

C++ course

Sébastien Ponce `sebastien.ponce@cern.ch`

CERN

January 2021



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
 - 314 slides, 436 pages, 17 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



Outline

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

Detailed outline

- 1 History and goals
 - History
 - Why we use it ?
- 2 Langage basics
 - Core syntax and types
 - Arrays and Pointers
 - Operators
 - Scopes / namespaces
 - Compound data types
 - Functions
 - References
 - Control instructions
 - Headers and interfaces
 - Auto keyword
- 3 Object orientation (OO)
 - Objects and Classes
 - inherit
- 4 Core modern C++
 - Constructors/destructors
 - Static members
 - Allocating objects
 - Advanced OO
 - Operators
 - Functors
 - Name Lookups
 - Constness
 - Constant Expressions
 - Exceptions
 - Move semantic
 - Copy elision
 - Templates
 - The STL
 - More STL
 - Lambdas
 - pointers and RAI
- 5 Expert C++
 - Variadic templates
 - Perfect forwarding
 - SFINAE
- 6 Useful tools
 - C++ editor
 - Code management
 - The Compiling Chain
 - Debugging
 - The Valgrind family
 - Static code analysis
- 7 Concurrency
 - Threads and async
 - Mutexes
- 8 C++ and python
 - Writing a module
 - Marrying C++ and C
 - The ctypes module
 - The cppy project



History and goals

1 History and goals

- History
- Why we use it ?

2 Langage basics

3 Object orientation (OO)

4 Core modern C++

5 Expert C++

6 Useful tools

7 Concurrency

8 C++ and python

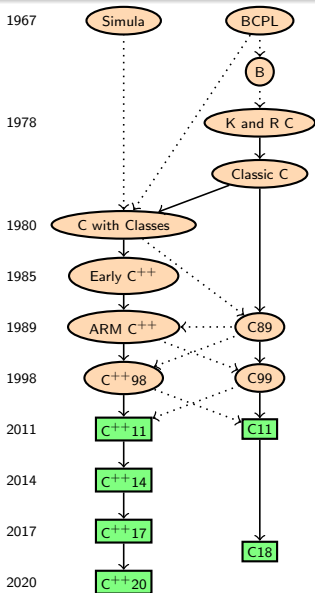


History

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



C/C++ origins



C inventor
Dennis M. Ritchie



C++ inventor
Bjarne Stroustrup

- Both C and C++ are born in Bell Labs
- C++ *almost* embeds C
- C and C++ are still under development
- We will discuss all C++ specs 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	≥4.8	≥3.3
14	≥4.9	≥3.4
17	≥7.3	≥5
20	>10	>10

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

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



Why is C++ our language of choice ?

Adapted to large projects

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

Fast

- compiled (unlike Java or C#)
- allows to go close to hardware when needed



Why is C++ our language of choice ?

Adapted to large projects

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

Fast

- compiled (unlike Java or C#)
- allows to go close to hardware when needed

What we get

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



Language basics

1 History and goals

2 Language basics

- Core syntax and types
- Arrays and Pointers
- Operators
- Scopes / namespaces
- Compound data types
- Functions
- References
- Control instructions
- Headers and interfaces

• Auto keyword

3 Object orientation (OO)

4 Core modern C++

5 Expert C++

6 Useful tools

7 Concurrency

8 C++ and python



Core syntax and types

- 2 Language basics
 - Core syntax and types
 - Arrays and Pointers
 - Operators
 - Scopes / namespaces
 - Compound data types
 - Functions
 - References
 - Control instructions
 - Headers and interfaces
 - Auto keyword



Hello World

C++98

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



Comments

```
1 // simple comment for integer declaration
2 int i;
3
4 /* multiline comment
5  * in case we need to say more
6  */
7 double d;
8
9 /**
10  * Best choice : doxygen compatible comments
11  * \fn bool isOdd(int i)
12  * \brief checks whether i is odd
13  * \param i input
14  * \return true if i is odd, otherwise false
15  */
16 bool isOdd(int i);
```



Basic types(1)

C++98

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



Basic types(2)

C++98

```
1  int i = -123456;           // 32 bits signed integer
2  unsigned int i = 1234567; // 32 bits signed integer
3
4  long l = 0L               // 32 or 64 bits (ptr size)
5  unsigned long l = 0UL;   // 32 or 64 bits (ptr size)
6
7  long long ll = 0LL;      // 64 bits signed integer
8  unsigned long long l = 0ULL; // 64 bits unsigned integer
9
10 float f = 1.23f;        // 32 (23+7+1) bits float
11 double d = 1.23E34;    // 64 (52+11+1) bits float
```



Portable numeric types

C++98

One needs to include specific header

```
1  #include <stdint>
2
3  int8_t c = -3;      // 8 bits, replaces char
4  uint8_t c = 4;     // 8 bits, replaces unsigned char
5
6  int16_t s = -444;  // 16 bits, replaced short
7  uint16_t s = 444; // 16 bits, replaced unsigned short
8
9  int32_t s = -0674; // 32 bits, replaced int
10 uint32_t s = 0674; // 32 bits, replaced unsigned int
11
12 int64_t s = -0x1bc; // 64 bits, replaced long long
13 uint64_t s = 0x1bc; // 64 bits, replaced unsigned long long
```



Arrays and Pointers

- 2 Language basics
 - Core syntax and types
 - **Arrays and Pointers**
 - Operators
 - Scopes / namespaces
 - Compound data types
 - Functions
 - References
 - Control instructions
 - Headers and interfaces
 - Auto keyword



Static arrays

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



Pointers

C++98

```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // not compiling
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;
```



Pointers

C++98

```
1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // not compiling
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;
```

Memory layout

	0x304A
	0x3049
	0x3048
	0x3047
	0x3046
	0x3045
	0x3044
	0x3043
	0x3042
	0x3041
	0x3040



Pointers

```

1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // not compiling
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;

```

Memory layout

	0x304A
	0x3049
	0x3048
	0x3047
	0x3046
	0x3045
	0x3044
	0x3043
	0x3042
	0x3041
<i>i = 4</i>	0x3040



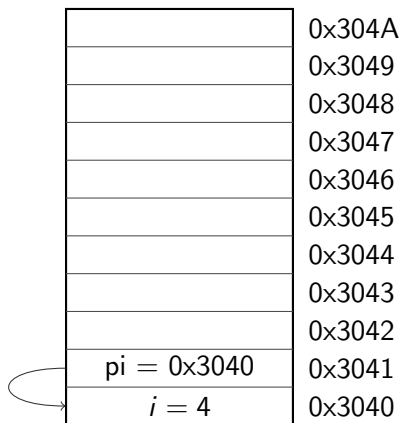
Pointers

```

1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // not compiling
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;

```

Memory layout



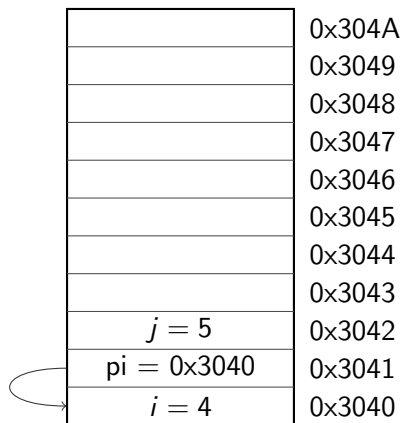
Pointers

```

1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // not compiling
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;

```

Memory layout



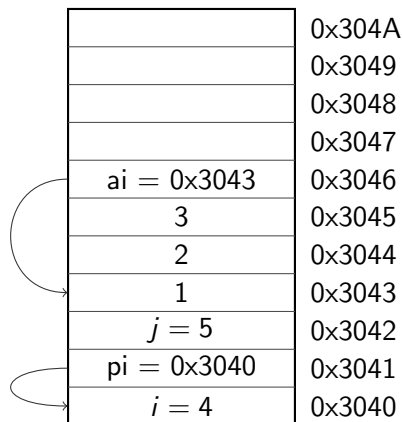
Pointers

```

1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // not compiling
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;

```

Memory layout



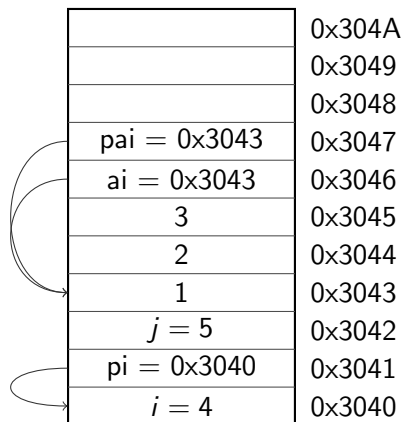
Pointers

```

1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // not compiling
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;

```

Memory layout



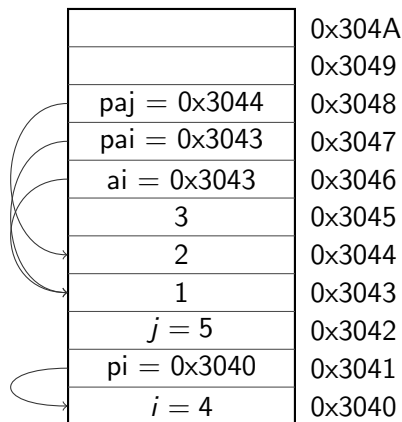
Pointers

```

1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // not compiling
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;

```

Memory layout



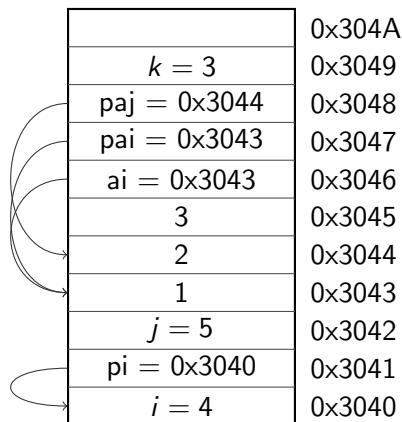
Pointers

```

1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // not compiling
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;

```

Memory layout



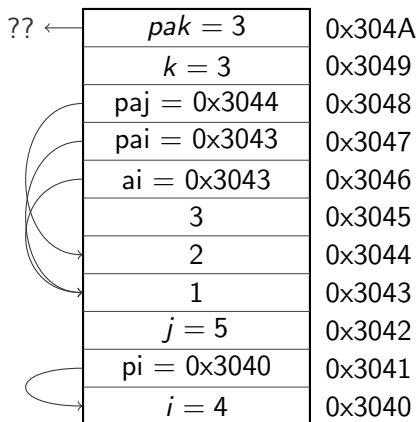
Pointers

```

1  int i = 4;
2  int *pi = &i;
3  int j = *pi + 1;
4
5  int ai[] = {1,2,3};
6  int *pai = ai;
7  int *paj = pai + 1;
8  int k = *paj + 1;
9
10 // not compiling
11 int *pak = k;
12
13 // seg fault !
14 int *pak = (int*)k;
15 int l = *pak;

```

Memory layout



Finally a C++ NULL pointer

- works like 0 or NULL in standard cases
- triggers compilation error when mapped to integer



Finally a C++ NULL pointer

- works like 0 or NULL in standard cases
- triggers compilation error when mapped to integer

Example code

```
1 void* vp = nullptr;
2 int* ip = nullptr;
3 int i = NULL;      // OK -> bug ?
4 int i = nullptr;  // ERROR
```



Dynamic Arrays

C++98

```
1  #include <cstdlib>
2  #include <cstring>
3
4  int *bad;           // pointer to random address
5  int *ai = nullptr; // better. Can be tested
6
7  // allocate array of 10 ints (not initialized)
8  ai = (int*) malloc(10*sizeof(int));
9  // and set them to 0
10 memset(ai, 0, 10*sizeof(int));
11
12 // Both in one go
13 ai = (int*) calloc(10, sizeof(int));
14
15 // liberate memory
16 free(ai);
```



