Intermediate C++

Tuesday, 06 September 2022
Joost Bonnet

Consultant

1

ALTEN

0. Introduction

2

Introduction – Joost Bonnet

- 1991 – 1996: MSc. Computer Science at Delft University of Technology
- 1997 – 2014: C++ developer, architect, technical project lead, team lead at Logica/LogicaCMG/Logica/CGI, many projects in oil & gas industry (and outside)
- 2014 – now: C++ developer, architect, C++ & OO trainer at Alten
- Currently software architect at Deltares

3

Introduction

- **Part 1**
  - Overview
  - Language and Syntax Changes
  - Uniform Initialization
- **Part 2**
  - Containers and Algorithms
  - Operator Overloading
  - Dynamic Memory
  - Move Semantics
  - Smart Pointers
- **Part 3**
  - Interfaces
  - TDD in C++
  - Dependency Injection
- **Part 4**
  - Compilers and Tooling
  - Concurrency

4

4

Introduction

- **House Rules**

5

5

Expectations

- **Be interactive, ask questions, start discussions**

6

6

1. Overview

7

Overview

Bjarne Stroustrup

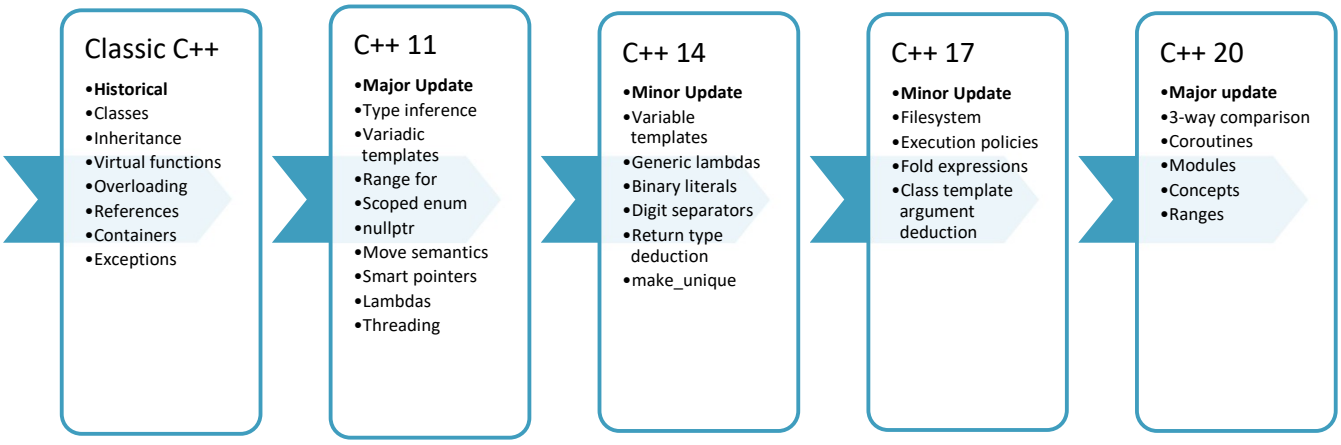
What is C++?

C++ is a multi-paradigm compiled language that evolved from C in 1979-2014 (2017)

8

8

Overview



9

9

Overview

- **Question: Which features of C++ do you use daily?**

10

10

Overview

Assumptions:

- **Comfortable with pointers and references**
- **Knowledge of Memory usage in C++**
- **Object Oriented Programming basics**
- **Container usage**
- **Exceptions**

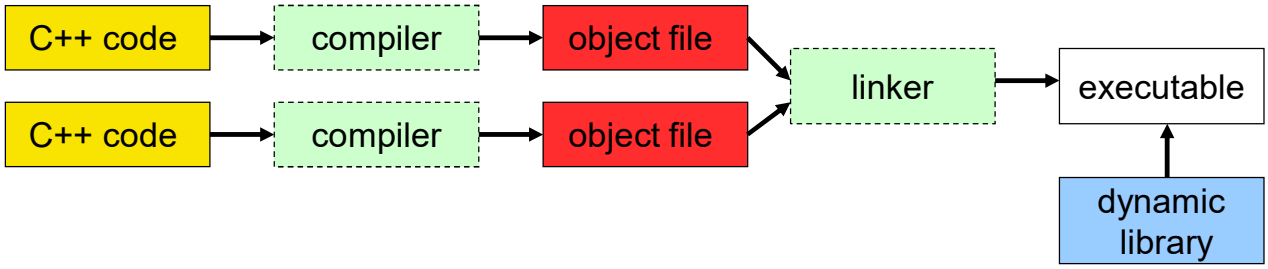
11

11

Overview

C++ is a compiled language.

- **It is completely translated to machine instructions before the program is executed.**



12

12

Overview

- **C++ is a multi-paradigm language, supporting imperative, procedural, object-oriented and generic programming:**
  - C++ is statically typed.
  - C++ is an intermediate-level language. It has both high-level and low-level features.
  - C++ supports const-correctness.
  - C++ does generally not provide features that will force your code to be less than optimal (*"what you don't use, you don't pay for"*).

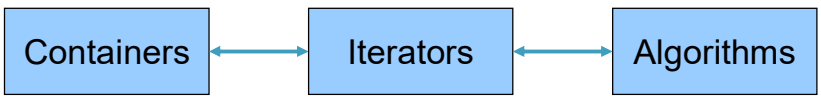
13

13

Containers and Algorithms

Introduction

- **Modern C++ makes extensive use of standard library (STL) containers and algorithms**
- **Algorithms are universal**
- **Iterators decouple containers from algorithms**



14

14

Objects

Object-oriented programming was the first reason for C++ to exist.

What are the three principles of OOP?

1. **Encapsulation**
   - Binding data and functions together
   - Hiding internal data and implementation
2. **Inheritance**
   - Deriving a class from another class to extend functionality
3. **Polymorphism**
   - Multiple implementations of the same interface

15

15

Encapsulation

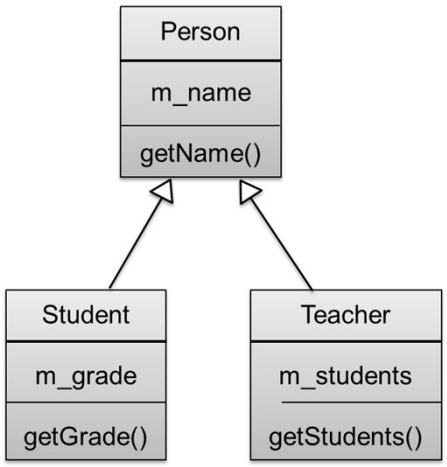
- **Use classes to group data and code**
- **Keep data and implementation code private**
- **Define public methods to be accessed by client code**

16

16

Inheritance

- Code reuse example:
- The classes Student and Teacher are derived from Person.
- Having a name is common functionality.
- Having grades is not.



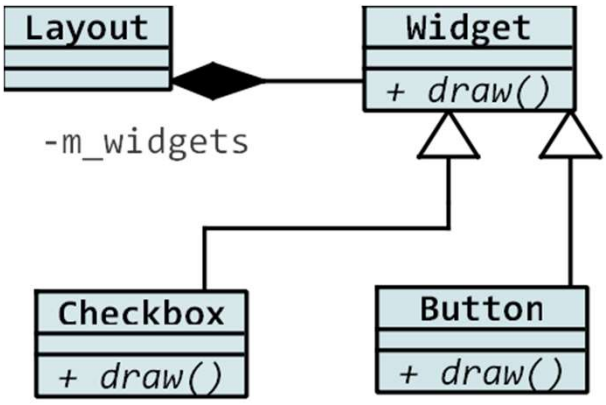
17

17

Polymorphism

What is the gain?

- Layout is unknowing of the types of Widgets that exist.
- Our coupling has decreased, which improves maintainability and extendibility.
- We can later add more widget types without having to alter existing code.



18

18

Object Oriented Design

Object Oriented Design

- **The quality of your code depends greatly on the design choices you make.**

These are your guidelines:

A. **A class should represent one thing, not more, not less.**
B. **Public inheritance should always represent the is-a relation.**
C. **Bind your data and functions together.**

19

19

Exceptions

- **During runtime, we can anticipate problems with the normal flow of the program.**
- **Some of these are common and can be handled locally, such as wrong user input.**
- **Sometimes a problem cannot be handled where it is detected, such as a file that can suddenly not be read, or the program has run out of memory.**
- **For these cases, we can use exceptions.**
- **Exceptions let us separate error handling from normal control flow.**

20

20

Exceptions

- Exceptions do not magically solve your problems.
- In fact, using exceptions can introduce new problems.
- In practice, *know* in which parts of your project exceptions are used, and how they ought to be dealt with.
- Do not use exceptions without having a plan.

21

21

2. Language and Syntax Changes

22

Language and Syntax Changes

- Changes to the C++ language are managed by the WG21 ISO C++ committee, supported by the Standard C++ Foundation
- (isocpp.org)
- formed in 1990-91, and consists of accredited experts from member nations of ISO/IEC JTC1/SC22 who are interested in C++ work

- Changes to the language are managed here

23

23

Language and Syntax Changes

- C++98, First ISO standard
- C++11: Significant evolution of the language
- C++14 and C++17: 'Minor' revisions of C++
- C++20 will be another major revision

24

24

Language and Syntax Changes

- **Core language features**
  - Improved compile-time evaluation.
    - constexpr
    - static_assert
  - Improved compile-time type checking
    - Null pointer constant.
    - Strongly typed enumerations.
  - Quality of life improvements
    - Range based for-loops
    - Initialization in if and switch statements
    - Lambda expressions
    - Type inference
    - More…

25

25

Language and Syntax Changes

- **Library features**
  - new containers (array, unique_ptr,…)
  - new abstractions (std::thread, std::chrono,…)
  - new functionality (std::regex,…)
  - And much more…

26

26

Language and Syntax Changes

Compile-time evaluation

- Trend in recent language versions ‘do more during compilation’
- Results in more efficient runtime
- Results in more errors found at compile time
- Most new additions are optional

Downside of this trend

- More complex compilers
- Many new keywords

27

27

Language and Syntax Changes

- Core language features
  - **Improved compile-time evaluation.**
    - **constexpr**
    - **static_assert**
  - Improved compile-time type checking:
    - Null pointer constant.
    - Strongly typed enumerations.
  - Quality of life improvements
    - Range based for-loops
    - Initialization in if and switch statements
    - Lambda expressions
    - Type inference
    - More…

28

28

Slide 27

BB0 Also everything that breaks at compile-time cannot fail at runtime
Bart Beumer, 2022-08-24T07:02:40.015

BB1 Where did you get that? For safety I think about rust.

Did you mean strongly typed?
Bart Beumer, 2022-08-24T07:04:22.720

BB2 NOTE: Removed ++ is known for type safety and runtime speed
Bart Beumer, 2022-08-24T15:11:04.197

constexpr – Generalized constant expressions

- **C++11 provides two related kinds of ‘‘constant’’:**
  - const: Do not modify in this scope
  - constexpr: Evaluate at compile time
- **A constant expression is an expression that a compiler can evaluate**
  - It cannot use values that are not known at compile time
  - It cannot have side effects
- **If the initializer for a constexpr can’t be evaluated at compile time, the compiler will give an error**

29

constexpr – Generalized constant expressions

```
int x1 = 7;
constexpr int x2 = 7;
constexpr int x3 = x1; // error : initializer is not a
                       // constant expression
constexpr int x4 = x2; // OK
```

30

constexpr – Generalized constant expressions

- constexpr also applies to functions
- constexpr functions must be simple: 'pure' and single return statement

```
enum Flags { good = 0, fail = 1, bad = 2, eof = 4 };

constexpr int operator|(Flags f1, Flags f2)
{
    return Flags(int(f1) | int(f2));
}

constexpr int x1 = bad | eof; // ok
```

31

constexpr – Generalized constant expressions

- A constexpr function is usable in constexpr expressions when given constexpr arguments
- It can also be used as a regular (non-constexpr) function

```
constexpr int fac(int n)
{
    return (n>1) ? n*fac(n-1) : 1;
}

constexpr int f9 = fac(9); // must be evaluated in compile time
int f5 = fac(5); // may be evaluated at compile time
int fn = fac(n); // evaluated at run time (since n is a variable)
```

32

constexpr – Generalized constant expressions

- **A class with a constexpr constructor is called a literal type.**
- **Such a constructor must have an empty body. All members must be initialized by potentially constant expressions**

```
struct Point {
int x, y;
    constexpr Point(int xx, int yy) : x(xx), y(yy) { }
};

constexpr Point origin(0, 0);
constexpr int z = origin.x;
```

33

Relaxed *constexpr* restrictions (C++14)

- **C++11 constexpr functions could only contain a single expression that is returned**
- **C++14 allows more things inside the body of constexpr functions, notably**
  - local variable declarations (not static or thread_local, and no uninitialized variables)
  - mutating objects whose lifetime began with the constant expression evaluation
  - if, switch, for, while, do-while (no goto)
- **constexpr methods are no longer implicitly const in C++14**

34

Relaxed *constexpr* restrictions (C++14)

- In C++11 we could write

```
constexpr int my_charcmp(char c1, char c2) {
    return (c1 == c2) ? 0 : (c1 < c2) ? -1 : 1;
}
```

- C++14 allows this

```
constexpr int my_strcmp(const char* str1, const char* str2) {
    int i = 0;
    for (; str1[i] && str2[i] && str1[i] == str2[i]; ++i) {
    }
    if (str1[i] == str2[i]) return 0;
    if (str1[i] < str2[i]) return -1;
    return 1;
}
```

35

If constexpr (C++17)

- C++17 introduces if constexpr. These if statements are checked once at compile time, effectively skipping/passing pieces of code for compilation

```
if constexpr (sizeof( int ) == 4 )
{
    std::cerr << "32-bit ints" << std::endl;
}
```

- If constexpr allows failing (else) clauses which would sometimes result in a compilation error.

