HEP C^{++}course

Based on the work of
Sébastien Ponce
sebastien.ponce@cern.ch

CERN

March 2022

Foreword

What this course is not

- It is not for absolute beginners
- It is not for experts
- It is not complete at all (would need 3 weeks...)
    - although is it already too long for the time we have
    - 240 slides, 336 pages, 10s of exercises...

How I see it

Adaptative pick what you want

Interactive tell me what to skip/insist on

Practical let's spend time on real code

Where to find latest version ?

- pdf format at http://cern.ch/sponce/C++Course
- full sources at https://github.com/hsf-training/cpluspluscourse

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. Useful tools
4. Object orientation (OO)
5. Core modern C++

Detailed outline

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

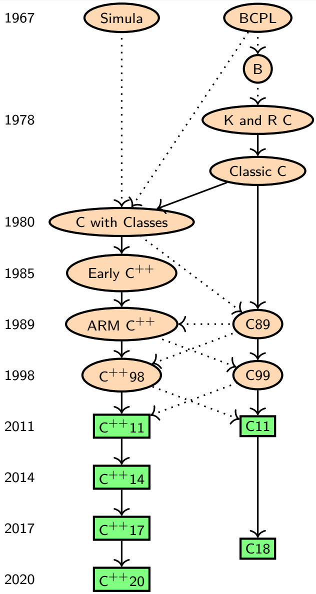
History and goals

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

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 but C^{++}20
- Each slide will be marked with first spec introducing the feature

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

status

- A new C++ specification every 3 years
  - C++20 is ready, officially published by ISO in December 2020
- Bringing each time a lot of goodies

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

status

- A new C++specification every 3 years
  - C++20 is ready, officially published by ISO in December 2020
- Bringing each time a lot of goodies

How to use C++XX features

- Use a compatible compiler
- add -std=c++xx to compilation flags
- e.g. -std=c++17

C++	gcc	clang
11	$\geq$4.8	$\geq$3.3
14	$\geq$4.9	$\geq$3.4
17	$\geq$7.3	$\geq$5
20	>11	>12

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

Why we use it?

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

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

Adapted to large projects

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

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

Adapted to large projects

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

Fast

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

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

Adapted to large projects

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

Fast

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

What we get

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

Language basics

1. History and goals

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

3. Useful tools

4. Object orientation (OO)

5. Core modern C^{++}

Core syntax and types

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

Hello World

C++98

```cpp
#include <iostream>

// This is a function
void print(int i) {
  std::cout << "Hello, world " << i << std::endl;
}

int main(int argc, char** argv) {
  int n = 3;
  for (int i = 0; i < n; i++) {
    print(i);
  }
  return 0;
}
```

Comments

C++98

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

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

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

Basic types(1)

C++98

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

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

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

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

Basic types(2)

$C^{++}98$

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

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

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

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

Portable numeric types

C++98

Requires inclusion of a specific header

```cpp
#include <cstdint>

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

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

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

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

Integer literals

C++98

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

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

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

Floating-point literals

C++98

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

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

double d = 0x4d2.1E6p3; // hexfloat, 0x4d2.1E6 * 2^3
                        // = 1234.12 * 2^3 = 9872.95

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

Useful aliases

C++98

Requires inclusion of headers

```
#include <cstddef> // and many other headers

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

#include <cstdint>

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

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

Arrays and Pointers

2 Language basics

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

Static arrays

C++98

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

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

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

Pointers

C++98

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

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

// compile error
int *pak = k;

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

Pointers

C++98

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

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

// compile error
int *pak = k;

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

Memory layout

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

Pointers

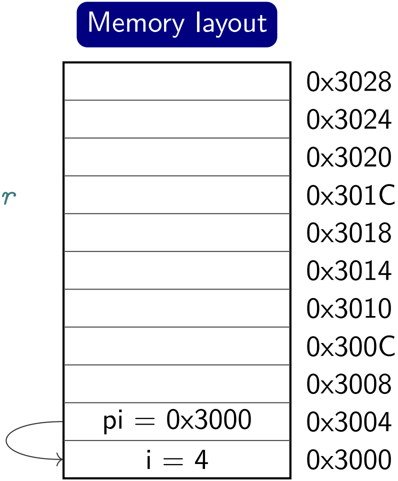
C++98

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

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

// compile error
int *pak = k;

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



Pointers

C++98

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

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

// compile error
int *pak = k;

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

Memory layout

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

Pointers $C^{++}98$

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

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

// compile error
int *pak = k;

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

Memory layout

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

Pointers

$C^{++}98$

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

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

// compile error
int *pak = k;

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

Memory layout

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

Pointers

C++98

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

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

// compile error
int *pak = k;

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

Memory layout

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

Pointers

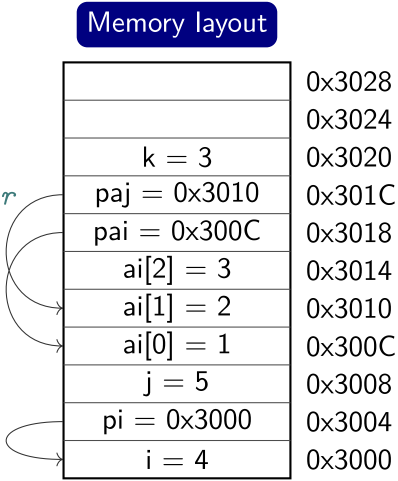
C++98

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

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

// compile error
int *pak = k;

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



Pointers

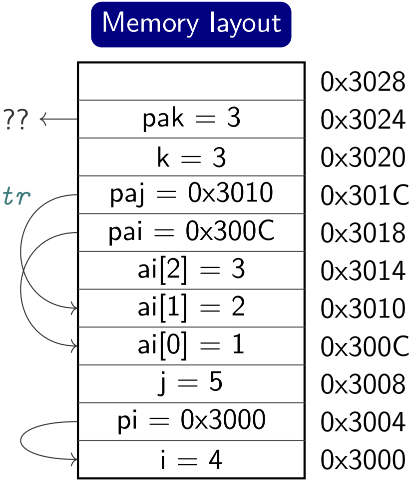
C++98

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

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

// compile error
int *pak = k;

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



nullptr

$C^{++}11$

Finally a C^{++} NULL pointer

- if a pointer doesn't point to anything, set it to `nullptr`
- works like 0 or NULL in standard cases
- triggers compilation error when mapped to integer

nullptr

$C^{++}11$

Finally a C^{++} NULL pointer

- if a pointer doesn't point to anything, set it to **nullptr**
- works like 0 or NULL in standard cases
- triggers compilation error when mapped to integer

Example code

```cpp
void* vp = nullptr;
int* ip = nullptr;
int i = NULL;       // OK -> bug ?
int i = nullptr;    // ERROR
```

Dynamic Arrays using C

C++98

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

int *bad;            // pointer to random address
int *ai = nullptr; // better, deterministic, can be tested

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

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

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

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
- Inline 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

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

Scope and lifetime of variables $C^{++}98$

Variable life time

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

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

Memory layout

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

Scope and lifetime of variables

$C^{++}98$

Variable life time

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

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

Memory layout

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

Scope and lifetime of variables

$C^{++}98$

Variable life time

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

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

Memory layout

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

Scope and lifetime of variables

C++98

Variable life time

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

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

Memory layout

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

Namespaces

$C^{++}98$

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

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

Nested namespaces

C++17

Easier way to declare nested namespaces

C++98

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

C++17

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

Unnamed namespaces

C++98

A namespace without a name !