Operators

- 2 Language basics
 - Core syntax and types
 - Arrays and Pointers
 - **Operators**
 - Scopes / namespaces
 - Compound data types
 - Functions
 - References
 - Control instructions
 - Headers and interfaces
 - Auto keyword



Operators(1)

C++98

Binary & Assignment Operators

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



Operators(1)

C++98

Binary & Assignment Operators

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

Increment / Decrement

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



Operators(1)

C++98

Binary & Assignment Operators

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

Increment / Decrement

Use wisely

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



Operators(2)

C++98

Bitwise and Assignment Operators

```
1  int i = 0xee & 0x55; // 0x44
2  i |= 0xee;          // 0xee
3  i ^= 0x55;          // 0xbb
4  int j = ~0xee;      // 0xfffff11
5  int k = 0x1f << 3;  // 0x78
6  int l = 0x1f >> 2;  // 0x7
```



Operators(2)

Bitwise and Assignment Operators

```
1  int i = 0xee & 0x55; // 0x44
2  i |= 0xee;          // 0xee
3  i ^= 0x55;          // 0xbb
4  int j = ~0xee;      // 0xfffff11
5  int k = 0x1f << 3;  // 0x78
6  int l = 0x1f >> 2;  // 0x7
```

Boolean Operators

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



Operators(3)

C++98

Comparison Operators

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



Operators(3)

Comparison Operators

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

Precedences

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



Operators(3)

C++98

Comparison Operators

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

Precedences

Don't use

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



Operators(3)

C++98

Comparison Operators

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

Precedences

Don't use - use parenthesis

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



Scopes / namespaces

- 2 Language basics
 - Core syntax and types
 - Arrays and Pointers
 - Operators
 - **Scopes / namespaces**
 - Compound data types
 - Functions
 - References
 - Control instructions
 - Headers and interfaces
 - Auto keyword



Definition

Portion of the source code where a given name is valid

Typically :

- simple block of code, within {}
- function, class, namespace
- source file for global declarations

Example

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



Namespaces

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

```

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

```



Nested namespaces

C++17

Easier way to declare nested namespaces

C++14

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

C++17

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



Anonymous namespaces

A namespace without a name !

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

Purpose

- groups a number of declarations
- visible 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

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



Compound data types

- 2 Language basics
 - Core syntax and types
 - Arrays and Pointers
 - Operators
 - Scopes / namespaces
 - **Compound data types**
 - Functions
 - References
 - Control instructions
 - Headers and interfaces
 - Auto keyword



struct

“members” grouped together under one name

```
1  struct Individual {           14  Individual *studentPtr =
2      unsigned char age;       15      &student;
3      float weight;           16      studentPtr->age = 25;
4  };
5
6  Individual student;
7  student.age = 25;
8  student.weight = 78.5;
9
10 Individual teacher = {
11     45, 67
12 };
```



struct

“members” grouped together under one name

```

1  struct Individual {
2      unsigned char age;
3      float weight;
4  };
5
6  Individual student;
7  student.age = 25;
8  student.weight = 78.5;
9
10 Individual teacher = {
11     45, 67
12 };

```

```

14 Individual *studentPtr =
15     &student;
16 studentPtr->age = 25;

```

Memory layout

				0x3053
				0x304F
				0x304B
				0x3047
				0x3043



struct

“members” grouped together under one name

```

1  struct Individual {
2      unsigned char age;
3      float weight;
4  };
5
6  Individual student;
7  student.age = 25;
8  student.weight = 78.5;
9
10 Individual teacher = {
11     45, 67
12 };

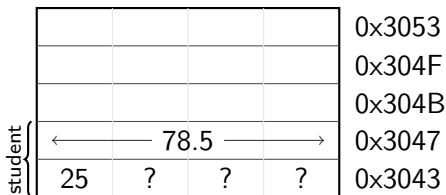
```

```

14 Individual *studentPtr =
15     &student;
16 studentPtr->age = 25;

```

Memory layout



struct

“members” grouped together under one name

```

1  struct Individual {
2      unsigned char age;
3      float weight;
4  };
5
6  Individual student;
7  student.age = 25;
8  student.weight = 78.5;
9
10 Individual teacher = {
11     45, 67
12 };

```

```

14 Individual *studentPtr =
15     &student;
16 studentPtr->age = 25;

```

Memory layout

					0x3053
student teacher	← 67.0 →				0x304F
	45	?	?	?	0x304B
	← 78.5 →				0x3047
	25	?	?	?	0x3043



“members” packed together at same memory location

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



“members” packed together at same memory location

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

Memory layout

				0x304F
				0x304B
				0x3047
				0x3043



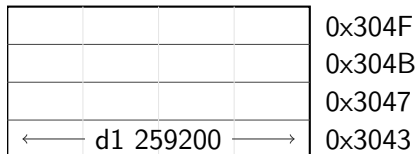
“members” packed together at same memory location

```

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

```

Memory layout



“members” packed together at same memory location

```

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

```

Memory layout

				0x304F	
				0x304B	
←	d2 72	→	?	?	0x3047
←	d1 259200		→		0x3043



“members” packed together at same memory location

```

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

```

Memory layout

				0x304F
d3 3	?	?	?	0x304B
← d2 72 →		?	?	0x3047
←	d1 259200		→	0x3043



“members” packed together at same memory location

```

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

```

Memory layout

				0x304F
d3 3	?	?	?	0x304B
← d2 72 →		?	?	0x3047
d1 3	?	?	?	0x3043



Enums

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

```
1 enum VehicleType {
2     BIKE = 3,
3     CAR = 5,
4     BUS = 7,
5 };
6 VehicleType t2 = BUS;
```



Scoped enumeration, aka enum class

C++11

Same syntax as enum, with scope

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



Scoped enumeration, aka enum class

C++11

Same syntax as enum, with scope

```

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

```

Only advantages

- scoping avoids name clashes
- strong typing, no automatic conversion to int

```

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

```



More sensible example

C++98

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



More sensible example

C++98

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



More sensible example

C++98

```

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

```



typedef and using

C++98 / C++11

Ways to create type aliases

C++98

```
1 typedef uint64_t myint;  
2 myint toto = 17;
```

C++11

```
1 using myint = uint64_t;  
2 myint toto = 17;  
3  
4 template <typename T> using myvec = std::vector<T>;  
5 myvec<int> titi;
```



Functions

- 2 Language basics
 - Core syntax and types
 - Arrays and Pointers
 - Operators
 - Scopes / namespaces
 - Compound data types
 - **Functions**
 - References
 - Control instructions
 - Headers and interfaces
 - Auto keyword



Functions

C++98

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



Parameter are passed by value

C++98

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

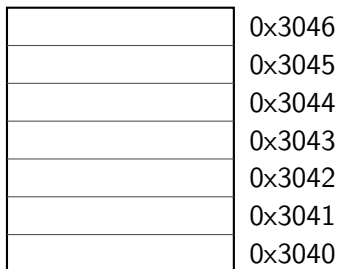


Parameter are passed by value

C++98

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

Memory layout



Parameter are passed by value

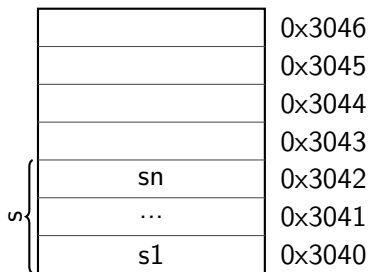
C++98

```

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

```

Memory layout



Parameter are passed by value

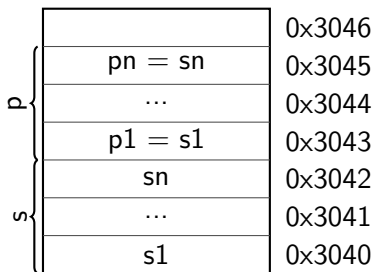
C++98

```

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

```

Memory layout



Parameter are passed by value

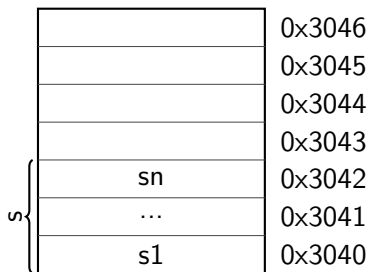
C++98

```

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

```

Memory layout



Parameter are passed by value

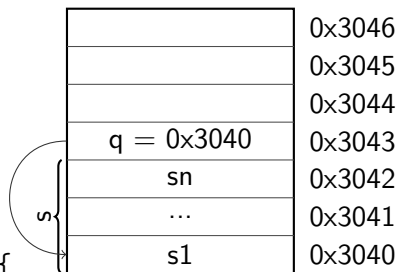
C++98

```

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

```

Memory layout



Parameter are passed by value

```
1  struct SmallStruct {int a};
2  SmallStruct s = {1};
3
4  void changeSS(SmallStruct p) {
5      p.a = 2;
6  }
7  changeSS(s);
8  // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11     q->a = 2;
12 }
13 changeSS2(&s);
14 // s.a = 2
```



Parameter are passed by value

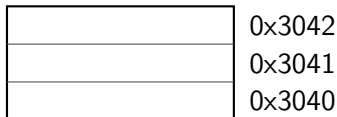
C++98

```

1  struct SmallStruct {int a};
2  SmallStruct s = {1};
3
4  void changeSS(SmallStruct p) {
5      p.a = 2;
6  }
7  changeSS(s);
8  // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11     q->a = 2;
12 }
13 changeSS2(&s);
14 // s.a = 2

```

Memory layout



Parameter are passed by value

C++98

```

1  struct SmallStruct {int a};
2  SmallStruct s = {1};
3
4  void changeSS(SmallStruct p) {
5      p.a = 2;
6  }
7  changeSS(s);
8  // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11     q->a = 2;
12 }
13 changeSS2(&s);
14 // s.a = 2

```

Memory layout

	0x3042
	0x3041
s.a = 1	0x3040



Parameter are passed by value

C++98

```

1  struct SmallStruct {int a};
2  SmallStruct s = {1};
3
4  void changeSS(SmallStruct p) {
5      p.a = 2;
6  }
7  changeSS(s);
8  // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11     q->a = 2;
12 }
13 changeSS2(&s);
14 // s.a = 2

```

Memory layout

	0x3042
p.a = 1	0x3041
s.a = 1	0x3040



Parameter are passed by value

C++98

```

1  struct SmallStruct {int a};
2  SmallStruct s = {1};
3
4  void changeSS(SmallStruct p) {
5      p.a = 2;
6  }
7  changeSS(s);
8  // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11     q->a = 2;
12 }
13 changeSS2(&s);
14 // s.a = 2

```

Memory layout

	0x3042
p.a = 2	0x3041
s.a = 1	0x3040



Parameter are passed by value

C++98

```
1 struct SmallStruct {int a};
2 SmallStruct s = {1};
3
4 void changeSS(SmallStruct p) {
5     p.a = 2;
6 }
7 changeSS(s);
8 // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11     q->a = 2;
12 }
13 changeSS2(&s);
14 // s.a = 2
```

Memory layout

	0x3042
	0x3041
s.a = 1	0x3040



Parameter are passed by value

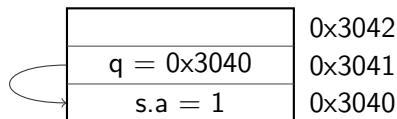
C++98

```

1  struct SmallStruct {int a};
2  SmallStruct s = {1};
3
4  void changeSS(SmallStruct p) {
5      p.a = 2;
6  }
7  changeSS(s);
8  // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11     q->a = 2;
12 }
13 changeSS2(&s);
14 // s.a = 2

