

# HEP C++ course

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

CERN

March 10, 2023



# Foreword

## What this course is not

- It is not for absolute beginners
- It is not for experts
- It is not complete at all (would need 3 weeks...)
  - although it is already too long for the time we have
  - 248 slides, 342 pages, 13 exercises...

## How I see it

**Adaptative** pick what you want

**Interactive** tell me what to skip/insist on

**Practical** let's spend time on real code

## Where to find latest version ?

- full sources at <https://github.com/hsf-training/cpluspluscourse>
- latest pdf under [raw/download/talk/C++Course\\_full.pdf](raw/download/talk/C++Course_full.pdf)



# 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 Useful tools



# Detailed outline

- 1 History and goals
  - History
  - Why we use it?
- 2 Language basics
  - Core syntax and types
  - Arrays and Pointers
  - Scopes / namespaces
  - Class and enum types
  - References
  - Functions
  - Operators
  - Control structures
- 3 Object orientation (OO)
  - Headers and interfaces
  - Auto keyword
  - Objects and Classes
  - Inheritance
  - Constructors/destructors
  - Static members
  - Allocating objects
  - Advanced OO
  - Operator overloading
  - Function objects
- 4 Core modern C++
- 5 Useful tools
  - Constness
  - Exceptions
  - Templates
  - Lambdas
  - The STL
  - RAI and smart pointers
  - C++ editor
  - Version control
  - Code formatting
  - The Compiling Chain
  - Web tools
  - Debugging



# History and goals

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

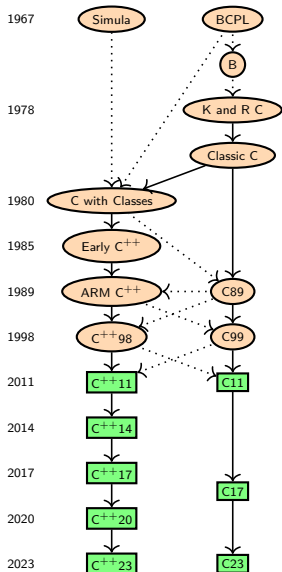


# History

- 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<sup>++</sup>11, C<sup>++</sup>14, C<sup>++</sup>17, C<sup>++</sup>20, C<sup>++</sup>23, C<sup>++</sup>26...

## status

- A new C<sup>++</sup> specification every 3 years
  - C<sup>++</sup>23 complete since 11th of Feb. 2023, awaiting ISO ballot
  - work on C<sup>++</sup>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	>12

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



# Why we use it?

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



# Why is C++ our language of choice?

## Adapted to large projects

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



# Why is C++ our language of choice?

## Adapted to large projects

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

## Fast

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



# Why is C++ our language of choice?

## Adapted to large projects

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

## Fast

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

## What we get

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



# Language basics

## 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

## 3 Object orientation (OO)

## 4 Core modern C++

## 5 Useful tools



# 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





# 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

C++98

```

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

C++98

```

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, 0x4d2.1 * 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

C++98

```
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);
```





# 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



## Static arrays

C++98

```
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

C++98

```
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

C++98

```

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

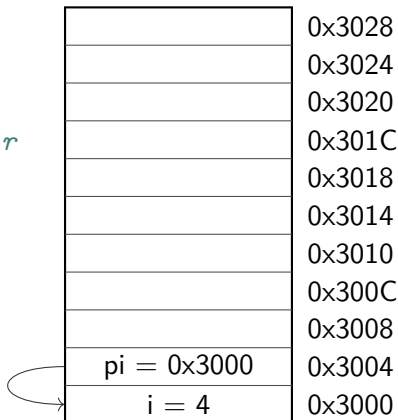


# Pointers

C++98

```
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

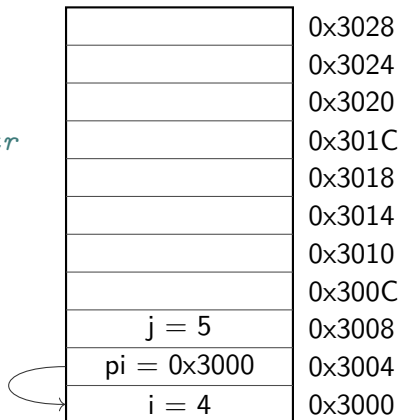
C++98

```

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

C++98

```

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

C++98

```
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
pai = 0x300C	0x3018
ai[2] = 3	0x3014
ai[1] = 2	0x3010
ai[0] = 1	0x300C
j = 5	0x3008
pi = 0x3000	0x3004
i = 4	0x3000





## Pointers

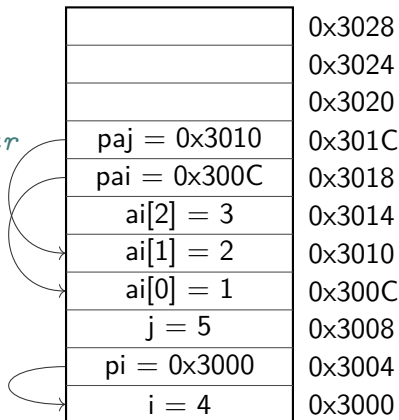
C++98