36

Static assertions

- **C++11 introduces static assertions. These are checked at compile time. A static assertion failure causes a compilation error.**

```
static_assert(sizeof(long) >= 8, "64-bit code generation required.");
struct S { X m1; Y m2; };
static_assert(sizeof(S) == sizeof(X) + sizeof(Y), "unexpected padding in S");
```

- **static_assert is evaluated at compile time, it cannot check runtime values**

```
int f(int* p, int n)
{
    static_assert(p == 0, "p is not null"); // error: not a constant
                                            // expression
}
```

37

Language and Syntax Changes

- **Core language features**
  - Improved compile-time evaluation.
    - constexpr
    - static_assert
  - **Improved compile-time type checking**
    - **Null pointer constant.**
    - **Strongly typed enumerations.**
  - Quality of life improvements
    - Range based for-loops
    - Initialization in if and switch statements
    - Lambda expressions
    - Type inference
    - More…

38

38

Language and Syntax Changes

Compile-time type checking

- **C++ is known for strong type checking.**
- **Prevents us from comparing apples to oranges (or pointers to integers)**

39

39

Null pointer constant

- **Question 1: How is NULL defined?**

40

Slide 39

BB0 Also everything that breaks at compile-time cannot fail at runtime
Bart Beumer, 2022-08-24T07:02:40.015

BB1 Where did you get that? For safety I think about rust.

Did you mean strongly typed?
Bart Beumer, 2022-08-24T07:04:22.720

BB2 NOTE: Removed ++ is known for type safety and runtime speed
Bart Beumer, 2022-08-24T15:11:04.197

Null pointer constant

- Question 1: How is NULL defined?
- Question 2: How to assign a null pointer?

41

Null pointer constant

- Before C++98, null pointers were set to NULL, like in C.
- In C++98, 0 was introduced as null pointer value
- This did not solve the ambiguity between int and pointers

```
void f(int p);
void f(char *p);
f(0); // which ‘f’ gets called?
```

- In C++11, the nullptr keyword was added
- nullptr is the only instance of std::nullptr_t

42

Strongly typed enumerations

C++98 Enum type

```
enum Color
{
  red,
  blue,
  green
};
Color buttonColor = red;
```

- Are internally represented by integer
- Can be converted to integer

43

43

Strongly typed enumerations

Be aware of implicit conversions!

- enum is implicitly convertible to int:

```
int value = blue;
```

- int is not implicitly convertible to enum:

```
Color color = Color(0); // ok, conversion needed
```

- Watch out with function overloading:

```
void getButton( int id ); // forward declaration
getButton( blue );        // not an error!
```

44

44

Strongly typed enumerations

- **C++98 enums have three major problems:**
  - Implicit conversion to integer causes errors when someone does not want an enumeration to act as an integer.
  - Enumerators are visible in the surrounding scope, causing name clashes.
  - The underlying type of an enum cannot be specified
- **C++11 solves these problems by introducing a new kind of enum: 'enum class'**

45

Strongly typed enumerations

```
enum Alert { green, yellow, orange, red }; // traditional enum
enum class Color { red, blue };   // scoped and strongly typed enum
          // no export of enumerator names into enclosing scope
          // no implicit conversion to int
enum class TrafficLight { red, yellow, green };
Alert a = 7;              // error (as ever in C++)
Color c = 7;              // error: no int->Color conversion
int a2 = red;             // ok: Alert->int conversion
int a3 = Alert::red;      // error in C++98; ok in C++11
int a4 = blue;            // error: blue not in scope
int a5 = Color::blue;     // error: not Color->int conversion
Color a6 = Color::blue;   // ok
```

46

Strongly typed enumerations

- **Being able to specify the underlying type allows simpler interoperability and guaranteed sizes of enumerations:**

```
enum class Color : char { red, blue };  // to save memory
enum class TrafficLight { red, yellow, green };  // default
                                                 // size is int
enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFF0U };
// size of E is implementation defined
enum EE : unsigned long { EE1 = 1, EE2 = 2, EEbig = 0xFFFFFFF0U };
// C++11 lets us be specific
```

47

Strongly typed enumerations

- **It also enables forward declaration of enums:**

```
enum class Color_code : char;      // forward declaration
void foobar(Color_code* p);        // use of forward
                                   // declaration

  // ...
enum class Color_code : char { red, yellow, green, blue };
// definition
```

48

Strongly typed enumerations

- **Exercise: correct the following code:**

```
enum Alert { red, yellow, none };
enum TrafficLightState { red, orange, green };
int main()
{
    Alert a = red;
    TrafficLightState ts = red;
    return 0;
}
```

```
main.cpp(4,26): error : redefinition of enumerator 'red'
main.cpp(3,14): message : previous definition is here
main.cpp(9,20): error : cannot initialize a variable of type 'TrafficLightState' with an rvalue of type 'Alert'
```

49

49

Language and Syntax Changes

- **Core language features**
  - Improved compile-time evaluation.
    - constexpr
    - static_assert
  - Improved compile-time type checking
    - Null pointer constant.
    - Strongly typed enumerations.
  - **Quality of life improvements**
    - **Range based for-loops**
    - **Initialization in if and switch statements**
    - **Lambda expressions**
    - **Type inference**
    - **More…**

50

50

Language and Syntax Changes

Quality of life improvements

- **C++98 required us to type a lot of code for some common problems.**
- **C++11 introduced more concise and more readable equivalents**

51

51

Range-based for loop

C++11 introduced an extension to the syntax of the 'for' loop:

```
std::vector<std::string> v;
for (std::string s : v) // or: const std::string &s
{
    do_something(s);
}
```

Instead of:

```
for (std::vector<std::string>::iterator it = v.begin();
     it != v.end(); ++it)
{
    do_something(*it);
}
```

52

Range-based for loop

- The range-based for will actually use members begin() and end() under the hood. Something like:

```
class NotARange
{
  int begin = 4;
  int end = 2;
  std::vector< double > data;
};

NotARange nar;
for ( double& d : nar )
    std::cout << d << std::endl; // triggers a compile error
```

53

Initialization in if and switch statements (C++17)

- If and switch statements can contain initializations:
- We can now write

```
std::string str;
if (auto it = str.find("secret"); it != std::string::npos)
    cout << "Found secret at position " << it << endl;
```

- Instead of

```
auto it = str.find("secret");
if (it != std::string::npos)
    cout << "Found secret at position " << it << endl;
```

54

54

Initialization in if and switch statements (C++17)

- In switch statements we can do things like

```
switch (Pen p = current_pen(); p.color())
{
case Color::red:
    // ...
    break;
// ...
default:
    // ...
}
```

55

55

Lambda expressions

- What is a function?
- What is a method?
- What is a functor?
- What is a predicate?

56

Lambda expressions

- C++11 introduced lambda expressions
- A lambda defines an anonymous function object
- Lamda expresions are useful for defining single-use functions

```
vector<int> v = { 50, -10, 20, -30 };
std::sort(v.begin(), v.end());
// now v should be { -30, -10, 20, 50 }
std::sort(v.begin(), v.end(), [](int a, int b) {
    return abs(a) < abs(b); });
// now v should be { -10, 20, -30, 50 }
```

57

Lambda expressions

- [](int a, int b) { return abs(a)<abs(b); } is a lambda expression
- Lambdas consist of three parts:
  - Capture list – inside '[]'
  - Parameter list – inside '()'
  - Body – inside '{}'
- The return type is (usually) deduced by the compiler

58

Lambda expressions

- A lambda expression can access local variables in the scope in which it is used. We call this 'variable capture'
- [] means nothing is captured
- [=] captures all variables in scope by value
- [&] captures all variables in scope by reference
- [a, &b] captures a by value and b by reference (and nothing else)
- Explicit capture is preferred
- Note that any non-local variables in scope can always be accessed

59

Lambda expressions

- Capture example

```
void f(vector<int>& v)
{
    bool sensitive = true;
    // ...
    sort(v.begin(), v.end(), [sensitive](int x, int y) {
       return sensitive ? x<y : abs(x)<abs(y); });
}
```

60

Lambda expressions

Exercise: Sort by last name

- **A Person is a struct containing two strings: firstName and lastName.**
- **Create a vector of Persons, then sort it by last name.**
- **Use std::find_if() to find a Person in the vector by last name.**

61

61

Generic lambdas (C++14)

- **C++11:**

```
for_each(begin(v), end(v), [](const decltype(*begin(v))& x) { cout << x; });
sort(begin(w), end(w), [](const shared_ptr<some_type>& a,
const shared_ptr<some_type>& b) { return *a<*b; });
auto size = [](const unordered_map<wstring, vector<string>>& m) { return m.size(); };
```

- **C++14:**

```
for_each(begin(v), end(v), [](const auto& x) { cout << x; });
sort(begin(w), end(w), [](const auto& a, const auto& b) { return *a<*b; });
// a, b are iterators
auto size = [](const auto& m) { return m.size(); };
// note: this will work with any class with 'size()'
```

62

Lambda capture expressions (C++14)

- C++14 generalizes lambda capture
- This allows capture-by-move (not easily possible in C++11)
- It also lets you introduce local variables in a lambda

```
auto u = make_unique<some_type>(some, parameters);  // a unique_ptr is move-only
go.run([u = move(u)]{ do_something_with(u); }); // move the unique_ptr into the lambda
go.run([u2 = move(u)]{ do_something_with(u2); }); // capture as "u2“
int x = 4;
int z = [&r = x, y = x + 1]{ // r, y are implicitly declared as ‘auto’
    r += 2;           // set x to 6
    return y + 2;     // returns 7
}();
```

63

Type inference

- C++11 supports type inference through the ‘auto’ keyword

```
auto i = 7;          // int
auto j = expression; // j will have the type of
                     // ‘expression’
```

- Note that the type is still fixed at compile time

64

Type inference

- Instead of

```
template<class T> void printall(const vector<T>& v)
{
    for (typename vector<T>::const_iterator p = v.begin();
        p != v.end(); ++p)
        cout << *p << "\n";
}
```

- We can now write

```
template<class T> void printall(const vector<T>& v)
{
    for (auto p = v.begin(); p != v.end(); ++p)
        cout << *p << "\n";
}
```

65

Type inference

- ‘auto’ may be easier to type and read
- ‘auto’ declarations are more resilient to change
- ‘auto’ always resolves to a value type
- Use ‘auto &’ or ‘const auto &’ when required

66

Type inference

- The keyword decltype can be used to determine the type of expression at compile-time

```
int some_int;
decltype(some_int) other_integer_variable = 5;
```

- Prefer using auto for declaring variables
- Use decltype to specify a type for something that is not a variable, such as a return type.

