Advanced Programming Techniques

(Amazing C++)

Bartosz Mindur

Applied Physics and Computer Science Summer School '21

Kraków, 2021-07-05

Agenda

References

C++ history

C++11/14 - must be used

C++17 - should be used

C++20 - may be used

...

References and tools

C++ links and cool stuff

C++ links

- ISO C++
- cpp references
- c & cpp programming
- `#include <C++>`
- LernCpp.com
- cplusplus.com
- Moderns C++
- CppCon
- CppCast
- Bartek Filipek
- ...

Online tools

- repl.it
- Compiler Explorer
- cpp insights
- cpp.sh
- Quick Bench
- Online GDB
- piaza.io
- codiva.io

Offline tools

- CMake
- Valgrind
- GDB
- Docker
- Clang Static Analyzer
- CI/CD - not exclusively for C++

Things you (probably) already know well

C++ basics

- variables
- references
- pointers
- functions
- statements
- loops
- declaration
- initialization
- operator overloading
- classes
  - constructors & destructor
  - fields
  - methods
- inheritance
  - types
  - virtual functions
  - polymorphism
  - multiple inheritance
- conversion and casts
  - implicit
  - explicit
  - static
  - dynamic
- exceptions
- function templates
- class templates
- smart pointers
- basics of the STL
  - containers
  - iterators
  - algorithms
- building programs
  - compilation
  - linking
  - libraries
  - tools

