template<typename T,
4 typename std::enable_if_t
5 <!is_iterable<T>, bool> = true>
6 void print(T const& t) {
7 std::cout << t << "\n";
8 }
9 template<typename T,
10 typename std::enable_if_t
11 <is_iterable<T>, bool> = true>
12 void print(T const& t) {
13 for (auto const& item : t) {
14 std::cout << item << "\n";
15 }
16 }
```

Note : using `if constexpr` would be simpler



## Back to variadic class templates

## The tuple\_element trait

```
1 template <size_t I, typename Tuple>
2 struct tuple_element;
3
4 template <typename T, typename... Ts>
5 struct tuple_element<0, tuple<T, Ts...>> {
6 using type = T;
7 };
8
9 template <size_t I, typename T, typename... Ts>
10 struct tuple_element<I, tuple<T, Ts...>> {
11 using type = typename
12 tuple_element<I - 1, tuple<Ts...>>::type;
13 };
```



## Back to variadic class templates

## The tuple get function

```
1 template <size_t I, typename... Ts>
2 std::enable_if_t<I == 0,
3 typename tuple_element<0, tuple<Ts...>>::type&>
4 get(tuple<Ts...>& t) {
5 return t.m_head;
6 }
7 template <size_t I, typename T, typename... Ts>
8 std::enable_if_t<I != 0,
9 typename tuple_element<I - 1, tuple<Ts...>>::type&>
10 get(tuple<T, Ts...>& t) {
11 return get<I - 1>(static_cast<tuple<Ts...>&>(t));
12 }
```



## The tuple get function

```
1 template <size_t I, typename T, typename... Ts>
2 auto& get(tuple<T, Ts...>& t) {
3 if constexpr(I == 0)
4 return t.m_head;
5 else
6 return get<I - 1>(static_cast<tuple<Ts...>&>(t));
7 }
```

## 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

## 5 Expert C++

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





## 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 Adv

## C++ 17 work around: SFINAE

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

## Practical code - godbolt

```
1 template
2 <typename T,
3 typename = std::enable_if_t<std::is_floating_point_v<T>>>
4 bool equal(T e1, T e2) {
5 return std::abs(e1-e2)<std::numeric_limits<T>::epsilon();
6 }
7 ... equal(10,5+5) ...
```



## The world before concepts

## C++ 17 work around: SFINAE

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

## Practical code - godbolt

```

1 template
2 <typename T,
3 typename = std::enable_if_t<std::is_floating_point_v<T>>>
4 bool equal(T e1, T e2) {
5 return std::abs(e1-e2)<std::numeric_limits<T>::epsilon();
6 }
7 ... 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;

```



## A new keyword

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

## With concepts - godbolt

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



## A new keyword

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

## With concepts - godbolt

```

1 template<typename T>
2 requires std::is_floating_point_v<T>
3 bool equal(T e1, T e2) {
4 return std::abs(e1-e2)<std::numeric_limits<T>::epsilon();
5 }
6 ... 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

## Example of several competing templates

```
1 template<typename T>
2 bool equal(T e1, T e2) { return (e1==e2); }
3
4 template< typename T>
5 requires std::is_floating_point_v<T>
6 bool equal(T e1, T e2)
7 {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.



## Definition

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

## A new keyword: concept

```
1 template< typename T>
2 concept MyFloatingPoint =
3 std::is_floating_point_v<T> &&
4 std::numeric_limits<T>::epsilon()>0;
5
6 template<typename T>
7 requires MyFloatingPoint<T>
8 bool equal(T e1, T e2)
9 {return std::abs(e1-e2)<std::numeric_limits<T>::epsilon();}
```



# Some usages of concepts

## Constrained template parameters

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

```
1 template<MyFloatingPoint T>
2 bool equal(T e1, T e2) {
3 return std::abs(e1-e2)<std::numeric_limits<T>::epsilon();
4 }
```

## Constrained variables

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

## Example code

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





# Some usages of concepts

## 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

```
1 // no template <...> here!
2 bool equal(MyFloatingPoint auto e1,
3 MyFloatingPoint auto e2) {
4 return std::abs(e1-e2) <
5 std::numeric_limits<decltype(e1)>::epsilon();
6 }
7
8 // unconstrained abbreviated function template:
9 void equal(auto e1, auto e2) { ... }
```



## 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

```
1 #include <concepts>
2 bool equal(std::floating_point auto e1,
3 std::floating_point auto e2) {
4 return std::abs(e1-e2) <
5 std::numeric_limits<decltype(e1)>::epsilon();
6 }
```



# Checking concepts

C++ 20 Adv

## 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`

```
1 template<typename T>
2 bool equal(T e1, T e2)
3 {
4 if constexpr (std::floating_point<T>) {
5 return std::abs(e1-e2)
6 < std::numeric_limits<T>::epsilon();
7 } else {
8 return (e1==e2);
9 }
10 }
```



# Advanced requirements overview

C++ 20 Adv

## 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

```
1 template<typename T>
2 concept StreamableAndComparableNumber =
3 requires(T v1, T v2, std::ostream os) {
4 requires std::integral<T> || std::floating_point<T>;
5 os<<v1<<v2;
6 { equal(v1,v2) } -> std::convertible_to<bool>;
7 };
```

Remember: use standard concepts first



## 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: 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  $\langle = \rangle$  operator
  - Variadic templates
  - Perfect forwarding
  - SFINAE
  - Concepts
  - **Modules**
  - Coroutines



## 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, or 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

```
1 #include "math.h"
```

## Module importation

```
1 import math;
```





# Writing modules

C++ 20 Adv

## ratio.cpp

```
1 export module math;
2 import <string>;
3 namespace utils { // ns. and module names orthogonal
4 export struct Ratio { float num, denom; };
5 export std::string toString(const Ratio& r) { ... }
6 void normalize(Ratio& r) { ... } // not exported
7 }
```

## main.cpp