67

Alternative function syntax

- In addition to

```
string to_string(int a); // prefix return type
```

- C++11 also allows

```
auto to_string(int a) -> string; // suffix return type
```

- This is mostly useful for deduced return types:

```
template<class T, class U>
auto product(const vector<T>& x, const vector<U>& y) -> decltype(T{} * U{} );
```

- We cannot use

```
template<class T, class U>
decltype(x*y) product(const vector<T>& x, const vector<U>& y);
```

- Because of scoping problems

68

Function return type deduction (C++14)

- C++11 allowed lambda functions to deduce the return type based on the type of the expression given to the return statement
- C++14 provides this ability to all functions
- It also introduces return type deduction for functions that are not of the form “return expression;”
- The function return type must be declared as ‘auto’, without a trailing return type specification

```
auto some_func() -> int { return 42; } // C++11
auto some_func() { return 42; } // C++14
```

69

Function return type deduction (C++14)

- C++11

```
[=]() -> int { return foo() * 42; };
[=] { return foo() * 42; }  // deduces "-> int"
```

- C++14

```
[=] {                           // deduces "-> int"
    while (something()) {
        if (expr) {
            return foo() * 42; // arbitrary control flow
        }
    }
    return bar(84);             // multiple returns (types must
                                // be the same)
};
```

70

Function return type deduction (C++14)

- **C++11**

```
int f() { return foo() * 42; }
auto f() -> int { return foo() * 42; }
```

- **C++14**

```
auto f() { return foo() * 42; } // deduces "-> int"
auto g() {                      // deduces "-> int"
    while (something()) {
        if (expr) {
            return foo() * 42;  // arbitrary control flow
        }
    }
    return bar(84);             // multiple returns (types must be the
                                // same)
}
```

71

decltype(auto) (C++14)

- **Given these functions:**

```
string  lookup1();
string& lookup2();
```

- **In C++11 we could write the following wrapper functions which remember to preserve the reference-ness of the return type:**

```
string  look_up_a_string_1() { return lookup1(); }
string& look_up_a_string_2() { return lookup2(); }
```

- **In C++14, we can automate that:**

```
decltype(auto) look_up_a_string_1() { return lookup1(); }
decltype(auto) look_up_a_string_2() { return lookup2(); }
```

72

Attributes

- C++11 provides a standardized syntax for compiler/tool extensions to the language.
- Such extensions were traditionally specified using #pragma directive or vendor-specific keywords like __attribute__ or __declspec.
- C++11 attributes are enclosed in double brackets: [[attr]].
- Attributes can be applied to various elements of source code (classes, functions, variables, etc.)

```
void f[[noreturn]]()  // f() will never return
{
    throw "error";  // OK
}
```

73

[[deprecated]] attribute (C++14)

- The [[deprecated]] attribute marks an entity deprecated
- This makes it still legal to use but notifies users on notice that its use is discouraged and may cause a warning message to be printed during compilation
- This attribute may be applied to the declaration of a class, a typedef-name, a variable, a non-static data member, a function, an enumeration, or a template specialization

74

[[fallthrough]] attribute (C++17)

- The [[fallthrough]] marks a fall-through case in a switch intentional

```
switch (n) {
   case 0: // no statement, hence intentional fallthrough
   case 1:
      doit(); // call doit, followed by next
      [[fallthrough]];
   case 4: // no warning on fallthrough
      domore();
}
```

75

[[nodiscard]] attribute (C++17)

- The [[nodiscard]] attribute triggers an error when its result is discarded

```
struct[[nodiscard]] Result{ };
Result foo(); // A function which returns a nodiscard Result

void bar()
{
    foo(); // Call foo and discard the result: trigger error
}
```

76

More attributes

- [[noreturn]]
- [[carries_dependency]]
- [[optimize_for_synchronized]]

- [[no_unique_address]] **(C++20)**

77

New string literals

- C++98 had two types of string literals:
  - `const char *str = "foo";`
  - `const wchar_t *wstr = L"bar";`

- wchar_t is an unspecified 'wide character' type
- The definition of the type char has been modified to be at least the size needed to store an eight-bit coding of UTF-8.

- C++11 adds two new character types: char16_t and char32_t.
- These are designed to store UTF-16 and UTF-32.

78

New string literals

- String literals for these encodings are created like this:

```
const char s1[] = u8"I'm a UTF-8 string.";
const char16_t s2[] = u"This is a UTF-16 string.";
const char32_t s3[] = U"This is a UTF-32 string.";
```

- C++11 provides a raw string literal:

```
const char s4[] = R"(The String Data \ Stuff " )";
```

- Prefixes can be combined:

```
const char32_t s5[] = UR"(This is a "raw UTF-32" string.)";
```

79

Digit separators (C++14)

- The single-quote character ' can now be used anywhere within a numeric literal for readability. It does not affect the numeric value.

```
auto one_million = 1'000'000;
auto pi = 3.14159'26535'89793;
```

80

Binary literals (C++14)

- C++14 adds support for binary literals:

```
auto a1 = 42;
auto a2 = 0x2A;
auto a3 = 0b101010;
```

- This works well in combination with the new ' digit separators, for example to separate nibbles or bytes:

```
auto a = 0b0100'0001;  // ASCII 'A'
```

81

User-defined literals

- C++ has always provided literals for a variety of built-in types

```
123    // int
1.2    // double
1.2F   // float
'a'    // char
1ULL   // unsigned long long
0xD0   // hexadecimal unsigned
"as"   // string
```

- However, in C++98 there are no literals for user-defined types

82

User-defined literals

- C++11 supports “user-defined literals”
- literal operators map literals with a given suffix to a desired type

```
constexpr complex<double> operator "" i(long double d)  // imaginary literal
{
    return{ 0,d };   // complex is a literal type
}

std::string operator "" s(const char* p, size_t n)   // std::string literal
{
    return string(p, n); // requires free store allocation
}

auto s = "hello"s; // s is an std::string
auto z = 2 + 1i;   // complex(2, 1)
```

83

User-defined literals

- Exercise: Duration class

- Create a Duration class that stores time in seconds
- Add user-defined literals to easily specify Durations in weeks, days, hours and minutes

84

84

User-defined literals

- **Exercise: Duration class**

```
class Duration
{
public:
    constexpr explicit Duration(long s) : seconds(s) {}
    explicit operator long() const { return seconds; }

    constexpr Duration operator + (const Duration& that) const;
private:
    long seconds;
};

int main()
{
    Duration d = 1_w + 2_d;
    std::cout << long(d) << " seconds" << std::endl;
}
```

85

85

3. Uniform Initialization

86

Uniform initialization

- C++98 offered several ways of initializing objects (depending on type and context)
- Question: What are all the different initialization types?

87

Uniform initialization

- C++98 offered several ways of initializing objects (depending on type and context)
- Mistakes could cause surprising errors and obscure error messages

```
// ok: initialize array variable
string a[] = { "foo", " bar" };

// error: initializer list for non-aggregate vector
vector<string> v = { "foo", " bar" };

void f(string a[]);
f({ "foo", " bar" }); // syntax error: block as argument
```

88

Uniform initialization

```
int a = 2;              // "assignment style"
int aa[] = { 2, 3 };    // assignment style with list
complex z(1, 2);        // "functional style" initialization
x = Ptr(y);             // "functional style" for
                        // conversion/cast/construction

int a(1);               // variable definition
int b();                // function declaration
int b(foo);             // variable definition or function
                        // declaration
```

89

Uniform initialization

- **C++11 solves all this by allowing {}-initializer lists for all initialization**

```
X x1 = X{ 1, 2 };
X x2 = { 1, 2 };
X x3{ 1, 2 };
X* p = new X{ 1, 2 };

struct D : X {
    D(int x, int y) : X{ x, y } { /* ... */ };
};
struct S {
    int a[3];
    S(int x, int y, int z) : a{ x, y ,z } { /* ... */ };
};
```

90

Initializer lists

- In C++98 (and C), arrays could be initialized using initializer lists

```
int a[] = { 1, 2, 3, 4, 5 };
```

- In C++11, initializer lists can do much more

```
vector<double> v = { 1.0, 2.0, 3.456, 99.99 };
list<pair<string, string>> languages = {
    { "Nygaard","Simula" },{ "Richards","BCPL" },{ "Ritchie","C" }
};
```

91

Initializer lists

- The mechanism for accepting a {}-list is a function (often a constructor) accepting an argument of type std::initializer_list<T>

```
void f(std::initializer_list<int> args)
{
    for (auto p = args.begin(); p != args.end(); ++p)
        std::cout << *p << "\n";
}

f({ 1,2 });
f({ 23,34,45,56,67,78,89 });
f({});  // empty list
```

92

Initializer lists

- Constructors and assignment operators of manny STL classes accept initializer lists
- Initializer lists can be used in range for loops

```
void f(initializer_list<int> args)
{
    for (auto n : args)
        std::cout << n << "\n";
}
```

93

Explicit conversion operators

- C++98 added the explicit keyword as a modifier on constructors to prevent single-argument constructors from being used as implicit type conversion operators.
- In C++11, the explicit keyword can now be applied to conversion operators

```
template <typename T, typename D = default_delete<T>>
class unique_ptr {
public:
    // ...
    // does *this hold a pointer != nullptr?
    explicit operator bool() const noexcept;
    // ...
};
```

94

Explicit conversion operators

```
void use(unique_ptr<Record> p, unique_ptr<int> q)
{
    if (!p) // OK: we want this use
        throw invalid_unique_ptr{};
    bool b = p; // error ; suspicious use
    int x = p + q; // error ; we definitly don't want this
}
```

- **Without 'explicit', the last two lines would have compiled**

95

In-class member initializers

- **C++11 allows initializers for non-static data members in the class declaration. For example:**

```
class A {
public:
    int a{ 7 };
    int b = 77;
};
```

- **For technical reasons, only '=' and '{}' style initializers can be used**

96

In-class member initializers

- Member initialization is done in declaration order
- Constructor initialization overrides in-class member initialization

```
class A {
public:
    A() : a{ 7 }, b{ 5 }, algorithm{ "MD5" }, state{ "Constructor run" } {}
    A(int a_val) : a{ a_val }, b{ 5 }, algorithm{ "MD5" }, state{ "Constructor run" } {}
    A(D d) : a{ 7 }, b{ g(d) }, algorithm{ "MD5" }, state{ "Constructor run" } {}
    // ...
private:
    int a, b;
    HashFunction algorithm;
    string state;
};
```

97

In-class member initializers

- This can now be simplified to

```
class A {
public:
    A() {}
    A(int a_val) :a{ a_val } {}
    A(D d) :b{ g(d) } {}
    // ...
private:
    int a{ 7 };
    int b{ 5 };
    HashFunction algorithm{ "MD5" };
    string state{ "Constructor run" };
};
```

98

Aggregate member initialization (C++14)

- C++11 aggregates did not allow member initializers
- C++14 relaxes this restriction, allowing aggregate initialization on such types. If the braced init list does not provide a value for that argument, the member initializer takes care of it.

```
struct Point {
    int x = 0;
    int y = 0;
};

Point p{ 1, 1 };  // allowed in C++14, not allowed in C++11
```

99

Tuple types

- The new std::tuple class implements an N-tuple
- N can range from 0 to a large value defined in <tuple>
- Element types of a tuple can be specified or deduced (using make_tuple())
- Elements can be access by (zero-based) index using get():

```
tuple<string, int> t2("Haddock", 123);
auto t = make_tuple(string("Cod"), 10, 1.23);
// t: tuple<string, int, double>
string s = get<0>(t);
int x = get<1>(t);
double d = get<2>(t);
```

100

Tuple types

- std::tie allows creating a tuple on the fly with references

```
int a = 1, b = 2;

const auto& [x, y] = std::tie(a, b); // x and y are of type int&

auto [z, w] = std::tie(a, b); // z and w are still of type int&

assert(&z == &a); // passes, because references to the same int
```

101

Structured Bindings (C++17)

- Structured Bindings do this as follows:

```
struct Data { double value; int size; };

Data d = { 1.0, 10 };
auto[ v, s ] = d; // Map deduced typed v and s onto Data
std::cout << "V = " << v << std::endl;
std::cout << "S = " << s << std::endl;
```

102

Structured Bindings (C++17)

- Structured Bindings with arrays

```
double v3[3] = { 1., 2., 3. };
auto& [ a, b, c ] = v3;
a = 10.; b = 100.; c = 1000.;
for ( double d : v3 )
    std::cout << d << std::endl;
```

103

Structured Bindings (C++17)

- Structured Bindings with range for and map

```
std::map<std::string, int> mymap;
for (const auto& [key, value] : mymap)
{
  std::cout << key << " ==> " << value << std::endl;
}
```

104

Designated Initializers (C++20)