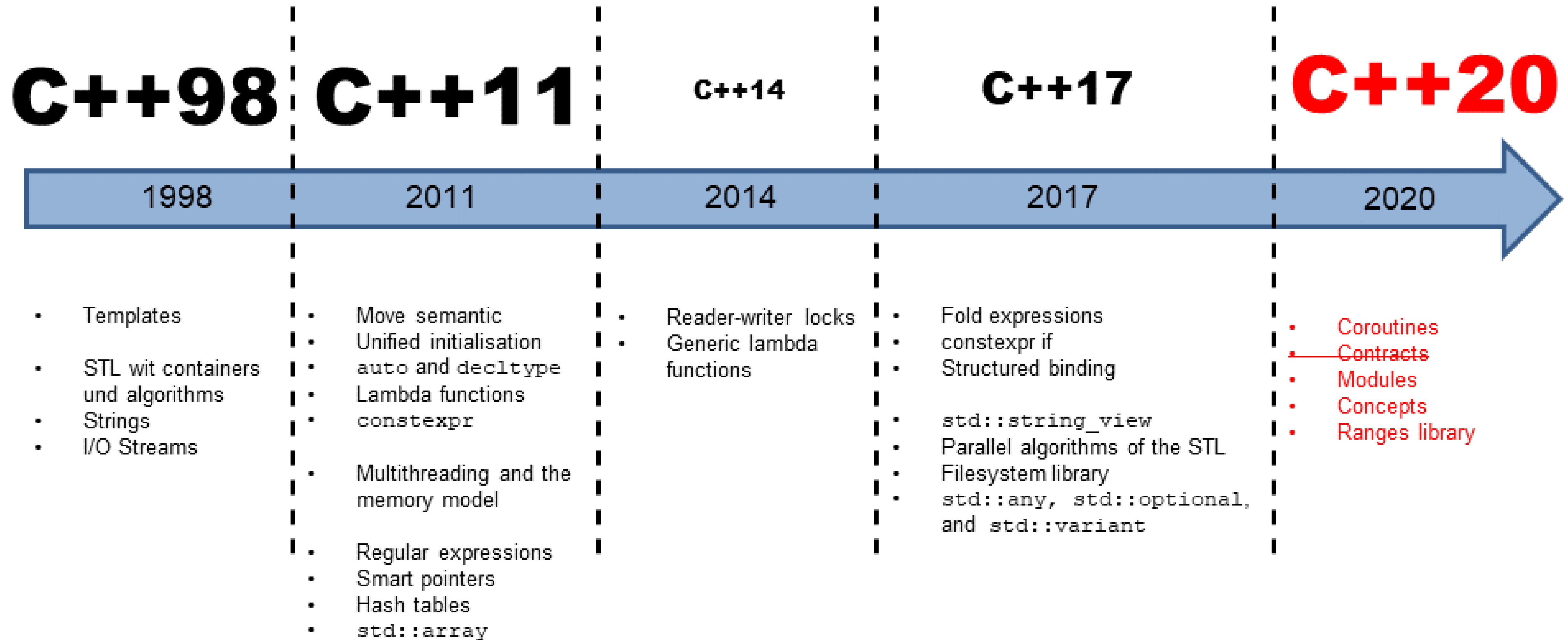
C++ history

The Design of C++

The Design of C++, a lecture by Bjarne Stroustrup

This video has been recorded in March, 1994 [link]

C++ Timeline



[link]

C++11/C++14

Move semantics

Value categories (simplified)

`lvalue`

- e.g. named variable
- something which stands on the left of the `operator=`

`prvalue` - pure `rvalue`

- e.g. temporary variable (without name)
- something which stands on the right of the `operator=`

`xvalue` - 'eXpiring' value

- `std::move(x)`

`rvalue` = `prvalue` + `xvalue`

Special members

- `T::T(const T&& other)` or `T::T(T&& other)`
- `T& operator=(T&& other)`

user declares	compiler implicitly declares					
	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

[Ref], [Ref], [Ref] and Code sample

Constant expressions - `constexpr`

The `constexpr` specifier declares that it is possible to evaluate the value of the function or variable at compile time

Such variables and functions can then be used where only compile time constant expressions are allowed

They are somehow similar to `const` but ordinary `const` refers to run-time

```
#include <iostream>

// C++11 constexpr functions use recursion rather than iteration
// C++14 constexpr functions may use local variables and loops
constexpr int factorial(int n) {
    const int res = n <= 1 ? 1 : (n * factorial(n - 1));
    return res;
}

// output function that requires a compile-time constant, for testing
template<int n>
struct constN {
    constN() { std::cout << n << '\n'; }
};

int main() {
    std::cout << "4! = " ;
    constN<factorial(4)> out1; // computed at compile time

    int n = std::rand()/((RAND_MAX + 1u)/6);
    int out2 = factorial(n); //computed at run time
    std::cout << n << "! = " << out2 << "\n";
}
```

[Ref] and Code sample #1, Code sample #2

Uniform initialization and `initializer_list`

Braced initialization list - `{}`

```
int foo(){
  int i{1};
  int arr[] = {1, 2, 3, 4, 5};
  int *pi = new int[5]{1, 2, 3, 4, 5};

  std::string s {"hello"};
  std::vector foo{1, 2};
  std::map<int,std::string> bar
    { {0,"zero"}, {1,"one"}, {2,"two"} };

  f({foo, bar});
  return {foo, bar};
}
```

Initialization with `std::initializer_list`

```
template <class T>
struct S {
    std::vector<T> v;
    S(std::initializer_list<T> l) : v(l) {
         std::cout << "constructed with a "
            << l.size()
            << "-element list\n";
    }
    void append(std::initializer_list<T> l) {
        v.insert(v.end(), l.begin(), l.end());
    }
};
```

[Ref], [Ref] and Code sample

Type interface `auto`

For variables, specifies that the type of the variable that is being declared will be automatically deduced from its initializer

For functions, specifies that the return type will be deduced from its return statements [Ref]

```
#include <utility>
#include <iostream>
int main() {
    auto a = 1 + 2;            // type of a is int
    auto b = 1 + 1.2;          // type of b is double
    static_assert(std::is_same_v<decltype(a), int>);
    static_assert(std::is_same_v<decltype(b), double>);

    const auto c0 = a;             // type of c0 is int, holding a copy of a

    auto d = {1, 2}; // OK: type of d is std::initializer_list<int>
    auto n = {5};    // OK: type of n is std::initializer_list<int>
//  auto e{1, 2};    // Error as of C++17, std::initializer_list<int> before
    auto m{5};       // OK: type of m is int as of C++17, initializer_list<int> before

//  auto int x; // valid C++98, error as of C++11
//  auto x;     // valid C, error in C++
}
```