```

Memory layout



Parameter are passed by value

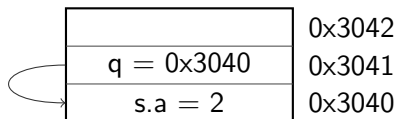
C++98

```

1  struct SmallStruct {int a};
2  SmallStruct s = {1};
3
4  void changeSS(SmallStruct p) {
5      p.a = 2;
6  }
7  changeSS(s);
8  // s.a = 1
9
10 void changeSS2(SmallStruct *q) {
11     q->a = 2;
12 }
13 changeSS2(&s);
14 // s.a = 2

```

Memory layout



References

- 2 Language basics
 - Core syntax and types
 - Arrays and Pointers
 - Operators
 - Scopes / namespaces
 - Compound data types
 - Functions
 - **References**
 - Control instructions
 - Headers and interfaces
 - Auto keyword



Value, pointers and references

C++98

Different ways to pass arguments to a function

- by default arguments are passed by value
- pointers can be used to avoid copies
- references are also available and are preferred



Value, pointers and references

C++98

Different ways to pass arguments to a function

- by default arguments are passed by value
- pointers can be used to avoid copies
- references are also available and are preferred

Syntax

```
1  struct T {...}; T a;  
2  void func    (T value); f(a);    // by value  
3  void funcPtr(T *value); f(&a);   // pointer  
4  void funcRef(T &value); f(a);   // by reference
```



Value versus pointers/reference

C++98

When passing by value

- a copy is created
- making the original argument unmodifiable (aka const)
- efficient for small types, e.g. numbers

When passing by pointer/reference

- no copy of your argument
- in the back only a memory address is copied
- argument can be modified (unless you use constness, see later)



Pointers vs References

C++98

Specificities of reference

- natural syntax
- will never be `nullptr`
- thus cannot reference temporary object

Advantages of pointers

- can be `nullptr`
- clearly indicates that argument may be modified



Pointers vs References

C++98

Specificities of reference

- natural syntax
- will never be `nullptr`
- thus cannot reference temporary object

Advantages of pointers

- can be `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



Control instructions

- 2 Language basics
 - Core syntax and types
 - Arrays and Pointers
 - Operators
 - Scopes / namespaces
 - Compound data types
 - Functions
 - References
 - **Control instructions**
 - Headers and interfaces
 - Auto keyword



Control instructions : if

C++98

if syntax

```
1  if (condition1) {
2      Instructions1;
3  } else if (condition2) {
4      Instructions2;
5  } else {
6      Instructions3;
7  }
```

- *else* and *else if* part are optional
- *else if* part can be repeated
- braces are optional if there is a single instruction



Control instructions : if

C++98

Practical example

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



Control instructions : conditional operator

C++98

Syntax

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

- if test is *true* expression1 is returned
- else expression 2 is returned



Control instructions : conditional operator

C++98

Syntax

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

- if test is *true* expression1 is returned
- else expression 2 is returned

Practical example

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



Control instructions : conditional operator

C++98

Syntax

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

- if test is *true* expression1 is returned
- else expression 2 is returned

Practical example

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

Do not abuse

explicit ifs are easier to read

to be used only when obvious and not nested



Control instructions : switch

C++98

Syntax

```
1 switch(identifier) {
2     case c1 : instructions1; break;
3     case c2 : instructions2; break;
4     case c3 : instructions3; break;
5     ...
6     default : instructiond; break;
7 }
```

- *break* is not mandatory but...
- cases are entry points, not independant pieces
- execution carries on with the next case if no *break* is present !
- *default* may be omitted



Control instructions : switch

C++98

Syntax

```
1 switch(identifier) {
2     case c1 : instructions1; break;
3     case c2 : instructions2; break;
4     case c3 : instructions3; break;
5     ...
6     default : instructiond; break;
7 }
```

- *break* is not mandatory but...
- cases are entry points, not independant pieces
- execution carries on with the next case if no *break* is present !
- *default* may be omitted

Use break

Do not try to make use of non breaking cases



Control instructions : switch

C++98

Practical example

```
1  enum Lang { FRENCH, GERMAN, ENGLISH, OTHER };
2  ...
3  switch (language) {
4  case FRENCH:
5      printf("Bonjour");
6      break;
7  case GERMAN:
8      printf("Guten tag");
9      break;
10 case ENGLISH:
11     printf("Good morning");
12     break;
13 default:
14     printf("I do not talk your langage");
15 }
```



[[fallthrough]] attribute

C++17

C++14

```
1  switch (c) {
2      case 'a':
3          f(); // Warning emitted
4      case 'c':
5          h();
6  }
```

C++17

```
1  switch (c) {
2      case 'a':
3          f();
4          [[fallthrough]]; // Warning suppressed
5      case 'c':
6          h();
7  }
```



init-statements for if and switch

C++17

Allows to simplify if and switch statements

C++14

```
1 Value val = GetValue();
2 if (condition(val)) {
3     // on success
4 } else {
5     // on false...
6 }
```

C++17

```
1 if (Value val = GetValue(); condition(val)) {
2     // on success
3 } else {
4     // on false...
5 }
```

val is visible only inside the if and else statements



Control instructions : for loop

C++98

for loop syntax

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

- initializations and increments are comma separated
- initializations can contain declarations
- braces are optional if there is a single instruction



Control instructions : for loop

C++98

for loop syntax

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

- initializations and increments are comma separated
- initializations can contain declarations
- braces are optional if there is a single instruction

Practical example

```
1  for(int i = 0, j = 0 ; i < 10 ; i++, j = i*i) {  
2      std::cout << i << "^2 is " << j << "\n";  
3  }
```



Control instructions : for loop

C++98

for loop syntax

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

- initializations and increments are comma separated
- initializations can contain declarations
- braces are optional if there is a single instruction

Practical example

```
1  for(int i = 0, j = 0 ; i < 10 ; i++, j = i*i) {  
2      std::cout << i << "^2 is " << j << "\n";  
3  }
```

Do not abuse the syntax

The for statement should fit in 1-3 lines



Range based loops

C++11

Reason of being

- simplifies loops tremendously
- especially with STL containers

Syntax

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

Example code

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



Control instructions : while loop

C++98

while loop syntax

```
1  while(condition) {  
2      instructions;  
3  }  
4  do {  
5      Instructions;  
6  } while(condition);
```

- braces are optional if there is a single instruction



Control instructions : while loop

C++98

while loop syntax

```
1  while(condition) {
2      instructions;
3  }
4  do {
5      Instructions;
6  } while(condition);
```

- braces are optional if there is a single instruction

Practical example

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



Control instructions : commands

C++98

control commands

`break` goes out of the loop

`continue` goes immediately to next iteration

`return` goes out of current function



Control instructions : commands

C++98

control commands

`break` goes out of the loop

`continue` goes immediately to next iteration

`return` goes out of current function

Practical example

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



Headers and interfaces

- 2 Language basics
 - Core syntax and types
 - Arrays and Pointers
 - Operators
 - Scopes / namespaces
 - Compound data types
 - Functions
 - References
 - Control instructions
 - **Headers and interfaces**
 - Auto keyword



Headers and interfaces

C++98

Interface

Set of declarations defining some functionality

- defined in a “header file”
- no implementation defined

Header : hello.hpp

```
void printHello();
```

Usage : myfile.cpp

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



Preprocessor

C++98

```
1 // file inclusion
2 #include "hello.hpp"
3 // macros
4 #define MY_GOLDEN_NUMBER 1746
5 // compile time decision
6 #ifdef USE64BITS
7     typedef uint64_t myint;
8 #else
9     typedef uint32_t myint;
10 #endif
```



Preprocessor

```
1 // file inclusion
2 #include "hello.hpp"
3 // macros
4 #define MY_GOLDEN_NUMBER 1746
5 // compile time decision
6 #ifdef USE64BITS
7     typedef uint64_t myint;
8 #else
9     typedef uint32_t myint;
10 #endif
```

Use only in very restricted cases

- include of headers
- hardcoded constants before C++11
- portability necessity



Auto keyword

- 2 Language basics
 - Core syntax and types
 - Arrays and Pointers
 - Operators
 - Scopes / namespaces
 - Compound data types
 - Functions
 - References
 - Control instructions
 - Headers and interfaces
 - Auto keyword



Reason of being

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

```
1 std::vector<int> v;  
2 int a = v[3];  
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 error if you mess up

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

Practical usage

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



Object orientation (OO)

- 1 History and goals
 - 2 Langage basics
 - 3 Object orientation (OO)**
 - Objects and Classes
 - inherit
 - Constructors/destructors
 - Static members
 - Allocating objects
 - Advanced OO
 - Operators
 - 4 Core modern C++
 - 5 Expert C++
 - 6 Useful tools
 - 7 Concurrency
 - 8 C++ and python
- Functors
 - Name Lookups



Objects and Classes

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



What are classes and objects

C++98

Classes

structs on steroids

- with inheritance
- including methods

Objects

instances of classes

A class encapsulates a concept

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



My First Class

C++98

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

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

Separating the interface

C++98

Header : MyFirstClass.hpp

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

Implementation : MyFirstClass.cpp

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



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

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



this keyword

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

```
1 void ext_func(MyFirstClass& c) {
2     ... do something with c ...
3 }
4
5 int MyFirstClass::some_method(...) {
6     ext_sum(*this);
7 };
```



Method overloading

C++98

The rules in C++

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

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



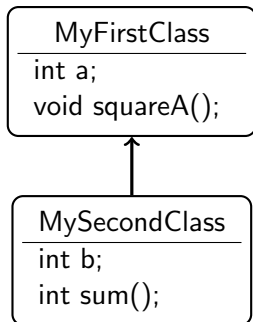
inherit

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



First inheritance

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



Managing access to class members

C++98

public / *private* keywords

private allows access only within the class

public allows access from anywhere

- Default is *private*
- A *struct* is a *class* where all members are public



Managing access to class members

C++98

public / *private* keywords

private allows access only within the class

public allows access from anywhere

- Default is *private*
- A *struct* is a *class* where all members are public

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

```
9  MyFirstClass obj;
10 obj.a = 5; // error !
11 obj.setA(5); // ok
12 obj.squareA();
13 int b = obj.getA();
```



Managing access to class members

C++98

public / *private* keywords

private allows access only within the class

public allows access from anywhere

- Default is *private*
- A *struct* is a *class* where all members are public

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

This breaks MySecondClass !



Managing access to class members(2)

C++98

Solution is *protected* keyword

Gives access to classes inheriting from base class

```
1  class MyFirstClass {
2  public:
3      void setA(int a);
4      int getA();
5      void squareA();
6  protected:
7      int a;
8  }
13 class MySecondClass :
14     public MyFirstClass {
15     public:
16         int sum() {
17             return a + b;
18         };
19     private:
20         int b;
21     }
```



Managing inheritance privacy

C++98

Inheritance can be public, protected or private

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

public privacy of inherited members remains unchanged

protected inherited public members are seen as protected

private all inherited members are seen as private
this is the default 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 if nothing is specified

Net result for external code

- only public members of public inheritance are accessible

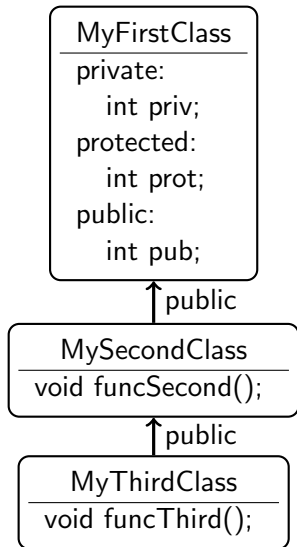
Net result for grand child code

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



Managing inheritance privacy - public

C++98



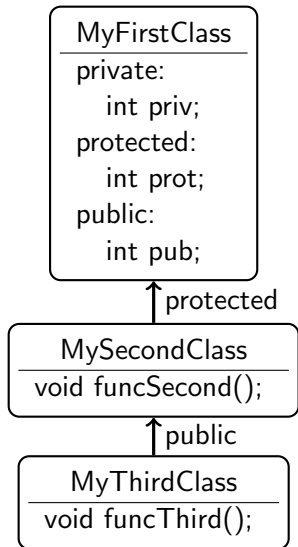
```

