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
  - 405 slides, 542 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

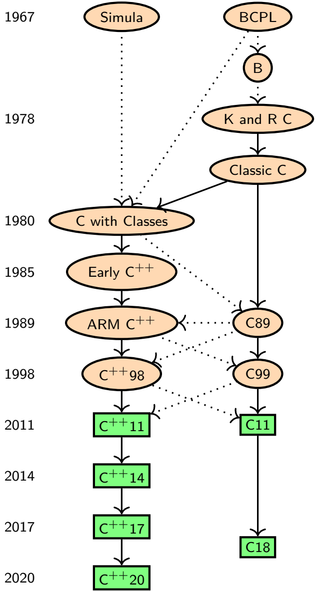
Detailed outline

History and goals

History

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

C/C++ origins



C inventor
Dennis M. Ritchie

C++ inventor
Bjarne Stroustrup

- Both C and C++ are born in Bell Labs
- C++ *almost* embeds C
- C and C++ are still under development
- We will discuss all C++ specs 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

Core syntax and types

Hello World

C++98

```cpp
#include <iostream>

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

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

Comments

C++98

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

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

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

Basic types(1)

C++98

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

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

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

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

Basic types(2) $C^{++}98$

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

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

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

float f = 1.23f;           // 32 (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

```
#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

```
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

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

```
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

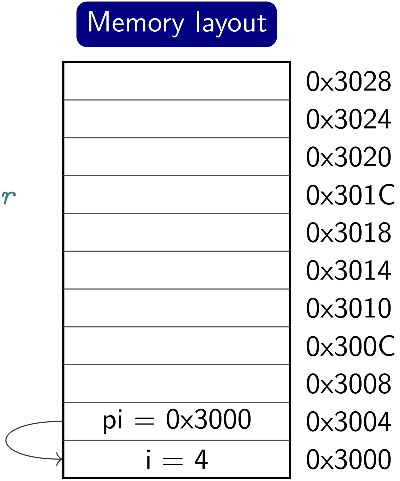
C++98

```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

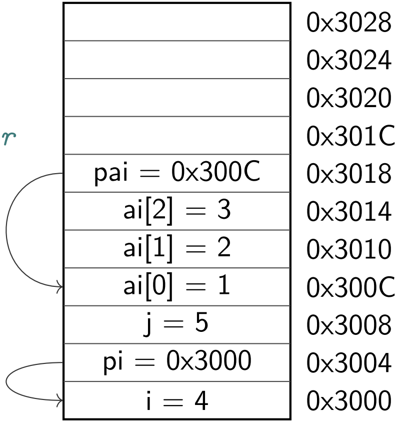
```cpp
int i = 4;
int *pi = &i;
int j = *pi + 1;

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



Pointers

C++98

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

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

// compile error
int *pak = k;

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

Memory layout

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

Pointers

C++98

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

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

// compile error
int *pak = k;

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

Memory layout

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

Pointers

C++98

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

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

// compile error
int *pak = k;

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

Memory layout

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

nullptr

C++11

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

```
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

Scope

C++98

Definition

Portion of the source code where a given name is valid

Typically :

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

Example

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

Scope and lifetime of variables

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!

```
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!

```
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

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

C++17

```
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

struct

C++98

"members" grouped together under one name

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

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

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

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

struct

C++98

“members” grouped together under one name

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

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

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

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

Memory layout

				0x3010
				0x300C
				0x3008
				0x3004
				0x3000

struct

C++98

"members" grouped together under one name

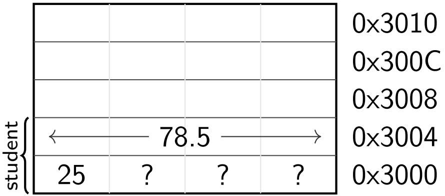
```cpp
struct Individual {
  unsigned char age;
  float weight;
};

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

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

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

Memory layout



struct

C++98

"members" grouped together under one name

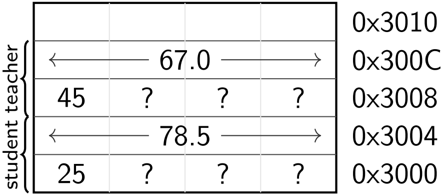
```cpp
struct Individual {
  unsigned char age;
  float weight;
};

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

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

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

Memory layout



struct

C++98

"members" grouped together under one name

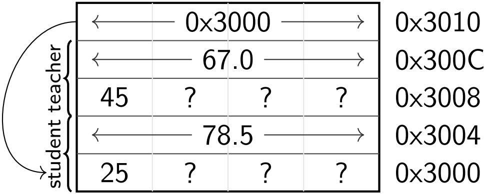
```cpp
struct Individual {
  unsigned char age;
  float weight;
};

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
```

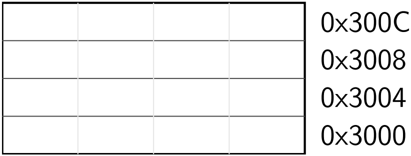
union

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



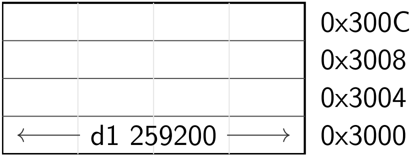
union

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

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

Memory layout

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

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

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

Only advantages

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

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

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

More sensible example

$C^{++}98$

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

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

More sensible example

$C^{++}98$

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

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

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

More sensible example

$C^{++}98$

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

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

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

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

```

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

typedef and using

C++98 / C++11

Used to create type aliases

C++98

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

C++11

```
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:

```
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

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

$C^{++}98$

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

```

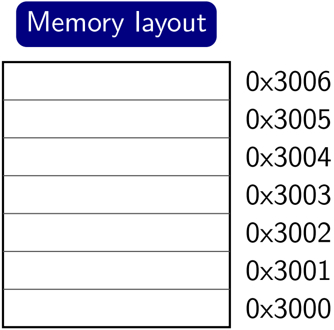
```cpp
// multiple default
// arguments are possible
int add(int a = 2,
        int b = 2) {
  return a + b;
}
// add() == 4
// add(3) == 5
```

Functions: parameters are passed by value C++98

```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

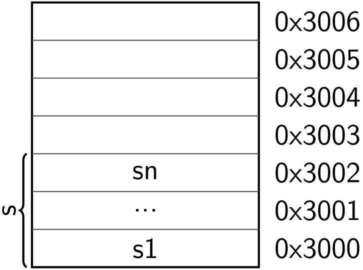
C++98

```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
```

Memory layout

		0x3006
p	pn = sn	0x3005
p	...	0x3004
p	p1 = s1	0x3003
s	sn	0x3002
s	...	0x3001
s	s1	0x3000

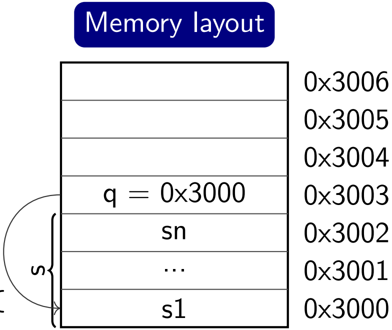
Functions: parameters are passed by value

$C^{++}98$

```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: 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
	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

	0x3008
p.a = 1	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 = 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

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

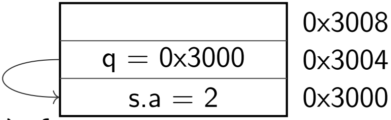
Functions: pass by value or reference? C++98

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

void 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 = 2	0x3000

Pass by value, reference or pointer

C++98

Different ways to pass arguments to a function

- by default, arguments are passed by value (= copy)
  good for small types, e.g. numbers
- 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

```
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

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

Operators(3)

C++98

Comparison Operators

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

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

```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**

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

Details can be found on cppreference

Operators(3)

C++98

Comparison Operators

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

Precedences

Don't use - use parentheses

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

Details can be found on cppreference

Control structures

Control structures: if

$C^{++}98$

if syntax

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

- `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

```
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

```
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

```cpp
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

$C^{++}17$

```
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

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

- braces are optional if the body is a single statement

Bad example

```cpp
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

```
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

$C^{++}98$

```
// 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

```
// 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

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.

