Sébastien Ponce sebastien.ponce@cern.ch

CERN

ESIPAP 2020
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
  - 298 slides, 420 pages, 15 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. Language basics
3. Object orientation (OO)
4. Core modern C++
5. Expert C++
6. Useful tools
7. Concurrency
8. C++ and python
Detailed outline

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

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

3. Object orientation (OO)
   - Objects and Classes
   - Inheritance

4. Core modern C++
   - Constructors/destructors
   - Static members
   - Allocating objects
   - Advanced OO
   - Operators
   - Functors

5. Expert C++
   - Core modern C++
   - Constness
   - Constant Expressions
   - Exceptions
   - Move semantic
   - Copy elision
   - Templates
   - The STL
   - More STL
   - Lambdas
   - Pointers and RAII

6. Useful tools
   - Variadic templates
   - Perfect forwarding
   - SFINAE
   - 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
History and goals

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

2. Language basics

3. Object orientation (OO)

4. Core modern C++

5. Expert C++

6. Useful tools

7. Concurrency

8. C++ and python
History and goals

1. History
2. Why we use it?
C/\textit{C++} origins

- 1967: Simula
- 1978: B
- 1980: K and R C
- 1980: Classic C
- 1985: Early \textit{C++}
- 1989: ARM \textit{C++}
- 1989: C89
- 1998: \textit{C++98}
- 1999: C99
- 2011: \textit{C++11}
- 2014: \textit{C++14}
- 2017: \textit{C++17}
- 2020: \textit{C++20}

\textit{C} inventor: Dennis M. Ritchie
\textit{C++} inventor: Bjarne Stroustrup

- Both \textit{C} and \textit{C++} are born in Bell Labs
- \textit{C++} \textit{almost} embeds \textit{C}
- \textit{C} and \textit{C++} are still under development
- We will discuss all \textit{C++} specs
- Each slide will be marked with first spec introducing the feature
status

- A new C++ specification every 3 years
  - C++20 is ready, supposed to be official in May
- Bringing each time a lot of goodies
status

- A new C++ specification every 3 years
  - C++20 is ready, supposed to be official in May
- 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

<table>
<thead>
<tr>
<th></th>
<th>gcc</th>
<th>clang</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>4.8</td>
<td>3.3</td>
</tr>
<tr>
<td>14</td>
<td>4.9</td>
<td>3.4</td>
</tr>
<tr>
<td>17</td>
<td>7.3</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>&gt;10</td>
<td>&gt;10</td>
</tr>
</tbody>
</table>

Table: Minimum versions of gcc and clang for a given C++ version
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?
1 History and goals

2 Language basics
- Core syntax and types
- Arrays and Pointers
- Operators
- 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 Langage basics

- Core syntax and types
- Arrays and Pointers
- Operators
- Compound data types
- Functions
- References
- Control instructions
- Headers and interfaces
- Auto keyword
```cpp
#include <iostream>

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

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

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

/**
 * Best choice: doxygen compatible comments
 * fn bool isOdd(int i)
 * brief checks whether i is odd
 * param i input
 * return true if i is odd, otherwise false
 */
bool isOdd(int i);
Basic types(1)

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)

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

One needs to include specific header

```cpp
#include <cstdint>

int8_t c = -3;   // 8 bits, replaces char
uint8_t c = 4;   // 8 bits, replaces unsigned char

int16_t s = -444;  // 16 bits, replaced short
uint16_t s = 444;  // 16 bits, replaced unsigned short

int32_t s = -0674;  // 32 bits, replaced int
uint32_t s = 0674;  // 32 bits, replaced unsigned int

int64_t s = -0x1bc;  // 64 bits, replaced long long
uint64_t s = 0x1bc;  // 64 bits, replaced unsigned long long
```
2 Langage basics

- Core syntax and types
- Arrays and Pointers
- Operators
- Compound data types
- Functions
- References
- Control instructions
- Headers and interfaces
- Auto keyword
### Static arrays

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

---

S. Ponce

C++ course
int i = 4;

int *pi = &i;

int j = *pi + 1;

int ai[] = {1, 2, 3};

int *pai = ai;

int *paj = pai + 1;

int k = *paj + 1;

// not compiling
int *pak = k;

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

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

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

// not compiling
int *pak = k;

// seg fault!
int *pak = (int*)k;
int l = *pak;
```
```cpp
int i = 4;
int *pi = &i;
int j = *pi + 1;

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

// not compiling
int *pak = k;

// seg fault!
int *pak = (int*)k;
int l = *pak;
```
```cpp
int i = 4;
int *pi = &i;
int j = *pi + 1;

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

// not compiling
int *pak = k;

// seg fault!
int *pak = (int*)k;
int l = *pak;
```
```cpp
int i = 4;
int *pi = &i;
int j = *pi + 1;

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

// not compiling
int *pak = k;

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

Memory layout:
```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3040</td>
<td>0x3040A</td>
</tr>
<tr>
<td>0x3049</td>
<td>0x3049</td>
</tr>
<tr>
<td>0x3048</td>
<td>0x3048</td>
</tr>
<tr>
<td>0x3047</td>
<td>0x3047</td>
</tr>
<tr>
<td>0x3046</td>
<td>0x3046</td>
</tr>
<tr>
<td>0x3045</td>
<td>0x3045</td>
</tr>
<tr>
<td>0x3044</td>
<td>0x3044</td>
</tr>
<tr>
<td>0x3043</td>
<td>0x3043</td>
</tr>
<tr>
<td>0x3042</td>
<td>0x3042</td>
</tr>
<tr>
<td>0x3040</td>
<td>0x3040</td>
</tr>
</tbody>
</table>

j = 5
pi = 0x3040
i = 4
```cpp
int i = 4;
int *pi = &i;
int j = *pi + 1;

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

// not compiling
int *pak = k;

// seg fault!
int *pak = (int*)k;
int l = *pak;
```
int i = 4;
int *pi = &i;
int j = *pi + 1;

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

// not compiling
int *pak = k;

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

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

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

// not compiling
int *pak = k;

// seg fault!
int *pak = (int*)k;
int l = *pak;
```
```
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;
```
```cpp
int i = 4;
int *pi = &i;
int j = *pi + 1;

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

// not compiling
int *pak = k;

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

```cpp
void* vp = nullptr;
int* ip = nullptr;
int i = NULL;  // OK -> bug ?
int i = nullptr;  // ERROR
```
```c++
#include <cstdlib>
#include <cstring>

int *bad;          // pointer to random address
int *ai = nullptr; // better. Can be tested

// allocate array of 10 ints (not initialized)
ai = (int*) malloc(10*sizeof(int));
// and set them to 0
memset(ai, 0, 10*sizeof(int));

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

// liberate memory
free(ai);
```
2. **Language basics**

- Core syntax and types
- Arrays and Pointers
- **Operators**
- Compound data types
- Functions
- References
- Control instructions
- Headers and interfaces
- Auto keyword
## Binary & Assignment Operators

```cpp
int i = 1 + 4 - 2;       // 3
i *= 3;                  // 9
i /= 2;                  // 4
i = 23 % i;              // modulo => 3
```
Binary & Assignment Operators

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

Increment / Decrement

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

Binary & Assignment Operators

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

Increment / Decrement

```c++
// Use wisely
int i = 0; i++;       // i = 1
int j = ++i;          // i = 2, j = 2
int k = i++;          // i = 3, k = 2
int l = --i;          // i = 2, l = 2
int m = i--;          // i = 1, m = 2
```
# Bitwise and Assignment Operators