```
1 import <iostream>;
2 import math;
3 int main() {
4 auto r = utils::Ratio{1, 3};
5 std::cout << utils::toString(r);
6 }
```



## 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 Adv

## What to export

```
1 export module math;
2
3 export void f(); // function
4 export float pi = 3.14f; // variable
5 export namespace utils { // namespace
6 enum Status { Yes, No }; // enum (exported)
7 }
8 namespace utils {
9 export struct Ratio { ... }; // class
10 export using R = Ratio; // alias
11 float e = 2.71f; // (not exported)
12 }
13 export { // content of export block
14 void h();
15 }
```



## What not to export

```
1 export module math;
2
3 namespace {
4 export void f(); // error
5 export float pi = 3.14f; // error
6 export struct Ratio { ... }; // error
7 }
8
9 export static void f(); // error
10 export static float pi = 3.14f; // error
11
12 class C { ... };
13 export using D = C; // error
```



## Visibility and reachability

C++ 20 Adv

## ratio.cpp

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

## main.cpp

```
1 import math;
2 import <iostream>;
3 int main() {
4 Ratio r = golden(); // error: Ratio not visible
5 auto r = golden(); // OK: Ratio is reachable
6 std::cout << r.num; // OK: members of reachable
7 // types are visible
8 using R = decltype(r);
9 R r2{1.f, 3.f}; // OK
10 }
```



## Module structure

```

1 module; // global module fragment (opt.)
2 #include <cassert> // only preprocessor
3 // ... // statements allowed
4
5 export module math; // module preamble starts with
6 // exported module name.
7 import <string>; // only imports allowed
8 // ... //
9
10 export struct S { ... }; // module purview started
11 export S f(); // at first non-import
12 std::string h() { ... }
13
14 module :private // private module fragment (opt.)
15 S f() { ... } // changes here trigger
16 // no recompilation of BMI

```



# Module structure

C++ 20 Adv

## Pitfall

```
1 export module math;
2 #include <header.h>
3 import <string>;
4 // ...
```

- 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

```
1 module;
2 #include <vector>
3 export module math;
4 import <string>;
5 // ...
```



# Module structure

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

## Interface

```
1 export module math;
2
3 export struct S { ... };
4 export S f();
```

## Implementation

```
1 module;
2 #include <cassert>
3 module math;
4 import <string>;
5
6 S f() { ... }
7 std::string b() { ... }
```





## Submodules and re-exporting

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

## Module one

```
1 export module one;
2
3 export struct S { ... };
```

## Module two

```
1 export module two;
2 import <string>;
3
4 export std::string b()
5 { ... }
```

## Module three

```
1 export module three;
2 import one;
3 export import two;
4
5 export S f() { ... }
```

## Module four

```
1 export module four;
2 import three;
3 // b and f visible
4 // S and std::string only
5 // reachable
```



# Module partitions

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

## Primary module interface

```
1 export module math;
2 export import :structs;
3
4 export S f();
```

## Module interface partition

```
1
2 export module math:structs;
3
4 export struct S { ... };
```

## Implementation

```
1 module math;
2 import <string>;
3
4 S f() { ... }
5 std::string b() { ... }
```



# Module implementation units

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

## Interface

```
1 export module math;
2 import :foo; // no export
3 import :bar; // no export
4 import <string>;
5
6 export struct S { ... };
7 export S f();
8 std::string b();
```

## Implementation unit foo

```
1 module math:foo;
2
3 S f() { ... }
```

## Implementation unit bar

```
1 module math:bar;
2
3 std::string b() { ... }
```



## 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`
  - Note that you need to prefix with `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.

```
1 import std; // everything C++
2 #include <cassert> // for assert()
3 #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.)



## 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

```
1 #include <set>
2 import <vector>;
3
4 #include "header.hpp"
5 import "otherheader.hpp";
```



## 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

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



# Automatic include to import translation

## 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"
...
```

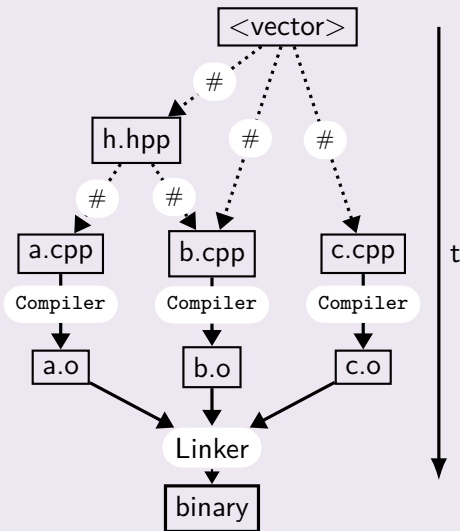
b.cpp

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

c.cpp

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

## Traditional workflow





## Build system

C++ 20 Adv

h.cpp

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

a.cpp

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

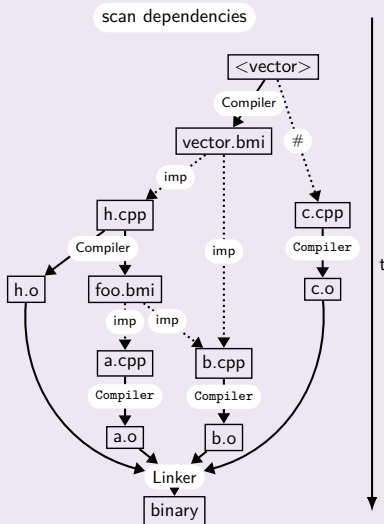
b.cpp

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

c.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



## 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



## 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)



## 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)



## 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](#)



### 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 `#includes` into header unit `imports`



# Coroutines

## 5 Expert C++

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





# Why do we need coroutines?

C++ 20 Adv

## The generator use case (one of several)

In python one can write

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

What about it in C++?

