Introduction to C++

Why C++?

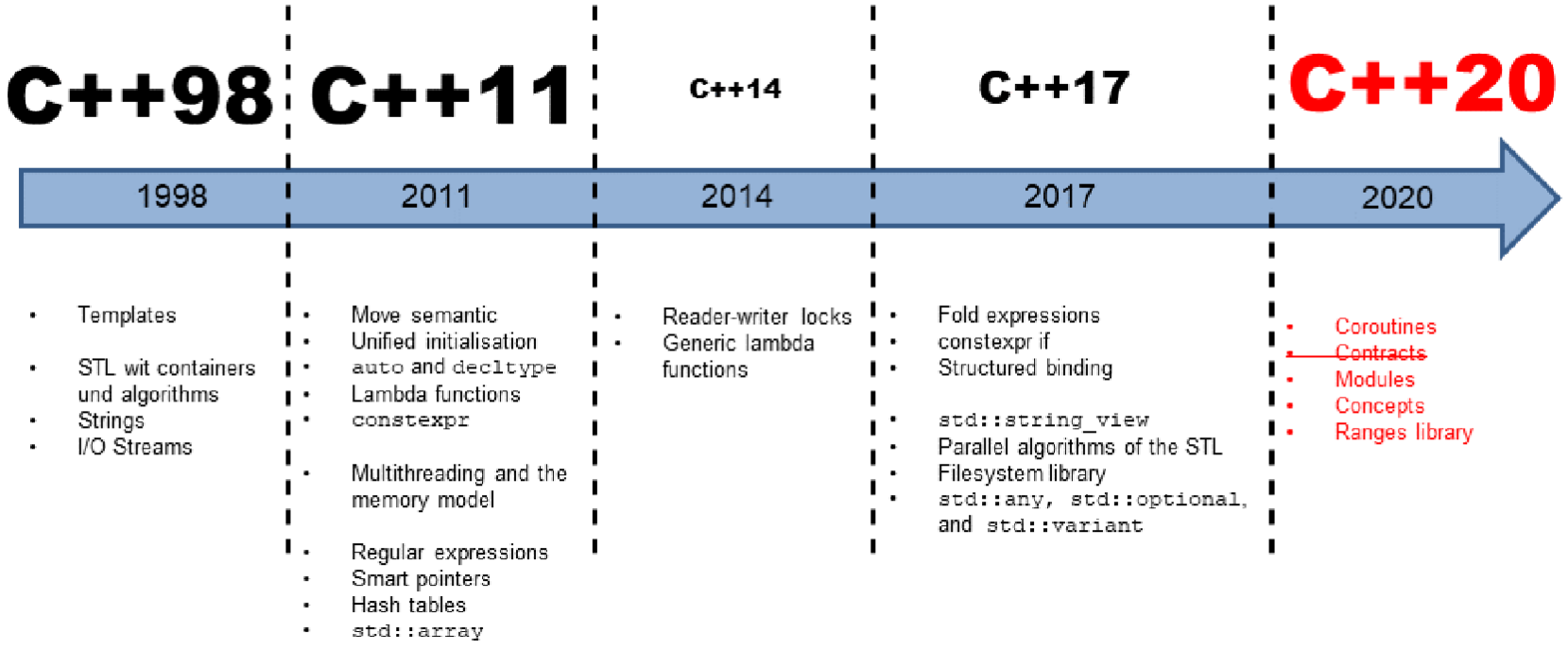
- Modern and efficient language
- Used by major companies, for all relevant Browsers and many programs that need to run efficiently

Table 4. Normalized global results for Energy, Time, and Memory

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

History of C++

- Developed by Bjarne Stroustrup in 1983
- Aimed for fast, simple to use, cross-platform language



Hello World!