1. `int i = 0xee & 0x55; // 0x44`
2. `i |= 0xee; // 0xee`
3. `i ^= 0x55; // 0xbb`
4. `int j = ~0xee; // 0xffffff11`
5. `int k = 0x1f << 3; // 0x78`
6. `int l = 0x1f >> 2; // 0x7`
### Bitwise and Assignment Operators

```c++
int i = 0xee & 0x55;  // 0x44
i |= 0xee;            // 0xee
i ^= 0x55;            // 0xbb
int j = ~0xee;        // 0xffffffff11
int k = 0x1f << 3;    // 0x78
int l = 0x1f >> 2;    // 0x7
```

### Boolean Operators

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

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

Don't use
Comparison Operators

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

precedences

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

Don't use - use parenthesis
Compound data types

2 Language basics
- Core syntax and types
- Arrays and Pointers
- Operators
- **Compound data types**
- Functions
- References
- Control instructions
- Headers and interfaces
- Auto keyword
```
struct Individual {
    unsigned char age;
    float weight;
};

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

Individual teacher = {
    .age = 45,
    .weight = 67
};
```

“members” grouped together under one name
```cpp
struct Individual {
    unsigned char age;
    float weight;
};

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

Individual teacher = {
    .age = 45,
    .weight = 67
};

Individual *studentPtr = &student;
studentPtr->age = 25;
```

```
Memory layout

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3053</td>
<td>0x304F</td>
</tr>
<tr>
<td>0x304B</td>
<td>0x3047</td>
</tr>
<tr>
<td>0x3043</td>
<td></td>
</tr>
</tbody>
</table>
```
struct Individual {
    unsigned char age;
    float weight;
};

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

Individual teacher = {
    .age = 45,
    .weight=67
};

Individual *studentPtr = &student;
studentPtr->age = 25;

Memory layout

<table>
<thead>
<tr>
<th>7</th>
<th>8</th>
<th>.</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>student</th>
<th>0x3053</th>
<th>0x304F</th>
<th>0x304B</th>
<th>0x3047</th>
<th>0x3043</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>0x3043</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>weight</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

"members" grouped together under one name
```
struct Individual {
    unsigned char age;
    float weight;
};

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

Individual teacher = {
    .age = 45,
    .weight = 67
};

Individual *studentPtr = &student;
studentPtr->age = 25;
```
union Duration {  
    int seconds;
    short hours;
    char days;
};

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

Duration d1, d2, d3;
d1.seconds = 259200;
d2.hours = 72;
d3.days = 3;
d1.days = 3; // d1.seconds overwritten
int a = d1.seconds; // d1.seconds is garbage
### union

“members” packed together at same memory location

```cpp
union Duration {
    int seconds;
    short hours;
    char days;
};

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

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

Duration d1, d2, d3;
d1.seconds = 259200;
d2.hours = 72;
d3.days = 3;
d1.days = 3; // d1.seconds overwritten
int a = d1.seconds; // d1.seconds is garbage

“members” packed together at same memory location

Memory layout

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3040</td>
<td>d1.days=3</td>
</tr>
<tr>
<td>0x3041</td>
<td>d2.hours=72</td>
</tr>
<tr>
<td>0x3042</td>
<td>d3.days=3</td>
</tr>
<tr>
<td>0x3043</td>
<td>d1.seconds=259200</td>
</tr>
</tbody>
</table>
“members” packed together at same memory location

```c++
union Duration {
    int seconds;
    short hours;
    char days;
};

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

<table>
<thead>
<tr>
<th></th>
<th>Memory layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3040</td>
<td>d3.days=3</td>
</tr>
<tr>
<td>0x3041</td>
<td>d2.hours=72</td>
</tr>
<tr>
<td>0x3042</td>
<td>d1.days=3</td>
</tr>
<tr>
<td>0x3043</td>
<td></td>
</tr>
</tbody>
</table>
# Enums

```cpp
enum VehicleType { 
    BIKE,   // 0
    CAR,    // 1
    BUS,    // 2
};

VehicleType t = CAR;
```

```cpp
enum VehicleType { 
    BIKE = 3,
    CAR = 5,
    BUS = 7,
};

VehicleType t2 = BUS;
```
Scoped enumeration, aka enum class

Same syntax as enum, with scope

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

Same syntax as enum, with scope

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

Only advantages

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

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

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

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

struct Rectangle {
    float width;
    float height;
};

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

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

Shape t;
t.type = ShapeType::Rectangle;
t.rect.width = 3;
t.rect.height = 4;
```
enum class ShapeType {
    Circle,
    Rectangle
};

struct Rectangle {
    float width;
    float height;
};

struct Shape {
    ShapeType type;
    union {
        float radius;
        Rectangle rect;
    }
};
```cpp
enum class ShapeType {
    Circle,
    Rectangle
};

struct Rectangle {
    float width;
    float height;
};

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

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

Shape t;
t.type = ShapeType::Rectangle;
t.rect.width = 3;
t.rect.height = 4;
```
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;
2 Language basics

- Core syntax and types
- Arrays and Pointers
- Operators
- Compound data types
- Functions
  - References
  - Control instructions
  - Headers and interfaces
  - Auto keyword
Functions

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

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

// no parameter
void hello() {
    printf("Hello World");
}

// no return
void log(char* msg) {
    printf("%s", msg);
}
```
Parameter are passed by value

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

BigStruct s;

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

printBS(s); // copy

// parameter by pointer
void printBSp(BigStruct *q) {
    ...
}

printBSp(&s); // no copy
```
Parameter are passed by value

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

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

// parameter by pointer
void printBSp(BigStruct *p) {
    ...
}
printBSp(&s); // no copy
```

Memory layout:
```
0x3040...
0x3045
0x3044
0x3043
0x3042
0x3041
0x3040
```
Parameter are passed by value

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

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

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

BigStruct s;

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

printBS(s); // copy

// parameter by pointer
void printBSp(BigStruct *q) {
    ...
}

printBSp(&s); // no copy
```
Parameter are passed by value

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

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

// parameter by pointer
void printBSp(BigStruct *q) {
    ...
}
printBSp(&s);  // no copy
```
Parameter are passed by value

```c++
struct BigStruct {...};
BigStruct s;

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

// parameter by pointer
void printBSp(BigStruct *q) {
    ...
}
printBSp(&s); // no copy
```
Parameter are passed by value

```
1 struct SmallStruct {int a};
2 SmallStruct s = {.a = 1};
3
4 void changeSS(SmallStruct p) {
5    p.a = 2;
6 }
7 changeSS(s);
8 // s.a = 1

9 void changeSS2(SmallStruct *q) {
10    q->a = 2;
11 }
12 changeSS2(&s);
13 // s.a = 2
```
Parameter are passed by value

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

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

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

<table>
<thead>
<tr>
<th>Memory layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3040</td>
</tr>
<tr>
<td>0x3041</td>
</tr>
<tr>
<td>0x3042</td>
</tr>
</tbody>
</table>

```print```
Parameter are passed by value

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

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

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

Memory layout:

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

S. Ponce  C++ course
Parameter are passed by value

```cpp
#include <iostream>

struct SmallStruct { int a; }

SmallStruct s = {.a = 1};

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

changeSS(s);

// s.a = 1

void changeSS2(SmallStruct *q) {
    q->a = 2;
}

changeSS2(&s);