```

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

C++98

```

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

## Pointers

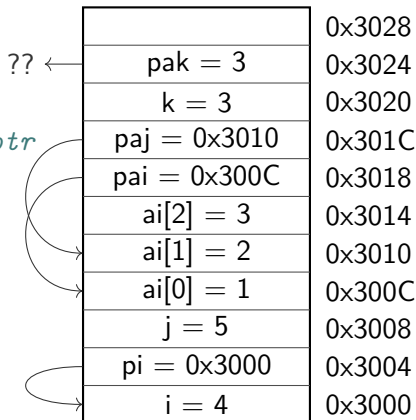
C++98

```

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



# Scopes / namespaces

## 2 Language basics

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



# Scope

C++98

## Definition

Portion of the source code where a given name is valid

Typically :

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

## Example

```
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

C++98

- 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 }
```



# Class and enum types

## 2 Language basics

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



## struct

“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;
```





## struct

C++98

“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



## struct

C++98

“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 };

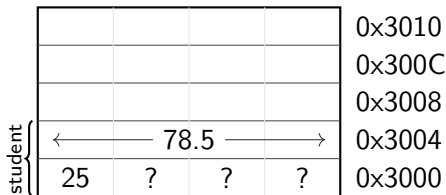
```

```

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

```

## Memory layout



## struct

C++98

“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

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



## struct

C++98

“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 };

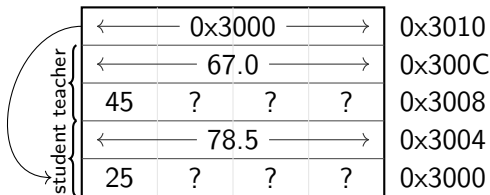
```

```

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

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
				0x3004
				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
				0x3008
				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
				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

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



## 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`



# Enums

- 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

C++11

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



# References

C++98

## References

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

## Example:

```
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



# Functions

C++98

```
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 }
```





# Function default arguments

C++98

```
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

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
{	sn	0x30A0
	...	0x3050
	s1	0x3000



# 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

p	pn = sn	0x31E0
	...	0x3190
	p1 = s1	0x30F0
s	sn	0x30A0
	...	0x3050
	s1	0x3000



# Functions: parameters are passed by value

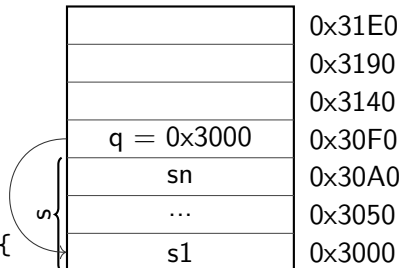
C++98

```

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

	0x3008
	0x3004
	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

	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

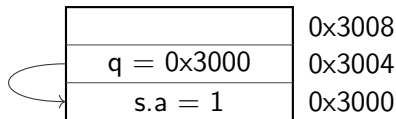


## Functions: pass by value or reference?

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

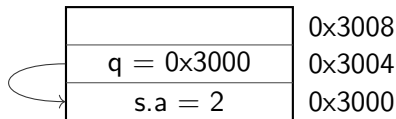


## Functions: pass by value or reference?

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

C++98

## Different ways to pass arguments to a function

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

## Syntax

```
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
- They must have different parameter lists
- A different return type alone is not allowed
- Default arguments can cause ambiguities
- All functions with the same name form an “overload set”

```
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 code/functions
- Look at functions.cpp
- Compile it (make) and run the program (./functions)
- Work on the tasks that you find in functions.cpp



# Functions: good practices

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;
                           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



# 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
```



# 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
```

## 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
```

## 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
```

## Precedences

**Avoid - use parentheses**

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

Details can be found on [cpreference](http://cpreference.com)



# Control structures

## 2 Language basics

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



# Control structures: if

C++98

## 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 emitted  
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  }
```

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



# Control structures: for loop

C++98

## for loop syntax

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

- Initializations and 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  }
```

- Initializations and 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& 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 code/control
- Look at control.cpp
- Compile it (make) and run the program (./control)
- Work on the tasks that you find in README.md



# Headers and interfaces

## 2 Language basics

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



# Headers and interfaces

C++98

## Interface

Set of declarations defining some functionality

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

## Header: hello.hpp

```
void printHello();
```

## Usage: myfile.cpp

```
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
```





# Auto keyword

## 2 Language basics

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



# Auto keyword

C++11

## Reason of being

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