- Simplest program:

```cpp
#include <iostream>

int main() {
  // Is this your first C++ program?
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

Comments by // or multi-line comments with /* <comment> */

Return value:
0 means OK

Connects to std output.
Also:
std::cin for input
std::cerr for errors

Header. Things you need but haven‘t written yourself go here. Distinguish between:
#include <file> for system files
#include “file” for local files

Compilation

- Unlike e.g. Python, C++ is a compiled language
- Code needs to be translated in machine-readable code
- We will usually use g++
- Example: g++ -o hello_world hello_world.cpp
- Will create a binary called hello_world that can be executed

General Concepts

Functions

- Declaration and definition can be separated
- Best Practice: Declare all funtions in a **header file** (*.h or *.hpp). The definition goes into *.cpp files
- Overloading: Define same function for different inputs

```
// some_file.hpp
Type SomeFunc(... args...);

// some_file.cpp
#include "some_file.hpp"
Type SomeFunc(... args...) {}  // implementation

// program.cpp
#include "some_file.hpp"
int main() {
  SomeFunc(/* args */);
  return 0;
}
```

Variables

- Variables always have a certain type, e.g. int or string
- You can also define a variable as *auto* to have the type automatically assigned according to the assigned value
- Pay attention to the scopes of the code:

```
1 int my_variable;  // "my_variable" is the name
2
3 {                 //{<-this defines a new scope
4   float var_fl;   // var_f is valid within this scope
5 }                 //}<-this defines end of the scope
6
7 var_fl;           // Error, var_fl outside its scope
8
9 int var_fl;       // Valid, var_fl not declared
```

All the Types

- Several types available out-of-the-box:

```
1  bool this_is_fun = true;     // Boolean: true or false.
2  char carret_return = '\n';   // Single character.
3  int meaning_of_life = 42;    // Integer number.
4  short smaller_int = 42;      // Short number.
5  long bigger_int = 42;        // Long number.
6  float fraction = 0.01f;      // Single precision float.
7  double precise_num = 0.01;   // Double precision float.
8  auto some_int = 13;          // Automatic type [int].
9  auto some_float = 13.0f;     // Automatic type [float].
10 auto some_double = 13.0;     // Automatic type [double].
```

If Statements

- Execute code conditionally, depending on boolean expression
- Can be extended with *else* statements

```
if (STATEMENT) {
  // This is executed if STATEMENT == true
} else if (OTHER_STATEMENT) {
  // This is executed if:
  // (STATEMENT == false) && (OTHER_STATEMENT == true)
} else {
  // This is executed if neither is true
}
```

- Ternary form: (STATEMENT) ? <if true> : <else>

Switch Statement

- Similar to if, but for multiple *cases*
- Execution will start at matching case and will **fall through**
- Use *break* to stop after one case

```
switch(STATEMENT) {
  case CONST_1:
    // This runs if STATEMENT == CONST_1.
    break;
  case CONST_2:
    // This runs if STATEMENT == CONST_2.
    break;
  default:
    // This runs if no other options worked.
}
```

While Loop

- Repeat certain actions, while a statement is true
- Variations: do-while and while-do
- Easy to trap yourself in an endless loop!

```
1 bool condition = true;
2 while (condition) {
3   condition = /* Magically update condition. */
4 }
```

- Use *break* to stop and *continue* to go to the next iteration

For Loop

- Repeat until some condition is met

```
for (INITIAL_CONDITION; END_CONDITION; INCREMENT) {
  // This happens until END_CONDITION == false
}
```

- Can also be used to iterate through some range

```
for (const auto& value : container) {
  // This happens for each value in the container.
}
```

I/O Streams

- Handle stdin, stdout, stderr
- Part of *#include <iostream>*

```
#include <iostream>
int main() {
  int some_number;
  std::cout << "please input any number" << std::endl;
  std::cin >> some_number;
  std::cout << "number = " << some_number << std::endl;
  std::cerr << "boring error message" << std::endl;
  return 0;
}
```

String Streams

- Combines non-strings into strings and vice versa

```
#include <iomanip>
#include <iostream>
#include <sstream>
using namespace std;

int main() {
  // Combine variables into a stringstream.
  stringstream filename{"00205.txt"};

  // Create variables to split the string stream
  int num = 0;
  string ext;

  // Split the string stream using simple syntax
  filename >> num >> ext;

  // Tell your friends
  cout << "Number    is: " << num << endl;
  cout << "Extension is: " << ext << endl;
  return 0;
}
```

Arguments

- Arguments passed to main function
- Argc is number of arg's, argv are the arg's as strings