// s.a = 2
```

Memory layout

```
0x3040 0x3041 0x3042

p.a = 1
s.a = 1
```
Parameter are passed by value

```c++
struct SmallStruct { int a; }
SmallStruct s = {.a = 1};

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

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

Memory layout:

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

Parameter are passed by value

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

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

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

Memory layout

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3040</td>
<td></td>
</tr>
<tr>
<td>0x3041</td>
<td></td>
</tr>
<tr>
<td>0x3042</td>
<td></td>
</tr>
</tbody>
</table>

s.a = 1

Parameter are passed by value

```c++
struct SmallStruct { int a; }
SmallStruct s = {.a = 1};

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

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

Memory layout

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3040</td>
<td>s.a</td>
</tr>
<tr>
<td>0x3041</td>
<td></td>
</tr>
<tr>
<td>0x3042</td>
<td>q</td>
</tr>
</tbody>
</table>
Parameter are passed by value

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

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

void changeSS2(SmallStruct *q) {
    q->a = 2;
}
changeSS2(&s);
// s.a = 2
```
2 Langage basics

- Core syntax and types
- Arrays and Pointers
- Operators
- Compound data types
- Functions

- References
  - Control instructions
  - Headers and interfaces
  - Auto keyword
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
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
```
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)
Specificities of reference

- natural syntax
- will never be NULL
- thus cannot reference temporary object

Advantages of pointers

- can be NULL
- clearly indicates that argument may be modified
Pointers vs References

Specificities of reference

- natural syntax
- will never be NULL
- thus cannot reference temporary object

Advantages of pointers

- can be NULL
- 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
2 Langage basics

- Core syntax and types
- Arrays and Pointers
- Operators
- Compound data types
- Functions
- References
- Control instructions
- Headers and interfaces
- Auto keyword
Control instructions: if

### if syntax

```
if (condition1) {
    Instructions1;
} else if (condition2) {
    Instructions2;
} else {
    Instructions3;
}
```

- `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

Practical example

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

Syntax

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

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

Practical example

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

Syntax

test ? expression1 : expression2;
• if test is true expression1 is returned
• else expression 2 is returned

Practical example

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

Syntax

```cpp
    test ? expression1 : expression2;
    • if test is true expression1 is returned
    • else expression 2 is returned
```

Practical example

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

Do not abuse

explicit ifs are easier to read

to be used only when obvious and not nested
Control instructions: switch

Syntax

```cpp
switch(identifier) {
    case c1 : instructions1; break;
    case c2 : instructions2; break;
    case c3 : instructions3; break;
    ...
    default : instructiond; break;
}
```

- `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

Syntax

```
switch(identifier) {
    case c1 : instructions1; break;
    case c2 : instructions2; break;
    case c3 : instructions3; break;
    ...
    default : instructiond; break;
}
```

- `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

Practical example

```c
enum Lang { FRENCH, GERMAN, ENGLISH, OTHER };
...
switch (language) {
  case FRENCH:
    printf("Bonjour");
    break;
  case GERMAN:
    printf("Guten tag");
    break;
  case ENGLISH:
    printf("Good morning");
    break;
  default:
    printf("I do not talk your language");
}
```
[[fallthrough]] attribute

### C++14

```cpp
switch (c) {
  case 'a':
    f(); // Warning emitted
  case 'c':
    h();
}
```

### C++17

```cpp
switch (c) {
  case 'a':
    f();
    [[fallthrough]]; // Warning suppressed
  case 'c':
    h();
}
```
init-statements for if and switch

Allows to simplify if and switch statements

```cpp
// C++14
Value val = GetValue();
if (condition(val)) {
    // on success
} else {
    // on false...
}

// C++17
if (Value val = GetValue(); condition(val)) {
    // on success
} else {
    // on false...
}
val is visible only inside the if and else statements
```
Control instructions: for loop

for loop syntax

```cpp
for(initializations; condition; increments) {
    instructions;
}
```

- initializations and increments are comma separated
- initializations can contain declarations
- braces are optional if there is a single instruction
Control instructions: for loop

**for loop syntax**

```cpp
for(initializations; condition; increments) {
    instructions;
}
```

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

**Practical example**

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

for loop syntax

```
for(initializations; condition; increments) {
    instructions;
}
```
- initializations and increments are comma separated
- initializations can contain declarations
- braces are optional if there is a single instruction

Practical example

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

Do not abuse the syntax

The for statement should fit in 1-3 lines
Range based loops

Reason of being

- simplifies loops tremendously
- especially with STL containers

Syntax

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

Example code

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

while loop syntax

```cpp
while(condition) {
    instructions;
}
```

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

- braces are optional if there is a single instruction
Control instructions : while loop

while loop syntax

```c++
while (condition) {
    instructions;
}

do {
    Instructions;
} while (condition);

braces are optional if there is a single instruction
```

Practical example

```c++
while (n != 1)
    if (0 == n%2) n /= 2;
else n = 3 * n + 1;
```
Control instructions: commands

control commands

- **break** goes out of the loop
- **continue** goes immediately to next iteration
- **return** goes out of current function
Control instructions: commands

**control commands**

- **break** goes out of the loop
- **continue** goes immediately to next iteration
- **return** goes out of current function

**Practical example**

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

2 Langage basics

- Core syntax and types
- Arrays and Pointers
- Operators
- Compound data types
- Functions
- References
- Control instructions
- Headers and interfaces
- Auto keyword
Interface

Set of declarations defining some functionality

- defined in a "header file"
- no implementation defined

Header: hello.hpp

```cpp
void printHello();
```

Usage: myfile.cpp

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

// macros
#define MY_GOLDEN_NUMBER 1746

// compile time decision
#ifdef USE64BITS
    typedef uint64_t myint;
#else
    typedef uint32_t myint;
#endif

Use only in very restricted cases
- include of headers
- hardcoded constants before C++11
- portability necessity
2 Language basics

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

- Auto keyword
Auto keyword

Reason of being

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

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

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

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

Practical usage

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

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

3 Object orientation (OO)

- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Operators
- Functors
What are classes and objects

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

MyFirstClass myObj;
myObj.a = 2;

// let's square a
myObj.squareA();
Separating the interface

Header: MyFirstClass.hpp

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

Implementation: MyFirstClass.cpp

```cpp
#include "MyFirstClass.hpp"

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

void MyFirstClass::sum(int b) {
    return a + b;
}
```
A word on namespaces

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

```cpp
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          int a = 2;  //same as above
17          p::a = 2;
18          p::inner::a = 4;
19          inner::a = 4;
20          n::a = 5;
21      }
22  }
23  using namespace p::inner;
24  int g() {
25      int a = 3; // using p::inner
26  }
```
Easier way to declare nested namespaces

C++14

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

C++17

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

Standard practice

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

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

int MyFirstClass::sum(int b) {
    int a = 0;  // do not do that!
    a += this->a;
    a += b;
    return a;
};
```
Method overloading

The rules in C++

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

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

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

int MyFirstClass::sum(int b, int c) {
    return a + b + c;
};
```
3 Object orientation (OO)
- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Operators
- Functors
First inheritance

```cpp
struct MyFirstClass {
    int a;
    void squareA() { a *= a; };
};

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

MySecondClass myObj2;
myObj2.a = 2;
myObj2.b = 5;
myObj2.squareA();
int i = myObj2.sum(); // i = 9
```
Managing access to class members

<table>
<thead>
<tr>
<th>public / private keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>private</strong> allows access only within the class</td>
</tr>
<tr>
<td><strong>public</strong> allows access from anywhere</td>
</tr>
</tbody>
</table>

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

**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

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

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

**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

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

This breaks MySecondClass !
```
Managing access to class members (2)