```
3 someType range(int first, int last) { ... }
4 for (auto i : range(0, 10)) {
5 std::cout << i << "\n";
6 }
```

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!



Interesting part of the implementation (see [cpreference](#))

```
1 template<std::integral T>
2 Generator<T> range(T first, const T last) {
3 while (first < last) {
4 co_yield first++;
5 }
6 }
7 for (const char i : range(65, 91)) {
8 std::cout << i << ' ';
9 }
10 std::cout << '\n';
```

## The bad news

- class `Generator` is not provided by the STL
- and is  $\sim 80$  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

## 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`



## Definition of an empty coroutine

```
3 #include <coroutine>
4 struct Task {
5 struct promise_type {
6 Task get_return_object() { return {}; }
7 std::suspend_never initial_suspend() { return {}; }
8 std::suspend_never final_suspend() noexcept { return {}; }
9 void return_void() {}
10 void unhandled_exception() { throw; }
11 };
12 };
13 Task myCoroutine() { co_return; }
14 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

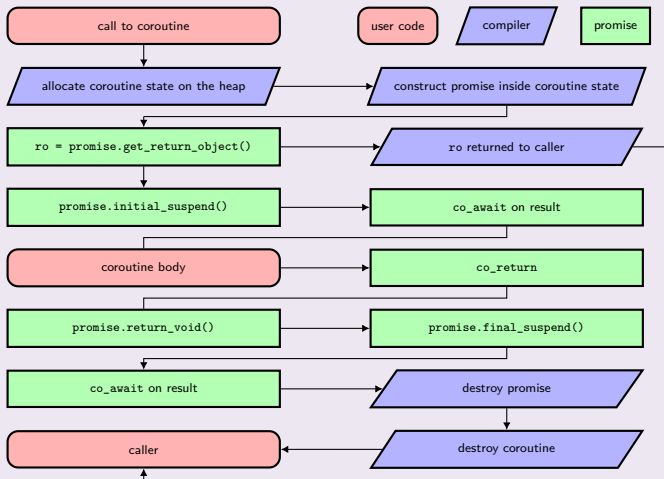


## Minimal sequence of events (nothing suspended)

C++ 20

Adv

## Compiler generated set of calls when calling myCoroutine

Credits: original idea for this graph comes from [this web site](#)

# Suspending a coroutine

## 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

```
1 Task myCoroutine() {
2 std::cout << "Step 1 of coroutine\n";
3 co_await std::suspend_always{};
4 std::cout << "Step 2 of coroutine\n";
5 co_await std::suspend_always{};
6 std::cout << "final step\n";
7 }
```



# Resuming a coroutine

C++ 20 Adv

## 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

```
1 struct Task {
2 struct promise_type {
3 Task get_return_object() {
4 // build Task with the coroutine handle
5 return {std::coroutine_handle<promise_type>::from_promise(*this)};
6 }
7 ... // rest is unchanged
8 };
9 std::coroutine_handle<promise_type> h_;
10 void resume() { h_.resume(); }
11 };
```



# Resuming a coroutine

## User code - godbolt

```
1 Task myCoroutine() {
2 std::cout << "Step 1 of coroutine\n";
3 co_await std::suspend_always{};
4 std::cout << "Step 2 of coroutine\n";
5 co_await std::suspend_always{};
6 std::cout << "final step\n";
7 }
8 int main() {
9 auto c = myCoroutine(); // Step 1 runs
10 std::cout << "In main, between Step 1 and 2\n";
11 c.resume(); // Step 2 runs
12 std::cout << "In main, between Step 2 and final step\n";
13 c.resume(); // final Step runs
14 // c.resume(); // would segfault!
15 }
```

```
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

## 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 Adv

## 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

```
1 template<typename T> struct Task {
2 struct promise_type {
3 T value_;
4 std::suspend_always yield_value(T&& from) {
5 value_ = std::forward<T>(from); // caching the result
6 return {};
7 }
8 ... // rest is unchanged
9 };
10 T getValue() { return h_.promise().value_; } // accessor for user code
11 };
```



## Generator behavior - more boiler plate code

```
1 template<typename T>
2 struct Task {
3 struct iterator {
4 std::coroutine_handle<promise_type> handle;
5 auto &operator++() {
6 handle.resume();
7 if (handle.done()) { handle = nullptr; } // == end iterator
8 return *this;
9 }
10 auto operator++(int) { ++*this; }
11 T const& operator*() const { return handle.promise().value_; }
12 ...
13 };
14 iterator begin() { resume(); return {h_}; }
15 iterator end() { return {nullptr}; }
16 ... // plus error handling is missing
17 };
```



Finally, we can use the generator nicely - `godbolt`

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



## co\_yield - final usage

## Finally, we can use the generator nicely - godbolt

```

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

```

10 struct Task {
11 struct promise_type {
12 ...
13 std::suspend_always initial_suspend() { return {}; }
14 ...
15 }
16 };

```



## Laziness allows for infinite ranges

## Generators do not need to be finite - godbolt

```
1 Task range(int first) {
2 while (true) {
3 co_yield first++;
4 }
5 }
6 for (int i : range(0) | std::views::take(10)) {
7 std::cout << i << "\n";
8 }
```



## Real life example

A Fibonacci generator, from [cppreference](#) - [godbolt](#)

```
1 Task<long long int> fibonacci(unsigned n) {
2 if (n==0) co_return;
3 co_yield 0;
4 if (n==1) co_return;
5 co_yield 1;
6 if (n==2) co_return;
7 uint64_t a=0;
8 uint64_t b=1;
9 for (unsigned i = 2; i < n; i++) {
10 uint64_t s=a+b;
11 co_yield s;
12 a=b;
13 b=s;
14 }
15 }
16
17 int main() {
18 for (int i : fibonacci(10)) {
19 std::cout << i << "\n";
20 }
21 }
```