```
#include <iostream>
#include <string>
using std::cout;
using std::endl;

int main(int argc, char const *argv[]) {
  // Print how many parameteres we received
  cout << "Got " << argc << " params\n";

  // First program argument is always the program name
  cout << "Program: " << argv[0] << endl;

  for (int i = 1; i < argc; ++i) {  // from 1 on
    cout << "Param: " << argv[i] << endl;
  }
  return 0;
}
```

Classes

Std::array

- Collection of items of same type
- Access via arr[i], starting at 0

```cpp
#include <array>
#include <iostream>
using std::cout;
using std::endl;

int main() {
  std::array<float, 3> data{10.0F, 100.0F, 1000.0F};

  for (const auto& elem : data) {
    cout << elem << endl;
  }

  cout << std::boolalpha;
  cout << "Array empty: " << data.empty() << endl;
  cout << "Array size : " << data.size() << endl;
}
```

Std::vector

- Part of #include <vector>
- Dynamic table, can be extended, shortened during runtime
- Fast and flexible

```cpp
#include <iostream>
#include <string>
#include <vector>
using std::cout;
using std::endl;

int main() {
  std::vector<int> numbers = {1, 2, 3};
  std::vector<std::string> names = {"Nacho", "Cyrill"};

  names.emplace_back("Roberto");

  cout << "First name : " << names.front() << endl;
  cout << "Last number: " << numbers.back() << endl;
  return 0;
}
```

Std::map

- List of pairs of two variable types, e.g. <int, string>
- Pair consituents are called <key, value>
- Keys have to be unique within a map

```
1 std::map<KeyT, ValueT> m{{key1, value1}, {..}};
```

Iterators

- Tie std library containers, e.g. vectors, to the data they contain
- Always point to specific element inside
- Can be incremented by itr++
- Useful for range-based for loops

```cpp
int main() {
  vector<double> x{1, 2, 3};
  for (auto it = x.begin(); it != x.end(); ++it) {
    cout << *it << endl;
  }
  // Map iterators
  map<int, string> m = {{1, "hello"}, {2, "world"}};
  map<int, string>::iterator m_it = m.find(1);
  cout << m_it->first << ":" << m_it->second << endl;

  auto m_it2 = m.find(1);  // same thing
  cout << m_it2->first << ":" << m_it2->second << endl;

  if (m.find(3) == m.end()) {
    cout << "Key 3 was not found\n";
  }
  return 0;
}
```

Class Basics - I

- Classes are created to “invent” new variable types
- Each class represents a concept, e.g. vehicles
- Potential connection between classes, e.g. car, truck, boat are conceptually all vehicles
- Helpful to structure code, as it mimics the way people think

Class Basics - II

- Classes consist of members, e.g. variables and functions
- Members can be accessed via “.” (objects) or “→” (pointers)
- Operators can be defined for each class
- Public members are interface to the rest of the code, private members are internal
- Alternative: *struct*, by default everything is *public*

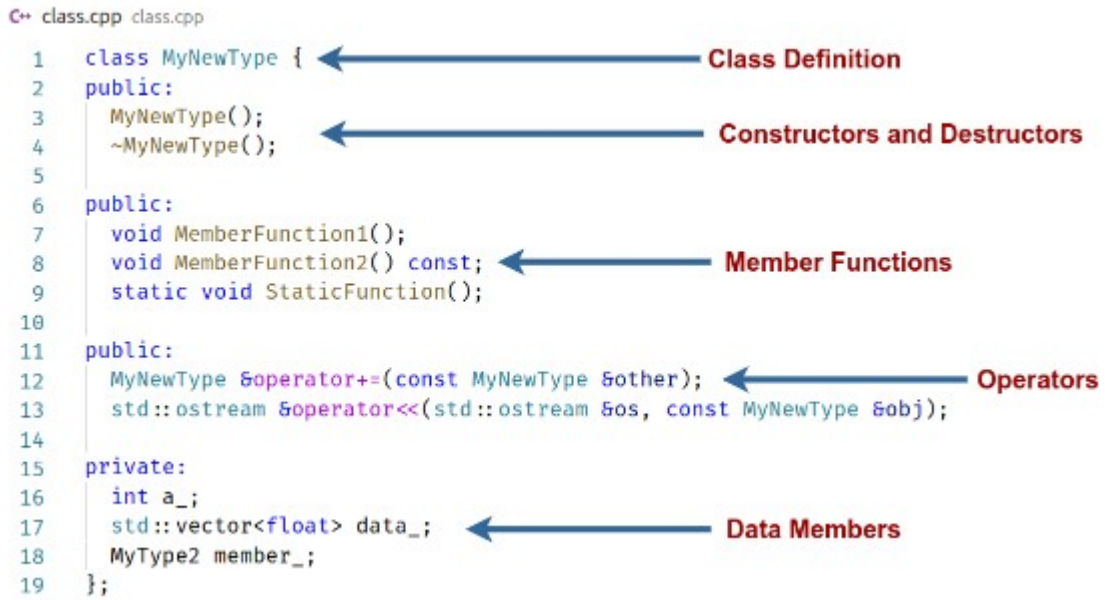
Example Class

- Class describes an image
- Every part of the code can define images and draw them
- Number of rows and columns are internal information