Solution is `protected` keyword

Gives access to classes inheriting from base class

```cpp
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

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 grandchild code:
- only public and protected members of public and protected parents are accessible
Managing inheritance privacy

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

MyFirstClass

private:
  int priv;
protected:
  int prot;
public:
  int pub;

MySecondClass

void funcSecond();

MyThirdClass

void funcThird();

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

MyFirstClass
---
private:
  int priv;
protected:
  int prot;
public:
  int pub;

MySecondClass
---
void funcSecond();

MyThirdClass
---
void funcThird();

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

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

void funcThird() {
    int a = priv; // Error
    int b = prot; // Error
    int c = pub; // Error
}

void extFunc(MyThirdClass t) {
    int a = t.priv; // Error
    int b = t.prot; // Error
    int c = t.pub; // Error
}
```
**Final class**

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

**Practically**

```cpp
struct Base final {
    ...
};
struct Derived : Base {    // compiler error
    ...
};
```
3 Object orientation (OO)

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

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 ~

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

// note special notation for
// initialization of members
MyFirstClass() : a(0) {}
MyFirstClass(int a_) : a(a_) {}
~MyFirstClass(){};


```cpp
class Vector {
public:
    Vector(int n);
    ~Vector();
    void setN(int n, int value);
    int getN(int n);

private:
    int len;
    int* data;
}

Vector::Vector(int n) : len(n) {
    data = (int*)malloc(n*sizeof(int));
}

Vector::~Vector() {
    free(data);
}
```

Class Constructors and Destructors

S. Ponce

C++98

C++ course

79 / 298
Constructor and inheritance

```cpp
struct MyFirstClass {
    MyFirstClass();
    MyFirstClass(int a);
}

struct MySecondClass : MyFirstClass {
    MySecondClass();
    MySecondClass(int b);
    MySecondClass(int a, int b);
}

MySecondClass() : MyFirstClass(), b(0) {};
MySecondClass(int b_) : MyFirstClass(), b(b_) {};
MySecondClass(int a_,
               int b_) : MyFirstClass(a_), b(b_) {};
```
Copy constructor

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
Copy constructor

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

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

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

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

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 Vector {
public:
    Vector(int n);
    Vector(const Vector &other);
    ~Vector();
    ...
};

Vector::Vector(int n) : len(n) {
    data = (int*)calloc(n, sizeof(int));
}

Vector::Vector(const Vector &other) : len(other.len) {
    data = (int*)malloc(len*sizeof(int));
    memcpy(data, other.data, len);
}

Vector::~Vector() { free(data); }
Default Constructor

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

Idea

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

Practically

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

**Idea**

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

**Practically**

```cpp
struct BaseClass {
    BaseClass(int value);
};

struct DerivedClass : BaseClass {
    using BaseClass::BaseClass;
};

DerivedClass a{5};
```
**Member initialization**

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

```cpp
struct BaseClass {
    int a{5};
    BaseClass() = default;
    BaseClass(int _a) : a(_a) {};
};

struct DerivedClass : BaseClass {
    int b{6};
    using BaseClass::BaseClass;
};

DerivedClass a{7}; // a = 7, b = 6
```
Calling constructors

After object declaration, arguments within `{}`

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

struct B {
    int a;
    float b;
};

A a{1, 2};  // A::A(int, int)
A a{1};    // A::A(int)
A a{};       // A::A()
A a;         // A::A()
A a = {1, 2}; // A::A(int, int)
B b = {1, 2.3}; // default constructor
```
Calling constructors the old way

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

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

struct B {
    int a;
    float b;
};

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

S. Ponce

C++ course
Calling constructors for arrays and vectors

10
11
12

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

list of items given within `{}`
### Calling constructors for arrays and vectors

#### C++11

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>int ip[3]{1,2,3};</td>
<td>OK</td>
</tr>
<tr>
<td>11</td>
<td>int* ip = new int[3]{1,2,3};</td>
<td>not allowed</td>
</tr>
<tr>
<td>12</td>
<td>std::vector&lt;int&gt; v{1,2,3};</td>
<td>not allowed</td>
</tr>
</tbody>
</table>

#### C++98 nightmare

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>int ip[3]{1,2,3};</td>
<td>OK</td>
</tr>
<tr>
<td>11</td>
<td>int* ip = new int[3]{1,2,3};</td>
<td>not allowed</td>
</tr>
<tr>
<td>12</td>
<td>std::vector&lt;int&gt; v{1,2,3};</td>
<td>not allowed</td>
</tr>
</tbody>
</table>
3 Object orientation (OO)
  - Objects and Classes
  - Inheritance
  - Constructors/destructors
  - Static members
  - Allocating objects
  - Advanced OO
  - Operators
  - Functors
Static members

Concept

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

```cpp
class Text {
   public:
      static std::string upper(std::string) {...};
   private:
      static int s_nbCallsToUpper;
};
int Text::s_nbCallsToUpper = 0;
std::string s = "my text";
std::string uppers = Text::upper("my text");
// now Text::s_nbCallsToUpper is 1
```
Allocating objects

3 Object orientation (OO)
- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Operators
- Functors
Process memory organization

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

```
heap
↓  ...  ↓
↓
↓
↓
stack
data segment
code segment
```
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

**On the stack**

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

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

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

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

On the heap

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

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

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

- arrays of objects are created by calling `new[]`
  default constructor is called for each object of the array
- arrays of objects are destructed by calling `delete[]`
  destructor is called for each object of the array

```cpp
{
    // default constructor called 10 times
    MyFirstClass *a = new MyFirstClass[10];
    ...
    delete[] a; // destructor called 10 times
}
```
3 Object orientation (OO)

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

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

```cpp
Polygon p;

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

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

**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

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

![Diagram showing the concept of virtual methods in C++](image)
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
```
override keyword

**Principle**

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

**Practically**

```cpp
struct Base {
    virtual void some_func(float);
};

struct Derived : Base {
    void some_func(float) override;
};
```
Why was `override` keyword introduced?

To detect the mistake in the following code:

```cpp
struct Base {
    virtual void some_func(float);
};

struct Derived : Base {
    void some_func(double); // oups!
};
```

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

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

**Practically**

```cpp
struct Base {
    virtual void some_func(float);
};

struct Intermediate : Base {
    void some_func(float) final;
};

struct Derived : Intermediate {
    void some_func(float) override;  // error
};
```
Pure Virtual methods

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

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

// ok, draw has been implemented
Polygon p;

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

```cpp
// Drawable
virtual void draw() = 0;

// Shape

// Polygon
void draw() override;
```
Pure Abstract Class aka Interface

Definition of pure abstract class

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

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

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

```cpp
struct BaseClass {
    int foo(std::string);
    int foo(int);
};

struct DerivedClass : BaseClass {
    using BaseClass::foo;
    int foo(std::string);
}

DerivedClass dc;
dc.foo(4); // error if no using
```
Polymorphism

Exercise Time

- go to code/polymorphism
- look at the code
- open test.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

Concept
- one class can inherit from multiple parents

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

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

**Problem**
- are the members of the grand parent replicated?
Virtual inheritance

Solution

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