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



Managing inheritance privacy - protected

C++98



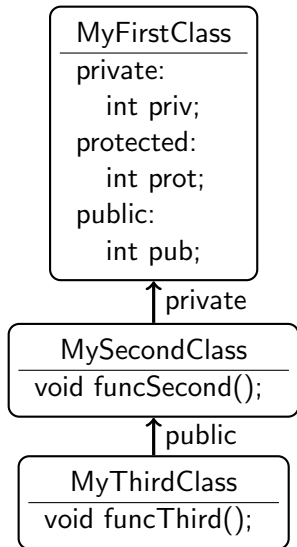
```

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



Managing inheritance privacy - private

C++98



```

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



Final class

C++11

Idea

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

Practically

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



Constructors/destructors

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



Class Constructors and Destructors

C++98

Concept

- special functions building/destroying an object
- a class can have several constructors
- the constructors have the name of the class
- same for the destructor with a leading ~

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



Class Constructors and Destructors

C++98

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



Constructor and inheritance

C++98

```
1  struct MyFirstClass {
2      MyFirstClass();
3      MyFirstClass(int a);
4  }
5
6  struct MySecondClass : MyFirstClass {
7      MySecondClass();
8      MySecondClass(int b);
9      MySecondClass(int a, int b);
10 }
11
12 MySecondClass() : MyFirstClass(), b(0) {};
13 MySecondClass(int b_) : MyFirstClass(), b(b_) {};
14 MySecondClass(int a_,
15                 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 ref to class
- will be implemented by the compiler if not provided
- in order to forbid copy, use delete (coming in 2 slides)
 - or private copy constructor with no implementation in C++98



Concept

- special constructor called for replicating an object
- takes a single parameter of type const ref to class
- will be implemented by the compiler if not provided
- in order to forbid copy, use delete (coming in 2 slides)
 - or private copy constructor with no implementation in C++98

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



Copy constructor

C++98

Concept

- special constructor called for replicating an object
- takes a single parameter of type const ref to class
- will be implemented by the compiler if not provided
- in order to forbid copy, use delete (coming in 2 slides)
 - or private copy constructor with no implementation in C++98

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

The rule of 3/5 (C++98/C++11 and newer)

- if a class has a destructor, a copy/move constructor or a (move) assignment operator, it should have all three/five



Class Constructors and Destructors

```
1  class Vector {
2  public:
3      Vector(int n);
4      Vector(const Vector &other);
5      ~Vector();
6      ...
7  }
8  Vector::Vector(int n) : len(n) {
9      data = (int*)calloc(n, sizeof(int));
10 }
11 Vector::Vector(const Vector &other) : len(other.len) {
12     data = (int*)malloc(len*sizeof(int));
13     memcpy(data, other.data, len);
14 }
15 Vector::~~Vector() { free(data); }
```



Default Constructor

C++11

Idea

- avoid writing explicitly default constructors
- by declaring them as default

Details

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

Practically

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



Constructor delegation

C++11

Idea

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

Practically

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



Constructor inheritance

C++11

Idea

- avoid declaring empty constructors inheriting parent's ones
- by stating that we inherit all parent constructors

Practically

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



Member initialization

C++11

Idea

- avoid redefining same default value for members n times
- by defining it once at member declaration time

Practically

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



Calling constructors

After object declaration, arguments within {}

```
1  struct A {
2      int a;
3      float b;
4      A();
5      A(int);
6      A(int, int);
7  };

12 A a{1,2};           // A::A(int, int)
13 A a{1};            // A::A(int)
14 A a{};             // A::A()
15 A a;               // A::A()
16 A a = {1,2};       // A::A(int, int)
17 B b = {1, 2.3};    // default constructor
```



Calling constructors the old way

C++98

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

```

1  struct A {
2      int a;
3      float b;
4      A();
5      A(int);
6      A(int, int);
7  };

12 A a(1,2);           // A::A(int, int)
13 A a(1);            // A::A(int)
14 A a();             // declaration of a function !
15 A a;               // A::A()
16 A a = {1,2};      // not allowed
17 B b = {1, 2.3};   // OK

```



Calling constructors for arrays and vectors

C++11

list of items given within {}

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



Calling constructors for arrays and vectors

C++11

list of items given within {}

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

C++98 nightmare

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



Static members

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



Static members

C++98

Concept

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

```
1 class Text {
2 public:
3     static std::string upper(std::string) {...};
4 private:
5     static int s_nbCallsToUpper;
6 };
7 int Text::s_nbCallsToUpper = 0;
8 std::string uppers = Text::upper("my text");
9 // now Text::s_nbCallsToUpper is 1
```



Allocating objects

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



Process memory organization

C++98

4 main areas

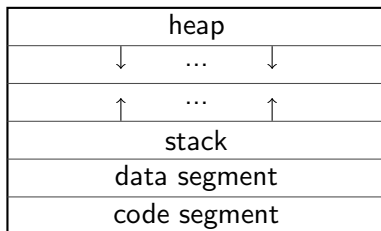
the **code segment** for the code of the executable

the **data segment** for global variables

the **heap** for dynamically allocated variables

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

Memory layout



Main characteristics

- allocation on the stack stays valid for the duration of the current scope. It is destroyed when it is popped off the stack.
- memory allocated on the stack is known at compile time and can thus be accessed through a variable.
- the stack is relatively small, it is not a good idea to allocate large arrays, structures or classes



Object allocation on the stack

C++98

On the stack

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

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



Main characteristics

- Allocated memory stays allocated until it is specifically deallocated
 - beware memory leaks
- Dynamically allocated memory must be accessed through pointers
- large arrays, structures, or classes should be allocated here



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)

```
1  {
2    // default constructor called
3    MyFirstClass *a = new MyFirstClass;
4    ...
5    delete a; // destructor is called
6  }
7
8  int f() {
9    // constructor called
10   MyFirstClass *a = new MyFirstClass(3);
11   ...
12  } // 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

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



Advanced OO

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

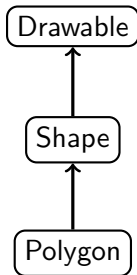


Polymorphism

the concept

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

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



Inheritance privacy and polymorphism

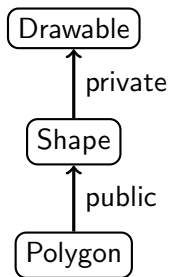
Only public inheritance is visible to code outside the class

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

```

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

```



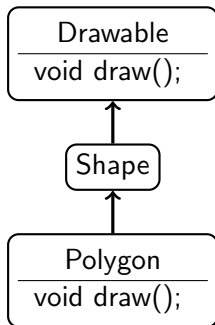
Method overriding

C++98

the problem

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

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



the concept

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

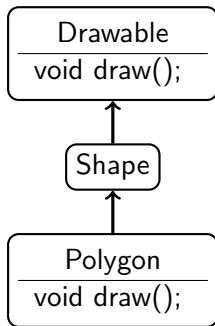


Virtual methods

the concept

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

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

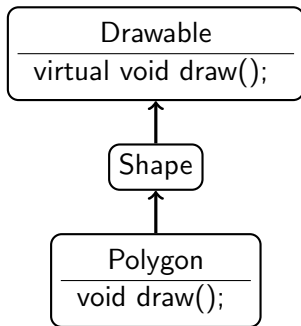


Virtual methods

the concept

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

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



Virtual methods - implications

C++11

Mechanics

- virtual methods are deduced at run time
 - while non virtual methods are deduced 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”

Consequences

- virtual methods are “slower” than standard ones
- and they cannot be inlined
- templates are an alternative for performance critical cases



override keyword

C++11

Principle

- when overriding a virtual method
- the `override` keyword needs to be used

Practically

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



Why was `override` keyword introduced ?

C++11

To detect the mistake in the following code :

Without `override` (C++98)

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

- 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 override further a given virtual method
- by declaring it final

Practically

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



Pure Virtual methods

C++98

Concept

- methods that exist but are not implemented
- marked by an “= 0” in the declaration
- makes their class abstract
- an object can only be instantiated for a non abstract class

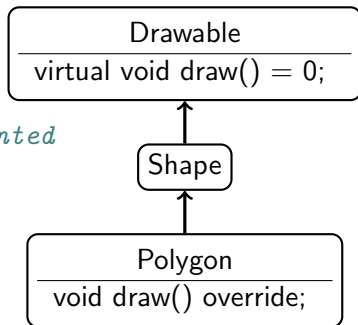


Pure Virtual methods

Concept

- methods that exist but are not implemented
- marked by an “= 0” in the declaration
- makes their class abstract
- an object can only be instantiated for a non abstract class

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



Pure Abstract Class aka Interface

C++98

Definition of pure abstract class

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

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

Drawable
virtual void draw() = 0;



Overriding overloaded methods

Concept

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

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



Exercise Time

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

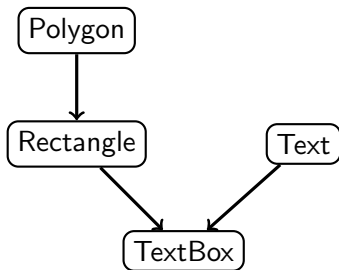


Multiple Inheritance

C++98

Concept

- one class can inherit from multiple parents



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



The diamond shape

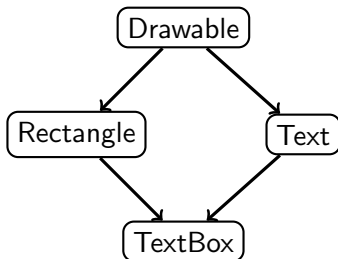
C++98

Definition

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

Problem

- are the members of the grand parent replicated ?

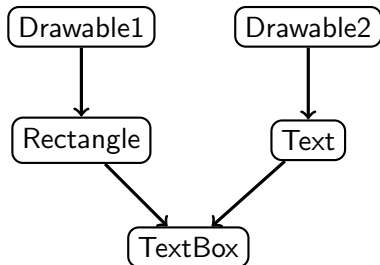
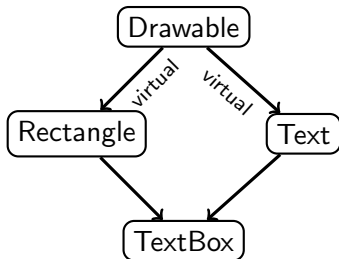


Virtual inheritance

Solution

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

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



Multiple inheritance advice

C++98

Do not use multiple inheritance

- Except for inheriting from interfaces
- and for very seldom special cases



Multiple inheritance advice

C++98

Do not use multiple inheritance

- Except for inheriting from interfaces
- and for very seldom 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 mind



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



Warning

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



Operators

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



Operators' example

C++98

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



Operators

Definition for 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

Symetry

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



Why to have non-member operators ?

C++98

Symetry

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



Other reason to have non-member operators ?

C++98

Extending existing classes

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



Functors

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



Functors

C++98

Concept

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

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



Typical usage

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

```
1 struct BinaryFunction {
2     virtual double operator() (double a, double b) = 0;
3 };
4 struct Add : public BinaryFunction {
5     double operator() (double a, double b) { return a+b; }
6 };
7 double binary_op(double a, double b, BinaryFunction &func)
8     return func(a, b);
9 }
10 Add addfunc;
11 double c = binary_op(a, b, addfunc);
```



Name Lookups

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



Basics of name lookup

C++98

Example code

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

How to find the declaration of a name ?