```
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; }
```



# Loops, references, auto

C++98

## Exercise: Loops, references, auto

Familiarise yourself with range-based for loops and references

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



# 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
- Operator overloading
- Function objects

## 4 Core modern C++

## 5 Useful tools



# Objects and Classes

## 3 Object orientation (OO)

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



# What are classes and objects

C++98

## Classes (or “user-defined types”)

C structs on steroids

- with inheritance
- with access control
- including methods/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

C++98

## Header : MyFirstClass.hpp

```
1  #pragma once
2  struct MyFirstClass {
3      int a;
4      void squareA();
5      int sum(int b);
6  };
```

## Implementation : MyFirstClass.cpp

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



# Implementing methods

C++98

## Standard practice

- usually in .cpp, outside of class declaration
- using the class name as “namespace”

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



# this keyword

## How to know an object's address?

- Sometimes we need to pass a reference to ourself to a different entity
- For example to implement operators, see later
- All class methods can use the keyword **this**
  - It returns the address of the current object
  - Its type is T\* in the methods of a struct/class T

```
1 void externalFunc(MyStruct & s);  
2  
3 struct MyStruct {  
4     void invokeExternalFunc() {  
5         externalFunc(*this); // Pass a reference to ourself  
6     }  
7 };
```



# 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)

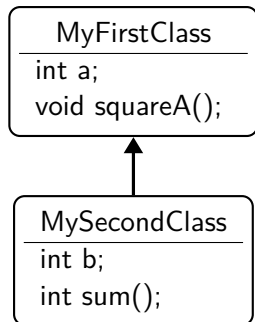
- Objects and Classes
- **Inheritance**
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Operator overloading
- Function objects



## 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
```

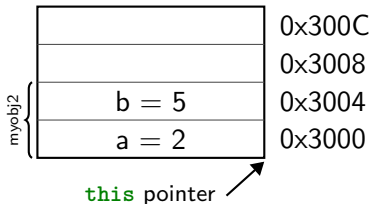


## First inheritance

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

C++98

## public / private keywords

`private` allows access only within the class

`public` allows access from anywhere

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





# Managing access to class members

C++98

## public / private keywords

`private` allows access only within the class

`public` allows access from anywhere

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

```
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`
- A `struct` is just a class that defaults to `public` access

```
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

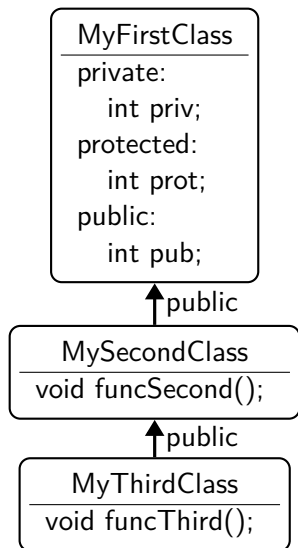
Net result for code in derived classes

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



## Managing inheritance privacy - public

C++98



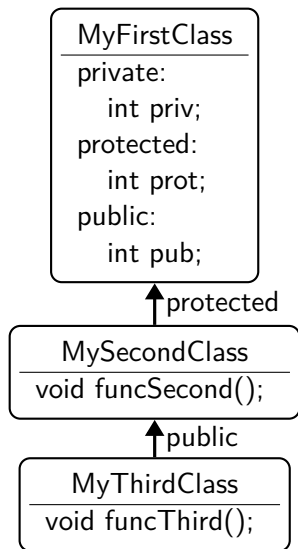
```

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



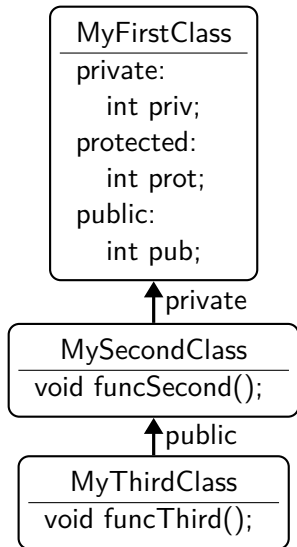
```

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 }
  
```





# Constructors/destructors

## 3 Object orientation (OO)

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



# Class Constructors and Destructors

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



# Class Constructors and Destructors

C++98

```
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

C++98

```
1  struct MyFirstClass {
2      int a;
3      MyFirstClass();
4      MyFirstClass(int a);
5  };
6  struct MySecondClass : MyFirstClass {
7      int b;
8      MySecondClass();
9      MySecondClass(int b);
10     MySecondClass(int a, int b);
11 };
12 MySecondClass::MySecondClass() : MyFirstClass(), b(0) {}
13 MySecondClass::MySecondClass(int b_)
14     : MyFirstClass(), b(b_) {}
15 MySecondClass::MySecondClass(int a_, int b_)
16     : MyFirstClass(a_), b(b_) {}
```