Object orientation (OO)

Objects and Classes

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

What are classes and objects $C^{++}98$

Classes (or “user-defined types”)

C structs on steroids

- with inheritance
- with access control
- including methods

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

```
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

```cpp
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

```
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

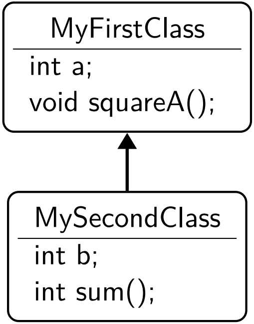
3 Object orientation (OO)
- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Type casting
- Operators
- Functors
- Name Lookups

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
```



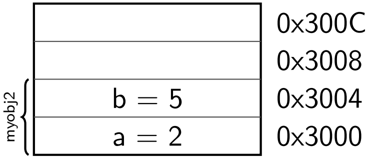
First inheritance

C++98

```
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
```

Memory layout



Managing access to class members $C^{++}98$

public / *private* keywords

private allows access only within the class

public allows access from anywhere

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

Managing access to class members

C++98

public / *private* keywords

private allows access only within the class

public allows access from anywhere

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

```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

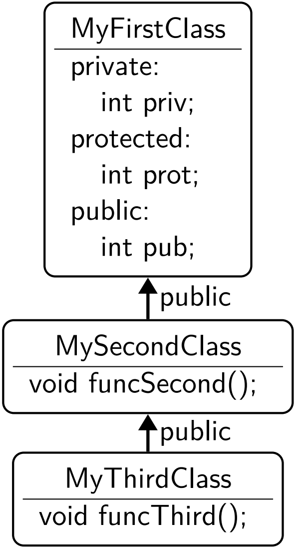
- only public members of public inheritance are accessible

Net result for grand child code

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

Managing inheritance privacy - public

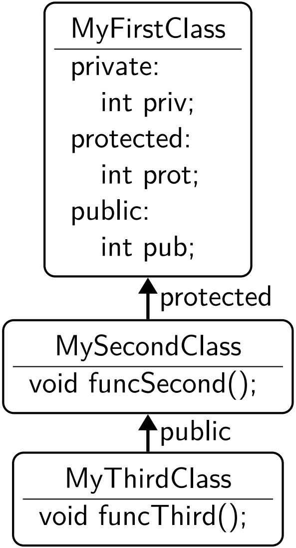
$C^{++}98$




```
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

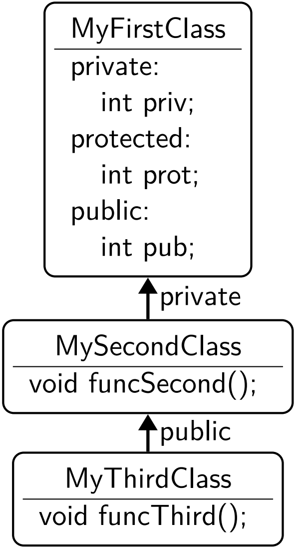
C++98




```
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
}
```

Final class

C++11

Idea

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

Practically

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

Constructors/destructors

3 Object orientation (OO)

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 $\sim$

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

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

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

~MyFirstClass() {}
```

Class Constructors and Destructors

C++98

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

Constructors and inheritance

C++98

```cpp
struct 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 $C^{++}98$

```cpp
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

```cpp
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.

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

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

Explicit unary constructor

C++98

Concept

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

```
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

```cpp
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

```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();            // 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 {}

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

$C^{++}98$ nightmare

```
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

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

3 Object orientation (OO)

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)

```cpp
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

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

Advanced OO

Polymorphism

C++98

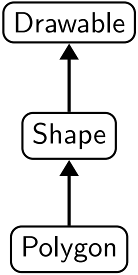
the concept

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

```cpp
Polygon p;

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

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



Polymorphism

C++98

the concept

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

```cpp
Polygon p;

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

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

Memory layout

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

Polymorphism

C++98

the concept

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

```cpp
Polygon p;

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

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

Memory layout

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

Inheritance privacy and polymorphism

C++98

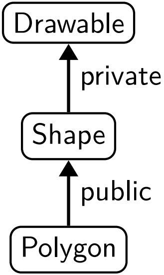
Only public inheritance is visible to code outside the class

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

```
Polygon p;

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

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



Method overriding

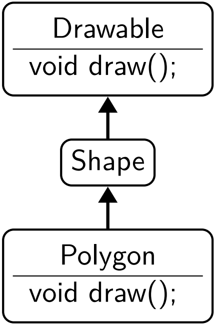
C++98

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

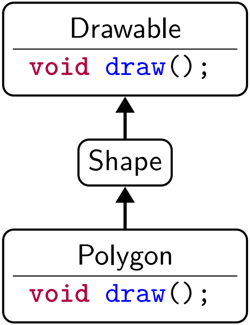
$C^{++}98$

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

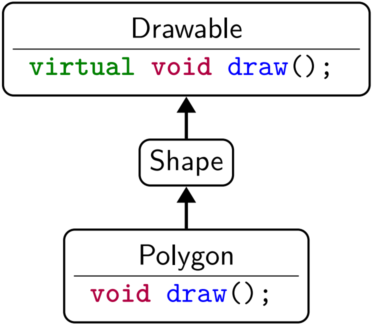
C++98

the concept

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

```
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

final keyword

C++11

Idea

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

Practically

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

Pure Virtual methods

C++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

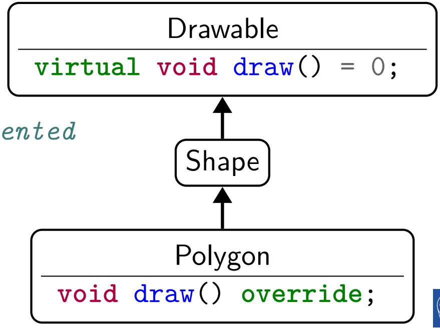
Concept

- 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();
```

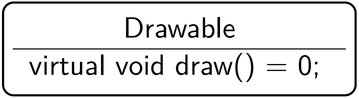


Pure Abstract Class aka Interface $C^{++}98$

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;
}
```



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

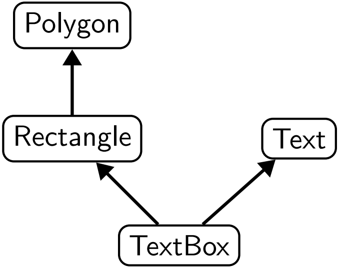
- go to code/polymorphism
- look at the code
- open trypoly.cpp
- create a Pentagon, call its perimeter method
- create a Hexagon, call its perimeter method
- create a Hexagon, call its parent's perimeter method
- retry with virtual methods

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
}
```

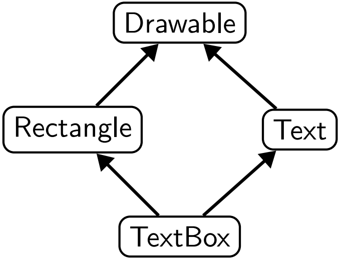
The diamond shape

Definition

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

Problem

- are the members of the grand parent replicated?



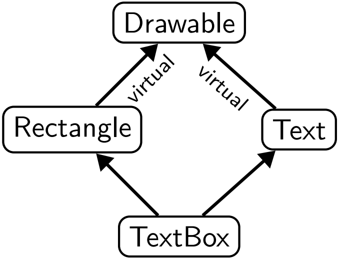
Virtual inheritance

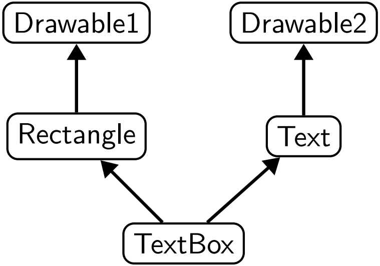
C++98

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

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.