Mainly 2 cases :

- qualified lookup, for names predated by '::'
 - here cout and endl
 - name is only looked for in given class/namespace/enum class
- unqualified lookup
 - here for std
 - 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, parent namespace/block, etc...
- current class if any, base class if any, etc...
- for names used to the left of the () operator, Argument Dependent Lookup (ADL)

Argument Dependent Lookup (simplified)

For each argument of the function call, extra scopes are searched

- the class of the argument's type if any
- its direct and indirect base classes if any
- the enclosing namespace of all these classes



ADL consequences (1)

C++98

Use standalone/non-member functions

When a method is not accessing the private part of a class

Don't write :

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

Prefer :

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

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++98

Prefer nested namespaces to using

Don't write :

```
1 using namespace MyProject;  
2 funcFromMyProject(...);
```

Prefer :

```
3 namespace MyProject::MySubpart {  
4     funcFromMyProject(...);  
5 }
```

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



ADL consequences (3)

C++98

Customization points and using

Don't write :

```
1 namespace N {  
2     A a,b;  
3     std::swap(a, b);  
4 }
```

Prefer :

```
5 namespace N {  
6     A a,b;  
7     using std::swap;  
8     swap(a, b);  
9 }
```

Advantages :

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

```
10 namespace N {  
11     // optimized swap for A  
12     void swap(A&, A&);  
13 }
```

Core modern C++

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



Constness

- 4 Core modern C++
 - Constness
 - Constant Expressions
 - Exceptions
 - Move semantic
 - Copy elision
 - Templates
 - The STL
 - More STL
 - Lambdas
 - pointers and RAI



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

```
1 // standard syntax
2 int const i = 6;
3
4 // error : i is constant
5 i = 5;
6
7 // also ok, when nothing on the left,
8 // const applies to element on the right
9 const int j = 6;
```



Constness and pointers

C++98

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



The *const* keyword for class functions

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

```
1 struct Exemple {
2     void foo() const {
3         m_member = 0; // Error : function is constant
4     }
5     int m_member;
6 };
```



Method constness

Constness is part of the type

- `const T` and `T` are different type
- however, `T` is automatically casted into `const T` when needed

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



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 semantic
 - Copy elision
 - Templates
 - The STL
 - More STL
 - Lambdas
 - pointers and RAI



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

```
1  constexpr int f(int x) {  
2      return x > 1 ? x * f(x - 1) : 1;  
3  }  
4  int a = f(5); // now computed at compile time
```



Generalized Constant Expressions(2)

Few limitations

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

Notes

- classes can have constexpr 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

```
1  constexpr float toSI(const float v, const char unit) {
2      switch (unit) {
3          case 'k': return 1000*v;
4          case 'm': return 0.001*v;
5          case 'y': return 0.9144*v;
6          case 'i': return 0.0254*v;
7          ...
8          default: return v;
9      }
10 }
11 constexpr float fromSI(const float v, const char unit) {
12     switch (unit) {
13         case 'k': return 0.001*v;
14         ...
15     }
16 }
```



Real life example(2)

C++11

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



Exceptions

- 4 Core modern C++
 - Constness
 - Constant Expressions
 - **Exceptions**
 - Move semantic
 - Copy elision
 - Templates
 - The STL
 - More STL
 - Lambdas
 - pointers and RAI



Exceptions

The concept

- exceptional Event breaking linearity of the code
- will be handled in dedicated place

Pratically

- you can throw any object with *throw*
- you handle them using *try ... catch* blocks

```
1 try {
2     if (0 == name) {
3         throw std::string("Expected non empty name");
4     }
5     printf("%s\n", name);
6 } catch (std::string& e) {
7     printf("empty name found\n");
8 }
```



Exceptions

C++98

Rules

- exception will skip all code until next *catch*
 - still destructors are called when exiting scopes
 - but your own cleanup may not be
- *catch* is selective on the exception type

```
1  class ZeroDivide {};  
2  
3  int divide(int a, int b) {  
4      if (0 == b) {  
5          throw ZeroDivide();  
6      }  
7      return a/b;  
8  }
```

```
9  int func(char* value) {  
10     try {  
11         errno = 0;  
12         long l = strtol(value,0,10);  
13         if (errno) {  
14             throw string("Bad Value");  
15         }  
16         divide(100, 1);  
17     } catch (string& e) {  
18         printf("%s\n", e.c_str());  
19     } catch (ZeroDivide e2) {  
20         printf("Division error\n");  
21     }  
22 }
```



Declaring expected exceptions

- each function can declare a set of expected exceptions
- using the *throw* statement in its declaration
- other exceptions won't exit the scope of the function
 - instead, the *unexpected* handler is called
 - by default, it terminates the program



Declaring expected exceptions

- each function can declare a set of expected exceptions
- using the *throw* statement in its declaration
- other exceptions won't exit the scope of the function
 - instead, the *unexpected* handler is called
 - by default, it terminates the program

```
1 int func(int a) throw(int) {
2     if (0 == a) {
3         throw 2; // ok, goes out
4     } else {
5         throw "hello"; // std::unexpected called
6     }
7 }
```



Good to know

- The check is done at runtime, not at compile time
 - unlike Java
- When the *throw* clause is absent, any exception can go out
- To block all exceptions, use *throw()*



Good to know

- The check is done at runtime, not at compile time
 - unlike Java
- When the *throw* clause is absent, any exception can go out
- To block all exceptions, use *throw()*

```
1 int func(int a) {
2     // any exception can go out
3 }
4 int otherfunc(int a) throw() {
5     // no exception can go out
6 }
```



Deprecation of C++98 exceptions

After a lot of thinking and experiencing, the conclusions of the community on exception handling are :

- Never write an exception specification
- Except possibly an empty one



Deprecation of C++98 exceptions

After a lot of thinking and experiencing, the conclusions of the community on exception handling are :

- Never write an exception specification
- Except possibly an empty one

Some of the reasons

- throw specification is runtime only
 - does not allow compiler optimizations
 - on the contrary forces extra checks
 - generally terminates your program if violated
- throw specification clashes with templates
 - one cannot “template” the throw clause



What remains

C++11

throw is dead

- throw statements are deprecated
- even throw() (no exceptions)



What remains

C++11

throw is dead

- throw statements are deprecated
- even throw() (no exceptions)

long live noexcept

- noexcept a somehow equivalent to throw()
- but is checked at compile time
- so allows compiler optimizations



Full power of noexcept

C++11

3 uses of noexcept

- standalone

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

- as an expression saying whether exceptions can be sent

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

- as an operator to know whether a function launches exceptions

```
template <typename T> void foo()  
    noexcept(noexcept(T())) {};
```



Move semantic

- 4 Core modern C++
 - Constness
 - Constant Expressions
 - Exceptions
 - **Move semantic**
 - Copy elision
 - Templates
 - The STL
 - More STL
 - Lambdas
 - pointers and RAI



Move semantics : the problem

C++11

Non efficient code

```
1 void swap(std::vector<int> &a,  
2          std::vector<int> &b) {  
3     std::vector<int> c = a;  
4     a = b;  
5     b = c;  
6 }  
7 std::vector<int> v, w;  
8 for (int i = 0; i < 10000; i++) v.push_back(i);  
9 for (int i = 0; i < 10000; i++) w.push_back(i);  
10 swap(v, w);
```



Move semantics : the problem

C++11

Non efficient code

```
1 void swap(std::vector<int> &a,  
2          std::vector<int> &b) {  
3     std::vector<int> c = a;  
4     a = b;  
5     b = c;  
6 }  
7 std::vector<int> v, w;  
8 for (int i = 0; i < 10000; i++) v.push_back(i);  
9 for (int i = 0; i < 10000; i++) w.push_back(i);  
10 swap(v, w);
```

What really happens during swap

- 10k allocations + 10k releases
- 30k copies



Move semantics : the problem

C++11

Dedicated efficient code

```
1  std::vector<int> v, w;  
2  for (int i = 0; i < 10000; i++) v.push_back(i);  
3  for (int i = 0; i < 10000; i++) w.push_back(i);  
4  v.swap(w);
```



Move semantics : the problem

Dedicated efficient code

```
1  std::vector<int> v, w;  
2  for (int i = 0; i < 10000; i++) v.push_back(i);  
3  for (int i = 0; i < 10000; i++) w.push_back(i);  
4  v.swap(w);
```

What probably happens during swap

- 1 allocations + 1 releases
- 3 copies

only the pointers to underlying arrays were swapped



Move semantics : the problem

C++11

Another non efficient code

```
1  std::vector<int> vrandom(unsigned int n) {
2      std::vector<int> result;
3      for (int i = 0; i < n; i++) {
4          result.push_back(rand());
5      }
6      return result;
7  }
8  std::vector<int> v = vrandom(10000);
```



Move semantics : the problem

C++11

Another non efficient code

```
1  std::vector<int> vrandom(unsigned int n) {
2      std::vector<int> result;
3      for (int i = 0; i < n; i++) {
4          result.push_back(rand());
5      }
6      return result;
7  }
8  std::vector<int> v = vrandom(10000);
```

What really happens during assignment

- 10k allocations + 10k releases
- 10k copies

Note that in this case, the compiler may optimize the copy away.



Move semantics : the problem

C++11

Dedicated efficient way

```
1 void vrandom(unsigned int n, std::vector<int> &v) {
2     for (int i = 0; i < n; i++) {
3         v.push_back(rand());
4     }
5 }
6 std::vector<int> v;
7 vrandom(10000, v);
```



Move semantics : the problem

C++11

Dedicated efficient way

```
1 void vrandom(unsigned int n, std::vector<int> &v) {
2     for (int i = 0; i < n; i++) {
3         v.push_back(rand());
4     }
5 }
6 std::vector<int> v;
7 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 references
 - used for move semantic
 - denoted by &&
- 2 new members in every class, with move semantic :
 - 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 references
 - used for move semantic
 - denoted by &&
- 2 new members in every class, with move semantic :
 - a **move constructor** similar to copy constructor
 - a **move assignment operator** similar to assignment operator
(now called copy assignment operator)

Practically

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



Move semantics

C++11

A few important points concerning move semantic

- the whole STL can understand the move semantic
- move assignment operator is allowed to destroy source
 - so do not reuse source afterward
 - still, I advice to never leave inconsistent objects
- if not implemented, move falls back to copy version
- move is called by the compiler whenever possible
 - e.g. when passing temporary



Move semantics

C++11

A few important points concerning move semantic

- the whole STL can understand the move semantic
- move assignment operator is allowed to destroy source
 - so do not reuse source afterward
 - still, I advice to never leave inconsistent objects
- if not implemented, move falls back to copy version
- move is called by the compiler whenever possible
 - e.g. when passing temporary

Practically

```
1 T a;  
2 T b = a; // 1. Copy assign  
3 T c = T(2); // 2. Move assign  
4 T d = func(); // 3. Move assign
```



Move semantics

C++11

In some cases, you want to force a move

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



Move semantics

In some cases, you want to force a move

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

There are mainly two ways

- casting to an rvalue reference
- using the `std::move` function

```
6 T a;  
7 T b = a; // Copy assign  
8 T c = static_cast<T&&>(a); // Move assign  
9 T d = std::move(a); // Move assign
```



Move semantics : the easy way

C++11

Use copy and swap idiom

- implement an efficient swap method to your class
 - preferably outside the class so that it is symmetric
- use swap for move constructor
 - create empty object with constructor delegation
 - swap it with source
- use swap in move assignment
 - pass parameter by value
 - this should force creation of a local replica of source
 - as we are in the move assignment
our move constructor will be called
and source will be filled with an empty object
 - swap local object with *this
 - let local object be destructed when exiting the method
this will actually destroy the original content of the target



Move semantics : the easy way

C++11

Practically

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



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

- 4 Core modern C++
 - Constness
 - Constant Expressions
 - Exceptions
 - Move semantic
 - **Copy elision**
 - Templates
 - The STL
 - More STL
 - Lambdas
 - pointers and RAI



Guaranteed copy elision

C++17

What is copy elision