```cpp
class TextBox : public virtual virtual Rectangle, Text
```

virtual

- Drawable
  - Rectangle
  - Text
  - TextBox

standard

- Drawable1
  - Rectangle
  - Text
  - TextBox

- Drawable2
  - Rectangle
  - Text
  - TextBox
Do not use multiple inheritance

- Except for inheriting from interfaces
- and for very seldom special cases
Multiple inheritance advice

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
Virtual inheritance

Exercise Time

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

Warning

in case of virtual inheritance it is the most derived class that calls the virtual base class’ constructor
Object orientation (OO)
- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Operators
- Functors
Operators’ example

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

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>As member</th>
<th>As non-member</th>
</tr>
</thead>
<tbody>
<tr>
<td>@a</td>
<td>(a).operator@()</td>
<td>operator@(a)</td>
</tr>
<tr>
<td>a@b</td>
<td>(a).operator@(b)</td>
<td>operator@(a,b)</td>
</tr>
<tr>
<td>a=b</td>
<td>(a).operator=(b)</td>
<td>cannot be non-member</td>
</tr>
<tr>
<td>a(b...)</td>
<td>(a).operator()(b...)</td>
<td>cannot be non-member</td>
</tr>
<tr>
<td>a[b]</td>
<td>(a).operator<a href="b"></a></td>
<td>cannot be non-member</td>
</tr>
<tr>
<td>a-&gt;</td>
<td>(a).operator-&gt;()</td>
<td>cannot be non-member</td>
</tr>
<tr>
<td>a@</td>
<td>(a).operator@()</td>
<td>operator@(a,0)</td>
</tr>
</tbody>
</table>
Why to have non-member operators?

Symmetry

```c++
struct Complex {
    float m_real, m_imaginary;
    Complex operator+(float other) {
        return Complex(m_real + other, m_imaginary);
    }
}

Complex c1(2, 3);
Complex c2 = c1 + 4;  // ok
Complex c3 = 4 + c1;  // not ok !!
**Symmetry**

```cpp
struct Complex {
    float m_real, m_imaginary;
    Complex operator+(float other) {
        return Complex(m_real + other, m_imaginary);
    }
}

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

Why to have non-member operators?

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

Extending existing classes

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

 std::ostream& operator<<(std::ostream& os, const Complex& obj) {
     os << "(" << obj.m_real << ", " << obj.m_imaginary << ")";
     return os;
 }

 Complex c1(2, 3);
 std::cout << c1 << std::endl;
```
Object orientation (OO)
- Objects and Classes
- Inheritance
- Constructors/destructors
- Static members
- Allocating objects
- Advanced OO
- Operators
- Functors
Functors

Concept

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

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

Adder inc1(1), inc10(10);
int i = 3;
int j = inc1(i);  // 4
int k = inc10(i);  // 13
```
Typical usage

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

```cpp
struct BinaryFunction {
    virtual double operator()(double a, double b) = 0;
};

struct Add : public BinaryFunction {
    double operator()(double a, double b) { return a + b; }
};

double binary_op(double a, double b, BinaryFunction &func) {
    return func(a, b);
}

Add addfunc;

double c = binary_op(a, b, addfunc);
```
Core modern C++

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

- Constness
- Constant Expressions
- Exceptions
- Move semantic
- Copy elision
- Templates
- The STL
- More STL
- Lambdas
- pointers and RAII
The *const* keyword

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

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

// error : i is constant
i = 5;

// also ok, when nothing on the left,
// const applies to element on the right
const int j = 6;
```
// pointer to a constant integer
int a = 1, b = 2;
int const *i = &a;
*i = 5; // error, int is const
i = &b; // ok, pointer is not const

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

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

```cpp
struct Exemple {
    void foo() const {
        m_member = 0; // Error : function is constant
    }
    int m_member;
};
```
Constness is part of the type

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

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

int a = 0;
const int b = 0;

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

- go to code/constness
- open test.cpp
- try pointer to constant
- try constant pointer
- try constant pointer to constant
- try constant arguments of functions
- try constant method in a class
4 Core modern C++

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

Reason of being

- compute constant expressions at compile time
- even if non trivial
Generalized Constant Expressions

Reason of being

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

Example

```cpp
constexpr int f(int x) {
    return x > 1 ? x * f(x - 1) : 1;
}

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
Real life example

```cpp
constexpr float toSI(const float v, const char unit) {
    switch (unit) {
        case 'k': return 1000*v;
        case 'm': return 0.001*v;
        case 'y': return 0.9144*v;
        case 'i': return 0.0254*v;
        ...
        default: return v;
    }
}

constexpr float fromSI(const float v, const char unit) {
    switch (unit) {
        case 'k': return 0.001*v;
        ...
    }
}
```
class DimLength {
    const float m_value;

public:
    constexpr DimLength(const float v, const char unit):
        m_value(convertToSI(v, unit)) {
    }

    constexpr float get(const char unit) const {
        return convertFromSI(m_value, unit);
    }
};

constexpr DimLength km(1, 'k');
constexpr float km_y = km.get('y');
constexpr float km_i = km.get('i');
std::cout << "1 km = " << km_y << " yards\n" << " = " << km_i << " inches\n";
Core modern C++

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

```c++
try {
  if (0 == name) {
    throw std::string("Expected non empty name");
  }
  printf("%s\n", name);
} catch (std::string e) {
  printf("empty name found\n");
}
```
### 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

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

```c++
int func(int a) throw(int) {
    if (0 == a) {
        throw 2; // ok, goes out
    } else {
        throw "hello"; // std::unexpected called
    }
}
```
Controlling exceptions

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

```cpp
int func(int a) throw(int) {
    if (0 == a) {
        throw 2;  // ok, goes out
    } else {
        throw "hello";  // std::unexpected called
    }
}
```
Controlling exceptions

**Good to know**

- The check in 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()`
Controlling exceptions

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()`

```cpp
def int func(int a) {
   // any exception can go out
}
def int otherfunc(int a) throw() {
   // no exception can go out
}
```
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
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
throw is dead

- throw statements are deprecated
- even throw() (no exceptions)
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
3 uses of noexcept

- standalone
  ```cpp
  int f() noexcept;
  ```
- as an expression saying whether exceptions can be sent
  ```cpp
  int f() noexcept(sizeof(long) == 8);
  ```
- as an operator to know whether a function launches exceptions
  ```cpp
  template <class T> void foo()
      noexcept(noexcept(T())) {}
  ```
4 Core modern C++

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

Non efficient code

```cpp
void swap(std::vector<int> &a,
          std::vector<int> &b) {
    T c = a;
    a = b;
    b = c;
}

std::vector<int> v, w;
for (int i = 0; i < 10000; i++) v.push_back(i);
for (int i = 0; i < 10000; i++) w.push_back(i);
swap(v, w);
```
Move semantics: the problem

Non efficient code

```cpp
void swap(std::vector<int> &a,
          std::vector<int> &b) {
    T c = a;
    a = b;
    b = c;
}
std::vector<int> v, w;
for (int i = 0; i < 10000; i++) v.push_back(i);
for (int i = 0; i < 10000; i++) w.push_back(i);
swap(v, w);
```

What really happens during swap

- 10k allocations + 10k releases
- 30k copies
Move semantics: the problem

Dedicated efficient code

```cpp
std::vector<int> v, w;

for (int i = 0; i < 10000; i++) v.push_back(i);
for (int i = 0; i < 10000; i++) w.push_back(i);

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

Dedicated efficient code

```cpp
std::vector<int> v, w;
for (int i = 0; i < 10000; i++) v.push_back(i);
for (int i = 0; i < 10000; i++) w.push_back(i);
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

Another non efficient code