```cpp
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

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

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

```
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

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

```
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);
```

Name Lookups

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

Basics of name lookup

C++98

Example code

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

How to find the declaration of a name ?

Mainly 2 cases :

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

Unqualified name lookup and ADL

$C^{++}98$

Ordered list of scopes (simplified)

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

Argument Dependent Lookup (simplified)

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

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

ADL consequences (1)

C++98

Use standalone/non-member functions

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

Don't write :

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

Prefer :

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

Advantages :

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

Notes :

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

ADL consequences (2)

C++17

Prefer nested namespaces to using

Don't write :

```cpp
using namespace MyProject;
funcFromMyProject(...);
```

Prefer :

```cpp
namespace MyProject::MySubpart {
  funcFromMyProject(...);
}
```

And let the compiler lookup (more easily) the function name

ADL consequences (3)

C++98

Customization points and using

Don't write :

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

Prefer :

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

Advantages :

- allows to use `std::swap` by default
- but also to benefit from any dedicated specialization

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

Core modern C++

1. History and goals
2. Language basics
3. Object orientation (OO)
4. Core modern C++
   - Constness
   - Constant Expressions
   - Exceptions
   - Move semantics
   - Copy elision
   - Templates
   - The STL
   - More STL
   - Lambdas
   - pointers and RAII
5. Expert C++
6. Useful tools
7. Concurrency
8. C++ and python

Constness

4 Core modern C++

- Constness
- Constant Expressions
- Exceptions
- Move semantics
- Copy elision
- Templates
- The STL
- More 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

```
// 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

```cpp
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

$C^{++}98$

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

Constant Expressions

4 Core modern C++

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

Generalized Constant Expressions

$C^{++}11$

Reason of being

- compute constant expressions at compile time
- even if non trivial

Generalized Constant Expressions

C++11

Reason of being

- compute constant expressions at compile time
- even if non trivial

Example

```
constexpr int f(int x) {
  return x > 1 ? x * f(x - 1) : 1;
}
constexpr int a = f(5); // computed at compile time
```

Generalized Constant Expressions $C^{++}11$

Reason of being

- compute constant expressions at compile time
- even if non trivial

Example

```
constexpr int f(int x) {
  return x > 1 ? x * f(x - 1) : 1;
}
constexpr int a = f(5); // computed at compile time
```

Example with $C^{++}14$

```
constexpr int f(int x) {
  if (x > 1) return x * f(x - 1);
  return 1;
}
constexpr int a = f(5); // computed at compile time
```

Static Assertions

$C^{++}11$

static_assert declaration

- Performs compile time assertions; meaning a failed assertion stops compilation
- The expression has to be a constexpr boolean expression
- Purely evaluated at compile time, no effect at runtime
- Often used in template programming to make assertion on types, can be used to validate boolean compile time expressions

Static Assertions

C++11

static_assert declaration

- Performs compile time assertions; meaning a failed assertion stops compilation
- The expression has to be a constexpr boolean expression
- Purely evaluated at compile time, no effect at runtime
- Often used in template programming to make assertion on types, can be used to validate boolean compile time expressions

Example

```
constexpr int f(int x) {
  return x > 1 ? x * f(x - 1) : 1;
}
static_assert(f(5)==120,"Expected f(5) to be 120!");
```

Generalized Constant Expressions(2) $C^{++}14$

Few limitations in C++14 (more in C++11)

- function's body cannot contain try-catch, uninitialized or static variables - details on cppreference
- arguments should be constexpr or literals in order to benefit from compile time computation

Notes

- classes can have constexpr member functions
- objects can be constexpr
    - if the constructor of their class is
- a constexpr function can also be used normally
- but a constexpr variable has to be evaluated at compile time

Real life example

C++11

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

Real life example(2)

$C^{++}11$

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

Exceptions

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, …

```cpp

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

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

  ```cpp
  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

  ```cpp
  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();
}
```

Move semantics

4 Core modern C++

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

Move semantics: the problem

C++11

Non efficient code

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

Move semantics: the problem

C++11

Non efficient code

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

What happens during swap

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

Move semantics: the problem

C++11

Dedicated efficient code

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

Move semantics: the problem

C++11

Dedicated efficient code

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

What happens during swap

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

Move semantics: the problem

C++98

Another potentially non efficient code

```
MyVector<int> vrandom(unsigned int n) {
  MyVector<int> result(n);
  ... // fill result
  return result;
}
MyVector<int> v = vrandom(10000);
```

Move semantics: the problem $C^{++}98$

Another potentially non efficient code

```
1 MyVector<int> vrandom(unsigned int n) {
2   MyVector<int> result(n);
3   ... // fill result
4   return result;
5 }
6 MyVector<int> v = vrandom(10000);
```

What could happen on line 4 and 6

- one unnecessary allocation and one release for 10k `ints`
- unnecessary copy of 10k `ints`
- The compiler may optimize the copy away, but there is no guarantee (before $C^{++}17$)

Move semantics: the problem

C++98

Dedicated efficient way before C++11

```
void vrandom(unsigned int n, MyVector<int> &v) {
  v.resize(n);
  ... // fill result
}
MyVector<int> v;
vrandom(10000, v);
```

Move semantics: the problem

C++98

Dedicated efficient way before C++11

```
void vrandom(unsigned int n, MyVector<int> &v) {
  v.resize(n);
  ... // fill result
}
MyVector<int> v;
vrandom(10000, v);
```

The ideal situation

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

Move semantics

$C^{++}11$

The idea

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

a move constructor similar to copy constructor

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

Move semantics

C++11

The idea

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

  a move constructor similar to copy constructor

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

Practically

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

Move semantics $C^{++}11$

A few points

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

Move semantics

$C^{++}11$

A few points

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

Practically

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

Move semantics

C++11

In some cases, you want to force a move

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

Move semantics

C++11

In some cases, you want to force a move

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

Explicitly request moving

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

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

Move semantics: recommended implementation C++11

Use copy and swap idiom

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

Move semantics: recommended implementation $C^{++}11$

Practically

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

Move semantics: alternative implementation $C^{++}11$

Practically

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

Move Semantic

$C^{++}11$

Exercise Time

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

prerequisite : be able to use simple templated code

Copy elision

Guaranteed copy elision

$C^{++}17$

What is copy elision

```
struct Foo { ... };
Foo f() {
  return Foo();
}
int main() {
  // compiler was authorised to elide the copy
  Foo foo = f();
}
```

From $C^{++}17$ on

The elision is guaranteed.

Guaranteed copy elision $C^{++}17$

Allows to write code not allowed with $C^{++}14$ (would not compile)

One case where the guarantee is needed