## awaitables vs awaiters

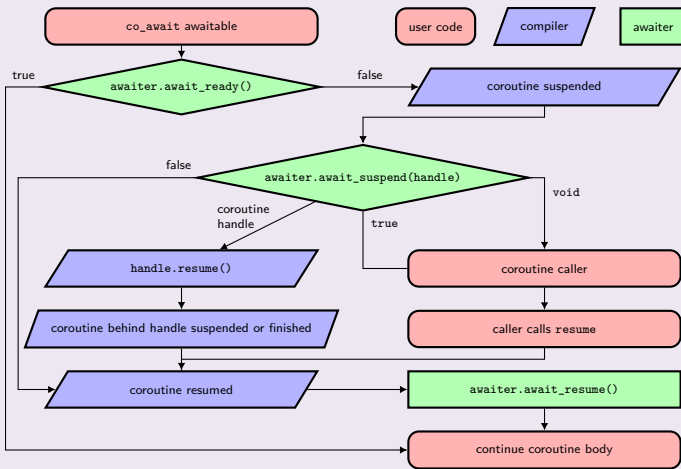
- awaitable is the type required by `co_await/co_yield`
  - or the expression given has to be convertible to awaitable
  - alternatively the `promise_type` can have an `await_transform` method returning an awaitable
- an awaiter is retrieved from the awaitable by the compiler
  - either by calling operator `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

## Compiler generated set of calls



Switch to new thread, from `cpreference` - godbolt

```
1 auto switch_to_new_thread(std::jthread& out) {
2 struct awaiter {
3 std::jthread* th;
4 bool await_ready() { return false; }
5 void await_suspend(std::coroutine_handle<> h) {
6 *th = std::jthread([h] { h.resume(); });
7 std::cout << "New thread ID: " << th->get_id() << '\n';
8 }
9 void await_resume() {}
10 };
11 return awaiter{&out};
12 }
13 Task<int> resuming_on_new_thread(std::jthread& out) {
14 std::cout << "Started on " << std::this_thread::get_id() << '\n';
15 co_await switch_to_new_thread(out);
16 std::cout << "Resumed on " << std::this_thread::get_id() << std::endl;
17 }
18 int main() {
19 std::jthread out;
20 resuming_on_new_thread(out);
21 }
```



## awaiter example

Switch to new thread, from `cpreference` - godbolt

```

1 auto switch_to_new_thread(std::jthread& out) {
2 struct awaiter {
3 std::jthread* th;
4 bool await_ready() { return false; }
5 void await_suspend(std::coroutine_handle<> h) {
6 *th = std::jthread([h] { h.resume(); });
7 std::cout << "New thread ID: " << th->get_id() << '\n';
8 }
9 void await_resume() {}
10 };
11 return awaiter{&out};
12 }
13 Task<int> resuming_on_new_thread(std::jthread& out) {
14 std::cout << "Started on " << std::this_thread::get_id() << '\n';
15 co_await switch_to_new_thread(out);
16 std::cout << "Resumed on " << std::this_thread::get_id() << std::endl;
17 }
18 int main() {
19 std::jthread out;
20 resuming_on_new_thread(out);
21 }

```

## Output

```

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

```



## 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

- 1 History and goals
- 2 Language basics
- 3 Object orientation (OO)
- 4 Core modern C++
- 5 Expert C++
- 6 Useful tools**
  - C++ editor
  - Version control
  - Code formatting
  - The Compiling Chain
  - Web tools
  - Debugging
  - Sanitizers
  - The Valgrind family
  - Static code analysis
  - Profiling
  - Doxygen
- 7 Concurrency
- 8 C++ and python



## 6 Useful tools

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



# 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

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





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



# clang-format

## .clang-format

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

## Run clang-format

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



# clang-format

## Exercise: clang-format

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



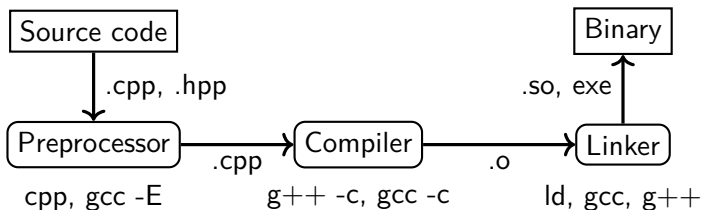
# The Compiling Chain

- 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

C++ 17



## 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
0000000000000008a T SlowToCopy::SlowToCopy(SlowToCopy const&)
00000000000000000 T SlowToCopy::SlowToCopy()
00000000000000064 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

```
1 cmake_minimum_required(VERSION 3.18)
2 project(hello CXX)
3
4 find_package(ZLIB REQUIRED) # for external libs
5
6 add_executable(hello main.cpp util.h util.cpp)
7 target_compile_features(hello PUBLIC cxx_std_17)
8 target_link_libraries(hello PUBLIC ZLIB::ZLIB)
```



# CMake - Building

## Building a CMake-based project

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

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



# 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:

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

## Typical usage

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





## Concept

Reveals the actual code behind C<sup>++</sup> syntactic sugar

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

## Typical usage

- understand how things work behind the C<sup>++</sup> 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

The screenshot shows the CppInsights web interface. The top bar includes navigation icons and the text "[C++ Standard: C++ 20]". The main content is split into two panels: "Source" and "Insight".