Range-based `for` loop

Executes a for loop over a range

```cpp
#include <iostream>
#include <array>

template <typename T, typename V>
void set_for_each(T cnt, V val){
  for(auto v: cnt)
    v = val;
}

template <typename T>
void printer(T cnt, std::ostream& o = std::cout) {
  for(auto v: cnt)
    o << v << ";";
}

int main() {
  std::array<int, 5> t = {0, 1, 2, 3};
  set_for_each(t, 13);
  printer(t);

}
```

```cpp
#include <iostream>
#include <array>

template <typename T, typename V>
void set_for_each(T& cnt, V val){
  for(auto& v: cnt)
    v = val;
}

template <typename T>
void printer(T cnt, std::ostream& o = std::cout) {
  for(auto v: cnt)
    o << v << ";";
}

int main() {
  std::array<int, 5> t = {0, 1, 2, 3};
  set_for_each(t, 13);
  printer(t);

}
```

Code sample #1

Code sample #2 and [Ref]

Explicit `override` and `final`

`virtual` function specifier

```
#include <iostream>
struct Base {
   virtual void f() {
       std::cout << "base\n";
   }
};
struct Derived : Base {
    void f() {
        std::cout << "derived\n";
    }
};
```

Do not forget of `virtual` destructors

[Ref] and Code sample

`override`

- Specifies that a virtual function overrides another virtual function.

`final`

- Specifies that a virtual function cannot be overridden in a derived class
- Or that a class cannot be inherited from

[Ref], [Ref] and Code sample

Lambdas

Constructs a closure

- an unnamed function object capable of capturing variables in scope

```
[ captures ] ( params ) spec -> ret { body }
```

- `captures` - a comma-separated list of zero or more captures with `=`, `&`, `this`
- `params` - a list of parameters, as in named functions
- `spec` - optional sequence of specifiers (e.g. `mutable`)
- `ret` - return type, if not present it's implied by the function return statements
- `body` - function body

[Ref]

```
#include <iostream>
int main() {
    auto f1 = [] () {
        std::cout << "simple lambda\n";
    };
    f1();

    int a = {5};
    auto f2 = [=] () {
        std::cout << "simple lambda: " << a << "\n";
    };
    f2();

    auto f3 = [&] () {
        a = 17;
        std::cout << "simple lambda: " << a << "\n";
    };
    f3();
}
```

Code sample #1 and Code explanation, Code sample #2

Code sample #3 and Code explanation

Template aliases and static assertions

`using`

- type alias is a name that refers to a previously defined type (similar to typedef)
- alias template is a name that refers to a family of types

```
template<class T>
struct Alloc { };

template<class T> // type-id is vector<T, Alloc<T>>
using Vec = vector<T, Alloc<T>>;

Vec<int> v; // Vec<int> == <int, Alloc<int>>
```

[Ref]

`static_assert`

- performs compile-time assertion checking
- no runtime costs

```
int main() {
    using T1 = char;
    using T2 = int;

    static_assert(sizeof(T1) >= 2,
        "Type size is too small."); // error
    static_assert(sizeof(T2) >= 2,
        "Type size is too small."); // ok
}
```

[Ref] and Code sample

Variadic templates & parameter pack

Parameter pack

- a template parameter pack is a template parameter that accepts zero or more template arguments
- a template with at least one parameter pack is called a variadic template
- template parameter pack `...` appears in
  - alias template
  - class template
  - variable template
  - function template parameter lists

```
template<class ... Types>
    struct Tuple {};
Tuple<> t0;             // no arguments
Tuple<int> t1;          // one argument: int
Tuple<int, float> t2;   // two arguments: int and float
Tuple<0> error;         // error: 0 is not a type
```