```cpp
struct Foo {
  Foo() { ... }
  Foo(const Foo &) = delete;
  Foo(Foo &&) = delete;
  Foo& operator=(const Foo &) = delete;
  Foo& operator=(Foo &&) = delete;
};
Foo f() {
  return Foo();  // ok
}
int main() {
  Foo foo = f(); // ok
}
```

Templates

4 Core modern C^{++}

- Constness
- Constant Expressions
- Exceptions
- Move semantics
- Copy elision
- Templates
- The STL
- More 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

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

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

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 $C^{++}20$

```cpp
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

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

```
#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

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

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

STL in practice

C++14

```
#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)

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

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

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

STL way (since C++98)

```cpp
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

$C^{++}98$

```
// 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

```cpp
// 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!

More STL

std::string_view

C++17

Non owning view of a continuous char sequence

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

Some example uses

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

std::optional $C^{++}17$

Manages an optional contained value

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

Code example

```cpp
std::optional<Phone> parse_phone(std::string_view in) {
  if (is_valid_phone(in))
    return in;   // equiv. to optional<Phone>(in);
  else
    return {};   // default constructs std::nullopt
}
auto v = parse_phone(...);
if (v) {               // alternatively v.is_valid()
  process_phone(v.value()); // *v is equivalent
}
```

std::variant

C++17

a type-safe union

- Allows the variable to hold any of the given types
- `std::get` reads the value of the variant
- and throws `std::bad_variant_access` for bad accesses
- Makes it easy to implement visitor pattern

Code example

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

// Or check the type before accessing it
if (std::holds_alternative<float>(opt))
    std::cout << std::get<float>(opt);
```

std::variant and the visitor pattern

C++17

std::visit

- Applies a "visitor" to given variant
- A visitor is a callable able to handle the different types

Practically

```cpp
using option_t = std::variant<int,float,string>;
struct Visitor {
  void operator() (int i) { std::cout<< "i32:"<< i;}
  void operator() (float f) { std::cout<< "f32:"<< f;}
  void operator() (string s) { std::cout<< "s:"<< s;}
};
void print_opt(option_t opt) {
  std::visit(Visitor{}, opt);
  std::cout << "\n";
}
print_opt(100); print_opt(3.14f); print_opt("example");
```

std::any

C++17

a type-safe container for single values of any type

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

Code example

```cpp
std::any val{100};            // holding int
val = std::string("hello"); // holding string
std::string s = std::any_cast<std::string>(val);
try {
  int val = std::any_cast<int>(val); // will throw
} catch (std::bad_any_cast const& ex) {...}
// Or check the type before accessing it
if (val.type() == typeid(int))
  std::cout << std::any_cast<int>(val);
```

non-member begin and end

C++11

The problem in C++98

STL containers and arrays have different syntax for loop

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

non-member begin and end

C++11

The problem in C++98

STL containers and arrays have different syntax for loop

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

A new syntax

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

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

The problem in $C^{++}98$

STL containers and arrays have different syntax for loop

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

A new syntax

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

Allowing the best syntax

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

Structured Binding Declarations $C^{++}17$

Helps when using `std::tuple` or tuple-like types as a return type.
Automatically creates variables and ties them.

$C^{++}14$

```
void foo(std::tuple<int, double, long> tuple) {
  int a = 0;
  double b = 0.0;
  long c = 0;
  // a, b, c need to be declared first
  std::tie(a, b, c) = tuple;
```

$C^{++}17$

```
void foo(std::tuple<int, double, long> tuple) {
  auto [ a, b, c ] = tuple; ...
}
for (const auto& [key, value] : map) { ... }
```

compile-time branches

C++17

if constexpr

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

Example code

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

Lambdas

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

```
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

```
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

```cpp
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

$C^{++}11$

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

4 Core modern C^{++}

- Constness
- Constant Expressions
- Exceptions
- Move semantics
- Copy elision
- Templates
- The STL
- More 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

```
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

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

They need to be released

```
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

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

They need to be released — Memory leak

```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, ...);
```

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**

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

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

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

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

```
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

```cpp
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

```
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

```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 ?

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 ?

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

```cpp
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

```cpp
{
  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.

Expert C++

Variadic templates

5 Expert C++

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

Basic variadic template

C++11

The idea

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

Recursive example

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

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

A couple of remarks

C++11

About performance

- do not be afraid of recursion
- everything is at compile time!
- unlike C-style variadic functions

Why is it better than variadic functions

- it's more performant
- type safety is included
- it applies to everything, including objects

Fold expressions

C++17

The idea

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

Example

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

Variadic class template

C++11

The tuple example, simplified

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

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

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

Perfect forwarding

The problem

C++11

How to write a generic wrapper function?

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

Example usage :

- emplace_back
- make_unique

Why is it not so simple? C++11

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

What about references ?

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

Second try, second failure ? C++11

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

- `const T&` won't work when passing something non const
- rvalues are not supported in either case

The solution: cover all cases

C++11

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

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

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

The new problem: scaling to n arguments

C++11

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

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

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

Exploding complexity

3^n complexity

you do not want to try n = 5...

Reference collapsing

C++11

Reference to references

- They are forbidden, but some compilers allow them

```cpp
template <typename T>
void foo(T t) { T& k = t; } // int& &
int ii = 4;
foo<int&>(ii);
```

C++11 added rvalues

- what about `int&& &` ?
- and int `&& &&` ?

Rule

`&` always wins

`&& &`, `& &&`, `& &` → `&`

`&& &&` → `&&`

rvalue in type-deducing context

C++11

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

Next to a template parameter, `&&` is not an rvalue, but a “forwarding reference” (aka. “universal reference”)

`T&&` actual type depends on the arguments passed to func

- if an lvalue of type U is given, T is deduced to `U&`
- otherwise, collapse references normally

```
func(4);          // rvalue -> T&& is int&&
double d = 3.14;
func(d);          // lvalue -> T&& is double&
float f() {...}
func(f());        // rvalue -> T&& is float&&
int foo(int i) {
  func(i);        // lvalue -> T&& is int&
}
```

std::remove_reference

C++11

Type trait to remove reference from a type

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

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

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

If `T` is a reference type, `remove_reference_t<T>::type` is the type referred to by `T`, otherwise it is `T`.

std::forward

C++11

Keeps references and maps non-reference types to rvalue references

```
template<typename T>
T&& forward(typename std::remove_reference<T>
               ::type& t) noexcept {
  return static_cast<T&&>(t);
}
template<typename T>
T&& forward(typename std::remove_reference<T>
               ::type&& t) noexcept {
  return static_cast<T&&>(t);
}
```

How it works

- if T is `int`, it returns `int&&`
- if T is `int&`, it returns `int& &&` ie. `int&`
- if T is `int&&`, it returns `int&& &&` ie. `int&&`

Perfect forwarding

C++11

Putting it all together

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

- if we pass an rvalue to wrapper (`U&&`)
  - arg will be of type `U&&`
  - func will be called with a `U&&`
- if we pass an lvalue to wrapper (`U&`)
  - arg will be of type `U&`
  - func will be called with a `U&`
- if we pass a plain value (`U`)
  - arg will be of type `U&&` (no copy in wrapper)
  - func will be called with a `U&&`
  - but func takes a `U&`, so copy happens there, as expected

Real life example

C++11

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

SFINAE

5 Expert C++
- Variadic templates
- Perfect forwarding
- SFINAE
- Concepts
- The <=> operator

Substitution Failure Is Not An Error (SFINAE) $C^{++}11$

The main idea

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

Example

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

void f(int a) { ... }

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

Note : SFINAE is largely superseded by concepts in $C^{++}20$

decltype

C++11

The main idea

- gives the type of the result of an expression
- the expression is not evaluated
- at compile time

Example

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

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

declval

C++11

The main idea

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