# Copy constructor

C++11

## Concept

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



# Copy constructor

C++11

## Concept

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

```
1  struct MySecondClass : MyFirstClass {  
2      MySecondClass();  
3      MySecondClass(const MySecondClass &other);  
4  };
```



# 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 MySecondClass : MyFirstClass {  
2      MySecondClass();  
3      MySecondClass(const MySecondClass &other);  
4  };
```

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

- if a class has a destructor, a copy/move constructor or a copy/move assignment operator, it should have all three/five. strive for having none by using RAII types as members.



# Class Constructors and Destructors

C++98

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





# Explicit unary constructor

C++98

## Concept

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

## Example - godbolt

```
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

C++98

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

## Practically

```
1  struct BaseClass {  
2      BaseClass(int a);  
3  };  
4  struct DerivedClass : BaseClass {  
5      using BaseClass::BaseClass;  
6      DerivedClass(int a, int b);  
7  };  
8  DerivedClass a{5};
```



# 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 BaseClass {
2      int a{5}; // also possible: int a = 5;
3      BaseClass() = default;
4      BaseClass(int _a) : a(_a) {}
5  };
6  struct DerivedClass : BaseClass {
7      int b{6};
8      using BaseClass::BaseClass;
9  };
10 DerivedClass d{7}; // a = 7, b = 6
```



# Calling constructors

C++11

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

```





## Calling constructors for arrays and vectors

C++11

list of items given within {}

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



## Calling constructors for arrays and vectors

C++11

list of items given within {}

```

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

```

C++98 nightmare

```

10  int ip[3]{1,2,3};           // OK
11  int* ip = new int[3]{1,2,3}; // not allowed
12  std::vector<int> v{1,2,3};  // not allowed

```



# Static members

## 3 Object orientation (OO)

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



# 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

```
1  class Text {
2  public:
3      static std::string upper(std::string) {...}
4  private:
5      static int callsToUpper; // add `inline` in C++17
6  };
7  int Text::callsToUpper = 0; // required before C++17
8  std::string uppers = Text::upper("my text");
9  // now Text::callsToUpper is 1
```



# Allocating objects

## 3 Object orientation (OO)

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



# Process memory organization

C++98

## 4 main areas

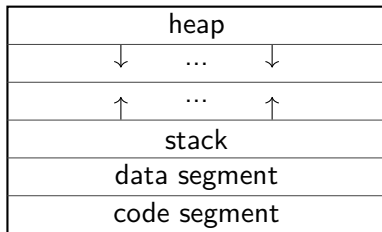
the **code segment** for the machine code of the executable

the **data segment** for global variables

the **heap** for dynamically allocated variables

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

## Memory layout



## Main characteristics

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



# Object allocation on the stack

C++98

## On the stack

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

```
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

C++98

## 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**
- Operator overloading
- Function objects



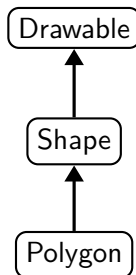
# Polymorphism

C++98

## 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

C++98

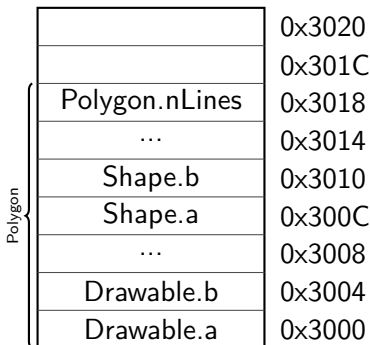
## the concept

- 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

C++98

## 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

C++98

## 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	Shape.b	0x3010
	Shape.a	0x300C
	...	0x3008
	Drawable.b	0x3004
	Drawable.a	0x3000





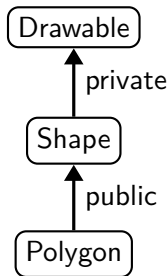
# Inheritance privacy and polymorphism

C++98

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 }
```



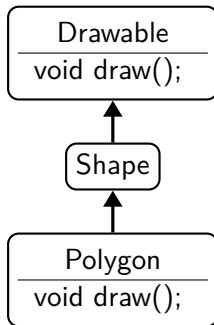
# Method overriding

C++98

## 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(); // ?
```



# Virtual methods

C++98

## the concept

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



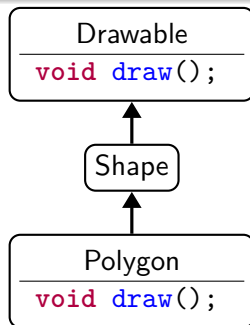
# Virtual methods

C++98

## the concept

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

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



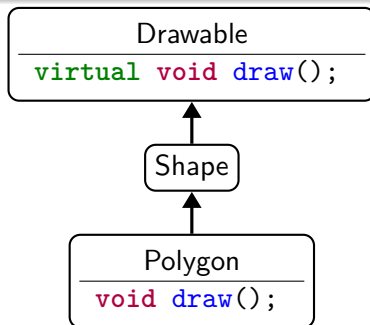
# Virtual methods

C++98

## the concept

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

```
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?