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

Purpose

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

Deprecates static

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

Class and enum types

2 Language basics

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

struct

C++98

"members" grouped together under one name

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

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

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

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

struct

$C^{++}98$

"members" grouped together under one name

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

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

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

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

Memory layout

				0x3010
				0x300C
				0x3008
				0x3004
				0x3000

struct

C++98

"members" grouped together under one name

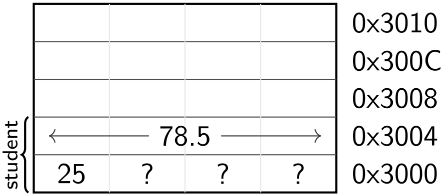
```cpp
struct Individual {
  unsigned char age;
  float weight;
};

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

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

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

Memory layout



struct

C++98

“members” grouped together under one name

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

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

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

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

Memory layout

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

struct

C++98

“members” grouped together under one name

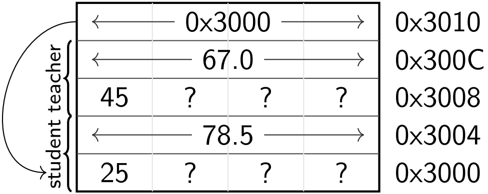
```cpp
struct Individual {
  unsigned char age;
  float weight;
};

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

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

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

Memory layout



union

C++98

"members" packed together at same memory location

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

union

$C^{++}98$

"members" packed together at same memory location

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

Memory layout

				0x300C
				0x3008
				0x3004
				0x3000

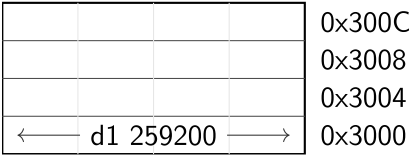
union

C++98

"members" packed together at same memory location

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

Memory layout



union

C++98

"members" packed together at same memory location

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

Memory layout

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

union

C++98

“members” packed together at same memory location

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

Memory layout

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

union

$C^{++}98$

"members" packed together at same memory location

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

Memory layout

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

union

C++98

"members" packed together at same memory location

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

Memory layout

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

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

Enums

C++98

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

```cpp
enum VehicleType {

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

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

Scoped enumeration, aka enum class $C^{++}11$

Same syntax as enum, with scope

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

Scoped enumeration, aka enum class

C++11

Same syntax as enum, with scope

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

Only advantages

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

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

enum class VT { Bus, Car }; enum class Col { Red, Blue };
VT t = VT::Bus;
if (t == Col::Red) { /* Compiler error */ }
int a = t * 5;         // Compiler error
```

More sensible example

C++98

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

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

More sensible example

$C^{++}98$

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

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

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

More sensible example

C++98

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

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

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

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

```

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

typedef and using

C++98 / C++11

Used to create type aliases

C++98

```cpp
typedef uint64_t myint;
myint toto = 17;
typedef int pos[3];
```

C++11

```cpp
using myint = uint64_t;
myint toto = 17;
using pos = int[3];

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

References

2 Language basics

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

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
- They can be used as function arguments

Example:

```cpp
int i = 2;
int &iref = i; // access to i
iref = 3;      // i is now 3

// const reference to a member:
struct A { int x; int y; } a;
const int &x = a.x; // direct read access to A's x
x = 4;              // doesn't compile
```

Pointers vs References

C++98

Specificities of reference

- natural syntax
- must be assigned when defined, cannot be `nullptr`
- cannot be reassigned
- non-const references to temporary objects are not allowed

Advantages of pointers

- can be reassigned to point elsewhere or to `nullptr`
- clearly indicates that argument may be modified

Pointers vs References

C++98

Specificities of reference

- natural syntax
- must be assigned when defined, cannot be `nullptr`
- cannot be reassigned
- non-const references to temporary objects are not allowed

Advantages of pointers

- can be reassigned to point elsewhere or to `nullptr`
- clearly indicates that argument may be modified

Good practice

- Always use references when you can
- Consider that a reference will be modified
- Use constness when it's not the case

Functions

2 Language basics

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

Functions

C++98

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

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

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

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

Function default arguments

C++98

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

```

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

Functions: parameters are passed by value

C++98

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

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

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

Memory layout

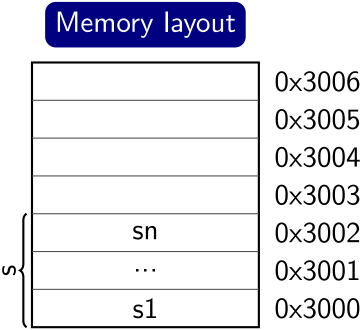
	0x3006
	0x3005
	0x3004
	0x3003
	0x3002
	0x3001
	0x3000

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

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

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

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



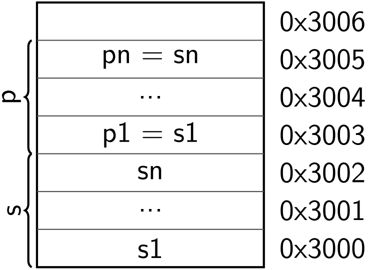
Functions: parameters are passed by value $C^{++}98$

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

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

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

Memory layout



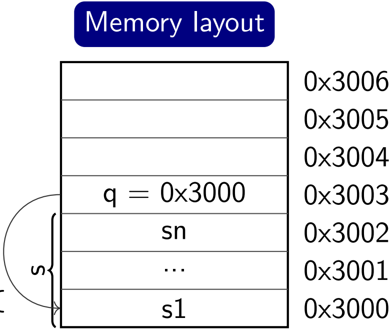
Functions: parameters are passed by value

C++98

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

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

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



Functions: pass by value or reference?

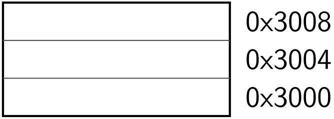
C++98

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

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

void changeSS2(SmallStruct &q) {
  q.a = 2;
}
changeSS2(s);
// s.a == 2
```

Memory layout



Functions: pass by value or reference? C++98

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

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

void changeSS2(SmallStruct &q) {
  q.a = 2;
}
changeSS2(s);
// s.a == 2
```

Memory layout

	0x3008
	0x3004
s.a = 1	0x3000

Functions: pass by value or reference? C++98

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

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

void changeSS2(SmallStruct &q) {
  q.a = 2;
}
changeSS2(s);
// s.a == 2
```

Memory layout

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

Functions: pass by value or reference? C++98

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

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

void changeSS2(SmallStruct &q) {
  q.a = 2;
}
changeSS2(s);
// s.a == 2
```

Memory layout

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

Functions: pass by value or reference? C++98

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

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

void changeSS2(SmallStruct &q) {
  q.a = 2;
}
changeSS2(s);
// s.a == 2
```

Memory layout

	0x3008
	0x3004
s.a = 1	0x3000

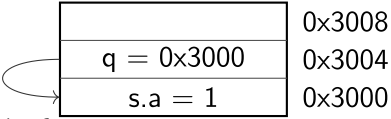
Functions: pass by value or reference? C++98

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

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

void changeSS2(SmallStruct &q) {
  q.a = 2;
}
changeSS2(s);
// s.a == 2
```

Memory layout



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

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

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

void changeSS2(SmallStruct &q) {
  q.a = 2;
}
changeSS2(s);
// s.a == 2
```

Memory layout

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

Functions: pass by value or reference?