```cpp
std::vector<int> vrandom(unsigned int n) {
    std::vector<int> result;
    for (int i = 0; i < n; i++) {
        result.push_back(rand());
    }
    return result;
}

std::vector<int> v = vrandom(10000);
```
Move semantics: the problem

Another non efficient code

```cpp
std::vector<int> vrandon(unsigned int n) {
    std::vector<int> result;
    for (int i = 0; i < n; i++) {
        result.push_back(rand());
    }
    return result;
}
std::vector<int> v = vrandon(10000);
```

What really happens during assignment

- 10k allocations + 10k releases
- 10k copies
Move semantics: the problem

Dedicated efficient way

```cpp
void vrandom(unsigned int n, std::vector<int> &v) {
    for (int i = 0; i < n; i++) {
        v.push_back(rand());
    }
}

std::vector<int> v;
vrandom(10000, v);
```
Move semantics: the problem

Dedicated efficient way

```cpp
void vrrandom(unsigned int n, std::vector<int> &v) {
    for (int i = 0; i < n; i++) {
        v.push_back(rand());
    }
}

std::vector<int> v;
vrandom(10000, v);
```

The ideal situation

Have a way to express that we move the vector’s content
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)
The idea

- A new type of reference: rvalue references
  - Used for move sematics
  - Denoted by `&&`
- 2 new members in every class, with move semantics:
  - 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`
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
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

```cpp
T a;
T b = a;   // 1. Copy assign
T c = T(2); // 2. Move assign
T d = func(); // 3. Move assign
```
In some cases, you want to force a move

```cpp
void swap(T &a, T &b) {
    T c = a;    // copy
    a = b;      // copy
    b = c;      // copy
}
```
Move semantics

In some cases, you want to force a move

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

There are mainly two ways

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

```cpp
T a;
T b = a;     // Copy assign
T c = static_cast<T&&>(a);  // Move assign
T d = std::move(a);         // Move assign
```
Move semantics: the easy way

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

Practically

```cpp
class Movable {
    Movable();
    Movable(Movable &&other) :
        Movable() { // constructor delegation
            swap(*this, other);
        }
    Movable& operator=(Movable other) { // by value
        swap(*this, other);
        return *this;
    }
    friend void swap(Movable &a, Movable &b);
};
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 RAII
What is copy elision

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

From C++17 on

The elision is guaranteed.
Guaranteed copy elision

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

One case where the guarantee is needed

```cpp
struct Foo {
    Foo() { ... }    // Calls C++14 copy constructor
    Foo(const Foo &) = delete;
    Foo(Foo &&) = delete;
};

Foo f() {
    return Foo();
}

int main() {
    Foo foo = f();
}
```
Templates

4 Core modern C++

- Constness
- Constant Expressions
- Exceptions
- Move semantic
- Copy elision

- Templates
- The STL
- More STL
- Lambdas
- pointers and RAII
Templates

Concept

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

```
template<class T>
const T & Max(const T &A, const T &B) {
    return A > B ? A : B;
}

template<class T>
struct Vector {
    int m_len;
    T* m_data;
}
```
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

```cpp
1 template<class T>
2 T func(T a) {
3     return a;
4 }

1 int func(int a) {
2     return a;
3 }

1 double func(double a) {
2     return a;
3 }
```
Arguments

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

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

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

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

templates can be specialized for given values of their parameter

```c++
1 template<unsigned int N> struct Polygon {
2   Polygon(float radius);
3   float perimeter();
4   float m_radius;
5
6 `template<>
7 struct Polygon<6> {
8   Polygon(float radius);
9   float perimeter() {return 6*m_radius;};
10  float m_radius;
11 };
12```
The full power of templates

Exercise Time

- go to code/template
- look at the OrderedVector code
- compile and run test.cpp. See the ordering
- modify test.cpp and reuse OrderedVector with Complex
- improve OrderedVector to template the ordering
- test reverse ordering of strings (from the last letter)
- test manhattan order with complex type
- check the implementation of Complex
- try ordering complex of complex
Core modern C++

- Constness
- Constant Expressions
- Exceptions
- Move semantic
- Copy elision
- Templates

The STL

- More STL
- Lambdas
- Pointers and RAII
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

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

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

std::transform(vi.begin(), vi.end(), // range1
               vi.begin(), // start range2
               vr.begin(), // start result
               std::multiplies<int>()); // function

for(auto n : vr) {
    std::cout << n << " ";
}
```
STL’s concepts

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
STL’s concepts

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
#include <vector>
#include <algorithm>

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

std::transform(vi.begin(), vi.end(), // range1
    vi.begin(), // start range2
    vr.begin(), // start result
    std::multiplies<int>()); // function

for(auto n : vr) {
    std::cout << n << " ";
};
#include <vector>

#include <algorithm>

std::vector<int> vi, vr(3);
vi.push_back(5); vi.push_back(3); vi.push_back(4);

std::transform(vi.begin(), vi.end(),
    vi.begin(),
    vr.begin(),
    std::multiplies<int>());

for(std::vector<int>::iterator it = vr.begin();
    it != vr.end();
    it++) {
    std::cout << *it << " ";
};
// Finds the first element in a list that lies in
// the range from 1 to 10.
list<int> L;
...

list<int>::iterator in_range =
    find_if(L.begin(), L.end(),
        compose2(logical_and<bool>(),
            bind2nd(greater_equal<int>(), 1),
            bind2nd(less_equal<int>(), 10)));

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

S. Ponce C++ course
Welcome to lego programming!
Exercise Time

- go to code/stl
- look at the non STL code in test.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 test.cpp and complete the “translation” to STL
- see how easy it is to reuse the code with complex numbers
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

Old way

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

New way

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

STL way

```cpp
std::vector<int> v = ...;
int sum = std::accumulate(v.begin(), v.end(), 0);
```
Loops and auto keyword with the STL

Old way

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

New way

```cpp
std::vector<int> v = ...;
int sum = 0;
for (auto a : v) { sum += a; }
```
Loops and auto keyword with the STL

Old way

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

New way

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

STL way

```cpp
std::vector<int> v = ...;
int sum = std::accumulate(v.begin(), v.end(), 0);
```
4 Core modern C++

- Constness
- Constant Expressions
- Exceptions
- Move semantic
- Copy elision
- Templates
- The STL
- More STL
- Lambdas
- pointers and RAII
Some new STL types

**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
The problem in C++98

STL containers and arrays have different syntax for loop

```cpp
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

```cpp
1 for(auto & element : v) {...}
2 for(auto & element : a) {...}
```
non-member begin and end

The problem in C++98

STL containers and arrays have different syntax for loop

```cpp
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

```cpp
1 for(auto it = begin(v); it != end(v); it++) {...}
2 for(auto i = begin(a); i != end(a); i++) {...}
```
The problem in C++98

STL containers and arrays have different syntax for loop

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

A new syntax

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

Allowing the best syntax

```cpp
for(auto & element : v) {...}
for(auto & element : a) {...}
```
Structured Binding Declarations

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

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

```cpp
void foo(std::tuple<int, double, long> tuple) {
    auto [ a, b, c ] = tuple;
}
```
4 Core modern C++

- Constness
- Constant Expressions
- Exceptions
- Move semantic
- Copy elision
- Templates
- The STL
- More STL
- Lambdas
- Pointers and RAII
A new way to specify function’s return type

```cpp
ReturnType fn_name(ArgType1, ArgType2);  //old
auto fn_name(ArgType1, ArgType2) -> ReturnType;
```
A new way to specify function's return type

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

**Advantages**

- Allows to simplify inner type definition

```cpp
class TheClass {
  using inner_type = int;
  inner_type func();
}
TheClass::inner_type TheClass::func() {...}
auto TheClass::func() -> inner_type {...}
```

- will be used for lambdas
Lambdas

Definition

A lambda is a function with no name.
**Definition**

A lambda is a function with no name.