```cpp
struct Default {
  int foo() const { return 1; }
};
struct NonDefault {
  NonDefault(int i) { }
  int foo() const { return 1; }
};
decltype(Default().foo()) n1 = 1;      // int
decltype(NonDefault().foo()) n2 = n1; // error
decltype(std::declval<NonDefault>().foo()) n2 = n1;
```

true_type and false_type

C++11

The main idea

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

Example

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

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

Using SFINAE for introspection

C++11

The main idea

- use a template specialization
  - that may or may not create valid code
- use SFINAE to choose between them
- inherit from true/false_type

Example

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

Not so easy actually... C++11

Example

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

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

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

Using void_t

C++17

Concept

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

Implementation in header type_traits

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

Previous example using void_t

C++17

Example

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

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

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

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

SFINAE and the STL

C++11/C++14/C++17

enable_if / enable_if_t

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

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

is_*< T >/is_*_v< T > (float/signed/object/final/abstract/...)

- Standard type traits in header `type_traits`
- Checks at compile time whether T is ...
- Result in boolean member `value`

Gaudi usage example

```
constexpr struct deref_t {
  template
    <typename In,
     typename = typename std::enable_if_t
                  <!std::is_pointer_v<In>>>
  In& operator()(In& in) const { return in; }

  template <typename In>
  In& operator()(In* in) const {
    assert(in!=nullptr); return *in;
  }
} deref{};
```

Back to variadic templated class

C++11

The tuple get method

```
template <size_t I, typename Tuple>
struct elem_type;

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

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

Back to variadic templated class

C++11

The tuple get function

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

with if constexpr

C++17

The tuple get function

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

Best practice

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

Concepts

5 Expert C++

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

Requirements and concepts $C^{++}20$

Motivation

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

The world before concepts

$C^{++}20$

$C^{++}17$ work around : SFINAE

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

Practical code

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

The world before concepts

C++20

C++17 work around : SFINAE

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

Practical code

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

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

Basic requirements

$C^{++}20$

A new keyword

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

$C^{++}20$ code

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

Basic requirements

$C^{++}20$

A new keyword

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

$C^{++}20$ code

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

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

Requirements and overloads

C++20

Example of several competing templates

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

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

Requirements affect overload resolution

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

Concepts

C++20

Definition

- a **concept** gives a name to a given set of requirements
- useful when requirements are reused often

A new keyword : *concept*

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

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

Some usages of concepts

$C^{++}20$

Concepts as template parameters

- concepts can be used in template parameter lists
- replacing *typename*

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

Concepts in abbreviated function arguments

- concepts can be used together with *auto* in abbreviated function templates

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

Standard concepts

C++20

Never reinvent the wheel

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

E.g. : the floating point concept

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

Concepts checking

$C^{++}20$

Concepts as boolean operators

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

Using concepts with *if constexpr*

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

Advanced requirements overview

C++20

requires as an expression

requires can express more than basic requirements. It can

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

Practically

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

Remember : use standard concepts first

Requirements and concepts $C^{++}20$

To be remembered

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

The <=> operator

5 Expert C++
- Variadic templates
- Perfect forwarding
- SFINAE
- Concepts
- The <=> operator

The three-way comparison operator

C++20

Motivation

- There are frequent situations where one needs < operator for a home-made class. Typically, if you want to sort a `std::vector` of such objects, or use them as a key for e.g. `std::set` or `std::map`.
- For completeness, one should also add >, >=, and <=, implemented reusing either < and ==, or < and >.
- Those operators should be defined as free functions, optionally friends, so that left and right arguments will be similarly convertible.
- Much boilerplate code to write. Too much...

The three-way comparison operator C++20

Idea

- C++20 introduces the **spaceship** operator: `<=>`. Well, the real official name is **three-way comparison operator**.
- It is provided by default for all predefined types, and returns **something** which can be compared to `0` (similar to `std::strcmp`).
- Greater, lower or equal to `0` means respectively **lower than** , **greater than** and **equivalent to**.

The three-way comparison operator

C++20

Example

```
template <typename T>
void three_way_compare( T lhs, T rhs )
{
  auto res = (lhs<=>rhs) ;
  std::cout
    <<"("<<lhs<<"<=>"<<rhs<<") : "
    <<(res<0)<<(res==0)<<(res>0)
    <<std::endl ;
}

int main()
{
  three_way_compare(1,2) ;
  three_way_compare(2,2) ;
  three_way_compare(2,1) ;
}
```

```
(1<=>2): 100
(2<=>2): 010
(2<=>1): 001
```

The three-way comparison operator C++20

Different kinds of ordering

- The real return type of `<=>` for integers is `std::strong_ordering`: whatever the values, you will always get `true` for exactly one test among `<0`, `==0`, and `>0`.
- On the contrary, the return type of `<=>` for floating point numbers is `std::partial_ordering`, because sometimes all tests may return `false`, typically if one one the number is `NaN`.

The three-way comparison operator

C++20

Example

```
int main()
{
  three_way_compare(+0.,-0.) ;
  three_way_compare(0./1.,1./0.) ;
  three_way_compare(0.,0./0.) ;
}
```

```
(0<=>-0)   : 010
(0<=>inf)  : 100
(0<=>-nan) : 000
```

The three-way comparison operator C++20

Different kinds of ordering

- Between the two, we have a class `std::weak_ordering`, where ==0 means that the two compared values are **equivalent** from a ranking point of view, but not necessarily **equal**. In any given expression, one cannot substitutes one value for the other and be sure to have the same result.
- I am not aware of some predefined type whose <=> would return an instance of `std::weak_ordering`, but it may make sense for some home-made class, such as the following.
- In the next example, we define a very basic class for positive rational numbers, and provide an implementation of <=>.

The three-way comparison operator C++20

Example

```
struct Ratio
{
  unsigned n, d ;
  friend std::weak_ordering operator<=>( Ratio const & a, Ratio const & b )
  { return (a.n*b.d)<=>(a.d*b.n) ; }
  friend std::ostream & operator<<( std::ostream & os, Ratio const & r )
  { return (os<<r.n<<'/'<<r.d) ; }
} ;

std::ostream & operator<<( std::ostream & os, std::weak_ordering cmp )
{ return (os<<(cmp<0)<<(cmp==0)<<(cmp>0)) ; }
```

Home-made class

- We convert above the result of `<=>` to `std::weak_ordering`, so to emphasize that `(a<=>b)==0` means that `a` and `b` are logically equivalent, but may lead to different results in other expressions (e.g. printing them).