```
1  struct Foo { ... };
2  Foo f() {
3      return Foo();
4  }
5  int main() {
6      // compiler was authorised to elude the copy
7      Foo foo = f();
8  }
```

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

```
1  struct Foo {
2      Foo() { ... }
3      Foo(const Foo &) = delete;
4      Foo(Foo &&) = delete;
5  };
6  Foo f() {
7      return Foo();
8  }
9  int main() {
10     Foo foo = f();
11 }
```



Templates

- 4 Core modern C++
 - Constness
 - Constant Expressions
 - Exceptions
 - Move semantic
 - Copy elision
 - **Templates**
 - The STL
 - More STL
 - Lambdas
 - pointers and RAI



Templates

C++98

Concept

- The C++ way to write reusable code
 - aka macros on steroids
- Applicable to functions and objects

```
1  template<typename T>
2  const T & max(const T &A, const T &B) {
3      return A > B ? A : B;
4  }
5
6  template<typename T>
7  struct Vector {
8      int m_len;
9      T* m_data;
10 };
```



Warning

These are really like macros

- they are compiled n times
- 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

1
2
3
4

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

func(3)

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

func(5.2)

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



Arguments

- can be a class,
- you can have several
- last ones can have a default value

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



Templates implementation

```
1  template<typename KeyType=int, typename ValueType=KeyType>
2  struct Map {
3      void set(const KeyType &key, ValueType value);
4      ValueType get(const KeyType &key);
5  }
6
7  template<typename KeyType, typename ValueType>
8  void Map<KeyType, ValueType>::set
9      (const KeyType &key, ValueType value) {
10     ...
11 }
12
13 template<typename KeyType, typename ValueType>
14 ValueType Map<KeyType, ValueType>::get
15     (const KeyType &key) {
16     ...
17 }
```



Non-type template parameter

C++98 / C++17 / C++20

template parameters can also be a value

- integral types, pointer, enums in C++98
- auto in C++17
- floats and literals in C++20

```
1  template<unsigned int N> struct Polygon {  
2      Polygon(float radius);  
3      float perimeter() {return 2*N*sin(PI/N)*m_radius;}  
4      float m_radius;  
5  };
```



Specialization

templates can be specialized for given values of their parameter

```
1  template<typename F, unsigned int N> struct Polygon {
2      Polygon(F radius) : m_radius(radius) {}
3      F perimeter() {return 2*N*sin(PI/N)*m_radius;}
4      F m_radius;
5  };
6
7  template<typename F>
8  struct Polygon<F, 6> {
9      Polygon(F radius) : m_radius(radius) {}
10     F perimeter() {return 6*m_radius;}
11     F m_radius;
12 };
```



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

- 4 Core modern C++
 - Constness
 - Constant Expressions
 - Exceptions
 - Move semantic
 - Copy elision
 - Templates
 - **The STL**
 - More STL
 - Lambdas
 - pointers and RAI



The Standard Template Library

C++98

What it is

- A library of standard templates
- Everything you need, or ever dreamed of
 - 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
- Everything you need, or ever dreamed of
 - strings, containers, iterators
 - algorithms, functions, sorters
 - functors, allocators
 - ...
- Portable
- Reusable
- Efficient

Just use it

and adapt it to your needs, thanks to templates



STL in practice

C++11

```
1  #include <vector>
2  #include <algorithm>
3
4  std::vector<int> vi{5, 3, 4}; // initializer list
5  std::vector<int> vr(3); // constructor taking int
6
7  std::transform(vi.begin(), vi.end(),          // range1
8                vi.begin(),                    // start range2
9                vr.begin(),                    // start result
10               std::multiplies<int>()); // function
11
12  for(auto n : vr) {
13      std::cout << n << " ";
14  };
```



containers

- a structure containing data
- with a given way of handling it
- irrespective of
 - the data itself (templated)
 - the memory allocation of the structure (templated)
 - the algorithms that may use the structure
- examples
 - string
 - tuple, list, vector, deque
 - map, set, multimap, multiset, hash_map, hash-set, ...
 - stack, queue, priority_queue



iterators

- generalization of pointers
- allowing 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



algorithms

- implementation of an algorithm working on data
- with a well defined behavior (defined complexity)
- irrespective of
 - the data handled
 - the container where data live
 - the iterator used to go through data
- examples
 - for_each, find, find_if, count, count_if, search
 - copy, swap, transform, replace, fill, generate
 - remove, remove_if
 - unique, reverse, rotate, random, partition
 - sort, partial_sort, merge, min, max
 - lexicographical_compare, iota, accumulate, partial_sum



functions / functors

- generic utility functions/functors
- mostly useful to be passed to STL algorithms
- implemented independently of
 - the data handled (templated)
 - the context (algorithm) calling it
- examples
 - plus, minus, multiply, divide, modulus, negate
 - equal_to, less, greater, less_equal, ...
 - logical_and, logical_or, logical_not
 - identity, project1st, project2nd
 - binder1st, binder2nd, unary_compose, binary_compose



STL in practice

C++11

```
1  #include <vector>
2  #include <algorithm>
3
4  std::vector<int> vi{5, 3, 4}; // initializer list
5  std::vector<int> vr(3); // constructor taking int
6
7  std::transform(vi.begin(), vi.end(),          // range1
8                vi.begin(),                    // start range2
9                vr.begin(),                    // start result
10               std::multiplies<int>()); // function
11
12  for(auto n : vr) {
13      std::cout << n << " ";
14  };
```



STL in practice using C++98

C++98

```
1  #include <vector>
2  #include <algorithm>
3
4  std::vector<int> vi, vr(3);
5  vi.push_back(5); vi.push_back(3); vi.push_back(4);
6
7  std::transform(vi.begin(), vi.end(),          // range1
8               vi.begin(),                    // start range2
9               vr.begin(),                    // start result
10              std::multiplies<int>()); // function
11
12  for(std::vector<int>::iterator it = vr.begin();
13      it != vr.end();
14      it++) {
15      std::cout << *it << " ";
16  };
```



STL and functors

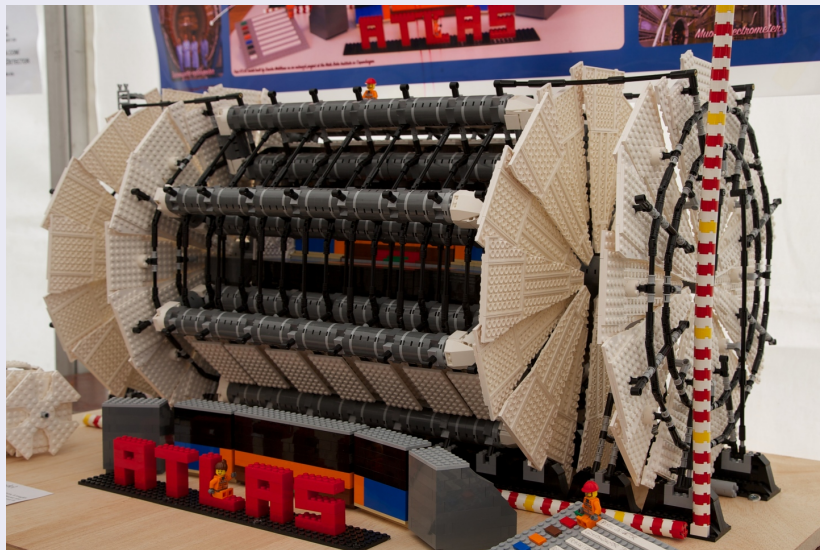
C++98

```
1 // Finds the first element in a list that lies in
2 // the range from 1 to 10.
3 list<int> L;
4 ...
5 list<int>::iterator in_range =
6     find_if(L.begin(), L.end(),
7             compose2(logical_and<bool>(),
8                     bind2nd(greater_equal<int>(), 1),
9                     bind2nd(less_equal<int>(), 10)));
10
11 // Computes  $\sin(x)/(x + \text{DBL\_MIN})$  for elements of a range.
12 transform(first, last, first,
13           compose2(divides<double>(),
14                   ptr_fun(sin),
15                   bind2nd(plus<double>(), DBL_MIN)));
```



Welcome to lego programming !

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

Some last warning

You may find the STL quite difficult to use.

- template syntax is simply awful
- it is hard to debug (compilers spit out mostly garbage)

However, this has improved a lot with C++11

And will again in C++20 with concepts



Loops and auto keyword with the STL

C++11

Old way

```
1  std::vector<int> a = ...;
2  int sum = 0;
3  for (std::vector<int>::iterator it = v.begin();
4       it != v.end(); it++) {
5     sum += *it;
6  }
```



Loops and auto keyword with the STL

C++11

Old way

```
1  std::vector<int> a = ...;
2  int sum = 0;
3  for (std::vector<int>::iterator it = v.begin();
4       it != v.end(); it++) {
5     sum += *it;
6  }
```

New way

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



Loops and auto keyword with the STL

C++11

Old way

```
1  std::vector<int> a = ...;
2  int sum = 0;
3  for (std::vector<int>::iterator it = v.begin();
4       it != v.end(); it++) {
5     sum += *it;
6  }
```

New way

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

STL way

```
10 std::vector<int> v = ...;
11 int sum = std::accumulate(v.begin(), v.end(), 0);
```



More STL

- 4 Core modern C++
 - Constness
 - Constant Expressions
 - Exceptions
 - Move semantic
 - Copy elision
 - Templates
 - The STL
 - **More STL**
 - Lambdas
 - pointers and RAI



Some new STL types

C++17

`std::optional`

- manages an optional contained value
- contextually converted to `bool`
- useful for the return value of a function that may fail

`std::any`

- a type-safe container for single values of any type
- the `any_cast` function provides type-safe access
- and throws `std::bad_any_cast` for bad access

`std::variant`

- a type-safe union
- `std::get` reads the value of the variant
- and throws `std::bad_variant_access` for bad accesses



non-member begin and end

C++11

The problem in C++98

STL containers and arrays have different syntax for loop

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



non-member begin and end

C++11

The problem in C++98

STL containers and arrays have different syntax for loop

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

A new syntax

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



non-member begin and end

C++11

The problem in C++98

STL containers and arrays have different syntax for loop

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

A new syntax

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

Allowing the best syntax

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



Structured Binding Declarations

C++17

Helps when using tuples as a return type.
Automatically creates variables and ties them.

C++14

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

C++17

```
7 void foo(std::tuple<int, double, long> tuple) {  
8     auto [ a, b, c ] = tuple;
```



Lambdas

- 4 Core modern C++
 - Constness
 - Constant Expressions
 - Exceptions
 - Move semantic
 - Copy elision
 - Templates
 - The STL
 - More STL
 - **Lambdas**
 - pointers and RAI



Function return type

C++11

A new way to specify function's return type

```
ReturnType fn_name(ArgType1, ArgType2); //old  
auto fn_name(ArgType1, ArgType2) -> ReturnType;
```



Function return type

A new way to specify function's return type

```
ReturnType fn_name(ArgType1, ArgType2); //old
auto fn_name(ArgType1, ArgType2) -> ReturnType;
```

Advantages

- Allows to simplify inner type definition

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

- ReturnType is not mandatory, compiler can deduce it
- will be used for lambdas



Lambdas

C++11

Definition

a lambda is a function with no name



Definition

a lambda is a function with no name

Python example

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



Simplified syntax

```
1 [] (args) -> type {  
2     code;  
3 }
```

The type specification is optional

Usage example

```
4 std::vector<int> data{1,2,3,4,5};  
5 for_each(begin(data), end(data),  
6         [](int i) {  
7             std::cout << "The square of " << i  
8                 << " is " << i*i << std::endl;  
9         });
```



Capture

C++11

Python code

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



Capture

C++11

Python code

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

First attempt in C++

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



Capture

C++11

Python code

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

First attempt in C++

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

Error

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



Variable capture

- external variables need to be explicitly captured
- captured variables are listed within initial []



Capture

Variable capture

- external variables need to be explicitly captured
- captured variables are listed within initial []

Example

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



Default capture is by value

C++11

Code example

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



Default capture is by value

C++11

Code example

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

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



Capture by reference

C++11

Simple example

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

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



Capture by reference

C++11

Simple example

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

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

Mixed case

One can of course mix values and references