**Python example**

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

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

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

Simplified syntax

```cpp
[](args) -> type {
    code;
}

The type specification is optional
```

Usage example

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

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

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

First attempt in C++

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

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

Error

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

Python code

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

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

Example

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

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

Example

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

Code example

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

Code example

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

Error

```cpp
error: assignment of read-only variable 'sum'
    [sum](int x) { sum += x; });
```
Default capture is by value

**Code example**

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

**Error**

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

**Explanation**

By default, variables are captured by value
Capture by reference

Simple example

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

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

Simple example

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

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

Mixed case

One can of course mix values and references

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

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

by value

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

by reference

```cpp
[&](...) { ... };
```
Capture all

by value

\[=\] (\ldots) \{ \ldots \};

by reference

[&] (\ldots) \{ \ldots \};

exceptions

[&, b] (\ldots) \{ \ldots \};

[=, &b] (\ldots) \{ \ldots \};
Closures

Example

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

How it works

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

```cpp
struct Incrementer {
    int m_inc;
    Incrementer(int inc) : m_inc(inc) {}
    int operator()(int value) {
        return value + m_inc;
    }
};

std::vector<int> v{1, 2, 3};
std::transform(begin(v), end(v), begin(v),
    Incrementer(1));
for (auto a : v) std::cout << a << " ";
```
With lambdas

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

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

Conclusion

Use the STL!
Lambdas

Exercise Time

- go to code/lambdas
- look at the code (it’s the solution to the stl exercise)
- use lambdas to simplify it
4 Core modern C++
- Constness
- Constant Expressions
- Exceptions
- Move semantic
- Copy elision
- Templates
- The STL
- More STL
- Lambdas

- pointers and RAII
Pointers: why they are error prone?

They need initialization

```cpp
char *s;
try {
    foo();
    s = (char*) malloc(...);
    strncpy(s, ...);
} catch (...) { ... }
bar(s);
```
Pointers: why they are error prone?

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?

They need initialization

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

They need to be released

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

Seg Fault

Seg Fault

char *s;
try {
    foo();
    s = (char*) malloc(...);
}
```
Pointers: why they are error prone?

They need initialization

```c++
char *s;
try {
    foo();
    s = (char*) malloc(...);
```

Seg Fault

They need to be released

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

Memory leak

S. Ponce
C++ course
Pointers: why they are error prone?

They need initialization

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

They need to be released

```
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?

They need initialization

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

They need to be released

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

They need clear ownership

```cpp
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

For example with a file

```cpp
try {
    FILE *handle = std::fopen(path, "w+");
    if (0 == handle) { throw ... }
    if (std::fputs(str, handle) == EOF) {
        throw ...
    }
    fclose(handle);
} catch (...) { ... }
```
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
RAII in practice

File class

class File {

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

private:
    FILE* m_file_handle;
};
void log_function() {
    // file opening, aka resource acquisition
    File logfile("logfile.txt") ;

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

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

an RAII pointer

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

Usage

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

**an RAII pointer**

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

**Usage**

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

What do you expect?
Foo *p = new Foo{};  // allocation
std::unique_ptr<Foo> uptr(p);

void f(std::unique_ptr<Foo> ptr);
f(uptr); // transfer of ownership

What do you expect?

Compilation Error

test.cpp:15:5: error: call to deleted constructor
of 'std::unique_ptr<Foo>'
f(uptr);
~~~

/usr/include/c++/4.9/bits/unique_ptr.h:356:7: note: 'unique_ptr' has been explicitly marked deleted here
unique_ptr(const unique_ptr&) = delete; ~
std::make_unique

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

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

**make_unique usage**

```cpp
// allocation of one Foo object,
// calling constructor with one argument
auto a = std::make_unique<Foo>(memberValue);
std::cout << a.get() << " points to "
    << a->someMember << std::endl;

// allocation of an array of Foos
// calling default constructor
auto b = std::make_unique<Foo[]>(10);

// deallocations
```
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
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

```cpp
unique_ptr<T> producer();
void observer(T*);
void consumer(unique_ptr<T>);

unique_ptr<T> pt{producer()};
observer(pt.get()); // Keep ownership
consumer(std::move(pt)); // Transfer ownership
```
unique_ptr usage summary

It's about lifetime management

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

```cpp
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
```
Expert C++

1. History and goals
2. Language basics
3. Object orientation (OO)
4. Core modern C++

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

6. Useful tools
7. Concurrency
8. C++ and python
Variadic templates

5 Expert C++

- Variadic templates
- Perfect forwarding
- SFINAE
Basic variadic template

The idea

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

Practically

```cpp
template<typename T, typename... Args>
T adder(T first, Args... args) {
    return first + adder(args...);
}

template<typename T>
T adder(T v) {
    return v;
}

long sum = adder(1, 2, 3, 8, 7);
```
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
### The tuple example

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

template <class T, class... Ts>
struct tuple<T, Ts...> : tuple<Ts...> {
    tuple(T t, Ts... ts) :
        tuple<Ts...>(ts...), m_tail(t) {}
    T m_tail;
};

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

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

Trying to write a generic wrapper function

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

Example usage:
- `emplace_back`
- `make_unique`
Why is it not so simple?

```cpp
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?

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

```cpp
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

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

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

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

... 
```

Exploding complexity

$3^n$ complexity

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

They are forbidden in C++
But still may happen

```cpp
template <typename T>
void foo(T t) {
    T& k = t;
    ...
}
int ii = 4;
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
& & & & & & & \rightarrow &
& & && \rightarrow &&
rvalue in type-deducing context

```cpp
template <class T>
void func(T&& t) {} // In this context, && is not an rvalue
// It means that the T 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

func(4); // rvalue -> T is int
double d = 3.14;
func(d); // lvalue -> T is double&
float f() {...}
func(f()); // rvalue -> T is float
int foo(int i) {
    func(i); // lvalue -> T is int&
}
```
Some template trickery removing reference from a type

```cpp
template< class T >
struct remove_reference
{
    typedef T type;
};

template< class T >
struct remove_reference<T&>
{
    typedef T type;
};

template< class T >
struct remove_reference<T&&>
{
    typedef T type;
}
```

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

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

**How it works**

- if \(T\) is int, it returns int &&
- if \(T\) is int&, it returns int& && ie int&
Perfect forwarding

Putting it all together

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

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&
Real life example

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

5. Expert C++
- Variadic templates
- Perfect forwarding
- SFINAE
Substitution Failure Is Not An Error

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

```cpp
template <typename T>
void f(const T& t,
       typename T::iterator* it = nullptr) {}

void f(...) {} // ``sink'' function

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

S. Ponce
C++ course
The main idea

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

Example

```cpp
struct A { double x; };
A a;
dcltype(a.x) y; // double
dcltype((a.x)) z = y; // double& (lvalue)

template<typename T, typename U>
auto add(T t, U u) -> decltype(t + u);
// 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

```cpp
struct Default {
    int foo() const { return 1; }
};

struct NonDefault {
    NonDefault(int i) {}  // int foo()
    int foo() const { return 1; }
};

decltype(Default().foo()) n1 = 1;  // int
decltype(NonDefault().foo()) n2 = n1;  // error
decltype(std::declval<NonDefault>().foo()) n2 = n1;
```
true_type and false_type