**Source:**

```

1 #include <array>
2 #include <iostream>
3
4 int main() {
5 std::array<int, 5> arr{2,4,6,8,10};
6 for(auto const & c : arr) {
7 std::cout << "c" << c << "\n";
8 }
9 }

```

**Insight:**

```

1 #include <array>
2 #include <iostream>
3
4 int main()
5 {
6 std::array<int, 5> arr = {{2, 4, 6, 8, 10}};
7 {
8 std::array<int, 5> & __range1 = arr;
9 int * __begin1 = __range1.begin();
10 int * __end1 = __range1.end();
11 for(__begin1 != __end1; ++ __begin1) {
12 const std::array<int, 5>::value_type & c = * __begin1;
13 std::operator<<(std::operator<<(std::cout, "c="),operator<<(c), "\n");
14 }
15 }
16 }
17 return 0;
18 }
19

```

At the bottom, the "Console" section shows the message: "Insights exited with result code: 0".



# Debugging

## 6 Useful tools

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



# 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



## Exercise: gdb

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



# Debugging UIs

## User interfaces for debuggers

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

**Native gdb** Try “tui enable” for a simple built-in UI

▶ **VSCode** Built-in support for gdb

▶ **CodeLLDB** VS Code plugin for LLDB

▶ **GDB dashboard** Poplar terminal UI for gdb

▶ **GEF** Modern terminal UI for gdb

- some editors and most IDEs have good debugger integration



# Sanitizers

## 6 Useful tools

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



# 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 ( $\sim 2x$  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)

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



# Address Sanitizer (ASan)

## How it works

- Compiler adds run-time checks ( $\sim 2x$  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)

```
1 if (IsPoisoned(address)) {
2 ReportError(address, kAccessSize, kIsWrite);
3 }
4 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)

```
1 void foo() {
2 char a[8];
3 char b[8];
4 a[8] = '1';
5 }
```

### Memory layout



## 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

		0x300C
b[0-7]	redzone3	0x3008
a[0-7]	redzone2	0x3004
	redzone1	0x3000



```
==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:
```

```
=>0x1ffffdd27db20: 00 00 00 00 00 00 00 00 00 f1 f1 f1 f1 00[f2]f2 f2
```

```
0x1ffffdd27db30: 00 f3 f3 f3 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0x1ffffdd27db40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
0x1ffffdd27db50: 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 (“LeakSanitizer”)
- Might be active already, otherwise 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/contrib/asan/asan.cpp:33
#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/contrib/asan/asan.cpp:33
#1 0x403846 in void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::basic_string(char const*) /usr/include/c++/string:113
#2 0x4033f4 in std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::basic_string(char const*) /usr/include/c++/string:113
#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>





# Undefined Behaviour Sanitizer (UBSan)

## Exercise: Finding evil run-time bugs

- Go to `exercises/ubsan`
- Compile and run the program following the instructions in README or in the program
- The program should run without observable issues
- Recompile with UBSan and see that almost every second line contains evil bugs



# 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 biased
- 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 `fibonacci` call to `fibonacci2`
- 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

## 6 Useful tools

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



# 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

```
1 /** 10 ///
2 * text/directives ... 11 /// text/directives ...
3 */ 12 ///
4 some_cpp_entity 13 some_cpp_entity
5
6 /*! 15 ///
7 * text/directives ... 16 /// text/directives ...
8 */ 17 ///
9 some_cpp_entity 18 some_cpp_entity
```

- More details available [here](#)

## Comment blocks for members

```
int a_class_member; /// trailing documentation
```



# Doxygen

## Comment directives

```

1 /**
2 * Long description here. May include HTML etc.
3 *
4 * \brief Checks whether i is odd
5 * \tparam Integral The integral type of the input
6 * \param i Input value
7 * \return True if i is odd, otherwise false
8 * \throw std::out_of_range if i is larger than 100
9 * \see isEven
10 */
11 template <typename Integral>
12 bool isOdd(Integral i);

```

- All directives can also start with @ instead of \
- A list of all commands can be found [here](#)



# Concurrency

- 1 History and goals
- 2 Language basics
- 3 Object orientation (OO)
- 4 Core modern C++
- 5 Expert C++
- 6 Useful tools
- 7 Concurrency**
  - Threads and async
  - Mutexes
  - Atomic types
  - Thread-local storage
  - Condition Variables
- 8 C++ and python



# Threads and async

## 7 Concurrency

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



## 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

## Example code

```
1 void foo() {...}
2 void bar() {...}
3 int main() {
4 std::thread t1{foo};
5 std::thread t2{bar};
6 for (auto t: {&t1,&t2}) t->join();
7 return 0;
8 }
```



# The thread constructor

Can take a function and its arguments

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



# The thread constructor

## Can take a function and its arguments

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

## Can take any function-like object

```
1 struct AdderFunctor {
2 AdderFunctor(int i): m_i(i) {}
3 int operator() (int j) const { return m_i+j; }
4 int m_i;
5 };
6 std::thread t2{AdderFunctor{2}, 5};
7 int a;
8 std::thread t3{[] (int i) { return i+2; }, a};
9 std::thread t4{[a] { return a+2; }};
```





Good practice: Prefer `std::jthread` if C++ 20 is available

- In `<thread>` header since C++ 20
- Automatically joins on destruction
- Can be stopped from outside

Example with `jthread`

```
1 void foo() {...}
2 void bar() {...}
3 int main() {
4 std::jthread t1{foo};
5 std::jthread t2{bar}; t2.request_stop();
6 // No join required
7 return 0;
8 }
```



Using `jthread`'s `stop_token`

- Like `std::thread`, `jthread` can take a function or functors/lambda and their arguments
- New: Can take functions that handle `stop_tokens`

## stop\_token in a function

```

1 void function(std::stop_token st, int j) {
2 while (!st.stop_requested()) {
3 // ...
4 }
5 std::jthread t1{function, 1};
6 t1.request_stop();