```
5  int sum = 0, offset = 1;
6  std::vector<int> data{1,9,3,8,3,7,4,6,5};
7  for_each(begin(data), end(data),
8           [&sum, offset](int x) {
9             sum += x + offset;
10          });
```



Capture all

C++11

by value

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



Capture all

C++11

by value

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

by reference

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



Capture all

C++11

by value

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

by reference

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

exceptions

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

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



Lambdas rather than functors

C++11

Example

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

How it works

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



lambdas makes the STL usable

Before lambdas

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



lambdas makes the STL usable

With lambdas

```
1  std::vector<int> v{1, 2, 3};
2  std::transform(begin(v), end(v), begin(v),
3                  [](int value) {
4                      return value + 1;
5                  });
6  for (auto a : v) std::cout << a << " ";
```



lambdas makes the STL usable

With lambdas

```
1  std::vector<int> v{1, 2, 3};
2  std::transform(begin(v), end(v), begin(v),
3                  [](int value) {
4                      return value + 1;
5                  });
6  for (auto a : v) std::cout << a << " ";
```

Conclusion

Use the STL !



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 RAI

- 4 Core modern C++
 - Constness
 - Constant Expressions
 - Exceptions
 - Move semantic
 - Copy elision
 - Templates
 - The STL
 - More STL
 - Lambdas
 - **pointers and RAI**



Pointers : why they are error prone ?

C++98

They need initialization

```
1  char *s;  
2  try {  
3      foo();  
4      s = (char*) malloc(...);  
5      strncpy(s, ...);  
6  } catch (...) { ... }  
7  bar(s);
```



Pointers : why they are error prone ?

C++98

They need initialization**Seg Fault**

```
1  char *s;  
2  try {  
3      foo();  
4      s = (char*) malloc(...);  
5      strncpy(s, ...);  
6  } catch (...) { ... }  
7  bar(s);
```



Pointers : why they are error prone ?

C++98

They need initialization

Seg Fault

```
1  char *s;  
2  try {  
3      foo();  
4      s = (char*) malloc(...);
```

They need to be released

```
1  char *s = (char*) malloc(...);  
2  strncpy(s, ...);  
3  if (0 != strcmp(s, ...)) return;  
4  foo(s);  
5  free(s);
```



Pointers : why they are error prone ?

C++98

They need initialization

Seg Fault

```
1  char *s;  
2  try {  
3      foo();  
4      s = (char*) malloc(...);
```

They need to be released

Memory leak

```
1  char *s = (char*) malloc(...);  
2  strncpy(s, ...);  
3  if (0 != strcmp(s, ...)) return;  
4  foo(s);  
5  free(s);
```



Pointers : why they are error prone ?

C++98

They need initialization

Seg Fault

```
1 char *s;  
2 try {  
3     foo();  
4     s = (char*) malloc(...);
```

They need to be released

Memory leak

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

They need clear ownership

```
1 char *s = (char*) malloc(...);  
2 strncpy(s, ...);  
3 someVector.push_back(s);  
4 someSet.add(s);  
5 std::thread t1(vecConsumer, someVector);  
6 std::thread t2(setConsumer, someSet);
```

Pointers : why they are error prone ?

C++98

They need initialization**Seg Fault**

```
1  char *s;  
2  try {  
3      foo();  
4      s = (char*) malloc(...);
```

They need to be released**Memory leak**

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

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

```
1  char *s = (char*) malloc(...);  
2  strncpy(s, ...);  
3  someVector.push_back(s);  
4  someSet.add(s);  
5  std::thread t1(vecConsumer, someVector);  
6  std::thread t2(setConsumer, someSet);
```

This problem exists for any resource

C++98

For example with a file

```
1  try {
2      FILE *handle = std::fopen(path, "w+");
3      if (0 == handle) { throw ... }
4      if (std::fputs(str, handle) == EOF) {
5          throw ...
6      }
7      fclose(handle);
8  } catch (...) { ... }
```



Resource Acquisition Is Initialization

C++98

Practically

Use object semantic to acquire/release resources

- wrap the resource inside an object
- acquire resource via object 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



File class

```
1  class File {
2  public:
3      File(const char* filename) :
4          m_file_handle(std::fopen(filename, "w+")) {
5          if (m_file_handle == NULL) { throw ... }
6      }
7      ~File() { std::fclose(m_file_handle); }
8      void write (const char* str) {
9          if (std::fputs(str, m_file_handle) == EOF) {
10             throw ...
11         }
12     }
13 private:
14     FILE* m_file_handle;
15 };
```



Usage of File class

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



std::unique_ptr

C++11

an RAII pointer

- wraps a regular pointer
- has move only semantic
 - the pointer is only owned once
- in <memory> header



std::unique_ptr

C++11

an RAII pointer

- wraps a regular pointer
- has move only semantic
 - the pointer is only owned once
- in <memory> header

Usage

```
1  Foo *p = new Foo{}; // allocation
2  std::unique_ptr<Foo> uptr(p);
3  std::cout << uptr.get() << " points to "
4      << uptr->someMember << std::endl;
5  void f(std::unique_ptr<Foo> ptr);
6  f(std::move(uptr)); // transfer of ownership
7  // deallocation when exiting f
8  std::cout << uptr.get() << std::endl; // 0
```



Quiz

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

What do you expect ?



Quizz

C++11

```
1  Foo *p = new Foo{}; // allocation
2  std::unique_ptr<Foo> uptr(p);
3  void f(std::unique_ptr<Foo> ptr);
4  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

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



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

make_unique usage

```
1 // allocation of one Foo object,  
2 // calling constructor with one argument  
3 auto a = std::make_unique<Foo>(memberValue);  
4 std::cout << a.get() << " points to "  
5         << a->someMember << std::endl;  
6 // allocation of an array of Foos  
7 // calling default constructor  
8 auto b = std::make_unique<Foo[]>(10);  
9 // deallocations
```



RAII or raw pointers

C++11

When to use what ?

- Always use RAII for allocations
- You thus never have to deallocate !
- Use raw pointers for observer functions (or references)
 - remember that `unique_ptr` is move only



RAII or raw pointers

When to use what ?

- Always use RAII for allocations
- You thus never have to deallocate !
- Use raw pointers for observer functions (or references)
 - remember that `unique_ptr` is move only

A question of ownership

```
1 unique_ptr<T> producer();
2 void observer(const T&);
3 void modifier(T&);
4 void consumer(unique_ptr<T>);
5 unique_ptr<T> pt{producer()};
6 observer(*pt.get()); // Keep ownership
7 modifier(*pt.get()); // Keep ownership
8 consumer(std::move(pt)); // Transfer ownership
```



It's about lifetime management

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



shared_ptr, make_shared

shared_ptr : a reference counting pointers

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

make_shared : creates a shared_ptr

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



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++

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



Variadic templates

- 5 Expert C++
 - Variadic templates
 - Perfect forwarding
 - SFINAE



Basic variadic template

C++11

The idea

- a template with an arbitrary number of parameters
- ... syntax as in good old printf
- using recursivity and specialization for stopping it

Practically

```
1  template<typename T>
2  T adder(T v) {
3      return v;
4  }
5  template<typename T, typename... Args>
6  T adder(T first, Args... args) {
7      return first + adder(args...);
8  }
9  long sum = adder(1, 2, 3, 8, 7);
```



A couple of remarks

About performance

- do not be afraid by recursivity
- 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



Variadic templated class

The tuple example, simplified

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



Perfect forwarding

- 5 Expert C++
 - Variadic templates
 - Perfect forwarding
 - SFINAE



The problem

Trying to write a generic wrapper function

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

Example usage :

- `emplace_back`
- `make_unique`



Why is is not so simple ?

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

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 ?

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

const ref won't help : you may want to pass something non const and rvalue are not yet supported...



The solution : cover all cases

C++11

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



The new problem : scaling to n arguments

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

Exploding complexity

3^n complexity

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



Reference collapsing in C++98

C++11

Reference to references

They are forbidden in C++

But still may happen

```
1  template <typename T>
2  void foo(T t) {
3      T& k = t;
4      ...
5  }
6  int ii = 4;
7  foo<int&>(ii);
```

Practically

all compilers were collapsing the 2 references



rvalues have been added

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

C++11 standardization

The rule is simple : `&` always wins

`&& &`, `& &&`, `& &` → `&`

`&& &&` → `&&`



rvalue in type-deducing context

```

1  template <class T>
2  void func(T&& t) {}

```

In this context, && is not an rvalue

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

- if an lvalue of type U is given, T is deduced to U&
- if an rvalue, T is deduced to U

```

3  func(4);           // rvalue -> T is int
4  double d = 3.14;
5  func(d);          // lvalue -> T is double&
6  float f() {...}
7  func(f());        // rvalue -> T is float
8  int foo(int i) {
9      func(i);       // lvalue -> T is int&
10 }

```



std::remove_reference

Some template trickery removing reference from a type

```
1  template< class T >
2  struct remove_reference
3  {using type = T;};
4
5  template< class T >
6  struct remove_reference<T&>
7  {using type = T;};
8
9  template< class T >
10 struct remove_reference<T&&>
11 {using type = T;};
```

If T is a reference type, T::type is the type referred to by T.
Otherwise T::type is T.



Another template trickery keeping references and mapping non reference types to rvalue references

```
1  template<class T>
2  T&& forward(typename std::remove_reference<T>::type& t)
3      noexcept {
4      return static_cast<T&&>(t);
5  }
```

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

Putting it all together

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

How it works

- if we pass an rvalue to T (U&&)
 - arg will be of type U&&
 - func will be called with a U&&
- if we pass an lvalue to T (U&)
 - arg will be of type U&
 - func will be called with a U&
- if we pass a plain T (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

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



SFINAE

- 5 Expert C++
 - Variadic templates
 - Perfect forwarding
 - SFINAE



Substitution Failure Is Not An Error

C++11

The main idea

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

Example

```
1  template <typename T>
2  void f(const T& t,
3         typename T::iterator* it = nullptr) { }
4  void f(int a) { }    // ``sink'' function
5
6  f(1); // Calls void f(...)
```

Note : SFINAE will be largely superseded by concepts in C++20



The main idea

- gives the type of the of the expression it will evaluate
- at compile time

Example

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



The main idea

- gives you a "fake reference" to an object at compile time
- useful for types that cannot be easily constructed

Example

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



true_type and false_type

The main idea

- encapsulate a constexpr boolean “true” and “false”
- can be inherited
- constexpr

Example

```
1  struct testStruct : std::true_type { };
2  constexpr bool testVar = testStruct();
3  bool test = testStruct::value; // 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

```
1  template <typename T, typename = void>
2  struct hasFoo : std::false_type {};
3
4  template <typename T>
5  struct hasFoo<T, decltype(std::declval<T>().foo())>
6    : std::true_type {};
7
8  std::cout << hasFoo<MyType>::value << std::endl;
```



enable_if / enable_if_t

```
1  template<bool B, class T = void> struct enable_if {};  
2  template<class T>  
3  struct enable_if<true, T> { typedef T type; };  
4  template< bool B, class T = void >  
5  using enable_if_t = typename enable_if<B,T>::type;
```

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

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

- Checks whether T is ...
- At compile time
- Has member value, as boolean telling whether it was



Gaudi usage example

```
1  constexpr struct deref_t {
2      template
3          <typename In,
4            typename = typename std::enable_if_t
5                <!std::is_pointer_v<In>>>
6      In& operator()( In& in ) const { return in; }
7
8      template <typename In>
9      In& operator()( In* in ) const {
10         assert(in!=nullptr); return *in;
11     }
12 } deref {};
```



Back to variadic templated class

The tuple get method