Expand parameter pack

```
#include <iostream>

template<class T, class ...Ts>
void print(T arg) {
    std::cout << "[" << arg << "] ";
}

template<class T, class ...Ts>
void print(T arg, Ts... args) {
    print(arg);
    print(args...);
}

int main() {
    print(1, 2.0, "Tekst", "Next text");
}
```

Code sample #1, Code sample #2 and [Ref]

Template template parameters (pre C++11 standard)

- Sometimes we would like to pass into the template a template type without fixing its values
- This is what template template parameters are created for

```
template<typename T>
using stack_v = stack<T, std::vector>;

int main() {
  using stack = stack<int, std::vector>;
  stack s1;

  stack_v<double> s2;
}
```

Code sample #1 and [Ref]

- the names of template template parameters do not have to be specified

```
template<typename T> class A
    { int x; }; // primary template
template<class T> class A<T*>
    { long x; }; // partial specialization

// class template with a template template parameter
template<template<typename> class V> class C
{
    V<int> y; // uses the primary template
    V<int*> z; // uses the partial specialization
};

C<A> c; // c.y.x has type int, c.z.x has type long
```

Code sample #2

Compile-time decisions and `type_traits`

Compile-time 'if' statement

```
#include <iostream>
#include <type_traits>
#include <typeinfo>

int main() {
using std::conditional<true, int, double>::type T1;
using std::conditional<false, int, double>::type T2;
using T3=std::conditional<sizeof(int) >=
    sizeof(double), int, double>::type;

    std::cout << typeid(T1).name() << '\n';
    std::cout << typeid(T2).name() << '\n';
    std::cout << typeid(T3).name() << '\n';
}
```

Code sample

Possible implantation

```
template<bool B, class T, class F>
struct conditional { typedef T type; };

template<class T, class F>
struct conditional<false, T, F> { typedef F type; };
```

Type traits

- type traits defines a compile-time template-based interface to query or modify the properties of types
  - type categories
  - type properties
  - type operations

[Ref] and Code sample

Asynchronous execution

- the function template `std::async` runs given function asynchronously [Ref]
  - potentially in a separate thread which may be part of a thread pool
  - returns a `std::future` that will eventually hold the result of that function call
- `std::future` [Ref]
  - provides a mechanism to access the result of asynchronous operations e.g. `std::async` or `std::promise`
- `std::promise` [Ref]
  - provides a facility to store a value or an exception that is later acquired asynchronously via a `std::future` object created by the `std::promise` object

```cpp
#include <future>
#include <iostream>

int main() {

  auto f1 =  // std::feature
    std::async(
      []() {  // lambda
        int sum = 0;
        for(int i = 1; i < 100; ++i)
          sum += i;
        return sum;
      });

  std::cout << "Sum: " << f1.get() << '\n';
}
```

Code sample

C++17

Structured binding declaration

Useful way to work with `pair`, `tupe` or `set` and `map`

`auto [a, b, c, ...] = expression;`

```
std::pair a(0, 1.0f);
auto [x, y] = a;
```

```
std::pair a(0, 1.0f);
const auto [x, y] = a;
```

```
std::tuple a {0, 1.0f, "string", 3.14, };
auto& [ refA, refB, refC, refD ] = a;
```

```
struct Point {
  double x;
  double y;
};

Point GetStartPoint() {
  return { 0.0, 0.0 };
}

const auto [x, y] = GetStartPoint();
```

Code sample #1, Code sample #2 and [Ref]

Class template argument deduction

In order to instantiate a class template, every template argument must be known

Before C++17 all templates arguments have to be specified directly, or using function templates -

```
// direct usage
std::tuple<int, double> p1{1, 1.0};

// helper function template
auto p2 = std::make_tuple(1, 1.0);
```