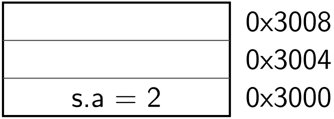
C++98

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

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

void changeSS2(SmallStruct &q) {
  q.a = 2;
}
changeSS2(s);
// s.a == 2
```

Memory layout



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
- prefer references for mandatory parameters to avoid copies
- use pointers for optional parameters to allow `nullptr`
- 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
- prefer references for mandatory parameters to avoid copies
- use pointers for optional parameters to allow `nullptr`
- use `const` for safety and readability whenever possible

Syntax

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

Functions

$C^{++}98$

Exercise

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

Ensure good readability/maintainability:

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

Example: Good

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

Functions: good practices

C++98

Example: don't! Everything in one long function

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

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

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

  return counter;
}
```

Operators

2 Language basics

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

Operators(1)

C++98

Binary & Assignment Operators

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

Operators(1) C++98

Binary & Assignment Operators

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

Increment / Decrement

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

Operators(1)

C++98

Binary & Assignment Operators

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

Increment / Decrement

Use wisely

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

Operators(2)

C++98

Bitwise and Assignment Operators

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

Operators(2) C++98

Bitwise and Assignment Operators

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

Boolean Operators

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

Operators(3)

$C^{++}98$

Comparison Operators

```cpp
bool a = (3 == 3);  // true
bool b = (3 != 3);  // false
bool c = (4 <  4);  // false
bool d = (4 <= 4);  // true
bool e = (4 >  4);  // false
bool f = (4 >= 4);  // true
```

Operators(3)

C++98

Comparison Operators

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

Precedences

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

Details can be found on cppreference

Operators(3)

C++98

Comparison Operators

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

Precedences

Don't use

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

Details can be found on cppreference

Operators(3)

C++98

Comparison Operators

```cpp
bool a = (3 == 3);  // true
bool b = (3 != 3);  // false
bool c = (4 <  4);  // false
bool d = (4 <= 4);  // true
bool e = (4 >  4);  // false
bool f = (4 >= 4);  // true
```

Precedences

Don't use - use parentheses

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

Details can be found on cppreference

Control structures

2 Language basics

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

Control structures: if

C++98

if syntax

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

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

Control structures: if

C++98

Practical example

```cpp
int collatz(int a) {
  if (a <= 0) {
    std::cout << "not supported";
    return 0;
  } else if (a == 1) {
    return 1;
  } else if (a%2 == 0) {
    return collatz(a/2);
  } else {
    return collatz(3*a+1);
  }
}
```

Control structures: conditional operator

$C^{++}98$

Syntax

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

- if test is `true` expression1 is returned
- else expression2 is returned

Control structures: conditional operator C++98

Syntax

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

- if test is `true` expression1 is returned
- else expression2 is returned

Practical example

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

Control structures: conditional operator

C++98

Syntax

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

- if test is `true` expression1 is returned
- else expression2 is returned

Practical example

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

Do not abuse

- explicit ifs are easier to read
- use only when obvious and not nested

Control structures: switch

C++98

Syntax

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

- `break` is not mandatory but...
- cases are entry points, not independent pieces
- execution falls through to the next case without a `break`!
- `default` may be omitted

Control structures: switch

$C^{++}98$

Syntax

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

- `break` is not mandatory but...
- cases are entry points, not independent pieces
- execution falls through to the next case without a `break`!
- `default` may be omitted

Use break

Do not try to make use of non breaking cases

Control structures: switch

C++98

Practical example

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

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

New compiler warning

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

$C^{++}17$

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

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

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

$C^{++}17$

```cpp
if (Value val = GetValue(); condition(val)) {
  f(val);
} else {
  g(val);
}
h(val); // compile error
```

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

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

$C^{++}17$

```
if (Value val = GetValue(); condition(val)) {
  f(val);
} else {
  g(val);
}
h(val); // compile error
```

$C^{++}98$

Don't confuse with a variable declaration as condition:

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

Control structures: for loop

C++98

for loop syntax

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

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

Control structures: for loop

C++98

for loop syntax

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

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

Practical example

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

Control structures: for loop

C++98

for loop syntax

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

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

Practical example

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

Do not abuse the 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

Syntax

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

Example code

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

Control structures: while loop

C++98

while loop syntax

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

- braces are optional if the body is a single statement

Control structures: while loop

C++98

while loop syntax

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

- braces are optional if the body is a single statement

Bad example

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

Control structures: jump statements

C++98

break	exits the loop and continues after it
continue	goes immediately to next loop iteration
return	exists the current function
goto	can jump anywhere inside a function, don't use!

Control structures: jump statements

C++98

break	exits the loop and continues after it
continue	goes immediately to next loop iteration
return	exists the current function
goto	can jump anywhere inside a function, don't use!

Bad example

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

Control structures

C++11

Exercise

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

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

Headers and interfaces

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

Headers and interfaces

C++98

Interface

Set of declarations defining some functionality

- put in a so-called “header file”
- the implementation exists somewhere else

Header : hello.hpp

```
void printHello();
```

Usage : myfile.cpp

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

Preprocessor

C++98

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

Preprocessor

C++98

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

Use only in very restricted cases

- inclusion of headers
- customization for specific compilers/platforms

Header include guards

$C^{++}98$

Problem: redefinition by accident

- a header 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

```
#ifndef MY_HEADER_INCLUDED
#define MY_HEADER_INCLUDED
... // content
#endif
```

Pragma once (non-standard)

```
#pragma once
... // 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
- Inline keyword

Auto keyword

$C^{++}11$

Reason of being

- many type declarations are redundant
- and lead to compiler errors if you mess up

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

Auto keyword

C++11

Reason of being

- many type declarations are redundant
- and lead to compiler errors if you mess up

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

Practical usage

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

Loops, references, auto

C++98

Exercise

Familiarise yourself with range-based for loops and references

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

Inline keyword

2 Language basics

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

Inline keyword

C++98

Inline functions originally

- applies to a function to tell the compiler to inline it
  - i.e. replace function calls by the function's content
  - similar to a macro
- only a hint, compiler can still choose to not inline
- avoids function call overhead
  - but may increase executable size

Major side effect

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

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

Inline keyword

C++98

Inline functions nowadays

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

Inline keyword C++17

Inline variables

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

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

- Avoid global variables! Constants are fine.

Useful tools

1 History and goals

2 Language basics

3 Useful tools
- C++editor
- Code management
- Code formatting
- The Compiling Chain
- Debugging

4 Object orientation (OO)

5 Core modern C++

C++ editor

3 Useful tools
- C++ editor
- Code management
- Code formatting
- The Compiling Chain
- Debugging

C++ editors and IDEs

Can dramatically improve your efficiency by

- coloring the code for you to "see" the structure
- helping with indenting and formatting properly
- allowing you to easily navigate in the source tree
- helping with compilation/debugging, profiling, static analysis
- showing you errors and suggestions while typing

▸ Visual Studio	heavy, fully fledged IDE for Windows
▸ Visual Studio Code	editor, open source, portable, many plugins
▸ Eclipse	IDE, open source, portable
▸ Emacs ▸ Vim	editors for experts, extremely powerful. They are to IDEs what latex is to PowerPoint

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

Choosing one is mostly a matter of taste

Code management

3 Useful tools

- C++editor
- Code management
- Code formatting
- The Compiling Chain
- Debugging

Code management tool

Please use one !

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

It will soon save your day

A few tools

git	THE mainstream choice. Fast, light, easy to use
mercurial	the alternative to git
Bazaar	another alternative
svn	historical, not distributed - DO NOT USE
CVS	archeological, not distributed - DO NOT USE

GIT crash course

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

# vim file.cpp; vim file2.cpp
# git add file.cpp file2.cpp
# git commit -m "Committing first 2 files"
[master (root-commit) c481716] Committing first 2 files
...

# git log --oneline
d725f2e Better STL test
f24a6ce Reworked examples + added stl one
bb54d15 implemented template part
...

# git diff f24a6ce bb54d15
```