The three-way comparison operator

C++20

Example

```cpp
template <typename T>
void compare( T lhs, T rhs )
{
  //std::cout<<lhs<<" == != "<<rhs<<": "<<(lhs==rhs)<<std::endl ;
  //  <<(lhs==rhs)<<" "<<(lhs!=rhs)
  //  <<std::endl ;
  std::cout<<lhs<<" <=> "<<rhs<<" (< <= > >=) : "<<(lhs<=>rhs)
    <<" ("<<(lhs<rhs)<<" "<<(lhs<=rhs)
    <<" "<<(lhs>rhs)<<" "<<(lhs>=rhs)<<")"
    <<std::endl ;
}
int main()
{
  compare<Ratio>({ 3, 4 },{ 2, 3 }) ;
  compare<Ratio>({ 3, 6 },{ 2, 3 }) ;
  compare<Ratio>({ 1, 2 },{ 2, 4 }) ;
}
```

```
3/4 <=> 2/3 (< <= > >=) : 001 (0 0 1 1)
3/6 <=> 2/3 (< <= > >=) : 100 (1 1 0 0)
1/2 <=> 2/4 (< <= > >=) : 010 (0 1 0 1)
```

The three-way comparison operator C++20

Home-made class

- As one can see from previous code, we have defined only <=>, but <, >, <= and => work as well. The compiler can express the latter operators in terms of <=>. Of course, one can also provide own implementations.
- It has NOT added a default implementation for == and !=. Those operators are generally expected to mean **equal**, rather than **equivalent**. If <=> does not provides a strong order, it is generally advised not to define ==.

The three-way comparison operator C++20

Default <=> implementation

- One can ask the compiler to provide a default implementation for <=> and/or ==. Logically enough, it will compare the first member variable of the two objects, and goes on to the next member variable as long as the current ones are equivalent.
- In the previous example, that would be wrong, because it will compare the numerators first, and conclude that 3/6 is greater than 2/3:
- On the contrary, for some tuple-like class like the next example, it makes sense.

The three-way comparison operator

C++20

Example

```
struct Grade
{
  double number ;
  char letter ;
  std::string name ;
  auto operator<=>( Grade const & other ) const = default ;
  std::ostream & operator<<( std::ostream & os, Grade const & g )
  { return (os<<g.letter<<", "<<g.number<<", "<<g.name) ; }
} ;

int main()
{
  std::set<Grade> grades
  { { 19,'A',"Djamila"}, {12,'C',"Charles"}, {16.5,'A',"Marc"} } ;
  for ( auto const & grade : grades )
  { std::cout<<grade<<std::endl ; }
}
```

```
C, 12, Charles
A, 16.5, Marc
A, 19, Djamila
```

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

Default <=> implementation

- We see above that the definition of <=> has been provided by the compiler, and the use of < by `std::set` to sorts its elements has been rewritten by the compiler in terms of <=>.
- If <=> is defaulted and no == is defined, then the compiler also provides a defaulted ==.

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

Summary

- Defining <=> allows you to use <, >,<=, and >= as well.
- The standard library defines a few different kinds of order (strong, weak and partial).
- If <=> does not define a strong order, avoid to define and/or use ==.

Resources

- https://blog.tartanllama.xyz/spaceship-operator/
- https://iq.opengenus.org/spaceship-operator-cpp/
- https://www.jonathanmueller.dev/talk/meetingcpp2019/
- https://quuxplusone.github.io/blog/2021/10/22/hidden-friend-outlives-spaceship/

Useful tools

C++ editor

6 Useful tools

- C++ editor
- Code management
- Code formatting
- The Compiling Chain
- Debugging
- Address Sanitizer
- The Valgrind family
- Static code analysis
- Profiling

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

6. Useful tools
   - C++ editor
   - Code management
   - Code formatting
   - The Compiling Chain
   - Debugging
   - Address Sanitizer
   - The Valgrind family
   - Static code analysis
   - Profiling

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

6 Useful tools
- C^{++}editor
- Code management
- Code formatting
- The Compiling Chain
- Debugging
- Address Sanitizer
- The Valgrind family
- Static code analysis
- Profiling

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

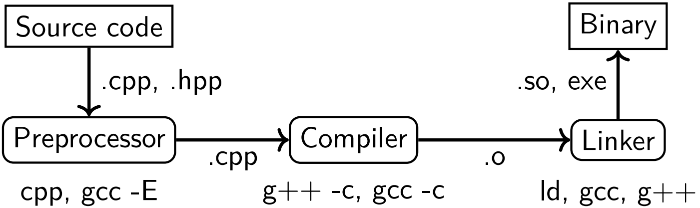
Exercise Time

- 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

6. Useful tools
   - C^{++}editor
   - Code management
   - Code formatting
   - The Compiling Chain
   - Debugging
   - Address Sanitizer
   - The Valgrind family
   - Static code analysis
   - Profiling

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

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

Address Sanitizer

6 Useful tools
- C++editor
- Code management
- Code formatting
- The Compiling Chain
- Debugging
- Address Sanitizer
- The Valgrind family
- Static code analysis
- Profiling

Address Sanitizer (asan)

asan introduction

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

Address Sanitizer (asan)

asan introduction

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

Usage (gcc/clang syntax)

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

Address Sanitizer (asan)

How it works

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

asan instrumentation (mock code)

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

Address Sanitizer (asan)

How it works

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

asan instrumentation (mock code)

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

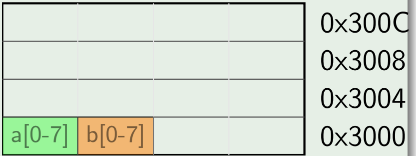
asan red zones

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

Illegal access (not detected without asan)

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

Memory layout



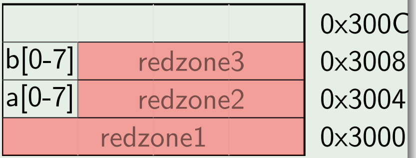
asan red zones

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

Illegal access (not detected without asan)

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

Memory layout



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

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

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

Finding memory leaks with asan

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

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

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

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

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

Address sanitizer (asan)

Wrap up

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

More info

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

Address sanitizer (asan)

Hands on

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

The Valgrind family

The valgrind family

Valgrind fundamentals

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

The valgrind family

Valgrind fundamentals

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

Main tools

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

memcheck

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

valgrind

Exercise Time

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

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

memcheck

Exercise Time

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

callgrind and kcachegrind

callgrind

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

kcachegrind

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

callgrind

Exercise Time

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

helgrind

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

helgrind

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

Note the "possible". It finds future problems !

helgrind

Exercise Time

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

Static code analysis

6 Useful tools

Static analysis

The problem

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

Static analysis

The problem

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

A (partial) solution : analyzing the source code

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

Static analysis

The problem

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

A (partial) solution : analyzing the source code

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

Existing tools

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

cppcheck

Exercise Time

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

clang-tidy

Documentation and supported checks

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

Run clang-tidy

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

Compilation flags

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

.clang-tidy file

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

Automatically collecting compilation flags

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

clang-tidy

Exercise Time

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

Profiling

6 Useful tools

- C^{++}editor
- Code management
- Code formatting
- The Compiling Chain
- Debugging
- Address Sanitizer
- The Valgrind family
- Static code analysis
- Profiling

Profiling

Conceptually

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

perf, VTune and uProf

perf

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

Intel VTune and AMD uProf

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

Concurrency

Threads and async

7 Concurrency
- Threads and async
- Mutexes
- Atomic types
- Condition Variables

Basic concurrency $C^{++}11$

Threading

- new object std::thread in <thread> header
- takes a function as argument of its constructor
- must be detached or joined before the main thread terminates
- $C^{++}20$: std::jthread automatically joins at destruction

Basic concurrency

$C^{++}11$

Threading

- new object std::thread in <thread> header
- takes a function as argument of its constructor
- must be detached or joined before the main thread terminates
- $C^{++}20$: std::jthread automatically joins at destruction

Example code

```cpp
void doSth() {...}
void doSthElse() {...}
int main() {
  std::thread t1(doSth);
  std::thread t2(doSthElse);
  for (auto t: {&t1,&t2}) t->join();
}
```

The thread constructor

C++11

Can take a function and its arguments

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

The thread constructor

C++11

Can take a function and its arguments

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

Can take any function-like object

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

Basic asynchronicity

C++11

Concept

- separation of the specification of what should be done and the retrieval of the results
- "start working on this, and ping me when it's ready"

Basic asynchronicity $C^{++}11$

Concept

- separation of the specification of what should be done and the retrieval of the results
- "start working on this, and ping me when it's ready"

Practically

- std::async function launches an asynchronous task
- std::future template allows to handle the result

Basic asynchronicity

C++11

Concept

- separation of the specification of what should be done and the retrieval of the results
- "start working on this, and ping me when it's ready"

Practically

- std::async function launches an asynchronous task
- std::future template allows to handle the result

Example code

```
int computeSth() {...}
std::future<int> res = std::async(computeSth);
std::cout << res->get() << std::endl;
```

Mixing the two

C++11

Is async running concurrent code ?