```
1  class Image {  // Should be in Image.hpp
2   public:
3    Image(const std::string& file_name);
4    void Draw();
5
6   private:
7    int rows_ = 0;  // New in C+=11
8    int cols_ = 0;  // New in C+=11
9  };
10
11 // Implementation omitted here, should be in Image.cpp
12 int main() {
13   Image image("some_image.pgm");
14   image.Draw();
15   return 0;
16 }
```

Classes – General Structure

- Each class has constructors and destructor that handle the data when defined or deleted
- By default everything is *private*



Constructor & Destructors

- At least one Constructor, exactly one Destructor
- Constructors
  - Named like the class
  - Can be called with arguments, will assign values to internal parameters
- Destructor
  - Named ~ClassName()
  - Last thing called during lifetime of an object

Constructor Calls - Example

- Many different ways of calling a constructor
- Dependent on the initialising values you want to pass

```
1  class SomeClass {
2   public:
3    SomeClass();                // Default constructor.
4    SomeClass(int a);           // Custom constructor.
5    SomeClass(int a, float b);  // Custom constructor.
6    ~SomeClass();               // Destructor.
7  };
8  // How to use them?
9  int main() {
10   SomeClass var_1;                // Default constructor
11   SomeClass var_2(10);            // Custom constructor
12   // Type is checked when using {} braces. Use them!
13   SomeClass var_3{10};            // Custom constructor
14   SomeClass var_4 = {10};         // Same as var_3
15   SomeClass var_5{10, 10.0};      // Custom constructor
16   SomeClass var_6 = {10, 10.0};   // Same as var_5
17   return 0;
18 }
```

Declaration & Definition

- Data members belong to declaration
- Class methods can be defined elsewhere (c.f. split into header and cpp files)
- Class name becomes part of function name

```
1  // Declare class.
2  class SomeClass {
3   public:
4    SomeClass();
5    int var() const;
6   private:
7    void DoSmth();
8    int var_ = 0;
9  };
10 // Define all methods.
11 SomeClass::SomeClass() {} // This is a constructor
12 int SomeClass::var() const { return var_; }
13 void SomeClass::DoSmth() {}
```

Classes - Keywords

- **Const**: after function states that object isn't changed

```cpp
#include <algorithm>
#include <vector>
class Human {
 public:
  Human(int kindness) : kindness_{kindness} {}
  bool operator<(const Human& other) const {
    return kindness_ < other.kindness_;
  }