Code formatting

3 Useful tools

- C++editor
- Code management
- Code formatting
- The Compiling Chain
- Debugging

clang-format

.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 Time

- go to any example
- format code with:
  `clang-format --style=GNU -i <file.cpp>`
- inspect changes, try `git diff`
- revert changes using `git checkout -- <file.cpp>`
- go to code directory and create a .clang-format file
  `clang-format -style=LLVM -dump-config > .clang-format`
- run `clang-format -i */*.cpp`
- revert changes using `git checkout .`

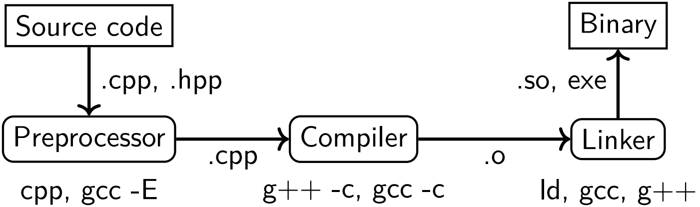
The Compiling Chain

3 Useful tools

- C++editor
- Code management
- Code formatting
- The Compiling Chain
- 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

Makefiles

Why to use them

- an organized way of describing building steps
- avoids a lot of typing

Several implementations

- raw Makefiles: suitable for small projects
- cmake: portable, the current best choice
- automake: GNU project solution

```
test : test.cpp libpoly.so
    $(CXX) -Wall -Wextra -o $@ $^
libpoly.so: Polygons.cpp
    $(CXX) -Wall -Wextra -shared -fPIC -o $@ $^
clean:
    rm -f *o *so *~ test test.sol
```

CMake

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

Example CMakeLists.txt

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

find_package(ZLIB REQUIRED) # for external libs

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

CMake - Building

Building a CMake-based project

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

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

Compiler chain

Exercise Time

- go to code/functions
- preprocess functions.cpp (cpp or gcc -E -o output)
- compile functions.o and Structs.o (g++ -c -o output)
- use nm to check symbols in .o files
- look at the Makefile
- try make clean; make
- see linking stage of the final program using g++ -v
  - just add a -v in the Makefile command for functions target
  - run make clean; make
  - look at the collect 2 line, from the end up to "-o functions"
- see library dependencies with 'ldd functions'

Debugging

3 Useful tools

- C^{++}editor
- Code management
- Code formatting
- The Compiling Chain
- Debugging

Debugging

The problem

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

Debugging

The problem

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

The solution : debuggers

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

Debugging

The problem

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

The solution : debuggers

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

Existing tools

- gdb THE main player
- lldb the debugger coming with clang/LLVM
- gdb-oneapi the Intel OneAPI debugger

They usually can be integrated into your IDE

gdb crash course

start gdb

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

inspect state

bt	prints a backtrace
print <var>	prints current content of the variable
list	show code around current point
up/down	go up or down in call stack

breakpoints

break <function>	puts a breakpoint on function entry
break <file>:<line>	puts a breakpoint on that line

gdb

Exercise Time

- 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
- use -Wall -Wextra and see warning

Object orientation (OO)

1 History and goals

2 Language basics

3 Useful tools

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

5 Core modern C++

Objects and Classes

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

What are classes and objects

$C^{++}98$

Classes (or "user-defined types")

C structs on steroids

- with inheritance
- with access control
- including methods

Objects

instances of classes

A class encapsulates state and behavior of "something"

- shows an interface
- provides its implementation
  - status, properties
  - possible interactions
  - construction and destruction

My First Class

$C^{++}98$

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

MyFirstClass myObj;
myObj.a = 2;

// let's square a
myObj.squareA();
```

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

Separating the interface

C++98

Header : MyFirstClass.hpp

```cpp
#pragma once
struct MyFirstClass {
  int a;
  void squareA();
  int sum(int b);
};
```

Implementation : MyFirstClass.cpp

```cpp
#include "MyFirstClass.hpp"
void MyFirstClass::squareA() {
  a *= a;
}
void MyFirstClass::sum(int b) {
  return a + b;
}
```

Implementing methods

C++98

Standard practice

- usually in .cpp, outside of class declaration
- using the class name as namespace
- when reference to the object is needed, use *this* keyword

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

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

this keyword

$C^{++}98$

- `this` is a hidden parameter to all class methods
- it points to the current object
- so it is of type `T*` in the methods of class `T`

```cpp
void ext_func(MyFirstClass& c) {
  ... do something with c ...
}

int MyFirstClass::some_method(...) {
  ext_func(*this);
}
```

Method overloading

$C^{++}98$

The rules in C^{++}

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

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

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

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

Inheritance

4 Object orientation (OO)

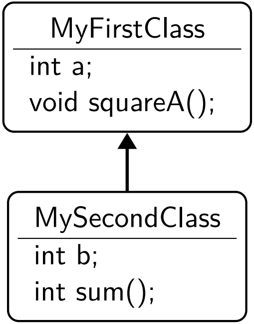
- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Type casting
- Operators
- Functors

First inheritance

C++98

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

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

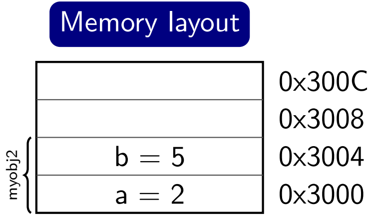


First inheritance

C++98

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

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



Managing access to class members

C++98

public / *private* keywords

private allows access only within the class

public allows access from anywhere

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

Managing access to class members

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

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

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

Managing access to class members

C++98

public / *private* keywords

private allows access only within the class

public allows access from anywhere

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

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

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

This breaks MySecondClass !

Managing access to class members(2)

C++98

Solution is *protected* keyword

Gives access to classes inheriting from base class

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

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

Managing inheritance privacy

C++98

Inheritance can be public, protected or private

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

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

Net result for external code

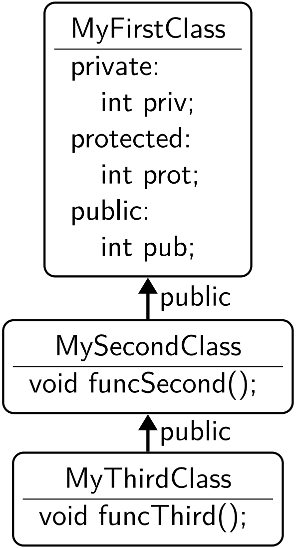
- only public members of public inheritance are accessible

Net result for grand child code

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

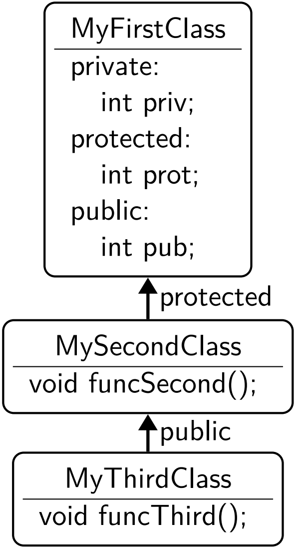
Managing inheritance privacy - public

C++98




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

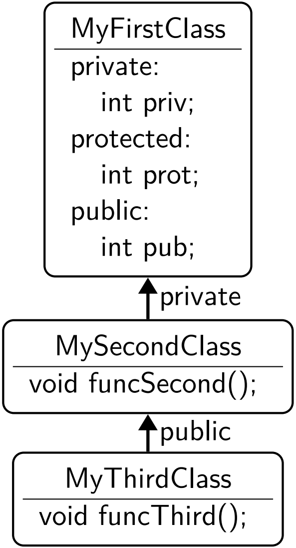
Managing inheritance privacy - protected $C^{++}98$




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

Managing inheritance privacy - private

C++98




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

Constructors/destructors

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

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 ~

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

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

MyFirstClass(int a_):a(a_) {}

~MyFirstClass() {}
```