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

Point p { .y = 2.0, .x = 1.0 }; // Wrong order,
                                // not allowed
Point q { .x = 2.0, .z = 1.0 }; // y initialized
                                // to zero
```

105

4. Containers and Algorithms

106

Containers and Algorithms

Introduction

- Modern C++ makes extensive use of standard library (STL) containers and algorithms
- Algorithms are universal
- Iterators decouple containers from algorithms

107

107

Containers and Algorithms

Containers

- Collection of elements
- Provide functions to access elements
- Implement common data structures like queues, lists and stacks
- Sequence containers, Associative containers, and Unordered associative containers
- Not (really) interested in internals

108

108

Containers and Algorithms

Container types

- **Sequence containers – Vector, Array, List, Deque, Forward_list**
- **Associative containers – Set, Map, Multiset, Multimap**
- **Unordered associative containers – Unordered_set, Unordered_map, Unordered_multiset, Unordered_multimap**
- **Container adaptors – Stack, Queue, Priority_queue**

109

109

Containers and Algorithms

Introduction

- **We have made extensive use of std::vector.**
- **std::vector is an example of a Standard Template Library container.**
- **We will now go into detail about how to choose and use the right container.**

110

110

Containers and Algorithms

std::vector

- std::vector is a dynamically sized array.
- It is by far the most used container.
- It is a good default choice. Even if other containers may perform better in theory under special circumstances, std::vector is in practice hard to beat for small (N<100) sizes in any case.

111

111

Containers and Algorithms

std::vector member functions

- push_back() to add an element to the end
- pop_back() to remove an element from the end
- front() returns first element
- back() returns last element
- size() current number of elements
- empty() returns true if it is empty
- resize() to resize the vector (pushing default-constructed elements if necessary)

112

112

Containers and Algorithms

std::vector member functions

- Note there is no push_front() function.
- Note there is no sort() function.
- This is very typical of STL: as a programmer, you are pushed in a certain direction to use the STL efficiently and correctly. You may not like this.
- push_front() would be very inefficient for a vector
- An efficient sorting algorithm exists that works on more than just vectors.

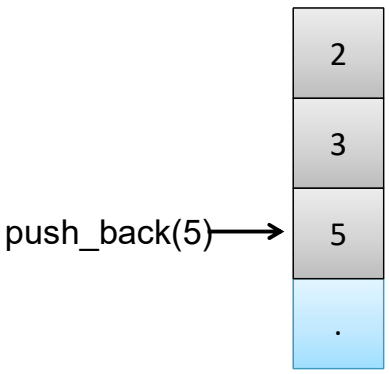
113

113

Containers and Algorithms

std::vector internals

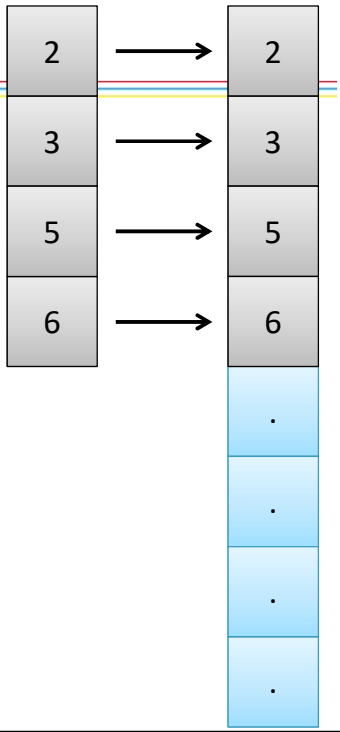
- A certain amount of memory is allocated.
- This memory fills up when you push_back elements.



114

114

Containers and Algorithms



std::vector internals

- If there is no more memory available, a larger piece of memory is allocated.
- The new piece of memory is typically twice as big.
- Then all elements are copied to make sure the elements remain consecutive.
- Oh no! This must be very inefficient!

115

115

Containers and Algorithms

- Actually: no. On average, push_back() is still O(1).
- The expected amount of work spent copying is of the same order as the amount spent push_backing.
- So don't worry.
- If you are still worried, you can call the member function reserve(n) to reserve memory for n elements (but don't confuse it with size(), which is the actual amount of items).
- You can also use a vector of pointers, if your objects are 'large'.

116

116

Containers and Algorithms

- O(1) is a mathematical notation called Big *O* notation
- It describes the complexity of computation for operations and algorithms
- Note: formally there is Space and Time complexity. We only focus on time complexity.

117

117

Containers and Algorithms

- Imagine the vector is *n* elements big
- The body of the loop runs for all elements.
- Its complexity is *O(n)*

```
// Vector can be any size
std::vector<int> numbers = {0, 1, 2, 3, 4, 5};
for(int numb : numbers)
{
    // Do something with numb
}
```

118

118

Containers and Algorithms

- Imagine the vector is *n* elements big
- The outer loop runs for all elements.
- The inner loop runs for half of the elements (on average)
- Its complexity is *O(n * n/2) ≈ O(n^2)*

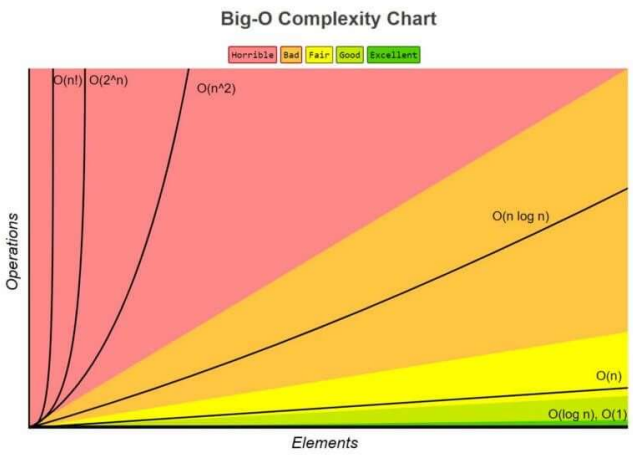
```
// Vector can be any size
std::vector<int> numbers = {0, 1, 2, 3, 4, 5};
for(int i=0; i < numbers.size(); ++i)
{
    for(int j=i; j < numbers.size(); ++j)
    {
        // Do something with numbers
    }
}
```

119

119

Containers and Algorithms

- Most common *Big-O* complexities
- All operations (access, search, insert, ….) on C++ containers have a certain complexity
- Useful when selecting most fitting container type



https://www.bigocheatsheet.com/

120

120

Containers and Algorithms

std::vector usage

- Iterating over a vector can be with simple indexing:

```
std::vector<int> v(10);
for ( unsigned int i = 0; i != v.size(); ++i )
{
  v[i] = 0;
}
```

- But you can also use 'iterators'…

121

121

Containers and Algorithms

Iterators

- Identify element in container
- Traverse elements in container
- Valid for limited time

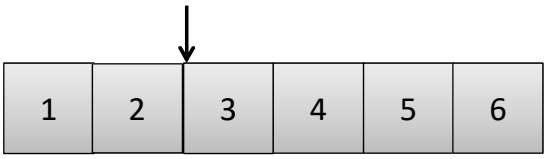
122

122

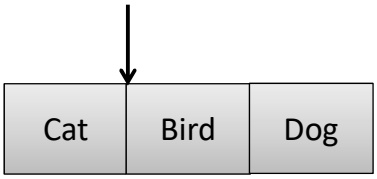
Containers and Algorithms

std::vector with Iterators

```
#include <algorithm>
std::vector<int> v = {1,2,3,4,5,6};
```




```
std::vector<std::string> v = {“Cat”, ”Bird”, ”Dog”};
```



123

123

Containers and Algorithms

std::vector usage

- Member function begin() returns an iterator pointing to the first element.
- Member function end() returns an iterator pointing to one-past-last element.
- Dereferencing iterators with * returns a reference to the element.
- (Note: the begin() function of a const vector returns a const_iterator)

124

124

Containers and Algorithms

std::vector usage

- Our loop can now look like this:

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

- What have we gained? (except a lot of extra typing)
- Iterators are a common method to access a range of elements in any type of container.
- In particular, it enables us to mix and match containers and algorithms (including our own).

125

125

Containers and Algorithms

std::vector usage

- Pay attention: iterators are high-performance and low-level.
- In particular, if you push_back, insert or remove an element, that invalidates all existing iterators to elements in that vector.
- These things can be found in any good C++ reference, e.g. www.cppreference.com or www.cplusplus.com.

126

126

Containers and Algorithms

std::vector usage

- In C++11, we can use the auto keyword:

```
for ( auto it = v.begin(); it != v.end(); ++it )
{
  *it = 0;
}
```

- It automatically derives the type of it from the result of v.begin(): std::vector<int>::iterator

127

127

Containers and Algorithms

std::vector usage

- In C++11, we can also use the ranged for loop:

```
for( int& i : v )
{
  i = 0;
}
```

- Mind the reference-indicating '&'.

128

128

Containers and Algorithms

std::vector insertion

- The insert() member function takes an iterator and an element argument.
- It inserts the element before the iterator.
- To insert an element at the front:

```
v.insert( v.begin(), 5 );
```

- To insert an element at the end:

```
v.insert( v.end(), 9 );
```

129

129

Containers and Algorithms

std::vector insertion

- Inserting a '3' before the first occurrence of '4':

```
std::vector<int>::iterator it;
for (it = v.begin(); it != v.end(); ++it )
{
  if ( *it == 4 )
    break;
}
v.insert( it, 3 );
```

- It is better to use a standard algorithm:

```
auto it = std::find( v.begin(), v.end(), 4 );
v.insert( it, 3 );
```

130

130

Containers and Algorithms

std::vector insertion

- Note how the standard find function is not a member of std::vector:

```
it = std::find( v.begin(), v.end(), 4 );
```

- The philosophy is that the std::find function should work with any sequence container that supports iterators, including your own.

131

131

Containers and Algorithms

std::vector removal

- To remove a single element, use the member function erase(), which takes an iterator:

```
myvector.erase( myvector.begin() + 3 ); // erase the 4th element
```

- To remove multiple elements, use erase() with two iterators:

```
v.erase( v.begin() + 1, v.end() ); // erase all except first
```

- To remove elements by value requires some attention.

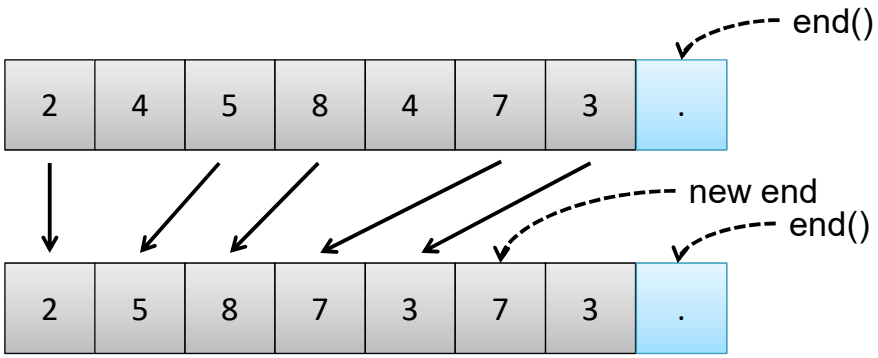
132

132

Containers and Algorithms

std::vector removal

- There is a standard algorithm to “remove” all occurrences of a value from a sequence container.
- But it only moves elements and returns an iterator to the new end.



133

133

Containers and Algorithms

std::vector removal

- You still need to erase the elements between the new end and the old end, using a member function.

```
std::vector<int>::iterator it = std::remove(v.begin(), v.end(), 4);
v.erase( it, v.end() );
```

- This is commonly written as:

```
v.erase( std::remove( v.begin(), v.end(), 4 ), v.end() );
```

134

134

Containers and Algorithms

Other sequence containers: deque

- std::deque is like std::vector, put supports efficient inserting at the beginning.
- It has member functions push_front() and pop_front().
- Often, a std::vector is better.

135

135

Containers and Algorithms

Other sequence containers: list

- std::list is a doubly linked list
- It supports efficient, O(1) inserting at any point.
- There is no random access, i.e. there is no list[n] access for the n-th element.
- Often, a std::vector is better.

136

136

Containers and Algorithms

Associative containers: set

- Is a container for storing unique elements
- The elements always remain sorted.
- Searching in a set has logarithmic complexity O(log(n)). Searching in a large set is more efficient than searching in a large vector.
- Inserting also has logarithmic complexity.

137

137

Containers and Algorithms

Associative containers: set

- Example:

```
#include <set>
std::set<int> s;
s.insert(3);
s.insert(5);
s.insert(3);
s.insert(2);
for ( auto it = s.begin(); it != s.end(); ++it )
{
  std::cout << *it << ",";
}
```

- Result: 2,3,5,

138

138

Containers and Algorithms

Associative containers: set

- To find out whether the set contains a value:

```
bool has_3 = ( s.find(3) != s.end() );
```

- Or:

```
bool has_4 = ( s.count(4) != 0 );
```

- Or:

```
bool has_4 = s.count(4);
```

- To remove by value:

```
s.erase(5);
```

139

139

Containers and Algorithms

Associative containers: map

- Is a container for storing key-value pairs.
- The elements are sorted by key.
- The key is unique.
- Searching in a map has logarithmic complexity O(log(n)).
- Inserting also has logarithmic complexity.

140

140

Containers and Algorithms

Associative containers: map

- **Example:**