 private:
  int kindness_ = 100;
};
int main() {
  std::vector<Human> humans = {Human{0}, Human{10}};
  std::sort(humans.begin(), humans.end());
  return 0;
}
```

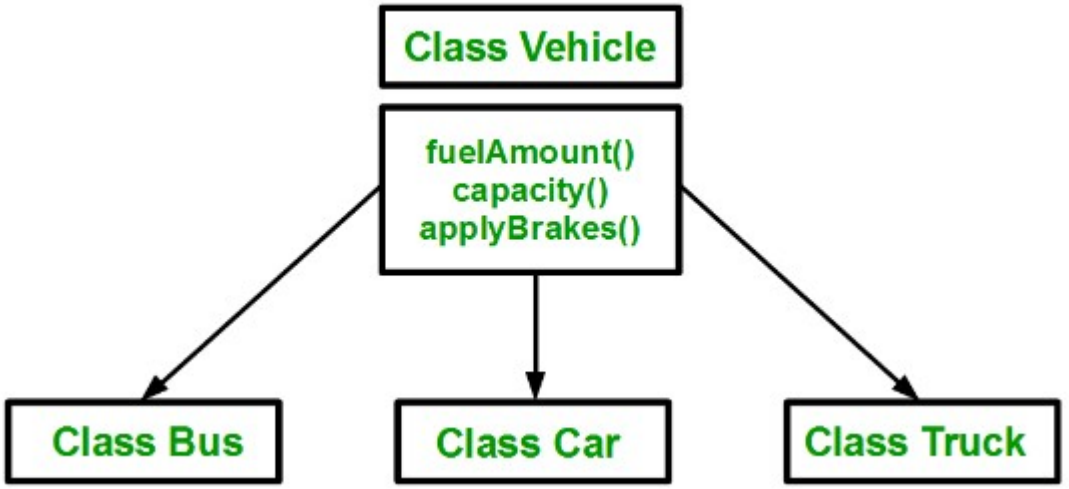
Classes - Keywords

Static:

- Member variables:
  - Shared between all objects of this type
- Member functions:
  - Don’t need to be called as <object>.function()
  - But as ClassName::function(<parameters>)
  - Independent of all instances of the class

Classes - Inheritance

- One general base class, e.g. Vehicle
- Many specialised derived classes



Classes - Inheritance

- Three types of inheritance in C++
- We'll be using public inheritance:
  - Keeps access specifiers of base class

```
class Derived : public Base {
  // Contents of the derived class.
};
```

- Corresponds to a "is a" relationship, e.g. a bus "is a" vehicle
- Can add specialised members that aren't generally applicable to vehicles

Inheritance - Example

- Base class are rectangles
- Implement squares as “specialised” rectangles

```cpp
#include <iostream>
using std::cout; using std::endl;
class Rectangle {
 public:
  Rectangle(int w, int h) : width_{w}, height_{h} {}
  int width() const { return width_; }
  int height() const { return height_; }
 protected:
  int width_ = 0;
  int height_ = 0;
};
class Square : public Rectangle {
 public:
  explicit Square(int size) : Rectangle{size, size} {}
};
int main() {
  Square sq(10);  // Short name to save space.
  cout << sq.width() << " " << sq.height() << endl;
  return 0;
}
```

Classes - Keywords

- Virtual:

```
virtual Func(<PARAMS>);
```

  - A virtual function can be overridden by derived classes

```
Func(<PARAMS>) override;
```

  - Pure virtual functions have to be overridden

```
virtual Func(<PARAMS>) = 0;
```

Memory & Pointers

Pointers

- Pointers store memory addresses of any kind of data
- <Type>* is a pointer to type <Type>
- Initialise pointers to either an address or NULL

```
1 int* a = nullptr;
2 double* b = nullptr;
3 YourType* c = nullptr;
```

- These are raw pointers and are disfavoured
- Better use smart pointers whenever suitable

(De)referencing

- Generally: & returns the address of an object, also true for pointers (referencing)
- This way you can call functions by reference:
  - function(&inputParameter)
  - No copying of the object required
- Dereferencing: Return object that sits at a certain address/reference
- If ptr is a pointer, *ptr will return the object

Stack & Heap

- Stack:
  - Static Memory available short term (scope)
  - Small but very fast
- Heap:
  - Dynamic Memory available during runtime
  - Slower than Stack
  - Modified by *new* and *delete*

new & delete

- Control of memory is completely up to you!