In C++17 deduction of template types is possible for classes in

- any declaration that specifies initialization of a variable
- `new` expressions
- function-style casts expressions like `unsigned(f)`

```
std::pair p(2, 4.5);
    // deduces to std::pair<int, double> p(2, 4.5);
std::tuple t(4, 3, 2.5);
    // same as auto t = std::make_tuple(4, 3, 2.5);
auto ptr = new std::pair{42, "World"s}; // for new
```

Code sample and [Ref]

Initialization statements for `if` and `switch`

In the init section you can specify a new variable, similarly to the init section in `for` loop

`if (init; condition)` and `switch (init; condition)`

```
{ //C++ code before C++17
  auto val = GetValue();
  if (condition(val))
    // on success
  else
    // on false...
}
```

```
if (auto val = GetValue(); condition(val))
  // on success
else
  // on false...
```

```
// Structured bindings + if initializer
if (auto [iter, succeeded] = m.insert(value);
  succeeded)
  {
  use(iter); // ok
  // ...
} // iter and succeeded are destroyed here
```

[Ref] and Code sample

Nested namespaces

Namespaces provide a method for preventing name conflicts in large projects

Nasted namespaces to be used in more convenient way of grouping namespace inside other namespace

```
// pre C++ 17 code
namespace MyCompany {
  namespace SecretProject {
    namespace SafetySystem {
      class SuperArmor {
        // ...
      };
      class SuperShield {
        // ...
      };
    } // SafetySystem
  } // SecretProject
} // MyCompany
```

```
// C++ 17 code
namespace MyCompany::SecretProject::SafetySystem {
  class SuperArmor {
    // ...
  };
  class SuperShield {
    // ...
  };
}
```

[Ref]

__has_include preprocessor directive

__has_include

- result of 1 only means that a file with the specified name exists
- it does not mean that the file, when included, would not cause an error etc.
- e.g. one could use experimental features or standard without changing the code

```
#if __has_include(<optional>)
#  include <optional>
#  define have_optional 1
#elif __has_include(<experimental/optional>)
#  include <experimental/optional>
#  define have_optional 1
#  define experimental_optional 1
#else
#  define have_optional 0
#endif

// rest of code
```

Code sample and [Ref]

Fold expressions

C++11 introduced variadic templates

Variadic templates required some additional code when you wanted to implement 'recursive' functions - rule to stop it

```
// C++11 code
auto Sum() {
  return 0;
}
template<typename T1, typename... T>
  auto sum(T1 s, T... ts) {
    return s + sum(ts...);
}
```

```
// C++17 code
template<typename ...Args> auto sum(Args ...args) {
  return (args + ...);
}
```

```
// C++17 code
template<typename ...Args>
void FoldPrint(Args... args) {
  (std::cout << ... << args) << '\n';
}
```

Code sample and [Ref]

Attributes

Introduces in C++11

- `[[noreturn]]`

In C++14 added

- `[[deprecated]]` and `[[deprecated("reason")]]`

In C++17 added

- `[[fallthrough]]` - for `switch`
- `[[maybe_unused]]` - OK if not used
- `[[nodiscard]]` - return value shall no be omitted

[Ref] and Code sample

```
[[noreturn]] void terminate() noexcept {}
[[deprecated("use BetterFunc")]] void f() { }
struct [[deprecated]] OldStruct { };

void switch_fun(char c) {
    switch (c) {
    case 'a':
    f(); // Warning! fallthrough
    case 'b':
    terminate();
    [[fallthrough]]; // Warning suppressed
    case 'c':
    terminate();
    }
}
void foo() {
    int x = 13; // warning
    [[maybe_unused]] int y = 13; // no warning
}

[[nodiscard]] inline int Compute() {return 1;}

int main () {
    Compute(); // Warning! return value of a
    // nodiscard function is discarded
}
```

C++20

Already a standard

May not be fully implemented in compilers

- C++ compiler support

Much to be learned

- Constraints and concepts
- Coroutines
- Ranges
- Modules
- ...

Thank you for your time

(any questions?)