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

  std::launch::async spawns a thread for immediate execution

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

  - execution starts when get() is called
- default is not specified!

Mixing the two

C++11

Is async running concurrent code ?

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

  std::launch::async spawns a thread for immediate execution

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

  - execution starts when get() is called
- default is not specified!

Usage

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

Fine grained control on asynchronous execution $C^{++}11$

std::packaged_task template

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

Fine grained control on asynchronous execution $C^{++}11$

std::packaged_task template

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

Usage

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

Mutexes

Races

C++11

Example code

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

Races

C++11

Example code

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

What do you expect ? Try it in code/race

Races

C++11

Example code

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

What do you expect ? Try it in code/race

Anything between 100 and 200 !!!

Atomicity

C++11

Definition (wikipedia)

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

Practically

- an operation that won't run concurrently to another one
- an operation that will have a stable environment during execution

Atomicity

C++11

Definition (wikipedia)

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

Practically

- an operation that won't run concurrently to another one
- an operation that will have a stable environment during execution

Is ++ operator atomic ?

Atomicity

C++11

Definition (wikipedia)

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

Practically

- an operation that won't run concurrently to another one
- an operation that will have a stable environment during execution

Is ++ operator atomic ?

Usually not. It behaves like :

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

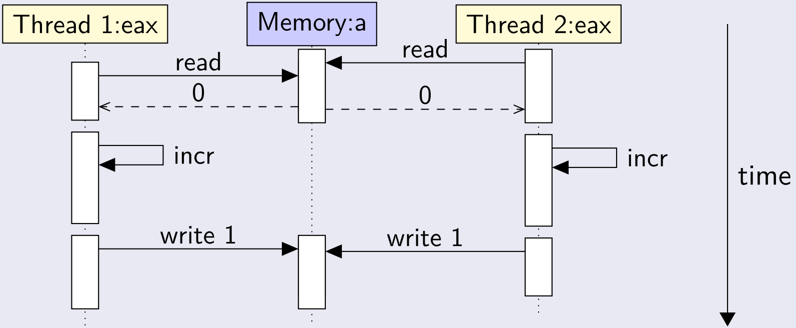
Timing

$C^{++}11$

Code

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

For 2 threads



Mutexes and Locks

C++17

Concept

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

Mutexes and Locks

C++17

Concept

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

The objects

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

Mutexes and Locks

C++17

Concept

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

The objects

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

Practically

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

Mutexes and Locks

$C^{++}17$

Good practice

- Generally use `scoped_lock` (C^{++}11: `lock_guard`)
- Hold as short as possible, wrap critical section in "`{ }`"
- Only if manual control needed, use `unique_lock`

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

Mutexes and Locks

Exercise Time

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

Dead lock

$C^{++}11$

Scenario

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

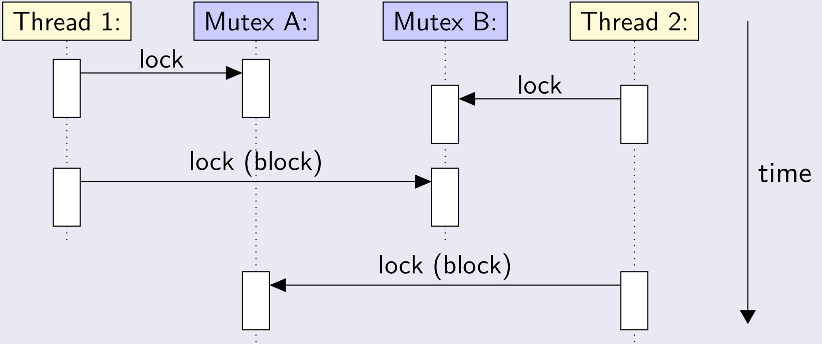
Dead lock

C++11

Scenario

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

Sequence diagram



How to avoid dead locks

C++11

Possible solutions

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

Shared mutex / locks

C++17

Sharing a mutex

- Normal `mutex` objects cannot be shared
- `shared_mutex` to the rescue, but can be slower
- *Either* exclusive *or* shared locking; never both

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

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

Atomic types

Atomic types in C^{++}

$C^{++}11$

std::atomic template

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

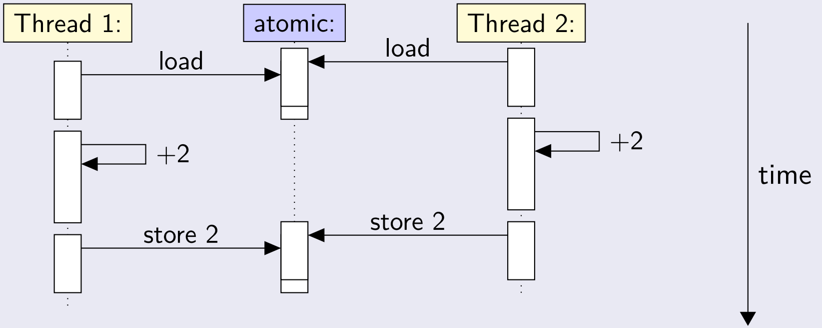
```cpp
std::atomic<int> a{0};
std::thread t1([&](){ a++; });
std::thread t2([&](){ a++; });
a += 2;
t1.join(); t2.join();
assert( a == 4 ); // Guaranteed to succeed
```

Expressions using an atomic type are *not* atomic!

- Atomic load; value+2; atomic store

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

Sequence diagram

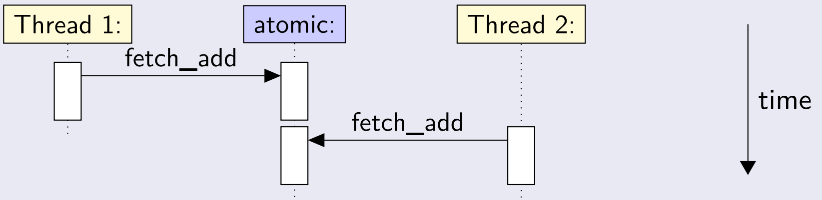


Use built-in atomic functions

- The built-in atomic functions are thread safe
- `fetch_add` (`operator+=()`): atomically { load; add; store }
- But don't confuse "`a += 2`" and "`a + 2`"

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

Sequence diagram



Atomic types in C^{++}

Exercise Time

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

Condition Variables

Condition variables

C++11

Communicating thread dependencies

- `std::condition_variable` from condition_variable header
- Allows for a thread to sleep (= conserve CPU time) until a given condition is satisfied

Condition variables

C++11

Communicating thread dependencies

- `std::condition_variable` from condition_variable header
- Allows for a thread to sleep (= conserve CPU time) until a given condition is satisfied