Class Constructors and Destructors

C++98

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

Constructors and inheritance

C++98

```cpp
struct MyFirstClass {
  int a;
  MyFirstClass();
  MyFirstClass(int a);
};
struct MySecondClass : MyFirstClass {
  int b;
  MySecondClass();
  MySecondClass(int b);
  MySecondClass(int a, int b);
};
MySecondClass::MySecondClass() : MyFirstClass(), b(0) {}
MySecondClass::MySecondClass(int b_)
  : MyFirstClass(), b(b_) {}
MySecondClass::MySecondClass(int a_, int b_)
  : MyFirstClass(a_), b(b_) {}
```

Copy constructor

$C^{++}98$

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

$C^{++}98$

Concept

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

```cpp
struct MySecondClass : MyFirstClass {
  MySecondClass();
  MySecondClass(const MySecondClass &other);
};
```

Copy constructor $C^{++}98$

Concept

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

```
struct MySecondClass : MyFirstClass {
  MySecondClass();
  MySecondClass(const MySecondClass &other);
};
```

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.

Class Constructors and Destructors

C++98

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

Explicit unary constructor

C++98

Concept

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

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

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

Explicit unary constructor $C^{++}98$

Concept

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

```cpp
class Vector {
public:
  explicit Vector(int n);
  Vector(const Vector &other);
  ~Vector();
  ...
};
```

Defaulted Constructor

C++11

Idea

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

Details

- when no user defined constructor, a default is provided
- any user-defined constructor disables the default one
- but they can be enforced
- rule can be more subtle depending on members

Practically

```
ClassName() = default;  // provide/force default
ClassName() = delete;   // do not provide default
```

Delegating constructor

C++11

Idea

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

Practically

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

Constructor inheritance C++11

Idea

- avoid having to re-declare parent's constructors
- by stating that we inherit all parent constructors

Practically

```
struct BaseClass {
  BaseClass(int value);
};
struct DerivedClass : BaseClass {
  using BaseClass::BaseClass;
};
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

```
struct BaseClass {
  int a{5}; // also possible: int a = 5;
  BaseClass() = default;
  BaseClass(int _a) : a(_a) {}
};
struct DerivedClass : BaseClass {
  int b{6};
  using BaseClass::BaseClass;
};
DerivedClass d{7}; // a = 7, b = 6
```

Calling constructors

C++11

After object declaration, arguments within {}

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

```cpp
struct B {
  int a;
  float b;
  // no constructor
};
```

```cpp
A a{1,2};        // A::A(int, int)
A a{1};          // A::A(int)
A a{};           // A::A()
A a;             // A::A()
A a = {1,2};     // A::A(int, int)
B b = {1, 2.3}; // aggregate initialization
```

Calling constructors the old way

C++98

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

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

```
struct B {
  int a;
  float b;
  // no constructor
};
```

```
A a(1,2);        // A::A(int, int)
A a(1);          // A::A(int)
A a();           // declaration of a function !
A a;             // A::A()
A a = {1,2};     // not allowed
B b = {1, 2.3};  // OK
```

Calling constructors for arrays and vectors

C++11

list of items given within {}

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

Calling constructors for arrays and vectors $C^{++}11$

list of items given within {}

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

$C^{++}98$ nightmare

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

Static members

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

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

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

Allocating objects

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

Process memory organization

$C^{++}98$

4 main areas

the code segment for the machine code of the executable

the data segment for global variables

the heap for dynamically allocated variables

the stack for parameters of functions and local variables

Memory layout

heap
$\downarrow$ ... $\downarrow$
$\uparrow$ ... $\uparrow$
stack
data segment
code segment

The Stack

$C^{++}98$

Main characteristics

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

Object allocation on the stack

C++98

On the stack

- objects are created when declared (constructor called)
- objects are destructed when out of scope (destructor is called)

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

{
  MyFirstClass a; // default constructor called
  ...
}  // destructor called
```

The Heap

$C^{++}98$

Main characteristics

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

Object allocation on the heap

C++98

On the heap

- object are created by calling `new` (constructor is called)
- object are destructed by calling `delete` (destructor is called)

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

int f() {
  // constructor called
  MyFirstClass *a = new MyFirstClass(3);
  ...
} // memory leak !!!
```

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

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

Advanced OO

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

Polymorphism

C++98

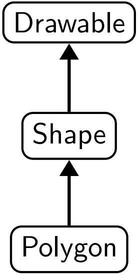
the concept

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

```
Polygon p;

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

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



Polymorphism

$C^{++}98$

the concept

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

```cpp
Polygon p;

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

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

Memory layout

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

Polymorphism

C++98

the concept

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

```cpp
Polygon p;

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

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

Memory layout

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

Polymorphism

C++98

the concept

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

```cpp
Polygon p;

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

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

Memory layout

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

Inheritance privacy and polymorphism

C++98

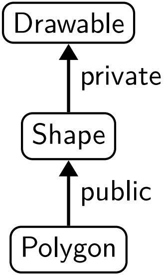
Only public inheritance is visible to code outside the class

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

```cpp
Polygon p;

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

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



Method overriding

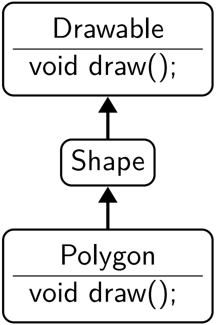
C++98

the problem

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

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

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



Virtual methods

$C^{++}98$

the concept

- methods can be declared `virtual`
- for these, the most derived object is always considered
- for others, the type of the variable decides

Virtual methods

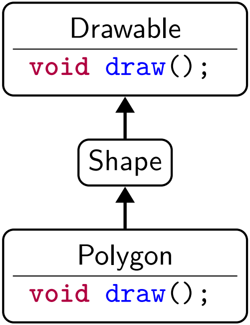
C++98

the concept

- methods can be declared `virtual`
- for these, the most derived object is always considered
- for others, the type of the variable decides