```

## A lambda with a stop\_token

```

1 std::jthread t1{
2 [](std::stop_token st, int i){ ... }, 1};

```



## 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”



## 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



## 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

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



## 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!



## 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 - gdbolt

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



## Fine-grained control on asynchronous execution C++ 11 Adv

`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 Adv

## 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

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



# Mutexes

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



## Example code - godbolt

```
1 int a = 0;
2 void inc() { a++; };
3 void inc100() {
4 for (int i=0; i < 100; i++) inc();
5 };
6 int main() {
7 std::thread t1{inc100};
8 std::thread t2{inc100};
9 for (auto t: {&t1,&t2}) t->join();
10 std::cout << a << "\n";
11 }
```



## Example code - godbolt

```
1 int a = 0;
2 void inc() { a++; };
3 void inc100() {
4 for (int i=0; i < 100; i++) inc();
5 };
6 int main() {
7 std::thread t1{inc100};
8 std::thread t2{inc100};
9 for (auto t: {&t1,&t2}) t->join();
10 std::cout << a << "\n";
11 }
```

What do you expect? (Exercise exercises/race)



## Example code - godbolt

```
1 int a = 0;
2 void inc() { a++; };
3 void inc100() {
4 for (int i=0; i < 100; i++) inc();
5 };
6 int main() {
7 std::thread t1{inc100};
8 std::thread t2{inc100};
9 for (auto t: {&t1,&t2}) t->join();
10 std::cout << a << "\n";
11 }
```

What do you expect? (Exercise exercises/race)

Anything between 100 and 200 !!!



## 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



## 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 ?



## 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 :

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



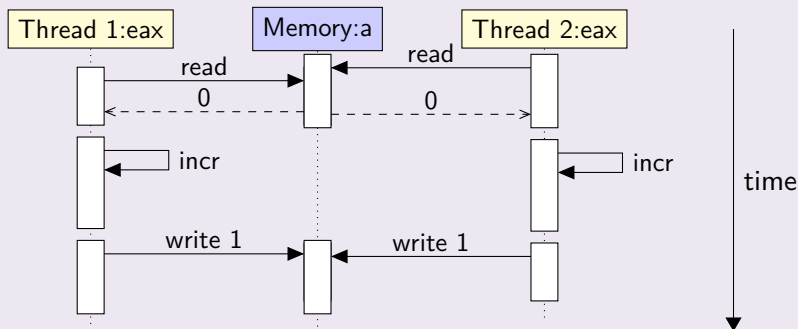


## Timing

## Code

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

## For 2 threads



# Mutexes and Locks

## Concept

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



# Mutexes and Locks

## Concept

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

## The objects

`std::mutex` in the `mutex` header. **Mutual exclusion**

`std::scoped_lock` RAII to lock and unlock automatically

`std::unique_lock` same, but can be released/relocked explicitly



# Mutexes and Locks

## Concept

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

## The objects

`std::mutex` in the `mutex` header. **Mutual exclusion**

`std::scoped_lock` RAII to lock and unlock automatically

`std::unique_lock` same, but can be released/relocked explicitly

## Practically - godbolt

```
1 int a = 0;
2 std::mutex m;
3 void inc() {
4 std::scoped_lock lock{m};
5 a++;
6 }
```



# Mutexes and Locks

## 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`

```
1 void function(...) {
2 // uncritical work ...
3 {
4 std::scoped_lock myLocks{mutex1, mutex2, ...};
5 // critical section
6 }
7 // uncritical work ...
8 }
```



# 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

## Scenario

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

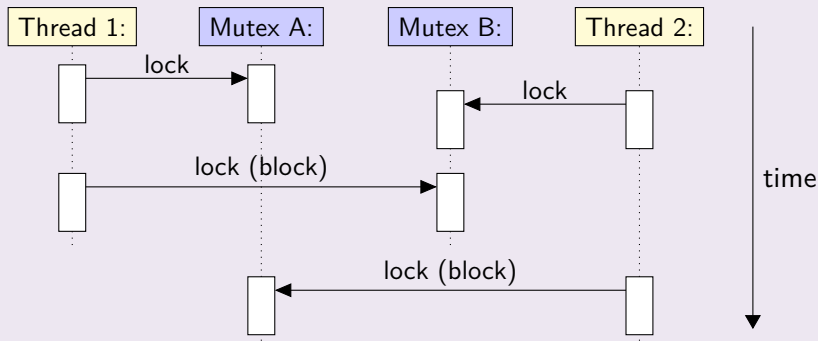


# Dead locks

## Scenario

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

## Sequence diagram





## 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

## 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

```
1 Data data; std::shared_mutex mutex;
2 auto reader = [&]() {
3 std::shared_lock lock{mutex};
4 read(data); // Many can read
5 };
6 std::thread r1{reader}, r2{reader}, ...;
7
8 std::thread writer([&]() {
9 std::scoped_lock lock{mutex}; // exclusive
10 modify(data); // Only one can write
11 });
```



## Parallel writing to streams

- Parallel writes to streams become garbled when not synchronised
- `std::osyncstream` wraps and synchronises output to streams
- Multiple instances of the `osyncstream` synchronise globally

```
1 void worker(const int id, std::ostream & os);
2 void func() {
3 std::osyncstream synccout1{std::cout};
4 std::osyncstream synccout2{std::cout};
5 std::jthread t1{worker, 0, std::ref(synccout1)};
6 std::jthread t2{worker, 1, std::ref(synccout2)};
7 }
```