C++11

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





# Pure Virtual methods

C++98

## Concept

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



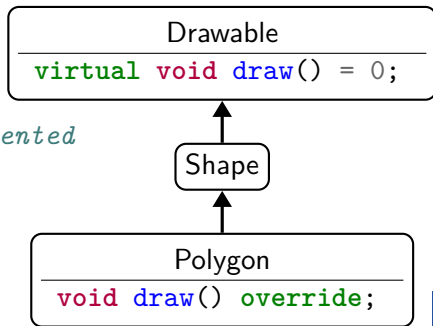
# Pure Virtual methods

C++98

## 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

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 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

C++11

## 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

C++11

## Definition of pure abstract class

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

```

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

```

Drawable
virtual void draw() = 0;



# Overriding overloaded methods

C++98

## Concept

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

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



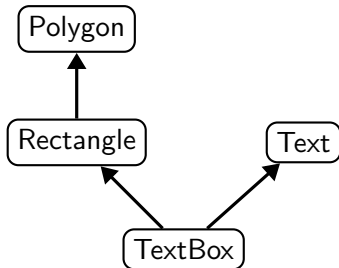


# Multiple Inheritance

C++98

## 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

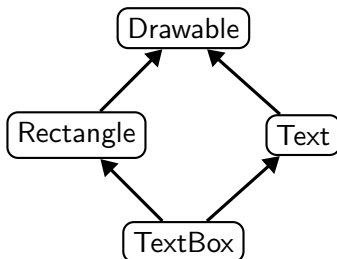
C++98

## Definition

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

## Problem

- are the members of the grand parent replicated?



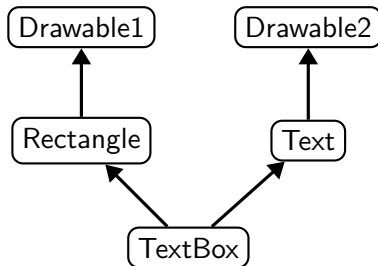
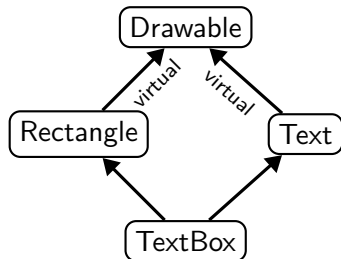
# Virtual inheritance

C++98

## Solution

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

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



# Multiple inheritance advice

C++98

## Good practice: Avoid multiple inheritance

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



# Multiple inheritance advice

C++98

## Good practice: Avoid multiple inheritance

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

## Good practice: Absolutely avoid diamond-shaped inheritance

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



### Exercise: Virtual inheritance

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



# Operator overloading

## 3 Object orientation (OO)

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



# Operator overloading example

C++98

```
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)
```





# Operator overloading

C++98

## Defining operators for a class

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

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



# Why have non-member operators?

C++98

## Symmetry

```
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?

C++98

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



# Function objects

## 3 Object orientation (OO)

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



# Function objects

C++98

## Concept

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

```
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          count += predicate(e);
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

```



# Core modern C<sup>++</sup>

- 1 History and goals
- 2 Language basics
- 3 Object orientation (OO)
- 4 Core modern C<sup>++</sup>
  - Constness
  - Exceptions
  - Templates
  - Lambdas
  - The STL
  - RAI and smart pointers
- 5 Useful tools



# Constness

- 4 Core modern C++
  - Constness
  - Exceptions
  - Templates
  - Lambdas
  - The STL
  - RAI and smart pointers



## 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

C++98

```
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

C++98

## The `const` keyword for member functions

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

```
1  struct Example {  
2      void foo() const {  
3          // type of 'this' is 'Example const*'  
4          data = 0; // Error: member function is const  
5      }  
6      int data;  
7  };
```



# Method constness

C++98

## Constness is part of the type

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

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



### Exercise: Constness

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





# Exceptions

- 4 Core modern C++
  - Constness
  - **Exceptions**
  - Templates
  - Lambdas
  - The STL
  - RAI and smart pointers



## 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 {
2      process_data(f);
3  } catch (const
4      std::out_of_range& e) {
5      std::cerr << e.what();
6  }

7  void process_data(file &f) {
8      ...
9      if (i >= buffer.size())
10         throw std::out_of_range{
11             "buf overflow"};
12     }

```



# Exceptions

## Throwing exceptions

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

## Good practice: Throwing exceptions

- prefer throwing standard exception classes
- throw objects by value

```
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
- ...