```
// pointer variable stored on stack
int* int_ptr = nullptr;
// 'new' returns a pointer to memory in heap
int_ptr = new int;
```

- New returns the heap address of the object
- You (!) need to free the memory after use

```
int* int_ptr = nullptr;
int_ptr = new int;
// delete frees memory to which the pointer points
delete int_ptr;
```

- Easy to create memory leaks (inaccessible addresses)
- Causes slow runtime and potential crash

Smart Pointers

- Manage lifetime of pointers and own the allocated memory
- Part of *#include <memory>*
- Focus on unique_ptr and shared_ptr
- Other than memory management, identical to raw pointers
- Dereference via *ptr, access via ptr->

Unique Pointer - Example

- Example based on vehicles:

```
std::unique_ptr<Vehicle> vehicle_1 =
std::make_unique<Bus>(20, 10, "Volkswagen", "LPM_");

std::unique_ptr<Vehicle> vehicle_2 =
std::make_unique<Car>(4, 60, "Ford", "Sony");
```

- Cannot be copied, but moved:

```
vehicle_2 = std::move(vehicle_1);
```

- Addresses before the move:

```
cout << "vehicle_1 = " << vehicle_1.get() << endl;
cout << "vehicle_2 = " << vehicle_2.get() << endl;
```

```
vehicle_1 = 0x56330247ce70
vehicle_2 = 0x56330247cec0
```

- And after the move:

```
vehicle_2 = 0x56330247ce70
vehicle_1 = 0
```

Shared Pointer - Example

- Many shared pointers can point to the same address
- Has a usage counter, +1 when copied, -1 when destructed, frees memory at 0

```
1  class MyClass {
2   public:
3    MyClass() { cout << "I'm alive!\n"; }
4    ~MyClass() { cout << "I'm dead... :(\n"; }
5  };
6
7  int main() {
8    auto a_ptr = std::make_shared<MyClass>();
9    cout << a_ptr.use_count() << endl;
10   {
11     auto b_ptr = a_ptr;
12     cout << a_ptr.use_count() << endl;
13   }
14   cout << "Back to main scope\n";
15   cout << a_ptr.use_count() << endl;
16   return 0;
17 }
```

When to use what?

- Generally: use smart pointers whenever you need to manage memory
- Raw pointers are fine as temporary storage for addresses for example
- Default should be unique_pointer
- If you need something copyable, use shared pointers
- Every single new or delete is a potential memory leak

Templates

Templates

- Templates can define two things in C++
- It can be a family of classes
- It can be a family of functions (possibly inside of a class)
- Motivation: Generic functions like abs()
  - Don’t want to define for every possible type like int, float, double, long, complex?, …

“absolute” Template

- Define generic abs<T>() function:

```
1 template <typename T>
2 T abs(T x) {
3   return (x >= 0) ? x : -x;
4 }
```

- Technically, templates aren’t functions but a pattern to make functions
- Only instantiated for used functions, if no one uses “abs<int>” no instantiating by the compiler
- Use:

```
6  int main() {
7    const double x = 5.5;
8    const    int y = -5;
9
10   auto abs_x = abs<double>(x);
11   int  abs_y = abs<int>(y);
12
13   double abs_x_2 = abs(x);  // type-deduction
14   auto   abs_y_2 = abs(y);  // type-deduction
15 }
```

Template classes

- Pattern for making classes

```
template <class T>
class MyClass {
 public:
  MyClass(T x) : x_(x) {}

 private:
  T x_;
};

int main() {
  MyClass<int> my_float_object(10);
  MyClass<double> my_double_object(10.0);
  return 0;
}
```

- E.g. vector is a template class

Lambda Functions

Lambda Functions

- Small anonymous functions of most basic form:

  [capture](parameter list){lambda body}

- The capture: which variables of the scope should be used inside of the function (can be empty)
- The parameter list: list of input parameters, like a regular function
- The lambda body: Like the body of any function

```
int main()
{
   using namespace std;
   int n = [] (int x, int y) { return x + y; }(5, 4);
   cout << n << endl;
}
```

Debugging & References

Debugging & References

- You will often face code that doesn’t work without knowing exactly why
- Easiest thing to do: Put couts into the code
  - This way, you’ll see where the code is before it crashes and which values variables have at any given time
- More complex machinery like gdb or valgrind (find out yourself)
- StackExchange, StackOverflow, Google in general