Usage

- Use RAII-style locks to protect shared data
- wait() will block until the condition is met
  - you can have several waiters sharing the same mutex
- notify_one() will wake up one waiter
- notify_all() will wake up all waiters

Using condition variables

C++17

Producer side

- Imagine multiple threads sharing data. Protect it with a mutex
- Use a `condition_variable` to notify consumers
- Optimisation: Don't hold lock while notifying (would block the waking threads)

```
std::mutex mutex;
std::condition_variable cond;
Data data;
std::thread producer([&](){
  {
    std::scoped_lock lock{mutex};
    data = produceData(); // may take long ...
  }
  cond.notify_all();
});
```

Consumer side I: Going into wait

- Start many threads which have to wait for shared data
- Provide a lock to be managed by `wait`
- `wait` will only lock while necessary; unlocked while sleeping
- Threads might wake up, but `wait` returns only when condition satisfied

```cpp
auto processData = [&](){

  std::unique_lock<std::mutex> lock{mutex};
  cond.wait(lock, [&](){ return data.isReady(); });

  process(data);
};
std::thread t1{processData}, t2{processData}, ...;
for (auto t : {&producer, &t1, &t2, ...}) t->join();
```

Consumer side II: Waking up

- `notify_all()` is called, threads wake up
- Threads try to acquire mutex, evaluate condition
- One thread succeeds to acquire mutex, exits from `wait`
- Problem: Other threads still blocked!

```cpp
auto processData = [&](){

  std::unique_lock<std::mutex> lock{mutex};
  cond.wait(lock, [&](){ return data.isReady(); });

  process(data);
};
std::thread t1{processData}, t2{processData}, ...;
for (auto t : {&producer, &t1, &t2, ...}) t->join();
```

Consumer side II: Waking up

- `notify_all()` is called, threads wake up
- Threads try to acquire mutex, evaluate condition
- One thread succeeds to acquire mutex, exits from `wait`
- Solution: Put locking and waiting in a scope

```cpp
auto processData = [&](){
  {
    std::unique_lock<std::mutex> lock{mutex};
    cond.wait(lock, [&](){ return data.isReady(); });
  }
  process(data);
};
std::thread t1{processData}, t2{processData}, ...;
for (auto t : {&producer, &t1, &t2, ...}) t->join();
```

Condition variables

Exercise Time

- Go to code/condition_variable
- Look at the code and run it
  See that it has a race condition
- Fix the race condition in the usage of the condition variable
- Try to make threads process data in parallel

C++ and python

Writing a module

How to build a python 3 module around C++ code

C++ code : mandel.hpp

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

Basic Module(1): wrap your method

mandelModule.cpp - see code/python exercise

```cpp
#include <Python.h>
#include "mandel.hpp"
PyObject * mandel_wrapper(PyObject * self,
                          PyObject * args) {
  // Parse Input
  float r, i;
  if (!PyArg_ParseTuple(args, "ff", &r, &i))
    return NULL;
  // Call C++ function
  int result = mandel(Complex(r, i));
  // Build returned objects
  return PyLong_FromLong(result);
}
```

Basic Module(2): create the python module

mandelModule.cpp - see code/python exercise

```cpp
// declare the modules' methods
PyMethodDef mandelMethods[] = {
    {"mandel", mandel_wrapper, METH_VARARGS,
    "computes nb of iterations for mandelbrot set"},
    {NULL, NULL, 0, NULL}
};
// declare the module
struct PyModuleDef mandelModule = {
  PyModuleDef_HEAD_INIT,
  "mandel", NULL, -1, mandelMethods
};
PyMODINIT_FUNC PyInit_mandel(void) {
  return PyModule_Create(&mandelModule);
}
```

Basic Module(3): use it

First compile the module

- as a regular shared library
- with '-I $(PYTHON_INCLUDE)'

mandel.py - see code/python exercise

```python
from mandel import mandel
v = mandel(0.7, 1.2)
```

Marrying C++ and C

8 C++ and python
- Writing a module
- Marrying C++ and C
- The ctypes module
- The cppyy project

A question of mangling

Mangling

the act of converting the name of variable or function to a symbol name in the binary code

C versus C++ symbol names

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

C mangling

Source : file.c

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

Binary symbols : file.o

```
# nm file.o
000000000000001a T square
0000000000000000 T sum
```

C^{++} mangling

Source : file.cpp

```cpp
float sum(float a, float b);
int square(int a);
// ok, signature is different
float square(float a);
```

Binary symbols : file.o

```
# nm file.o
0000000000000000 T _Z3sumff
000000000000002a T _Z6squaref
000000000000001a T _Z6squarei
```

Forcing C mangling in C^{++}

extern "C"

These functions will use C mangling :

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

Forcing C mangling in C^{++}

extern “C”

These functions will use C mangling :

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

You can now call these C^{++}functions from C code

Forcing C mangling in C^{++}

extern "C"

These functions will use C mangling :

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

You can now call these C^{++}functions from C code

Limitations

- no C^{++}types should go out
- no exceptions either (use noexcept here)
- member functions cannot be used
  - they need to be wrapped one by one

The ctypes module

8 C++ and python
- Writing a module
- Marrying C++ and C
- The ctypes module
- The cppyy project

The ctypes python module

From the documentation

- provides C compatible data types
- allows calling functions in DLLs or shared libraries
- can be used to wrap these libraries in pure Python

ctypes: usage example

C++ code : mandel.hpp

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

"C" code : mandel_cwrapper.hpp

```cpp
extern "C" {
  int mandel(float r, float i) {
    return mandel(Complex(r, i));
  };
}
```

calling the mandel library

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

Marrying C^{++} and python

Exercise Time

- go to code/python
- look at the original python code mandel.py
- time it (‘time python3 mandel.py‘)
- look at the code in mandel.hpp/cpp
- look at the python module mandel_module.cpp
- compile and modify mandel.py to use it
- see the gain in time
- look at the C wrapper in mandel_cwrapper.cpp
- modify mandel.py to use libmandelc directly with ctypes

Note : you may have to add '.' to LD_LIBRARY_PATH and PYTHONPATH

The cppyy project

8 C^{++} and python

- Writing a module
- Marrying C^{++} and C
- The ctypes module
- The cppyy project

Automatic Python-C++ bindings

The cppyy project

- originated from the ROOT project
- still young, version 1.0 from Mid 2018
- but very active, current version 2.1.0
- extremely powerful for interfacing C++ and python

How it works

- uses Just in Time compilation through cling
  - an interactive C++ interpreter

cppyy crash course(1)

Shamelessly copied from the cppyy documentation

```
>>> import cppyy
>>> cppyy.include('zlib.h')        # bring in C++ definitions
>>> cppyy.load_library('libz')     # load linker symbols
>>> cppyy.gbl.zlibVersion()        # use a zlib API
'1.2.11'
```

```
>>> import cppyy
>>> cppyy.cppdef("""
... class MyClass {
... public:
...     MyClass(int i) : m_data(i) {}
...     virtual ~MyClass() {}
...     virtual int add_int(int i) { return m_data + i; }
...     int m_data;
... };""")
True
```

cppyy crash course(1)

```
>>> from cppyy.gbl import MyClass
>>> m = MyClass(42)
>>> cppyy.cppdef("""
... void say_hello(MyClass* m) {
...     std::cout << "Hello, the number is: " << m->m_data << std::endl;
... }""")
True
>>> MyClass.say_hello = cppyy.gbl.say_hello
>>> m.say_hello()
Hello, the number is: 42
>>> m.m_data = 13
>>> m.say_hello()
Hello, the number is: 13
```

This is the end

Questions ?

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