# Exceptions

C++98

## Catching exceptions

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

```
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  }
```



# Exceptions

C++98

## 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 }
```



# Exceptions

C++98

## Stack unwinding

- all objects on the stack between a **throw** and the matching **catch** are destructed automatically
- this should cleanly release intermediate resources
- make sure you are using the RAI 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 c5;
14         g();
15         cout << "done"; // not run
16     } catch(const exception&) {
17         // c1, c3 and c5 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](#)



# Exceptions

## A more illustrative example

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

```

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 specifier

C++11

## noexcept

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

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

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

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

## Good practice: noexcept

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



# Templates

- 4 Core modern C++
  - Constness
  - Exceptions
  - **Templates**
  - Lambdas
  - The STL
  - RAI and smart pointers



# Templates

C++17

## Concept

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

```
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);
```

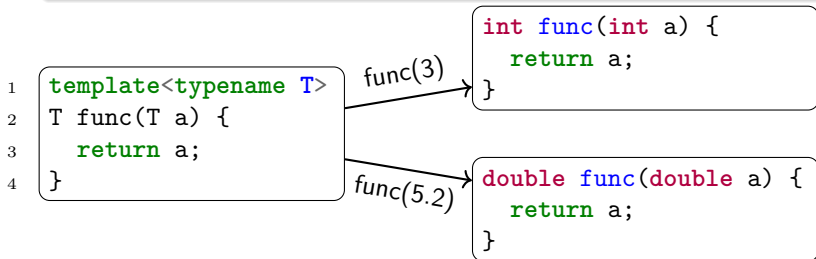


# Templates

C++98

## Warning

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



# Templates

C++98

## 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  Map<std::string, int> m1;
8  Map<float> m2;      // Map<float, float>
9  Map<> m3;          // Map<int, int>
```





# Template parameters

C++98

## typename vs. class keyword

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

```
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 }
```



## 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

C++98

## 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 };
```



# The full power of templates

C++98

## Exercise: Templates

- go to code/templates
- look at the OrderedVector code
- compile and run playwithsort.cpp. See the ordering
- modify playwithsort.cpp and reuse OrderedVector with Complex
- improve OrderedVector to template the ordering
- test reverse ordering of strings (from the last letter)
- test order based on [Manhattan distance](#) with complex type
- check the implementation of Complex
- try ordering complex of complex



# Lambdas

## 4 Core modern C++

- Constness
- Exceptions
- Templates
- **Lambdas**
- The STL
- RAI and smart pointers



# Trailing function return type

C++11

An alternate way to specify a function's return type

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



# Trailing function return type

C++11

An alternate way to specify a function's return type

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

## Advantages

- Allows to simplify inner type definition

```
1 class Class {
2     using ReturnType = int;
3     ReturnType func();
4 }
5 Class::ReturnType Class::func() {...}
6 auto Class::func() -> ReturnType {...}
```

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





# Lambda expressions

C++11

## Definition

a lambda expression is a function with no name



# Lambda expressions

C++11

## Definition

a lambda expression is a function with no name

## Python example

```
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);
```



# Capturing variables

C++11

## Adaptable lambdas

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



# Capturing variables

C++11

## Adaptable lambdas

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

## First attempt in C++

```
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; });
                        ^
```



# Capturing variables

C++11

## The capture list

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



# Capturing variables

C++11

## The capture list

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

## Example

```
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, and the lambda's `operator()` is `const`.



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



# Capture list

C++11

all by value

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



# Capture list

C++11

all by value

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

all by reference

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



# Capture list

C++11

all by value

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

all by reference

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

mix

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

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





# The STL

- 4 Core modern C++
  - Constness
  - Exceptions
  - Templates
  - Lambdas
  - The STL
  - RAI and smart pointers



# The Standard Template Library

C++98

## What it is

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



# The Standard Template Library

C++98

## What it is

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

## Use it

and adapt it to your needs, thanks to templates



## STL in practice

C++14

## STL example - godbolt

```
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(),      // range1
10               in.begin(),                // start range2
11               out.begin(),                // start result
12               std::multiplies{});        // function obj
13  std::copy(out.begin(), out.end(),        // 25 9 16
14            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 cppreference)

- string, string\_view (C++17)
- list, forward\_list (C++11), vector, deque, array (C++11)
- [multi]map, [multi]set (C++23: flat\_[multi]map, flat\_[multi]set)
- unordered\_[multi]map (C++11), unordered\_[multi]set (C++11)
- stack, queue, priority\_queue
- span (C++20)
- non-containers: bitset, pair, tuple (C++11), optional (C++17), variant (C++17), any (C++17), expected (C++23)



## Containers: std::vector

C++11

```
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
```



# STL's concepts

C++11

## iterators

- generalization of pointers
- allow iteration over some data, irrespective of:
  - the container used (templated)
  - the data itself (container is templated)
  - the consumer of the data (templated algorithm)
- examples
  - `std::reverse_iterator`, `std::back_insert_iterator`, ...