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

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



Virtual methods

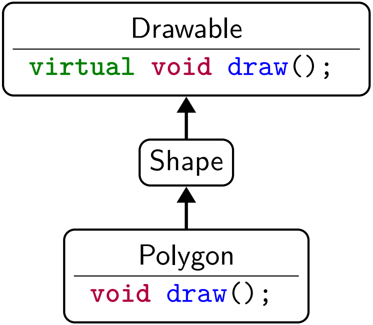
C++98

the concept

- methods can be declared `virtual`
- for these, the most derived object is always considered
- for others, the type of the variable decides

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

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



Virtual methods - implications

C++11

Mechanics

- virtual methods are dispatched at run time
  - while non-virtual methods are bound at compile time
- they also imply extra storage and an extra indirection
  - practically the object stores a pointer to the correct method
  - in a so-called "virtual table" ("vtable")

Consequences

- virtual methods are "slower" than standard ones
- and they can rarely be inlined
- templates are an alternative for performance-critical cases

override keyword

C++11

Principle

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

Practically

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

Why was override keyword introduced? $C^{++}11$

To detect the mistake in the following code :

Without override ($C^{++}98$)

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

- with `override`, you would get a compiler error
- if you forget `override` when you should have it, you 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 $C^{++}98$

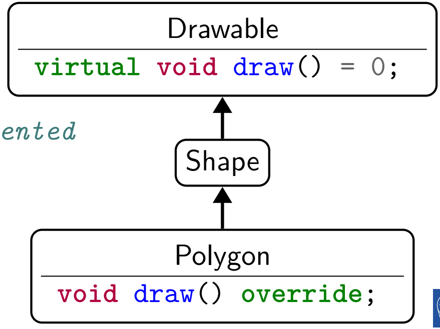
Concept

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

```cpp
// Error : abstract class
Shape s;

// ok, draw has been implemented
Polygon p;

// Shape type still usable
Shape & s = p;
s.draw();
```



Pure Abstract Class aka Interface

C++98

Definition of pure abstract class

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

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

Drawable

virtual void draw() = 0;

Overriding overloaded methods

C++98

Concept

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

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

Polymorphism

$C^{++}98$

Exercise Time

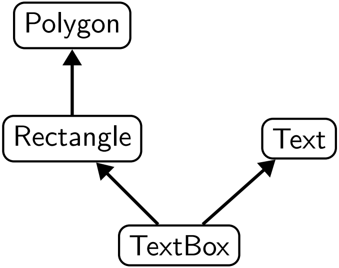
- go to code/polymorphism
- look at the code
- open trypoly.cpp
- create a Pentagon, call its perimeter method
- create a Hexagon, call its perimeter method
- create a Hexagon, call its parent's perimeter method
- retry with virtual methods

Multiple Inheritance

C++98

Concept

- one class can inherit from multiple parents




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

The diamond shape

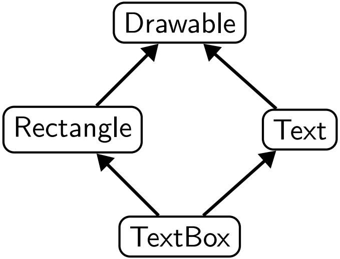
$C^{++}98$

Definition

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

Problem

- are the members of the grand parent replicated?



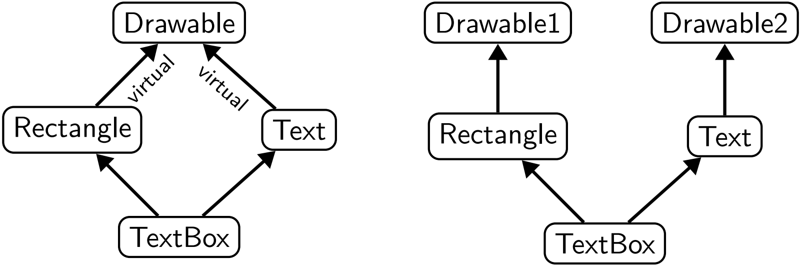
Virtual inheritance

$C^{++}98$

Solution

- inheritance can be *virtual* or not
- *virtual* inheritance will "share" parents
- standard inheritance will replicate them

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



Multiple inheritance advice

C++98

Do not use multiple inheritance

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

Multiple inheritance advice

C++98

Do not use multiple inheritance

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

Do not use diamond shapes

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

Virtual inheritance

C++98

Exercise Time

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

Virtual inheritance

C++98

Good practice

if you write a class and expect users to inherit from it, declare its destructor `virtual`

Warning

in case of virtual inheritance it is the most derived class that calls the virtual base class's constructor

Type casting

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

Type casting

C++98

5 types of casts in C++

- `static_cast<Target>(arg)`: Convert type if the static types allow it
- `dynamic_cast<Target>(arg)`: Check if object at address of "arg" is compatible with the type Target. Throw std::bad_cast if it's not.

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

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

B& f = dynamic_cast<B&>(c); // OK, c is a B
B& g = dynamic_cast<B&>(a); // Exception: not a B
```

Type casting $C^{++}98$

5 types of casts in C^{++}

- `static_cast<Target>(arg)`: Convert type if the static types allow it
- `dynamic_cast<Target>(arg)`: Check if object at address of "arg" is compatible with the type Target. Return `nullptr` if it's not.

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

if (auto bPtr = dynamic_cast<B*>(&c)) {
  // OK, we will get here
}
```

Type casting

$C^{++}98$

5 types of casts in C^{++}

- `const_cast`: Remove constness from a type. If you think you need this, first try to improve the design!
- `reinterpret_cast<Target>(arg)`: Change type irrespective of what 'arg' is. *Almost never* a good idea!
- C-style: `(Target)arg`: Force-change type in C-style. No checks. Don't use this.

Casts to avoid

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

Operators

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

Operators' example

C++98

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

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

Operators

$C^{++}98$

Defining operators of a class

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

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 to have non-member operators? C++98

Symmetry

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

Why to have non-member operators? $C^{++}98$

Symmetry

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

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

Extending existing classes

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

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

Operators

C++98

Exercise

Write a simple class representing a fraction and pass all tests

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

Functors

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

Functors

C++98

Concept

- a class that implements `operator()`
- allows to use objects in place of functions
- and as objects have constructors, allow to construct functions

```cpp
struct Adder {
  int m_increment;
  Adder(int increment) : m_increment(increment) {}
  int operator()(int a) { return a + m_increment; }
};

Adder inc1{1}, inc10{10};
int i = 3;
int j = inc1(i);  // 4
int k = inc10(i); // 13
int l = Adder{25}(i); // 28
```

Functors

C++98

Typical usage

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

```
struct BinaryFunction {
  virtual double operator() (double a, double b) = 0;
};
double binary_op(double a, double b, BinaryFunction &func) {
  return func(a, b);
}
struct Add : BinaryFunction {
  double operator() (double a, double b) override
  { return a+b; }
};
Add addfunc;
double c = binary_op(a, b, addfunc);
```

Core modern C++

1 History and goals

2 Language basics

3 Useful tools

4 Object orientation (OO)

5 Core modern C++

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

Constness

5 Core modern C++

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

Constness

C++98

The *const* keyword

- indicate that the element to the left is constant
- this element won't be modifiable in the future
- this is all checked at compile time

```cpp
// standard syntax
int const i = 6;

// error : i is constant
i = 5;

// also ok, when nothing on the left,
// const applies to the element on the right
const int j = 6;
```

Constness and pointers

C++98

```cpp
// pointer to a constant integer
int a = 1, b = 2;
int const *i = &a;
*i = 5; // error, int is const
i = &b; // ok, pointer is not const

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

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

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

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

Method constness

C++98

The *const* keyword for member functions

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