# Atomic types

## 7 Concurrency

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



`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

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



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

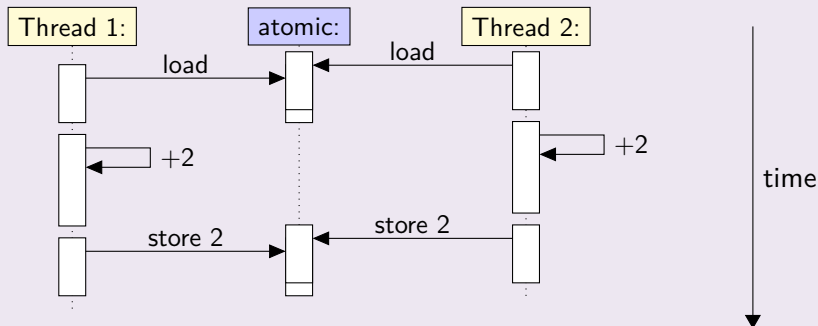
```

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

```

- Atomic load; value+2; atomic store

## Sequence diagram



## Solution: Use atomic member functions

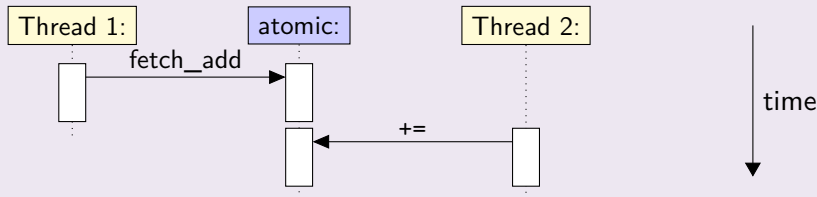
- The member functions of `std::atomic` are thread safe
- `fetch_add` and `operator+=`: `atomic { load; add; store }`
- But don't confuse "a += 2" and "a = a + 2"

```

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

```

## Sequence diagram



# Atomic references

## std::atomic\_ref template

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

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

## Don't mix concurrent atomic and non-atomic access

```
1 std::thread t3([&]{ std::atomic_ref{a}++; });
2 a += 2; // data race
3 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` 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

```
1 thread_local int a{0};
2 {
3 std::jthread t1([&] { a++; });
4 std::jthread t2([&] { a++; });
5 a += 2;
6 }
7 assert(a == 2); // Guaranteed to succeed
```



# Condition Variables

## 7 Concurrency

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



## 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



## 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

## 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

```
1 std::mutex mutex;
2 std::condition_variable cond;
3 Data data;
4 std::thread producer([&](){
5 {
6 std::scoped_lock lock{mutex};
7 data = produceData(); // may take long ...
8 }
9 cond.notify_all();
10 });
```



# Using condition variables: wait

## Mechanics of wait

- Many threads can wait for shared data
- Pass `unique_lock` and optional predicate to `wait()`
- `wait()` keeps threads asleep while predicate is `false`
  - Threads might wake up spuriously, but `wait()` returns only when lock available *and* predicate `true`
- `wait()` will only lock when necessary; unlocked while sleeping

## Naïve waiting

```
1 auto processData = [&]() {
2 std::unique_lock<std::mutex> lock{mutex};
3 cond.wait(lock, [&]() { return data.isReady(); });
4 process(data);
5 };
```





# Using condition variables: wait

## Waiting / waking up

- When `notify_all()` is called, threads wake up
- Threads try to lock mutex, and evaluate predicate
- One thread succeeds to acquire mutex, starts processing data
- **Problem:** One thread holds mutex, other threads are blocked!

## Naïve waiting

```
1 auto processData = [&]() {
2 std::unique_lock<std::mutex> lock{mutex};
3 cond.wait(lock, [&]() { return data.isReady(); });
4 process(data);
5 };
```



## 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

```

1 auto processData = [&]() {
2 {
3 std::unique_lock<std::mutex> lock{mutex};
4 cond.wait(lock, [&]() { return data.isReady(); });
5 }
6 process(data);
7 };
8 std::thread t1{processData}, t2{processData}, ...;
9 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

- 1 History and goals
- 2 Language basics
- 3 Object orientation (OO)
- 4 Core modern C++
- 5 Expert C++
- 6 Useful tools
- 7 Concurrency
- 8 C++ and python
  - Writing a module
  - Marrying C++ and C
  - The ctypes module
  - The cppy project



# 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

```
1 int mandel(Complex const & a);
```



## Basic Module(1): wrap your method

mandelModule.cpp - see exercises/python exercise

```

1 #include <Python.h>
2 #include "mandel.hpp"
3 PyObject * mandel_wrapper(PyObject * self,
4 PyObject * args) {
5 // Parse Input
6 float r, i;
7 if (!PyArg_ParseTuple(args, "ff", &r, &i))
8 return nullptr;
9 // Call C++ function
10 int result = mandel(Complex(r, i));
11 // Build returned objects
12 return PyLong_FromLong(result);
13 }

```



## Basic Module(2): create the python module

mandelModule.cpp - see exercises/python exercise

```

1 // declare the modules' methods
2 PyMethodDef mandelMethods[] = {
3 {"mandel", mandel_wrapper, METH_VARARGS,
4 "computes nb of iterations for mandelbrot set"},
5 {nullptr, nullptr, 0, nullptr}
6 };
7 // declare the module
8 struct PyModuleDef mandelModule = {
9 PyModuleDef_HEAD_INIT,
10 "mandel", nullptr, -1, mandelMethods
11 };
12 PyMODINIT_FUNC PyInit_mandel() {
13 return PyModule_Create(&mandelModule);
14 }

```





# 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<sup>++</sup> and C

- 8 C<sup>++</sup> and python
  - Writing a module
  - Marrying C<sup>++</sup> and C
  - The ctypes module
  - The cppyy project



# 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<sup>++</sup> symbol names

- C uses bare function name
- C<sup>++</sup> allows overloading of functions by taking the signature into account
- so C<sup>++</sup> mangling has to contain signature



# C mangling

Source : file.c

```
1 float sum(float a, float b);
2 int square(int a);
3 // won't compile : conflicting types for 'square'
4 // float square(float a);
```

Binary symbols : file.o

```
nm file.o
0000000000000001a T square
00000000000000000 T sum
```