```
#include <map>
std::map<std::string, int> m;
m.insert( std::make_pair( "two", 2 ) );
m.insert( std::make_pair( "four", 4 ) );
m.insert( std::make_pair( "five", 5 ) );
for ( std::map<std::string, int>::iterator it = m.begin(); it !=
m.end(); ++it )
{
  std::cout << it->first << " -> " << it->second << ", ";
}
```

- **Result:** `five -> 5, four -> 4, two -> 2,`

141

141

Containers and Algorithms

Associative containers: map

- **It is more common to use the [] operator:**

```
m["two"] = 2;
m["four"] = 4;
m["five"] = 5;
```

- **Be aware of the deeper meaning. The expression m["two"] tests whether a key-value pair with key "two" already exists, but creates one if it doesn't. The value is then set to 2.**
- **Therefore, [] is a modifying operator!**

142

142

Containers and Algorithms

Other associative containers

- **std::multiset is like a set, but can contain duplicate values.**
- **std::multimap is like a map, but can contain duplicate keys.**

143

143

Containers and Algorithms

Containers in memory

- **There are four main internal storage systems used**
  - Continuous block memory for std::vector and std::array
  - Black-Red tree (or binary search tree) for std::set and std::map
  - Doubly linked list for std::list
  - Hashtable for std::unordered_set and std::unordered_map

144

144

Containers and Algorithms

Continuous block memory for std::vector and std::array

1	5	3	2	8	5	3	2	8

- Random-access is *O(1)*
- Search is *O(n)*
- Insert is *O(n)*
- Append is *O(1)*
- Remove is *O(n)*

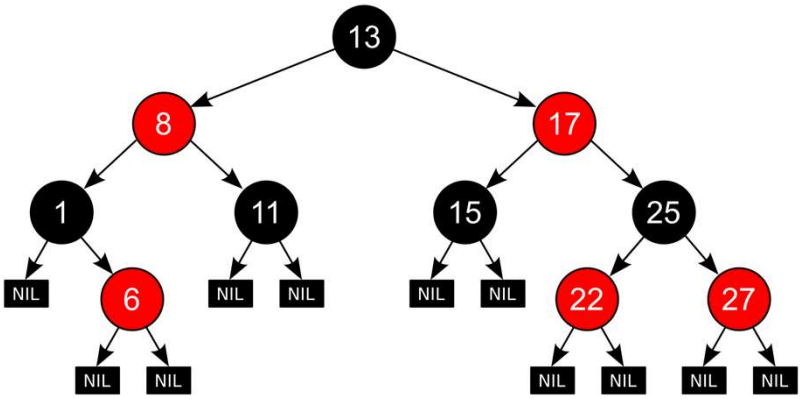
145

145

Containers and Algorithms

Black-Red tree (or binary search tree) for std::set and std::map

- Data representation
- Memory representation



13	8	17	1	11	15	25	NIL	6	NIL	NIL	NIL	NIL	22	27

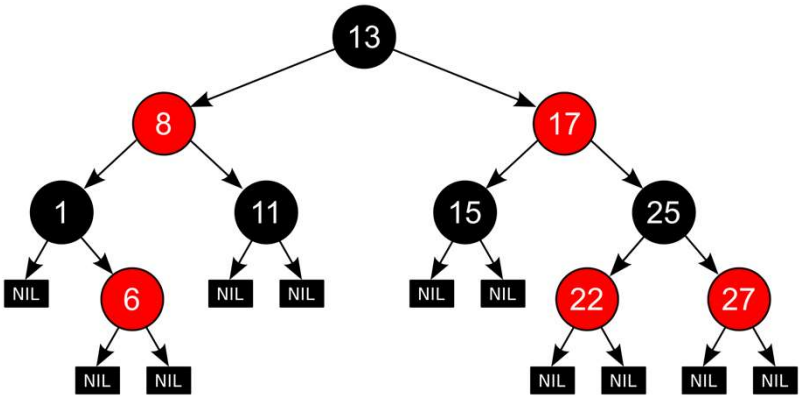
146

146

Containers and Algorithms

Black-Red tree (or binary search tree) for std::set and std::map

- Search is *O(log n)*
- Insert is *O(log n)*
- Remove is *O(log n)*



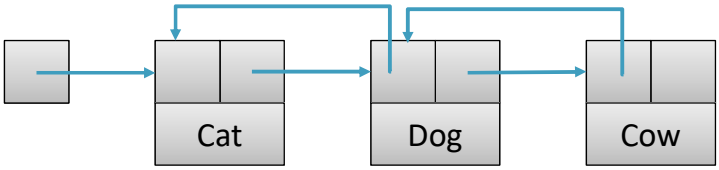
147

147

Containers and Algorithms

Doubly linked list for std::list uses

- Random-access is *O(n)*
- Search is *O(n)*
- Insert is *O(1)*
- Remove is *O(1)*

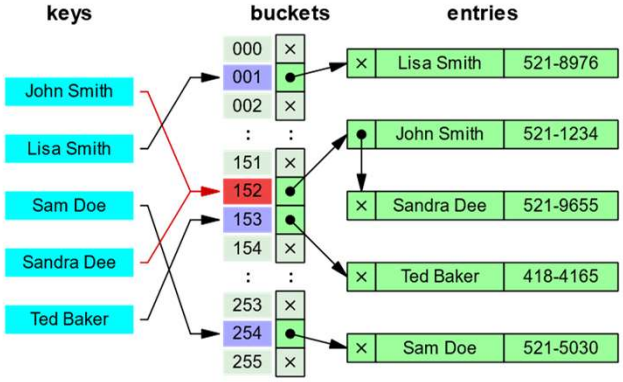


148

148

Containers and Algorithms

- Hashtable for unordered_map and unordered_set



https://en.wikipedia.org/wiki/Hash_table#Separate_chaining

149

149

Containers and Algorithms

- Hashtable for unordered_map and unordered_set
- Simplified implementation:

```
template<typename KEY, typename VAL>
class unordered_map {
private:
    list<pair<KEY, VAL>>* buckets;
    size_t bucket_count;
    size_t total_elements;
    float max_load_factor;
};
```

150

150

Containers and Algorithms

- Hashtable for unordered_map and unordered_set

- load_factor = total_elements / bucket_count
- When load_factor exceeds max_load_factor, a *rehash* is performed
- max_load_factor is 1.0 by default

- On average, the number of elements per bucket is <= 1

151

151

Containers and Algorithms

- Hashtable for unordered_map and unordered_set

- Search is *O(1)*
- Insert is *O(1)*
- Remove is *O(1)*

152

152

Containers and Algorithms

STL algorithms

- Algorithms of the STL usually work on iterators.
- This means they can be used on different types of containers.
- If you create your own container, and you provide iterators for your containers, you can use the STL algorithms on them.
- Use standard containers and algorithms if you can.
- As an example, we will look at std::sort.

153

153

Containers and Algorithms

Sort

- Use std::sort to sort a sequence container.

```
#include <algorithm>
std::vector<int> v = { 3, 2, 7, 5 }; // C++11 only
std::sort( v.begin(), v.end() );
for ( int i : v )
{
  std::cout << i << ", ";
}
```

- Output: 2, 3, 5, 7,

154

154

Containers and Algorithms

Sort

- You can also supply your own comparison function:

```
// custom Student comparison function
bool compareGrade( const Student& lhs, const Student& rhs )
{
  return lhs.getGrade() < rhs.getGrade();
}

void sortByGrade( std::vector<Student>& v )
{
  // sort with custom Student comparison function
  std::sort( v.begin(), v.end(), &compareGrade );
}
```

155

155

Containers and Algorithms

Lambda functions

- With C++11 you can also use an anonymous function or lambda function:

```
void sortByGrade( std::vector<Student>& v )
{
  std::sort( v.begin(),
             v.end(),
             [](const Student& lhs, const Student& rhs)
             {
                return lhs.getGrade() < rhs.getGrade();
             }
           );
}
```

- The [](){} syntax declares the lambda function.

156

156

Containers and Algorithms

Lambda functions

- You can also capture local variables with lambda functions:

```
double getGrade( const std::vector<Student>& v,
                 const std::string& name )
{
  auto it = std::find_if( v.begin(), v.end(),
                          [&name](const Student& s)
                          { return s.getName() == name; } );
  return it->getGrade();
}
```

157

157

Containers and Algorithms

Exercise 1: Use iterator

- Start with a list of names, stored as std::vector<std::string>
- Print the elements in reversed order
- Create a std::set from the vector
- Print all elements

158

158

Containers and Algorithms

Exercise 2: Merge two lists and sort

- Start with two lists of numbers, both stored as std::vector<int>
- Merge the lists and eliminate duplicate numbers.
- Print the elements of the merged list in sorted order.

159

159

Containers and Algorithms

Exercise 3: Sort by last name

- A Person is a struct containing two strings: firstName and lastName.
- Create a vector of Persons, then sort it by last name.

160

160

Containers and Algorithms

Exercise 4: Frequency table

- Write a program that reads text from a file, then displays the number of occurrences for each word.
- Use the following code to read the words:

```
std::ifstream file ( "test.txt" );
std::string word;
while (file >> word)
{
 ...
}
```

161

161

5. Operator Overloading

162

Operator Overloading

Assignment operator overloading

- **In C++, you can rewrite what happens when a class is assigned to another.**
- **This is useful for the same reasons for writing a copy constructor.**
- **If a class does not have an assignment operator, there will be a default assignment available, which will simply assign all member variables.**

163

163

Operator Overloading

- **Example:**

```
class Square
{
public:
  Square& operator=( const Square& source ); // assignment operator
private:
  double m_size;
};

Square& Square::operator=( const Square& source )
{
  m_size = source.m_size;
  return *this;       // returns reference to itself
}
```

164

164

Operator Overloading

- **Usage:**

```
int main()
{
  Square s1(1.1);
  Square s2(2.2);
  Square s3(s2); // calls copy constructor
  s1 = s2;       // calls assignment operator
  return 0;
}
```

165

165

Operator Overloading

Note that the assignment operator

```
Square& operator=( const Square& source ); // assignment operator
```

Has this syntax:

- **It returns a reference to itself. This enables us to do things like:**

```
s1 = s2 = s3;
```

- **It takes a const reference as the source argument**

166

166

Operator Overloading

- **Implementation example:**

```
School::School( const School& source ) // copy constructor
{
  // example: reuse assignment operator
  // this is less efficient, but prevents duplicate code
  *this = source;
}

School& School::operator=( const School& source )
{
  if ( this == &source )
  {
    return *this; // do nothing for self-assignment
  }
  // [...] (copy students one by one)
}
```

167

167

Operator Overloading

- **Note when the copy constructor is called:**

```
int main()
{
  Square s1( 3 );  // calls constructor
  Square s2( s1 ); // calls copy constructor
  Square s3 = s2;  // calls copy constructor (!)
  Square s4( 3 );  // calls constructor
  s4 = s3;         // calls assignment operator
  return 0;
}
```

168

168

Operator Overloading

- Note that the same rule applies for the assignment operator and the copy-constructor.
- If you delete one, you probably also want to delete the other.