## Iterator example - godbolt

```
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

C++11

## 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.



# 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 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 sin(x)/(x + DBL_MIN); });
14
15 // reduce/fold (using addition)
16 const auto sum = std::reduce(v.begin(), v.end());
```



# More examples

C++11

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



# Using the STL

C++98

## Be brave and persistent!

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



# RAII and smart pointers

- 4 Core modern C++
  - Constness
  - Exceptions
  - Templates
  - Lambdas
  - The STL
  - RAI and smart pointers



# 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?

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

```
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

C++11

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?



# Resource Acquisition Is Initialization (RAII)

C++98

## Practically

Use variable construction/destruction and scope semantics:

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



## RAII in practice

C++98

## An 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 RAI pointer

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



## std::unique\_ptr

## A RAI 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);
```



## Quiz

C++11

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

C++14

## std::make\_unique

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



## std::make\_unique

C++14

## std::make\_unique

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

## std::make\_unique usage

```
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
```



# RAII or raw pointers

C++11

## When to use what?

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



# RAII or raw pointers

C++11

## When to use what?

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

## A question of ownership

```
1  std::unique_ptr<T> produce();
2  void observe(const T&);
3  void modifyRef(T&);
4  void modifyPtr(T*);
5  void consumer(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
```





# std::unique\_ptr usage summary

C++11

## Good practice: std::unique\_ptr

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



## std::shared\_ptr

C++11

**std::shared\_ptr** : a reference counting pointer

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

**std::make\_shared** : creates a `std::shared_ptr`

```
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
```



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

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();
```

```
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
```



## Exercise: Smart pointers

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

```
$ valgrind --leak-check=full --track-origins=yes ./smartPointers
```
- In the *essentials course*, go through problem1() and problem2() and fix the leaks using smart pointers.
- In the *advanced course*, go through problem1() to problem4() and fix the leaks using smart pointers.
- problem4() is the most difficult. Skip if not enough time.



# Useful tools

- 1 History and goals
- 2 Language basics
- 3 Object orientation (OO)
- 4 Core modern C++
- 5 Useful tools
  - C++ editor
  - Version control
  - Code formatting
  - The Compiling Chain
  - Web tools
  - Debugging



# C++ editor

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





# C++ editors and IDEs

Can dramatically improve your efficiency by

- Coloring the code for you to “see” the structure
- Helping with indenting and formatting properly
- Allowing you to easily navigate in the source tree
- Helping with compilation/debugging, profiling, static analysis
- Showing you errors and suggestions while typing

▶ Visual Studio Heavy, fully fledged IDE for Windows

▶ Visual Studio Code Editor, open source, portable, many plugins

▶ Eclipse IDE, open source, portable

▶ Emacs ▶ Vim Editors for experts, extremely powerful.

They are to IDEs what latex is to PowerPoint

CLion, Code::Blocks, Atom, NetBeans, Sublime Text, ...

Choosing one is mostly a matter of taste



# Version control

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



# Version control

## Please use one!

- Even locally
- Even on a single file
- Even if you are the only committer

It will soon save your day

## A few tools

- ▶ **git** THE mainstream choice. Fast, light, easy to use
- ▶ **mercurial** The alternative to git
- ▶ **Bazaar** Another alternative
- ▶ **Subversion** Historical, not distributed - don't use
- ▶ **CVS** Archeological, not distributed - don't use



# Git crash course

```
$ git init myProject
```

```
Initialized empty Git repository in myProject/.git/
```

```
$ vim file.cpp; vim file2.cpp
```

```
$ git add file.cpp file2.cpp
```

```
$ git commit -m "Committing first 2 files"
```

```
[master (root-commit) c481716] Committing first 2 files
```

```
...
```

```
$ git log --oneline
```

```
d725f2e Better STL test
```

```
f24a6ce Reworked examples + added stl one
```

```
bb54d15 implemented template part
```

```
...
```

```
$ git diff f24a6ce bb54d15
```



# Code formatting

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



# clang-format

## .clang-format

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

## Run clang-format

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



# clang-format

## Exercise: clang-format

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



# The Compiling Chain

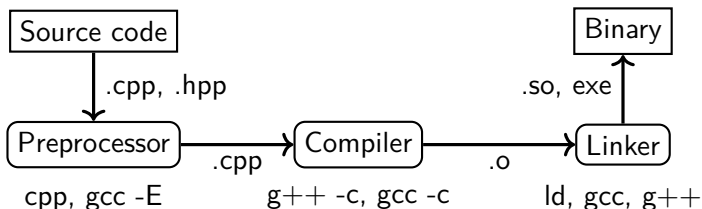
- 5 Useful tools
  - C++ editor
  - Version control
  - Code formatting
  - **The Compiling Chain**
  - Web tools
  - Debugging





# The compiling chain

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

`-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_string
```