# C++ mangling

Source : file.cpp

```
1 float sum(float a, float b);
2 int square(int a);
3 // ok, signature is different
4 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 :

```
1 extern "C" {
2 float sum(float a, float b);
3 int square(int a);
4 }
```



# Forcing C mangling in C++

```
extern "C"
```

These functions will use C mangling :

```
1 extern "C" {
2 float sum(float a, float b);
3 int square(int a);
4 }
```

You can now call these C++ functions from C code



# Forcing C mangling in C++

```
extern "C"
```

These functions will use C mangling :

```
1 extern "C" {
2 float sum(float a, float b);
3 int square(int a);
4 }
```

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

```
1 int mandel(Complex const & a);
```

"C" code : mandel\_cwrapper.hpp

```
1 extern "C" {
2 int mandel(float r, float i) {
3 return mandel(Complex(r, i));
4 };
5 }
```

calling the mandel library

```
from ctypes import *
libmandel = CDLL('libmandelc.so')
v = libmandel.mandel(c_float(0.3), c_float(1.2))
```



# Marrying C<sup>++</sup> and python

## Exercise: C<sup>++</sup> 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 cppy project

- 8 C++ and python
  - Writing a module
  - Marrying C++ and C
  - The ctypes module
  - The cppy 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



# cpyyy crash course(1)

Shamelessly copied from the cppy documentation

```
>>> import cppy
>>> cppy.include('zlib.h') # bring in C++ definitions
>>> cppy.load_library('libz') # load linker symbols
>>> cppy.gbl.zlibVersion() # use a zlib API
'1.2.11'
```

```
>>> import cppy
>>> cppy.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
```



# cpyyy crash course(1)

```
>>> from cppy.gbl import MyClass
>>> m = MyClass(42)
>>> cppy.cppdef("""
... void say_hello(MyClass* m) {
... std::cout << "Hello, the number is: " << m->m_data << std::endl;
... }""")
True
>>> MyClass.say_hello = cppy.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/raw/download/talk/C++Course_full.pdf)  
<https://github.com/hsf-training/cpluspluscourse>



# Index of Good Practices

1	C's memory management	27	17	Operator overloading	171
2	Manual memory management	28	18	Throwing exceptions	190
3	Initialisation	31	19	Catching exceptions	192
4	Avoid unions	38	20	Exceptions	196
5	References	45	21	noexcept	199
6	Write readable functions	54	22	Specialization vs. overloading	232
7	for syntax	67	23	STL and lambdas	263
8	preprocessor	75	24	C random library	294
9	Header / source	77	25	Use <code>std::fstream</code>	305
10	Assert	87	26	<code>std::unique_ptr</code>	311
11	Implementing methods	94	27	Single responsibility principle	317
12	Rule of 3/5	109	28	Rule of zero	317
13	Prefer smart pointer	127	29	Initialization	327
14	Prefer containers	128	30	SFINAE vs. <code>if constexpr</code>	378
15	Virtual destructors	141	31	<code>jthread</code> vs. <code>thread</code>	502
16	Avoid multiple inheritance	148	32	Locking	512



# Index of Exercises

1	Functions	53	12	std::variant	281	23	valgrind	479
2	Control structs	72	13	Smart pointers	318	24	memcheck	480
3	Loops, refs, auto	80	14	Variadic tpl	349	25	callgrind	482
4	Polymorphism	144	15	Concepts	391	26	helgrind	484
5	Virtual OO	149	16	Modules	416	27	cppcheck	487
6	Operators	160	17	Header units	416	28	clang-tidy	490
7	Constness	178	18	clang-format	444	29	Mutexes/Locks	513
8	Move semantics	214	19	Compiler chain	454	30	Atomics	523
9	Templates	236	20	gdb	463	31	Condition vars	531
10	STL	268	21	address sanitizer	472	32	C++ and python	546
11	std::optional	276	22	Finding evil run-time bugs	475			



## Index of Godbolt code examples

1	assert - godbolt	86	20	shared_ptr quiz - godbolt	314
2	Unary constructor - godbolt	111	21	shared/weak_ptr - godbolt	316
3	casting - godbolt	152	22	<=> output - godbolt	333
4	function objects - godbolt	163	23	various orderings - godbolt	335
5	Nested templates - godbolt	224	24	tuple code - godbolt	345
6	CTAD - godbolt	233	25	hasFoo failing - godbolt	370
7	lambda - godbolt	249	26	hasFoo working - godbolt	372
8	lambda C++ 20 - godbolt	250	27	enable_if - godbolt	375
9	STL - godbolt	253	28	SFINAE failure - godbolt	381
10	Iterator example - godbolt	259	29	concept failure - godbolt	382
11	span - godbolt	273	30	Coroutine resumption - godbolt	425
12	expected - godbolt	277	31	co_yield - godbolt	429
13	variant visitor - godbolt	279	32	Infinite generator - godbolt	430
14	variant $\lambda$ visitor - godbolt	280	33	Fibonacci generator - godbolt	431
15	Structured binding - godbolt	286	34	Thread switching - godbolt	434
16	timing - godbolt	289	35	std::async - godbolt	505
17	Views - godbolt	297	36	Race - godbolt	508
18	Lazy view - godbolt	299	37	mutex - godbolt	511
19	unique_ptr copy - godbolt	307			

# 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](http://cppcon.org) —  CppCon
- C++ Now — [cppnow.org](http://cppnow.org) —  BoostCon
- Code::Dive — [codedive.pl](http://codedive.pl) —  codediveconference
- ACCU Conference — [accu.org](http://accu.org) —  ACCUConf
- Meeting C++ — [meetingcpp.com](http://meetingcpp.com) —  MeetingCPP
- See link below for more information  
<https://isocpp.org/wiki/faq/conferences-worldwide>