```
School& operator=( const School& source ) = delete;
School( const School& source ) = delete;
```

- Alternatively, to avoid problems with dynamic memory management and copying objects, use smart pointers (later).

169

169

Operator Overloading

Operator overloading

- It is a feature of C++ that operators likes +, *, ==, << can be overloaded for classes, i.e. given a special meaning.
- We have seen operator overloading in practice with the overloading of the assignment operator.
- The idea is that the code that uses these classes can be concise, clear and correct.

170

170

Operator Overloading

- **Operator overloading is sometimes criticized:**
  - Object a = b + c;

    What is the meaning? It depends on the context of 'Object'.
- **Always work with the Principle of Least Surprise!**
- **Don't use it if you cannot guarantee there will be no surprises.**

171

171

Operator Overloading

- **As usual, C++ is extremely powerful: it can be persuaded to do anything you like.**
- **An example of the possibilities of operator overloading (or: how not to use it):**

```
#include "analogliterals.hpp"

int vol = ( o-------------o
            |L             \
            | L             \
            |  L             \
            |   o-------------o
            |   !             !
            !   !             !
            o   |             !
             L  |             !
              L |             !
               L|             !
                o-------------o ).volume;
```

172

172

Operator Overloading

Operator overloading example: the Duration class

```
class Duration
{
public:
  Duration( int hours, int minutes ); // constructor
  void print() const;
private:
  int m_hours;
  int m_minutes;
};
```

173

173

Operator Overloading

Implementation:

```
Duration::Duration( int hours, int minutes )  // constructor
  : m_hours( hours ),
    m_minutes( minutes )
{
}

void Duration::print() const
{
  std::cout << m_hours << " hours, ";
  std::cout << m_minutes << " minutes";
  std::cout << std::endl;
}
```

174

174

Operator Overloading

Usage:

```
int main()
{
  Duration t( 2, 30 ); // constructs a duration
  t.print();       // prints: 2 hours, 30 minutes
  return 0;
}
```

175

175

Operator Overloading

It would make sense to add durations:

```
int main()
{
  Duration t1( 2, 30 );
  Duration t2( 1, 45 );
  Duration t3 = t1 + t2;
  t3.print();         // prints: 4 hours, 15 minutes
  return 0;
}
```

176

176

Operator Overloading

- Therefore, we overload the + operator

```
class Duration
{
  /*...*/
 Duration operator+( const Duration& rhs ) const;
};

Duration Duration::operator+( const Duration& rhs ) const
{
 Duration sum( m_hours + rhs.m_hours,
               m_minutes + rhs.m_minutes );
  sum.m_hours += sum.m_minutes / 60;
  sum.m_minutes %= 60;
  return sum;
}
```

177

177

Operator Overloading

Note that the overloaded operator

```
Duration operator+( const Duration& rhs ) const
```

Has this syntax:

- It takes one parameter: the right-hand side.
- The left-hand side is the this object itself.
- It can be const, since the left-hand side is not changed.
- It returns the sum duration.

178

178

Operator Overloading

- **Likewise, we can overload the * operator to multiply duration with a factor.**

```
Duration Duration::operator*( double factor ) const
{
  Duration sum( m_hours * factor, m_minutes * factor );
  sum.m_hours += sum.m_minutes / 60;
  sum.m_minutes %= 60;
  return sum;
}
```

179

179

Operator Overloading

- **Usage:**

```
int main()
{
  Duration t1( 1, 45 );
  Duration t2 = t1 * 2;
  t2.print(); // prints: 3 hours, 30 minutes
  return 0;
}
```

180

180

Operator Overloading

- But there is a problem:

```
int main()
{
  Duration t1( 1, 45 );
  Duration t2 = 2 * t1;  // compiler error: '2' is not an object
  t2.print();
  return 0;
}
```

- We cannot freely reverse the order, which is not friendly for anyone using our class.

181

181

Operator Overloading

- The solution is to overload the operator not as a member function, but as nonmember (ordinary) function.

```
Duration operator*( double factor, const Duration& t )
{
  Duration sum( t.m_hours * factor, t.m_minutes * factor );
  sum.m_hours += sum.m_minutes / 60;
  sum.m_minutes %= 60;
  return sum;
}
```

- This poses another problem: we cannot access t.m_hours since it is a private member!

182

182

Operator Overloading

- We can declare the function a friend of the Duration class:

```
class Duration
{
public:
  Duration( int hours, int minutes );
  Duration operator+( const Duration& rhs ) const;
  Duration operator*( double factor ) const;
  void print() const;
  friend Duration operator*( double factor, const Duration& t );
private:
  int m_hours;
  int m_minutes;
};
```

183

183

Operator Overloading

- In this case, it is better to write the function in terms of the reversed overloaded operator (why?):

```
Duration operator*( double factor, const Duration& t )
{
  return t * factor;
}
```

184

184

Operator Overloading

Overloading the << operator for output streams

- **We already have a print function**

```
void Duration::print() const
```

- **This function only prints to std::cout. It would be nice to print to any kind of output stream (for example, a file stream).**
- **Therefore, we overload the << operator for output stream (ostream) and Duration.**

185

185

Operator Overloading

- **Implementation**

```
void operator<<( std::ostream& os, const Duration& t )
{
  os << t.m_hours << " hours, ";
  os << t.m_minutes << " minutes";
}
```

- **Make it a friend of Duration**

```
class Duration
{
  /*...*/
  friend void operator<<( std::ostream& os, const Duration& t );
};
```

186

186

Operator Overloading

- **Usage:**

```
int main()
{
  Duration t( 1, 45 );
  std::ofstream outfile ("test.txt"); // opens a file for writing
  outfile << t;    // calls overloaded operator
  std::cout << t;  // calls overloaded operator
  return 0;
}
```

187

187

Operator Overloading

- **One final issue:**

```
int main()
{
  Duration t( 1, 45 );
  std::cout << "The duration is " << t << std::endl;
  return 0;
}
```

- **This gives a compile error, because we have not provided a way to chain our operator. The order is left to right:**

```
((std::cout << "The duration is ") << t) << std::endl;
```

188

188

Operator Overloading

- **Solution: return the stream!**

```
std::ostream& operator<<( std::ostream& os, const Duration& t )
{
  os << t.m_hours << " hours, ";
  os << t.m_minutes << " minutes";
  return os; // enables chaining
}
```

189

189

Operator Overloading

- **So now, you finally know how the magic works!**

```
std::cout << 3 << " times " << duration << std::endl;
```

- **There simply exist overloaded operators << for everything:**

```
std::ostream& operator<<( std::ostream& os, int i );
std::ostream& operator<<( std::ostream& os, const std::string& s );
std::ostream& operator<<( std::ostream& os, const Duration& t );
std::ostream& operator<<( std::ostream& os, /* whatever type std::endl
is*/ );
```

190

190

Operator Overloading

```
void foo()
 {
    Duration a;
    Duration b;
    Duration c;

    c = a + b; // really is ...
    c.operator=( a.operator+(b) ); // ... this
}
```

191

191

Operator Overloading

Exercise: create a Matrix class

- **A matrix is a rectangular arrangement of real numbers.**

$$\mathbf{A} = \begin{bmatrix} 9 & 13 & 6 \\ 1 & 11 & 7 \\ 3 & 9 & 2 \\ 6 & 0 & 7 \end{bmatrix}$$

192

192

Operator Overloading

- **Matrices of the same size can be added.**

$$\begin{bmatrix} 9 & 1 \\ 13 & 11 \end{bmatrix} + \begin{bmatrix} 3 & 6 \\ 9 & 0 \end{bmatrix} = \begin{bmatrix} 12 & 7 \\ 22 & 11 \end{bmatrix}$$

193

193

Operator Overloading

- **Matrices can be multiplied by a number.**

$$3 \begin{bmatrix} 1.5 & 1 \\ 2 & -1 \end{bmatrix} = \begin{bmatrix} 4.5 & 3 \\ 6 & -3 \end{bmatrix}$$

194

194

Operator Overloading

Exercise: create a Matrix class that:

- Can be constructed with two parameters, indicating number of rows and columns.
- Can be copy-constructed and assigned.
- Allows getting and setting elements.
- Overloads operators + for adding, * for multiplication with a number, << for printing the matrix.
- For simplicity, start indexing at 0.

195

195

Operator Overloading

Exercise: create a Matrix class

This exercise teaches you how to build a well-designed class. The client code (where the class is used) naturally turns out to be clear, concise and correct.

196

196

6. Dynamic Memory Allocation

197

Dynamic Memory Allocation

Dynamic memory allocation

- Besides creating named variables, we can also store data by dynamically allocating it on the heap.
- The heap is a place in memory, where we can allocate memory for an indefinite lifetime.
- (The stack is where the memory is automatically released with the 'last in first out' principle.)
- Memory management is manual: we are responsible to release it when we no longer need it.

198

198

Dynamic Memory Allocation

Dynamic memory allocation

- The new operator is used to allocate memory.
- We specify the type of the object that we wish to create.
- The right amount of space is allocated, and the constructor for the object is called.
- The delete operator is used to deallocate memory.
- The right amount of space is deallocated, and the destructor for the object is called.

199

199

Dynamic Memory Allocation

- Example:

```
int main()
{
  Square* s = new Square(1.5);
  std::cout << (*s).area() << std::endl;
  std::cout << s->area()  << std::endl;  // alternative syntax
  delete s;
  return 0;
}
```

- The constructor `Square::Square(double size)` is called with `size = 1.5`.
- Don't forget: every time you use new you also need delete!

200

200

BB0

Dynamic Memory Allocation

- *Don't forget: every time you use new you also need delete!*

- **Actually no: in C++ we use delete as little as possible, and we can strive to write programs that do not contain any delete at all! (we will come back to this)**

201

201

Dynamic Memory Allocation

- **Further examples with plain old data:**

```
int* p1 = new int;    // value not initialized
int* p2 = new int();  // value set to 0
int* p3 = new int(3); // value set to 3
bool* p4 = new bool;
delete p1;
delete p2;
delete p3;
delete p4;
```

202

202

Slide 201

BB0 Consider removing this slide, seems more like something for Foundation training.
Bart Beumer, 2022-08-24T10:06:37.918

Dynamic Memory Allocation

- In contrast to C, there is no need to check whether the pointer is 0 after allocation:

```
Square* s = new Square(1.5);
if ( s == 0 ) // wrong: s can never be zero
{
  std::cout << "No more memory" << std::endl;
}
```

- The operator new will never return 0, but throw an exception when out of memory.

203

203

Dynamic Memory Allocation

Exercise: students and a school

Students go to a school, which has three courses: “C++”, “OO Design”, and “Scrum”.

- Create a school class. It maintains a list of all students and also for each course a list of enrolled students.
- Add a function in the school to create a new student.
- Add a function to add a student to a course.
- Add a function to list all students of a given course.

These instructions are not very specific on purpose!

204

204

Dynamic Memory Allocation

- **Check your code: can a school be copied?**

```
class School
{
  std::vector<Student*> students;
};
School school2 = school1;   // copies all student pointers, trouble?
```

- **Better decide what you want when you create the class.**

205

205

Dynamic Memory Allocation

- **Create a copy constructor to make sure your class behaves exactly like you expect it to behave.**
- **This is usually necessary whenever you have pointer member variables.**
- **Create a copy constructor only if you think it makes sense to directly copy the object.**

206

206

Dynamic Memory Allocation

Two types of objects

- **In practice we see a distinction between “value types” and “reference types”**
- **Value type objects are essentially data objects, that can be copied and assigned. Examples: money, string.**
- **Reference type objects are unique and persistent entities in your program. They have no copy constructor, use inheritance and are often dynamically allocated. Examples: widget, student.**

207

207

Dynamic Memory Allocation

- **Rule of Three:**
  - If a class requires a user-defined destructor, a user-defined copy constructor, or a user-defined copy assignment operator, it almost certainly requires all three.
- **Rule of Five (C++11):**
  - Because the presence of a user-defined destructor, copy-constructor, or copy-assignment operator prevents implicit definition of the move constructor and the move assignment operator, any class for which move semantics are desirable, has to declare all five special member functions:

208

Dynamic Memory Allocation

- **Rule of Zero:**
  - Classes that have custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership (which follows from the Single Responsibility Principle).
  - Other classes should not have custom destructors, copy/move constructors or copy/move assignment operators.

209

7. Move semantics

210

Rvalue references and move constructors

- **In C++98:**
  - References to non-const can bind to lvalues
  - references to const can bind to lvalues or rvalues
  - Nothing can bind to a non-const rvalue (i.e. modify a temporary object).
- **For safety – so you can't do this:**

```
void incr(int& a) { ++a; }
int i = 0;
incr(i);    // i becomes 1
incr(0);    // error: 0 in not an lvalue
```

211

Rvalue references and move constructors

- **C++11 added the 'rvalue reference'**
- **An rvalue reference can bind to an rvalue (not to an lvalue)**
- **Rvalue references are declared with '&&'**

```
X a;
X f();
X& r1 = a;      // bind r1 to a (an lvalue)
X& r2 = f();    // error: f() is an rvalue
X&& rr1 = f();  // bind rr1 to temporary
X&& rr2 = a;    // error: bind a is an lvalue
```

212

Rvalue references and move constructors

- **Consider this C++98-style swap function:**

```
template<class T> swap(T& a, T& b)
{
    T tmp(a);   // now we have two copies of a
    a = b;      // now we have two copies of b
    b = tmp;    // now we have two copies of tmp (aka a)
}
```

- **Expensive (i.e. slow) if T is big**

213

Rvalue references and move constructors

- **In C++11, we can define "move constructors" and "move assignment"**

```
template<class T> class vector {
    vector(const vector&);              // copy constructor
    vector(vector&&);                   // move constructor
    vector& operator=(const vector&);   // copy assignment
    vector& operator=(vector&&);        // move assignment
};
```

- **Note: arguments are non-const && (must be writeable)**
- **Compiler knows which one to call (based on lvalue or rvalue argument)**

214

Rvalue references and move constructors

- **C++11-style swap function:**

```
template<class T>
void swap(T& a, T& b)
{
    T tmp = std::move(a);     // may invalidate a
    a = std::move(b);         // may invalidate b
    b = std::move(tmp);       // may invalidate tmp
}
```

- **std::move does not move anything – it casts to an rvalue reference**

215

Rvalue references and move constructors