```cpp
struct Example {
  void foo() const {
    // type of 'this' is 'Example const*'
    m_member = 0; // Error: member function is const
  }
  int m_member;
};
```

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

```
void func(int & a);
void funcConst(int const & a);

int a = 0;
int const b = 0;

func(a);      // ok
func(b);      // error
funcConst(a); // ok
funcConst(b); // ok
```

constness

Exercise Time

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

Exceptions

5 Core modern C^{++}

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

Exceptions

C++98

The concept

- to handle *exceptional* events that happen rarely
- and cleanly jump to a place where the error can be handled

In practice

- add an exception handling block with `try` ... `catch`
  - when exceptions are possible *and can be handled*
- throw an exception using `throw`
  - when a function cannot proceed or recover internally

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

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

Exceptions

C++98

Rules and behavior

- objects of any type can be thrown
  - prefer standard exception types from the `<stdexcept>` header
  - define your own subclass of `std::exception` if needed
- an exception will be caught if the type in the catch clause matches or is a base class of the thrown object's static type
  - if no one catches an exception then `std::terminate` is called
- you can have multiple catch clauses, will be matched in order
- all objects on the stack between the `throw` and the `catch` are destructed automatically during stack unwinding
  - this should cleanly release intermediate resources
  - make sure you are using the RAII idiom for your own classes

Exceptions

$C^{++}17$

Advice

- throw exceptions by value, catch them by (const) reference
- use exceptions for *unlikely* runtime errors outside the program's control
  - bad inputs, files unexpectedly not found, DB connection, ...
- *don't* use exceptions for logic errors in your code
  - consider `assert` and tests
- *don't* use exceptions to provide alternative return values (or to skip them)
  - you can use `std::optional` or `std::variant`
  - avoid using the global C-style `errno`
- See also the C^{++}core guidelines and the ISO C^{++}FAQ

Exceptions

C++98

A more illustrative example

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

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

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

Exceptions

C++98

Catching everything

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

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

Error Handling and Exceptions

$C^{++}98$

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

Avoid

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

Prefer

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

noexcept specifier

C++11

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

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

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

```
int safe_if_8B_long() noexcept(sizeof(long)==8);
```

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

noexcept operator

C++11

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

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

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

Templates

5 Core modern C++
- Constness
- Exceptions
- Templates
- The STL
- Lambdas
- pointers and RAII

Templates

C++17

Concept

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

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

Templates

C++98

Warning

- they are compiled for each instantiation
- they need to be defined before used
  - so all templated code has to be in headers
- this may lead to longer compilation times and bigger libraries

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

func(3)

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

func(5.2)

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

Templates

C++98

Template parameters

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

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

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

Templates implementation

C++98

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

template<typename KeyType, typename ValueType>
void Map<KeyType, ValueType>::set
   (const KeyType &key, ValueType value) {
  ...
}

template<typename KeyType, typename ValueType>
ValueType Map<KeyType, ValueType>::get
   (const KeyType &key) {
  ...
}
```

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 $C^{++}17$
- floats and literal types in $C^{++}20$

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

Templates

C++98

Specialization

templates can be specialized for given values of their parameter

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

template<typename F>
struct Polygon<F, 6> {
  Polygon(F radius) : m_radius(radius) {}
  F perimeter() {return 6*m_radius;}
  F m_radius;
};
```

The full power of templates

C++98

Exercise Time

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

The STL

5 Core modern C++

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

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

```cpp
#include <vector>
#include <algorithm>

std::vector<int> vi{5, 3, 4}; // initializer list
std::vector<int> vr(3); // constructor taking int

std::transform(vi.begin(), vi.end(),        // range1
               vi.begin(),                  // start range2
               vr.begin(),                  // start result
               std::multiplies{}); // function objects

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

STL's concepts

C++98

containers

- data structures for managing a range of elements
- irrespective of
  - the data itself (templated)
  - the memory allocation of the structure (templated)
  - the algorithms that may use the structure
- examples
  - string, string_view (C++17)
  - list, forward_list (C++11), vector, deque, array (C++11)
  - map, set, multimap, multiset
  - unordered_map (C++11), unordered_set (C++11)
  - stack, queue, priority_queue
  - span (C++20)
- non-containers: pair, tuple (C++11), optional (C++17), variant (C++17), any (C++17)
- see also the string and container library on cppreference

Containers: std::vector

C++11

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

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

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

Containers: std::vector

C++11

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

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

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

STL's concepts

C++98

iterators

- generalization of pointers
- allow iteration over some data
- irrespective of
  - the container used (templated)
  - the data itself (container is templated)
  - the consumer of the data (templated algorithm)
- examples
  - iterator
  - reverse_iterator
  - const_iterator

STL's concepts

C++98

algorithms

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

STL's concepts

C++98

functors / function objects

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

Functors / function objects

C++11

Example

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

  int operator()(int value) const {
    return value + m_inc;
  }
};
std::vector<int> v;
v.push_back(5); v.push_back(3); ...
std::transform(v.begin(), v.end(), v.begin(),
               Incrementer{42});
```

STL in practice

$C^{++}14$

```cpp
#include <vector>
#include <algorithm>

std::vector<int> vi{5, 3, 4}; // initializer list
std::vector<int> vr(3); // constructor taking int

std::transform(vi.begin(), vi.end(),        // range1
               vi.begin(),                  // start range2
               vr.begin(),                  // start result
               std::multiplies{}); // function objects

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

Range-based for loops with STL containers

C++11

Iterator-based loop (since C++98)

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

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

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

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

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

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

Range-based for loops with STL containers

C++11

Iterator-based loop (since C++98)

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

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

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

STL way (since C++98)

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

STL and functors

C++98

```cpp
// Finds the first element in a list between 1 and 10.
list<int> l = ...;
...
list<int>::iterator it =
  find_if(l.begin(), l.end(),
          compose2(logical_and<bool>(),
                   bind2nd(greater_equal<int>(), 1),
                   bind2nd(less_equal<int>(), 10)));

// Computes sin(x)/(x + DBL_MIN) for elements of a range.
transform(first, last, first,
          compose2(divides<double>(), // non-standard
                   ptr_fun(sin),
                   bind2nd(plus<double>(), DBL_MIN)));
```

Deprecation warning

Binders and function adaptors were removed in C++17 or C++20

STL and lambdas

C++14

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

// Computes sin(x)/(x + DBL_MIN) for elements of a range.
std::transform(first, last, first,
  [](auto x) { return sin(x)/(x + DBL_MIN); });
```

Welcome to lego programming!

C++98

Using the STL

C++98

Exercise Time

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

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!

Lambdas

5. Core modern C++
   - Constness
   - Exceptions
   - Templates
   - The STL
   - Lambdas
   - pointers and RAII

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

```
class Class {
  using ReturnType = int;
  ReturnType func();
}
Class::ReturnType Class::func() {...}
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

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

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

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

C++Lambdas

C++11

Simplified syntax

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

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

Usage example

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

Capturing variables

$C^{++}11$

Python code

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

Capturing variables

C++11

Python code

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

First attempt in C++

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

Capturing variables

C++11

Python code

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

First attempt in C++

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

Error

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

Capturing variables

C++11

The capture list

- local variables outside the lambda must be explicitly captured
- captured variables are listed within initial []