```
1  template <size_t k, typename TUPLE>
2  struct elem_type_holder;
3
4  template <class T, class... Ts>
5  struct elem_type_holder<0, tuple<T, Ts...>> {
6      typedef T type;
7  };
8
9  template <size_t k, class T, class... Ts>
10 struct elem_type_holder<k, tuple<T, Ts...>> {
11     typedef typename elem_type_holder
12         <k - 1, tuple<Ts...>>::type type;
13 };
```



Back to variadic templated class

The tuple get method

```
1  template <size_t k, class... Ts>
2  typename std::enable_if_t<k == 0,
3      typename elem_type_holder
4          <0, tuple<Ts...>>::type&>
5  get(tuple<Ts...>& t) {
6      return t.m_tail;
7  }
8  template <size_t k, class T, class... Ts>
9  typename std::enable_if_t<k != 0,
10     typename elem_type_holder
11         <k-1, tuple<Ts...>>::type&>
12  get(tuple<T, Ts...>& t) {
13     tuple<Ts...>& base = t;
14     return get<k - 1>(base);
15 }
```



Useful tools

1 History and goals

2 Langage basics

3 Object orientation (OO)

4 Core modern C++

5 Expert C++

6 Useful tools

- C++ editor
- Code management
- The Compiling Chain
- Debugging
- The Valgrind family
- Static code analysis

7 Concurrency

8 C++ and python



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



C++ editor

Choose it wisely

- it can improve dramatically your efficiency by
 - coloring the code for you to “see” the structure
 - helping indenting properly
 - allowing you to navigate easily in the source tree
 - helping for compilation/debugging

A few tools

▶ Visual Studio the Microsoft way

▶ Visual Studio Code more lightweight, open source and portable

▶ Eclipse similar, also open source and portable

▶ Emacs the expert way. Extremely powerful. Programmable
It is to IDEs what latex is to PowerPoint

Choosing one is mostly a matter of taste

Code management

6 Useful tools

- C++ editor
- **Code management**
- The Compiling Chain
- Debugging
- The Valgrind family
- Static code analysis



Code management tool

Please use one !

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

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

```
# mkdir myProject; cd myProject; git init
Initialized empty Git repository in /tmp/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
```

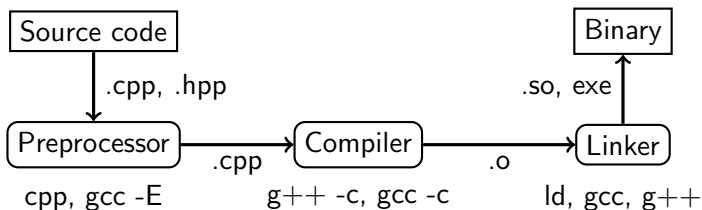


The Compiling Chain

- 6 Useful tools
 - C++ editor
 - Code management
 - **The Compiling Chain**
 - Debugging
 - The Valgrind family
 - Static code analysis



The compiling chain



The steps

`cpp` the preprocessor

handles the `#` directives (macros, includes)
creates “complete” source code

`g++` the compiler

creates assembly 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
- ▶ `icc` the intel compiler
proprietary but now free
optimized for Intel hardware
- ▶ Visual C++ the Windows way

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` the way to get all warnings

`-Werror` the way to force yourself to look at warnings

Around optimization

`-g` add debug symbols

`-O0, -O2` 0 = no optimization, -O2 = optimized

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 : portable but complex

```
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
```



Compiler chain

Exercise Time

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



Debugging

- 6 Useful tools
 - C++ editor
 - Code management
 - The Compiling Chain
 - **Debugging**
 - The Valgrind family
 - Static code analysis



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, still young
- ▶ `idb` the intel debugger, proprietary



`gdb` crash course

start `gdb`

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

inspect state

`bt` prints a backtrace

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

`list` show code around current point

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

breakpoints

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

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



Exercise Time

- go to code/debug
- compile, run, see the crash
- run it in gdb
- inspect backtrace, variables
- find problem and fix bug
- try stepping, breakpoints
- use `-Wall -Wextra` and see warning



The Valgrind family

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



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 non allocated memory
- and even tell you when it was deallocated if it was
- or what it 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 biased
- 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
 - C++ editor
 - Code management
 - The Compiling Chain
 - Debugging
 - The Valgrind family
 - **Static code analysis**



Static analysis

The problem

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



Static analysis

The problem

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

A (partial) solution : analyzing the source code

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



Static analysis

The problem

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

A (partial) solution : analyzing the source code

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

Existing tools

- ▶ Coverity proprietary tool, the most complete
- ▶ cppcheck free and opensource, but less complete
- ▶ scan-build the clang source 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



Concurrency

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



Threads and async

- 7 Concurrency
 - Threads and async
 - Mutexes



Threading

- new object `std::thread` in `<thread>` header
- takes a function as argument of its constructor
- must be called on `join` or program is terminated



Threading

- new object `std::thread` in `<thread>` header
- takes a function as argument of its constructor
- must be called on `join` or program is terminated

Example code

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



The thread constructor

C++11

Can take a function and its arguments

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



The thread constructor

Can take a function and its arguments

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

Can take any function like object

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



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”



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”

Pratically

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



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”

Pratically

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

Example code

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



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

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

```
1  int computeSth() {...}
2  auto res = std::async(std::launch::async,
3                       computeSth);
4  auto res2 = std::async(std::launch::deferred,
5                          computeSth);
```



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



std::packaged_task template

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

Usage

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



Mutexes

- 7 Concurrency
 - Threads and async
 - **Mutexes**



Example code

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



Example code

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

What do you expect ? Try it in code/race



Example code

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

What do you expect ? Try it in code/race

Anything between 100 and 200 !!!



Definition (wikipedia)

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

Practically

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



Definition (wikipedia)

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

Practically

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

Is ++ operator atomic ?



Definition (wikipedia)

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

Practically

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

Is ++ operator atomic ?

Usually not. It behaves like :

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

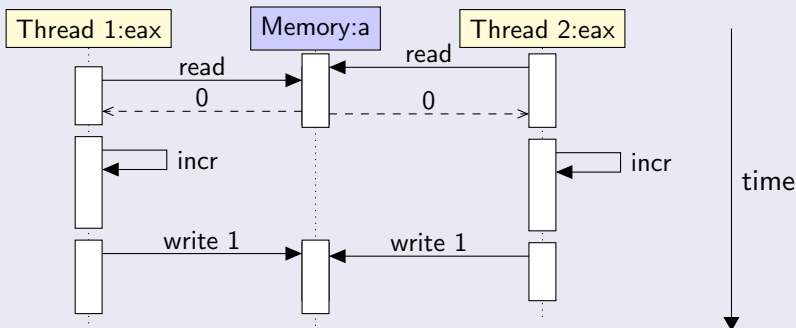


Timing

Code

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

For 2 threads



Concept

- a lock to serialize access to a non atomic piece of code



Concept

- a lock to serialize access to a non atomic piece of code

The objects

`std::mutex` in the mutex header

`std::lock_guard` for an RAII version of it

`std::unique_lock` same and can be released/relocked



Mutexes

Concept

- a lock to serialize access to a non atomic piece of code

The objects

`std::mutex` in the mutex header

`std::lock_guard` for an RAII version of it

`std::unique_lock` same and can be released/relocked

Practically

```
1  int a = 0;
2  std::mutex m;
3  void inc() {
4      std::lock_guard<std::mutex> guard(m);
5      a++;
6  };
```



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



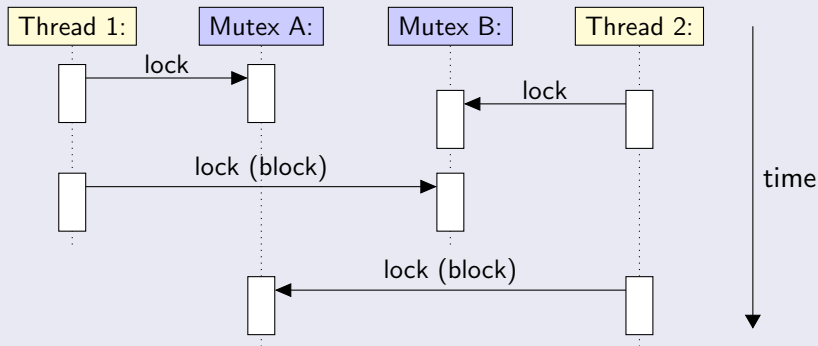
Dead lock

C++11

Scenario

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

Sequence diagram



How to avoid dead locks

Possible solutions

- 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



How to express thread dependencies

- Allows a thread to sleep until a given condition is satisfied
- `std::condition_variable` object from `condition_variable` header



How to express thread dependencies

- Allows a thread to sleep until a given condition is satisfied
- `std::condition_variable` object from `condition_variable` header

Usage

- wraps an RAII lock around a mutex
- `wait()` will hang until the condition is met
 - you can have several waiters sharing the same mutex
- `notify_one()` will wake up on waiter
- `notify_all()` will wake up all waiters



Condition variable usage

Example code

```
1  int value = -1;
2  std::mutex mutex;
3  std::condition_variable cond;
4  auto t1 = std::thread([&cond] () {
5      value = ... long process ...;
6      cond.notify_all();
7  });
8  auto t2 = std::thread([&cond,&mutex,&value] () {
9      std::unique_lock<std::mutex> lock{mutex};
10     cond.wait(lock, [&value] { return value > 0; });
11     ... use value ...
12 });
13 { std::unique_lock<std::mutex> lock{mutex};
14     cond.wait(lock, [] { return value > 0; });
15     std::cout << value << std::endl; }
```



C++ and python

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



Writing a module

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



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

C++ code : mandel.hpp

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



Basic Module(1) : wrap your method

mandelModule.cpp - see code/python exercise

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



Basic Module(2) : create the python module

mandelModule.cpp - see code/python exercise

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



Basic Module(3) : use it

First compile the module

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

mandel.py - see code/python exercise

```
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 cppy 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

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

Binary symbols : file.o

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



C++ mangling

Source : file.cpp

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

Binary symbols : file.o

```
# nm file.o  
000000000000000000 T _Z3sumff  
00000000000000002a T _Z6squaref  
00000000000000001a T _Z6squarei
```



Forcing C mangling in C++

```
extern "C"
```

These functions will use C mangling :

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



Forcing C mangling in C++

```
extern "C"
```

These functions will use C mangling :

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

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



Forcing C mangling in C++

```
extern "C"
```

These functions will use C mangling :

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

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

Limitations

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



The ctypes module

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



The ctypes python module

From the documentation

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



ctypes : usage example

C++ code : mandel.hpp

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

"C" code : mandel_cwrapper.hpp

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

calling the mandel library

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



Marrying C⁺⁺ 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 cppy project

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



Automatic Python-C++ bindings

The `cpyy` project

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

How it works

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



cpyyy crash course

Shamelessly copied from the cppy documentation

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

```
>>> import cppy
>>> cppy.cppdef("""
... class MyClass {
... public:
...     MyClass(int i) : m_data(i) {}
...     virtual ~MyClass() {}
...     virtual int add_int(int i) { return m_data + i; }
...     int m_data;
... };""")
True
>>> from cppy.gbl import MyClass
>>> m = MyClass(42)
>>> cppy.cppdef("""
... void say_hello(MyClass* m) {
...     std::cout << "Hello, the number is: " << m->m_data << std::endl;
... }""")
True
>>> MyClass.say_hello = cppy.gbl.say_hello
>>> m.say_hello()
Hello, the number is: 42
>>> m.m_data = 13
>>> m.say_hello()
Hello, the number is: 13
```



This is the end

Questions ?

<http://gitlab.cern.ch/sponce/cpluspluscourse>
<http://cern.ch/sponce/C++Course>