```
template<class T>
class Matrix {
    std::array<int, 2> dim;
    T* elem;
public:
    Matrix(int d1, int d2) : dim{ d1,d2 }, elem{ new T[d1*d2] } {}
    int size() const { return dim[0]*dim[1]; }
    Matrix(const Matrix&);            // copy constructor
    Matrix(Matrix&&);                 // move constructor
    Matrix& operator=(const Matrix&); // copy assignment operator
    Matrix& operator=(Matrix&&);      // move assignment operator
    ~Matrix();
};
```

216

Rvalue references and move constructors

- The argument of a move operation must always be left in a state that the destructor can handle (preferably efficiently)

```
template<class T>
Matrix<T>::Matrix(Matrix&& a)
  : dim{ a.dim }, elem{ a.elem }
{
    a.dim = { 0,0 };
    a.elem = nullptr;
}
```

217

Rvalue references and move constructors

- Move assignment can often use std::swap

```
template<class T>
Matrix<T>& Matrix<T>::operator=(Matrix&& a)
{
    swap(dim, a.dim);
    swap(elem, a.elem);
    return *this;
}
```

- When not using swap, make sure that the argument is left in a valid state!

218

Rvalue references and move constructors

- **Exercise: String class**

- **Write a simple String class**
- **Use a dynamically allocated array of characters to store the data**
- **Implement *copy* and *move* semantics**

Rvalue references and move constructors

- **Large objects can now be returned efficiently:**

```
template<class T>
Matrix operator+(const Matrix& a, const Matrix& b)
{
    if (a.dim[0] != b.dim[0] || a.dim[1] != b.dim[1])
        throw std::runtime_error("unequal Matrix sizes in +");
    Matrix res{ a.dim[0],a.dim[1] };
    for (int i = 0; i < a.size(); ++i)
        res.elem[i] = a.elem[i] + b.elem[i];
    return res; // return value will be moved into assignment result
}
```

Rvalue references and move constructors

- Rvalue references can be used with template parameters to provide perfect forwarding.

```
template<typename T, typename U>
std::pair<T, U> make_pair_wrapper(T&& t, U&& u)
{
    return std::make_pair(std::forward<T>(t), std::forward<U>(u));
}
```

- Perfect forwarding allows us to preserve an argument's value category (lvalue/rvalue) and const/volatile modifiers.
- Perfect forwarding is performed in two steps: receive a forwarding reference (also known as universal reference), then forward it using std::forward.

221

8. Smart Pointers

222

Smart Pointers

Rationale

- **There is an important complication with exceptions and dynamic memory management:**

```
Student* s = 0;
try
{
  s = new Student;
  foo( s ); // may throw
  delete s;
}
catch ( std::exception& e )
{
  std::cout << e.what() << std::endl;
}
```

223

223

Smart Pointers

- **Corrected version:**

```
Student* s = 0;
try
{
  s = new Student;
  foo( s ); // may throw
  delete s;
}
catch ( std::exception& e )
{
  delete s; // to prevent memory leak
  std::cout << e.what() << std::endl;
}
```

224

224

Smart Pointers

- Let's be fair: exceptions and manual resource management gives trouble.
- This is an opportunity to make the switch to modern C++. Using proper methods, garbage collecting becomes automatic.
- The idea is to solve the problem of resource management (including memory, open files, etc.) right when the resource is acquired.
- This technique is also called RAII (Resource Acquisition Is Initialization).

225

225

Smart Pointers

- Every resource is wrapped up in an object. The destructor of the object will release the resource.
- An example is the use of smart pointers.
- A smart pointer is a pointer with explicit ownership. The smart pointer object will take caring of destroying the object.

226

226

Smart Pointers

Simple smart pointer

- An example is std::unique_ptr. You can use it when you want to use a pointer, but also implicate ownership.
- Example:

```
#include <memory>
void foo()
{
  std::unique_ptr<Student> p( new Student("Jack") );
  /*...*/
  // Student is automatically destroyed
}
```

227

227

Smart Pointers

- This will solve our problem with throwing exceptions:

```
try
{
  std::unique_ptr<Student> p( new Student("Jack") );
  foo( *p ); // may throw
  // delete not needed
}
catch ( std::exception& e )
{
  std::cout << e.what() << std::endl;
}
```

228

228

Smart Pointers

- **Note that it is guaranteed that only one unique_ptr will be the owner of the student.**
- **Note that you need std::move to transfer ownership:**

```
std::unique_ptr<Student> p1( new Student("Jack") );
std::unique_ptr<Student> p2;
assert( p1 );                 // p1 is not empty
assert( !p2 );                // p2 is empty
p1 = p2;                      // compile-time error!
p2 = std::move(p1);           // ok
assert( !p1 );
assert( p2 );                 // student has moved from p1 to p2!
```

229

229

Smart Pointers

- **You can also use unique_ptr in containers:**

```
std::vector<std::unique_ptr<Student> > // ok
```

- **Using a smart pointer also clarifies your design:**
  - a normal pointer indicates simple association
  - a unique_ptr indicates a life cycle dependency (composition)

230

230

Smart Pointers

Shared pointers

- **Sometimes you want to share ownership of an object.**
- **The object should be destroyed if the last owner is removed.**
- **For this purpose, we have std::shared_ptr**
- **It uses reference counting internally.**

231

231

Smart Pointers

- **Example:**

```
std::shared_ptr<Student> p1( new Student );
std::shared_ptr<Student> p2 = p1; //share ownership
p1.reset(); // release 1
p2.reset(); // release 2: Student is deleted
```

- **It is of course more common to have shared pointers as object members.**
- **Shared pointers can also be used in containers:**

```
std::vector< std::shared_ptr<Student> > v; // okay
```

232

232

Smart Pointers

Exercise: Unique ownership

- **Create a School class.**
- **The School has a function to add a Student.**
- **The School has a function to remove and return a Student.**
- **Use a unique_ptr in the class and in both functions to indicate (transfer of) ownership**
- **It is best practice to wrap at construction:**

```
std::unique_ptr<Student> p1( new Student );
```

233

General-purpose smart pointers

- **Before 2010: Lots of libraries had implemented their own "smart" pointers:**
  - std::auto_ptr was "broken"
  - boost::scoped_ptr, shared_ptr, unique_ptr, weak_ptr
  - Loki::SmartPtr, StrongPtr
  - Poco::SharedPtr
  - Vtk::SmartPointer
- **Most often using reference counting, sometimes with associated weak-like ptr**

General-purpose smart pointers

- **Before 2010:**
  - Atomic reference count updates are ***hard***
  - A smart pointer does still not solve ***all*** memory problems
  - Transfer of ownership is ***hard*** *(without move semantics)*

235

General-purpose smart pointers

- **C++11 introduces three types of smart pointers:**
  - unique_ptr,
  - shared_ptr
  - weak_ptr

- **A unique_ptr is a container for a raw pointer, it deletes the raw pointer when out of scope**
- **A shared_ptr is a container for a raw pointer, it deletes the raw pointer when the last copy of the shared_ptr is destroyed**
- **A weak_ptr is a container for a raw pointer, it becomes empty when all shared_ptr instances have been destroyed.**

236

General-purpose smart pointers

- unique_ptr cannot be copied because its copy constructor and copy assignment operator are explicitly deleted
- std::move can be used to transfer ownership of the contained pointer to another unique_ptr

```
std::unique_ptr<int> p1(new int(5));
std::unique_ptr<int> p2 = p1; // Compiler error.
std::unique_ptr<int> p3 = std::move(p1); // p3 owns the
                                         // memory, p1 is invalid
p3.reset(); // deletes the memory
p1.reset(); // does nothing
```

237

General-purpose smart pointers

- shared_ptr uses reference counting to manage ownership

```
std::shared_ptr<int> p1(new int(5));
std::shared_ptr<int> p2 = p1; // both now own the memory

p1.reset(); // memory still exists because of p2
p2.reset(); // deletes the memory, since no owners remain
```

238

General-purpose smart pointers

- **Weak pointers can be used to break loops among shared pointers**

```
std::shared_ptr<int> p1(new int(5));
std::weak_ptr<int> wp1 = p1; // p1 owns the memory.
{
    std::shared_ptr<int> p2 = wp1.lock(); // p1 and p2 own the memory.
    if (p2)
    // As p2 is initialized from a weak pointer, you have to check if
    // the memory still exists!
    {
        // do something with p2
    }
} // p2 is destroyed. Memory is owned by p1.
p1.reset(); //Memory is deleted.
```

239

General-purpose smart pointers

- **Smart pointers overload operator*, operator->, etc.**
- **Shared pointers can be constructed using make_shared**

```
auto sp = std::make_shared<int>(12);
```

- **Shared pointers are not the answer to all your memory management problems**
- **When you have a choice:**
  - Prefer unique_ptr to shared_ptr
  - Prefer ordinary scoped objects to objects on the heap owned by a unique_ptr

240

General-purpose smart pointers

- Exercise: factory method

- Write a function that returns a unique_ptr<Person>
- Call the function. Throw an exception.
- Verify that the destructor of Person is called
- Does this also work for a shared_ptr?

241

241

9. Interfaces

242

Interfaces

What is an interface in programming?

- **Simplest form is access to a class**
- **All public methods and variables are part of its interface**

```
class Remote{
    public:
        Remote();
        virtual ~Remote();
        bool pressbutton(int button_index);
    private:
        bool setChannel(int channel);
        void resetMenu();
        void resetSubmenu();
        void powerToggle();
        int m_last_button_pressed;
        int m_battery_percentage;
};
```

243

243

Interfaces

Why do we use interfaces in programming?

- **High cohesion & low coupling (part of GRASP)**
- **“talking to an interface”**

244

244

Interfaces

Low coupling

- **Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.**
- ***Low coupling* is an evaluative pattern that dictates how to assign responsibilities for the following benefits:**
  - less dependency between the classes
  - change in one class having a lower impact on other classes
  - higher reuse potential

245

245

Interfaces

High cohesion

- ***High cohesion* is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of low coupling. High cohesion means that the responsibilities of a given set of elements are strongly related and highly focused a specific topic.**
  - Logical grouping
  - Easy to understand from outside

246

246

Interfaces in C++

- Not formally supported in C++ (due to multiple inheritance)
- ‘Interface’ in C++ is actually an Abstract Base Class
- C# and Java do have formal implementations
- This means nothing about interfaces is enforced in C++. Any class can be used as an interface.

247

247

Interfaces

Properties of an Interface:

- Every function is pure virtual
- No constructors
- No implementation (excluding destructor)
- No member variables

```
class Remote{
    public:
        virtual ~Remote(){};
        virtual std::string status() const =0;
        virtual void play()=0;
        virtual void stop()=0;
};
```

248

248

Interfaces