Capturing variables

C++11

The capture list

- local variables outside the lambda must be explicitly captured
- captured variables are listed within initial []

Example

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

Default capture is by value

C++11

Code example

```
int sum = 0;
std::vector<int> data{1,9,3,8,3,7,4,6,5};
for_each(begin(data), end(data),
         [sum](int x) { sum += x; });
```

Default capture is by value $C^{++}11$

Code example

```cpp
int sum = 0;
std::vector<int> data{1,9,3,8,3,7,4,6,5};
for_each(begin(data), end(data),
         [sum](int x) { sum += x; });
```

Error

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

Default capture is by value

C++11

Code example

```cpp
int sum = 0;
std::vector<int> data{1,9,3,8,3,7,4,6,5};
for_each(begin(data), end(data),
         [sum](int x) { sum += x; });
```

Error

```
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

```
int sum = 0;
std::vector<int> data{1,9,3,8,3,7,4,6,5};
for_each(begin(data), end(data),
         [&sum](int x) { sum += x; });
```

Capture by reference

C++11

Simple example

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

```
int sum = 0;
std::vector<int> data{1,9,3,8,3,7,4,6,5};
for_each(begin(data), end(data),
         [&sum](int x) { sum += x; });
```

Mixed case

One can of course mix values and references

```
int sum = 0, offset = 1;
std::vector<int> data{1,9,3,8,3,7,4,6,5};
for_each(begin(data), end(data),
         [&sum, offset](int x) {
           sum += x + offset;
         });
```

Capture list

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

Anatomy of a lambda

C++11

```cpp
int sum = 0, off = 1;
auto l =
  [&sum, off]
  (int x) {
    sum += x + off;
};
l(42);
```

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

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

See also result on cppinsights.io.

Higher-order lambdas

C++11

Example

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

How it works

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

Prefer lambdas over functors

C++11

Before lambdas

```
struct Incrementer {
  int m_inc;
  Incrementer(int inc) : m_inc(inc) {}
  int operator() (int value) {
    return value + m_inc;
  };
};
std::vector<int> v{1, 2, 3};
std::transform(begin(v), end(v), begin(v),
               Incrementer(1));
for (auto a : v) std::cout << a << " ";
```

Prefer lambdas over functors

C++11

With lambdas

```
std::vector<int> v{1, 2, 3};
std::transform(begin(v), end(v), begin(v),
               [](int value) {
                 return value + 1;
               });
for (auto a : v) std::cout << a << " ";
```

Prefer lambdas over functors

C++11

With lambdas

```
std::vector<int> v{1, 2, 3};
std::transform(begin(v), end(v), begin(v),
               [](int value) {
                 return value + 1;
               });
for (auto a : v) std::cout << a << " ";
```

Conclusion

Use the STL with lambdas!

Lambdas

Exercise Time

- go to code/lambdas
- look at the code (it's the solution to the stl exercise)
- use lambdas to simplify it

pointers and RAII

5 Core modern C^{++}
- Constness
- Exceptions
- Templates
- The STL
- Lambdas
- pointers and RAII

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

They need initialization

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

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

They need initialization **Seg Fault**

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

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

They need initialization — Seg Fault

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

They need to be released

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

Pointers: why they are error prone?

C++98

They need initialization — **Seg Fault**

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

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

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

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

They need initialization — **Seg Fault**

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

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

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

They need clear ownership

```
char *s = (char*) malloc(...);
strncpy(s, ...);
someVector.push_back(s);
someSet.add(s);
std::thread t1(vecConsumer, someVector);
std::thread t2(setConsumer, someSet);
```

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

They need initialization — Seg Fault

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

They need to be released — Memory leak

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

They need clear ownership — Who should release ?

```cpp
char *s = (char*) malloc(...);
strncpy(s, ...);
someVector.push_back(s);
someSet.add(s);
std::thread t1(vecConsumer, someVector);
std::thread t2(setConsumer, someSet);
```

This problem exists for any resource

C++11

For example with a file

```
try {
  FILE *handle = std::fopen(path, "w+");
  if (nullptr == handle) { throw ... }
  if (std::fputs(str, handle) == EOF) {
    throw ...
  }
  fclose(handle);
} catch (...) { ... }
```

Resource Acquisition Is Initialization (RAII) C++98

Practically

Use object semantic to acquire/release resources

- wrap the resource inside an object
- acquire resource in constructor
- release resource in destructor
- create this object on the stack so that it is automatically destructed when leaving the scope, including in case of exception
- use move semantics to pass the resource around

RAII in practice

C++98

File class

```
class File {
public:
  File(const char* filename) :
    m_file_handle(std::fopen(filename, "w+")) {
    if (m_file_handle == NULL) { throw ... }
  }
  ~File() { std::fclose(m_file_handle); }
  void write (const char* str) {
    if (std::fputs(str, m_file_handle) == EOF) {
      throw ...
    }
  }
private:
  FILE* m_file_handle;
};
```

RAII usage

C++98

Usage of File class

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

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

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

- on real projects, use `std::fstream` to handle files

std::unique_ptr

C++11

an RAII pointer

- wraps a regular pointer
- has move only semantic
  - the pointer has unique ownership
  - copying will result in a compile error
- in `<memory>` header

std::unique_ptr

C++11

an RAII pointer

- wraps a regular pointer
- has move only semantic
  - the pointer has unique ownership
  - copying will result in a compile error
- in `<memory>` header

Usage

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

Quiz

C++11

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

What do you expect ?

Quiz

C++11

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

What do you expect ?

Compilation Error

```
test.cpp:15:5: error: call to deleted constructor
of 'std::unique_ptr<Foo>'
  f(uptr);
    ^~~~
/usr/include/c++/4.9/bits/unique_ptr.h:356:7: note:
 'unique_ptr' has been explicitly marked deleted here
 unique_ptr(const unique_ptr&) = delete;
 ^
```

std::make_unique

C++14

- directly allocates a unique_ptr
- no `new` or `delete` calls anymore!

std::make_unique

$C^{++}14$

- directly allocates a unique_ptr
- no `new` or `delete` calls anymore!

make_unique usage

```
// allocation of one Foo object,
// calls new Foo(arg1, arg2) internally
auto a = std::make_unique<Foo>(arg1, arg2);
std::cout << a.get() << " points to "
          << a->someMember << '\n';
// allocation of an array of Foos
// calls default constructor
auto b = std::make_unique<Foo[]>(10);
// deallocations at end of scope
```

RAII or raw pointers

C++11

When to use what ?

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

RAII or raw pointers

C++11

When to use what ?

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

A question of ownership

```
unique_ptr<T> producer();
void observer(const T&);
void modifier(T&);
void consumer(unique_ptr<T>);
unique_ptr<T> pt{producer()}; // Receive ownership
observer(*pt);                // Keep ownership
modifier(*pt);                // Keep ownership
consumer(std::move(pt));      // Transfer ownership
```

unique_ptr usage summary $C^{++}11$

It's about lifetime management

- Use `unique_ptr` in functions taking part in lifetime management
- Otherwise use raw pointers or references

shared_ptr, make_shared

C++11

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

make_shared : creates a shared_ptr

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

smart pointers

C++98

Exercise Time

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

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

- Go through `problem1()` to `problem3()` and fix the leaks using smart pointers.
- `problem4()` is the most difficult. Skip if not enough time.

This is the end

Questions ?

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