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

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

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

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

std::cout << hasFoo<MyType>::value << std::endl;
```
Lot’s of useful stuff there

### enable_if

```cpp
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> { typedef T type; };
```

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

### is_*(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

```cpp
constexpr struct deref_t {
    template <typename In,
        typename = typename std::enable_if<!std::is_pointer<In>::value>::type>
    In& operator()( In& in ) const { return in; }

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

} deref {};
```
The tuple get method

template <class T, class... Ts>
struct elem_type_holder<0, tuple<T, Ts...>> {
    typedef T type;
};

template <size_t k, class T, class... Ts>
struct elem_type_holder<k, tuple<T, Ts...>> {
    typedef typename elem_type_holder
    <k - 1, tuple<Ts...>>::type type;
};
The tuple get method

template <size_t k, class... Ts>
typename std::enable_if<k == 0,
    typename elem_type_holder
    <0, tuple<Ts...>::type&>::type
get(tuple<Ts...>& t) {
    return t.m_tail;
}

template <size_t k, class T, class... Ts>
typename std::enable_if<k != 0,
    typename elem_type_holder
    <k-1, tuple<Ts...>::type&>::type
get(tuple<T, Ts...>& t) {
    tuple<Ts...>& base = t;
    return get<k - 1>(base);
}
Useful tools

1. History and goals
2. Language basics
3. Object orientation (OO)
4. Core modern C++
5. Expert C++

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

7. Concurrency

8. C++ and python
Useful tools

- **C++ editor**
- Code management
- The Compiling Chain
- Debugging
- The Valgrind family
- Static code analysis
**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
- Eclipse: similar, but open source and portable
- NetBeans: similar again, also 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
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 best choice. Fast, light, easy to use
- **mercurial** the main alternative
- **Bazzar** 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 "commiting first 2 files"
[master (root-commit) c481716] commiting first 2 files
...

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

# git diff f24a6ce bb54d15
Useful tools
- C++ editor
- Code management
- The Compiling Chain
- Debugging
- The Valgrind family
- Static code analysis
The compiling chain

Source code

cpp, .hpp

Preprocessor

.cpp

Compiler

g++ -c, gcc -c

Linker

.so, exe

ld, gcc, g++

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
  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
```
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
- see link statement using g++ -v
- see library dependencies with ldd
- look at the Makefile
- try make clean; make
Useful tools

- C++ editor
- Code management
- The Compiling Chain
- Debugging
- The Valgrind family
- Static code analysis
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
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)
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
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
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
- 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!
Exercise Time

- go to code/helgrind
- compile, run
- check it with valgrind. See strange behavior but no explanation
- check it with valgrind --tool=helgrind
- understand issue and fix
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)
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 open source, 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. Language 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
7 Concurrency

- Threads and async
- Mutexes
Basic concurrency

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

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

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

Can take a function and its arguments

```cpp
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

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

Can take any function like object

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

Concept

- separation of the specification of what should be done and the retrieval of the results
- “start working on this, and ping me when it’s ready”
Basic asynchronicity

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
Basic asynchronicity

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

```cpp
int computeSth() { ... } 
auto res = std::async(std::launch::async, computeSth); 
auto res2 = std::async(std::launch::deferred, computeSth); 
```
Fine grained control on asynchronous 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
Fine grained control on asynchronous 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

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

- Threads and async
- Mutexes
Example code

```cpp
int a = 0;

void inc() { a++; };

void inc100() {
    for (int i=0; i < 100; i++) inc();
};

int main() {
    std::thread t1(inc100);
    std::thread t2(inc100);
    for (auto t: {&t1,&t2}) t->join();
    std::cout << a << std::endl;
}
```
### Example code

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

What do you expect? Try it in code/race

Anything between 100 and 200!!!

S. Ponce
Example code

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

What do you expect? Try it in code/race

Anything between 100 and 200 !!!
Atomicity

**Definition (wikipedia)**

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

**Practically**

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

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?

S. Ponce C++ course
Atomicity

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. \texttt{eax = a}  // memory to register copy
2. \texttt{increase eax}  // increase (atomic CPU instruction)
3. \texttt{a = eax}  // copy back to memory

For 2 threads

Thread 1: \texttt{eax}  
Memory: \texttt{a}  
Thread 2: \texttt{eax}

read 0  
\texttt{incr}  
write 1  
\texttt{read}  
\texttt{0}  
\texttt{incr}  
write 1  
time
Mutexes

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

```cpp
int a = 0;
std::mutex m;
void inc() {
    std::lock_guard<std::mutex> guard(m);
    a++;
}
```
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

Scenario

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

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

**Sequence diagram**

Thread 1:
- Lock
- Lock (block)

Mutex A:
- Lock

Mutex B:
- Lock
- Lock (block)

Thread 2:
- Lock
- Lock (block)

Time
How to avoid deadlocks

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
Condition variables

How to express thread dependencies

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

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
Example code

```cpp
int value = -1;
std::mutex mutex;
std::condition cond;
auto t = std::thread([] () {
    value = ... long process ...;
    cond.notify_all();
});
auto t = std::thread([] () {
    std::unique_lock<std::mutex> lock{mutex};
    cond.wait(lock, [] { return value > 0; });
    ... use value ...
});
{ std::unique_lock<std::mutex> lock{mutex};
    cond.wait(lock, [] { return value > 0; });
    std::cout << value << std::endl; }
```
C++ and python

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

C++ and python
- Writing a module
- Marrying C++ and C
- The ctypes module
How to build a python module around C++ code

C++ code: mandel.hpp

```cpp
int mandel(const Complex &a);
```
Basic Module(1) : wrap your method

mandelModule.cpp

```cpp
#include <Python.h>
#include "mandel.hpp"

static PyObject * mandel_wrapper(PyObject * self, PyObject * args) {
    // Parse Input
    float r, i;
    if (!PyArg_ParseTuple(args, "ff", &r, &i))
        return NULL;
    // Call C++ function
    int result = mandel(Complex(r, i));
    // Build returned objects
    return Py_BuildValue("i", result);
}
```

Basic Module(2): create the python module

```cpp
// declare the modules' methods
static PyMethodDef MandelMethods[] = {
    {"mandel", mandel_wrapper, METH_VARARGS,
     "computes nb of iterations for mandelbrot set"},
    {NULL, NULL, 0, NULL}
};

// initialize the module
PyMODINIT_FUNC initmandel(void) {
    (void) Py_InitModule("mandel", MandelMethods);
}
```
Basic Module(3) : use it

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

Marrying C++ and C

**C++ and python**
- Writing a module
- Marrying C++ and C
- The ctypes module
A question of mangling

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

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

Source: file.c

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

Binary symbols: file.o

```
# nm file.o
0000000000000001a T square
0000000000000000 T sum
```
**C++ manangling**

**Source: file.cpp**

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

**Binary symbols: file.o**

```
# nm file.o
0000000000000000 T _Z3sumff
0000000000000002a T _Z6squaref
0000000000000001a T _Z6squarei
```
Forcing C mangling in C++

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

These functions will use C mangling:

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

You can now call these C++ functions from C code
Forcing C mangling in C++

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

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

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

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

From the documentation

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

### C++ code: mandel.hpp

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

### “C” code: mandel_cwrapper.hpp

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

### calling the mandel library

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

- go to code/python
- look at the original python code `mandel.py`
- time it
- 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`
This is the end

Questions ?

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