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



# Web tools

## 5 Useful tools

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



# Godbolt / Compiler Explorer

## Concept

An online generic compiler with immediate feedback. Allows:

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

## Typical usage

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



# Godbolt by example

## Check effect of optimization flags

<https://godbolt.org/z/Pb8WsWjEx>

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

The screenshot displays the Godbolt Compiler Explorer interface. On the left, the C++ source code is shown for a program that calculates the factorial of a number. The code includes `<iostream>`, defines a `constexpr` function `fact`, and a `main` function that calls `fact` and prints the result.

The right pane shows the assembly output for two optimization levels: `-O0` (unoptimized) and `-O3` (highly optimized). The `-O0` assembly is verbose, showing many instructions for stack management and function calls. The `-O3` assembly is significantly shorter and more efficient, demonstrating the impact of optimization flags.

At the bottom, the output window shows the program's execution results: "ASM generation compiler returned: 0", "Execution build compiler returned: 0", and "Program returned: 0".

## 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 displays the CppInsights interface with a dark theme. At the top, there are navigation icons and a header with 'C++ Standard: C++ 20'. The main content is split into two panels: 'Source' on the left and 'Insight' on the right. The 'Source' panel shows a C++ program using a range-based for loop. The 'Insight' panel shows the equivalent code using iterators and explicit range construction. Below these panels is a 'Console' area showing the execution result.

```

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> & __rangel = arr;
9         int * __begin1 = __rangel.begin();
10        int * __end1 = __rangel.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 }

Console:
Insights exited with result code: 0
  
```



# Debugging

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



# Debugging

## The problem

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





# Debugging

## The problem

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

## The solution: debuggers

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



# Debugging

## The problem

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

## The solution: debuggers

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

## Existing tools

▶ `gdb` THE main player

▶ `lldb` the debugger coming with clang/LLVM

▶ `gdb-oneapi` the Intel OneAPI debugger



# `gdb` crash course

## start `gdb`

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

## inspect state

`bt` prints a backtrace

`print <var>` prints current content of the variable

`list` show code around current point

`up/down` go up or down in call stack

## breakpoints

`break <function>` puts a breakpoint on function entry

`break <file>:<line>` puts a breakpoint on that line



## Exercise: gdb

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



# Debugging UIs

## User interfaces for debuggers

- offer convenience on top of command line
- windows for variables, breakpoints, call stack, active threads, watch variables in-code, disassembly, run to cursor ...
  - **VSCode** Built-in support for gdb
  - **CodeLLDB** VS Code plugin for LLDB
  - **GDB dashboard** Poplar terminal UI for gdb
    - **GEF** Modern terminal UI for gdb
- some editors and most IDEs have good debugger integration



# 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

❶	C's memory management	26	❶❷	Absolutely avoid diamond-shaped inheritance	133
❷	Initialisation	29	❶❸	Avoid multiple inheritance	133
❸	Avoid unions	34	❶❹	Absolutely avoid diamond-shaped inheritance	133
❹	References	41	❶❺	Throwing exceptions	154
❺	Write readable functions	50	❶❻	Catching exceptions	156
❻	for syntax	63	❶❼	Exceptions	160
❼	preprocessor	71	❶❽	noexcept	163
❽	Prefer smart pointer	114	❶❾	STL and lambdas	192
❾	Prefer containers	115	❶❿	Use <code>std::fstream</code>	204
❿	Virtual destructors	126	❶❶	<code>std::unique_ptr</code>	209
❶❶	Avoid multiple inheritance	133			



# Index of Exercises

❶	Functions	49	❸	Operators	141	❶❶	clang-format	222
❷	Control structs	68	❹	Constness	151	❶❷	Compiler chain	232
❸	Loops, refs, auto	75	❺	Templates	172	❶❸	gdb	241
❹	Polymorphism	129	❻	STL	197			
❺	Virtual OO	134	❼	Smart pointers	213			





# Index of Godbolt code examples

①	Unary constructor in action - godbolt	98	④	Iterator example - godbolt	188
②	function objects - godbolt	144	⑤	unique_ptr copy - godbolt	206
③	STL - godbolt	184	⑥	std::shared_ptr quiz - godbolt	211



# 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>