```
class TVRemote : public Remote {
    public:
        TVRemote(const std::string& name);
        virtual ~TVRemote() = default;
        TVRemote(const TVRemote& src) = default;

        virtual std::string status() const;
        virtual void play();
        virtual void stop();

        void rewind();
        void forward();
    private:
        std::string m_name;
        std::string m_state;
};
```

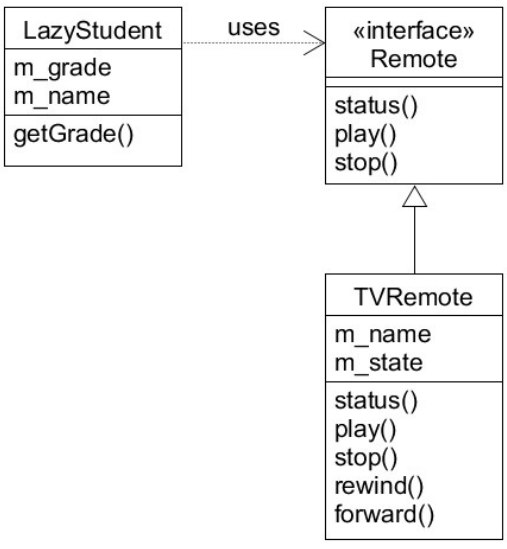
```
class Remote {
    public:
        virtual ~Remote(){};
        virtual std::string status() const =0;
        virtual void play()=0;
        virtual void stop()=0;
};
```

249

249

Interfaces

- **Can be different than public class interface**
- **The LazyStudent could use any remote that adheres the interface**



250

250

Interfaces

What did we achieve?

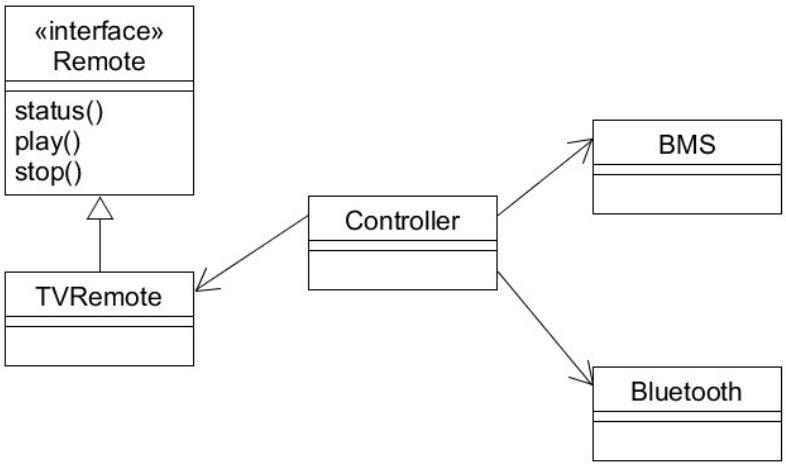
- Even more files
- More defined usage -> forced low coupling
- Intent is clear and easy to read

251

251

Interfaces

- Change is easy when interface stays the same
- High cohesion
- Low coupling

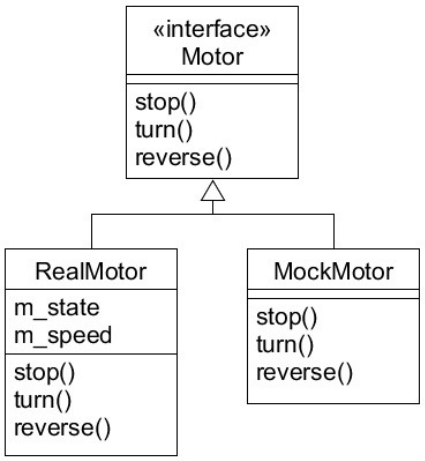


252

252

Interfaces

- Easy solution to abstract away hardware
- More code is testable



253

253

Interfaces

Interface is also a concept

- Interface to a class
- To a module
- To a library or application

254

254

Interface

How not to use

- Do not use interface for everything
- Avoid very big interfaces

```
class Remote {
    public:
        virtual ~Remote() = default;
        virtual std::string status() const =0;
        virtual void play()=0;
        virtual void stop()=0;
};
```

Advised:

- Every function is pure virtual
- No constructor
- No implementation (excluding destructor)
- No member variables

255

255

Interfaces

Exercise:

- Design a program and determine the location for interfaces
- Consider a Robot Vacuum that needs new software/architecture
- The Robot consists of vacuum unit, sensors, motors and a way of navigating.
- We want some flexibility of those components
- Consider testability

256

256

Interfaces

Exercise:

- Create a design which includes the class list below
- You are allowed to add extra classes

- Navigation, distance sensor, collision sensor, motor controller, BMS, display, vacuum unit, vacuum motor
- Give your robot a name

257

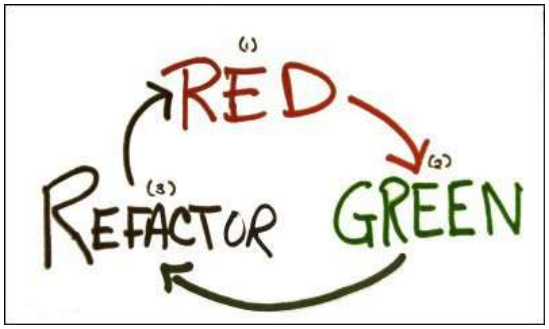
257

10. TDD in C++

258

TDD in C++

Red-Green-Refactor loop



Restricted version by Robert C. Martin

1. **Don't write any production code until you have written a failing test**
2. **Don't write more of a unit test than is sufficient to fail or fail to compile**
3. **Don't write any more production code than is sufficient to pass the failing test**

[1] Robert C. Martin & Micah Martin – Agile Principles, Patterns and Practices in C# (2007)

259

TDD in C++

- **Year that is divisible by 4**
- **Except centuries where year is not divisible by 400**

260

TDD in C++

- **Lets start with the basis test**

```
#include "gtest/gtest.h"

TEST(LeapYearProgramTest, is_leap_year_input) {
    EXPECT_EQ (true, is_leap_year(2020));
}
```

```
bool is_leap_year(int year)
{
    return true;
}
```

261

261

TDD in C++

- **Second assert**

```
#include "gtest/gtest.h"

TEST(LeapYearProgramTest, is_leap_year_input) {
    EXPECT_EQ (true, is_leap_year(2020));
    EXPECT_EQ (false, is_leap_year(2021));
}
```

```
bool is_leap_year(int year)
{
    if(year % 4 != 0)
    {
        return false;
    }
    return true;
}
```

262

262

TDD in C++

- **Third assert**

```
#include "gtest/gtest.h"

TEST(LeapYearProgramTest, is_leap_year_input) {
    EXPECT_EQ (true, is_leap_year(2000));
    EXPECT_EQ (true, is_leap_year(2020));
    EXPECT_EQ (false, is_leap_year(2021));
}
```

```
bool is_leap_year(int year)
{
    if(year % 4 != 0)
    {
        return false;
    }
    else if(year % 400 == 0)
    {
        return true;
    }
    return true;
}
```

263

263

TDD in C++

- **Forth and last assert**

```
#include "gtest/gtest.h"

TEST(LeapYearProgramTest, is_leap_year_input) {
    EXPECT_EQ (true, is_leap_year(2000));
    EXPECT_EQ (true, is_leap_year(2020));
    EXPECT_EQ (false, is_leap_year(2021));
    EXPECT_EQ (false, is_leap_year(2100));
}
```

```
bool is_leap_year(int year)
{
    if(year % 4 != 0)
    {
        return false;
    }
    else if(year % 400 == 0)
    {
        return true;
    }
    else if(year % 100 == 0)
    {
        return false;
    }
    return true;
}
```

264

264

TDD in C++

- Red -> Green -> Refactor
- Nice concise test with smallest number of cases

265

265

TDD in C++

- C++ classes are fundamentally difficult to unit test
- Only public methods and members can be used
- Private Keyword does not help testability
- No reflection and introspection

```
class Remote{
    public:
        Remote();
        virtual ~Remote();
        bool pressbutton(int button_index);
    private:
        bool setChannel(int channel);
        void resetMenu();
        void resetSubmenu();
        void powerToggle();
        int m_last_button_pressed;
        int m_battery_percentage;
};
```

266

266

TDD in C++

- **Suboptimal solutions:**

```
class Remote{
    public:
        Remote();
        virtual ~Remote();
        bool pressbutton(int button_index);
    protected:
        bool setChannel(int channel);
        void resetMenu();
        void resetSubmenu();
        void powerToggle();
        int m_last_button_pressed;
        int m_battery_percentage;
};

class TestWrapRemote{
    public:
        bool setChannel(int channel);
}
```

```
class Remote{
    public:
        Remote();
        virtual ~Remote();
        bool pressbutton(int button_index);
#ifdef TEST_MODE
    public:
#else
    private:
#endif
        bool setChannel(int channel);
        void resetMenu();
        void resetSubmenu();
        void powerToggle();
        int m_last_button_pressed;
        int m_battery_percentage;
};
```

267

267

TDD in C++

- **Alternative is to not test private functions**
- **View class as a blackbox and only test its interface**

268

268

TDD in C++

Test Doubles

- **A test double replaces parts (usually objects) of production code during test execution with a test implementation**
- **Why is this important?**
  - Keep tests focussed
  - Keep tests stable
  - Keep tests fast

269

269

TDD in C++

Formal types of test doubles

- **Dummy**
  - The implementation or return value does not matter for the test
- **Stub**
  - Provides the correct input for the function under test
- **Spy**
  - Tracks if a function has been called (and potentially how often)
- **Mock**
  - Verifies if a function has been called optionally including the provided parameters
- **Fake**
  - Actual logic that stands in for a function

270

270

TDD in C++

In practice there is one

- **Mock**
  - Verifies if a function has been called optionally including the provided parameters
  - With Dummy, stub, spy, and fake behaviour

271

271

TDD in C++

- **What is a unit?**
  - "A unit is the smallest testable part of an application"[1]
- **This implies full isolation of dependencies for class under test**
- **Too much isolation can hinder the ability to refactor and even affect the design itself: Is TDD Dead? (2014)[2]**

[1] This definition was taken from http://istqbexamcertification.com/what-is-unit-testing/
[2] Conversation between Kent Beck, Martin Fowler and David Heinemeier Hansson, https://www.youtube.com/watch?v=z9quxZsLcfo (30 minutes)

272

272

11. Dependency Injection

273

Inversion of Control

- In ‘traditional’ procedural code, application specific code *calls into* framework or library components
- Inversion of control means that application specific code *is called from* framework or library components
- This usually requires some form of setup or registration
- The Hollywood Principle: “Don't call us, we'll call you!"

274

274

Inversion of Control

- **According to Wikipedia:**
  - *"IoC inverts the flow of control as compared to traditional control flow. In IoC, custom-written portions of a computer program receive the flow of control from a generic framework. A software architecture with this design inverts control as compared to traditional procedural programming: in traditional programming, the custom code that expresses the purpose of the program calls into reusable libraries to take care of generic tasks, but with inversion of control, it is the framework that calls into the custom, or task-specific, code."*

275

275

Inversion of Control

- **Examples of Inversion of Control:**
  - Plug-ins
  - Callbacks
  - Event loops
  - Template method pattern
  - Dependency injection

276

276

Dependency Injection

- Dependency injection is a form of inversion of control
- Dependency injection removes the responsibility for creating an implementation from the user of its interface
- Dependency injection reduces coupling
- Dependency injection can make classes easier to test

277

277

Dependency Injection

- Without DI – Dependency on implementation

```
class A
{
public:
  void do_work()
  {
     _d->do_work();
  }
private:
  D* _d = new D();
};
```

```
class D
{
public:
  void do_work();
};
```

278

278

Dependency Injection

- **Preparation for DI – introduce an interface**

```
class A
{
public:
  void do_work()
  {
     _b->do_work();
  }
private:
  B* _b = new D();
};
```

```
class B // 'interface'
{
public:
  virtual void do_work() = 0;
};

class D : public B
{
public:
  void do_work() override;
};
```

279

279

Dependency Injection

- **Dependency Injection strategies**
  - Constructor injection
  - Property injection
  - Injection using template parameters
  - Service locator
  - DI framework

280

280

Dependency Injection

- Constructor Injection

```
class A
{
public:
  explicit A(B* b) : _b(b) {}
  void do_work() { _b->do_work(); }
private:
  B* _b;
};
```

281

281

Dependency Injection

- Property Injection

```
class A
{
public:
  void set_b(B* b) { _b = b; }
  void do_work() { _b->do_work(); }
private:
  B* _b = nullptr;
};
```

282

282

Dependency Injection

- Dependency injection using templates

```
template<typename B>
class A
{
public:
  void do_work() { _b->do_work(); }
private:
  B* _b = new B();
};
```

283

283

Dependency Injection and Testing

First we start with a common problem

- Imagine the following code
- Now we want to (unit) test this

```
class B{
    public:
        int this_work();
        bool that_work();
    private:
        int hidden_work();
};

class A{
    public:
        int other_work();
        bool work_using_b();
    private:
        B instance_of_B;
};
```

- Testing class B is easy
- Testing class A will automatically use class B
- class A can never be tested separately

- Unfortunately a not uncommon response is:
  - “class A is too difficult to test”

284

284

Dependency Injection and Testing

- Certain ‘main’ objects will auto create multiple sub objects which influence the behaviour
- In a larger design this will be problem
- Luckily there is a solution…….
- With some minor changes we can improve a lot

```
class B{
    public:
        int this_work();
        bool that_work();
    private:
        int hidden_work();
};

class A{
    public:
        int other_work();
        bool work_using_b();
    private:
        B instance_of_B;
};
```

285

285

Dependency Injection and Testing

- With some minor changes we have more influence over class A

```
class B{
    public:
        int this_work();
        bool that_work();
    private:
        int hidden_work();
};

class A{
    public:
        int other_work();
        bool work_using_b();
    private:
        B instance_of_B;
};
```

```
class B{
    public:
        int this_work();
        bool that_work();
    private:
        int hidden_work();
};

class improved_A{
    public:
        improved_A(B& injected_B);
        int other_work();
        bool work_using_b();
    private:
        B& instance_of_B;
};

int main(int)
{
    B my_B;
    A my_A(B);
}
```

286

286

Dependency Injection and Testing

- We can still test class B as before.
- When testing of class A we can change/influence class B
- Result is more control during testing
- Note: object my_B needs to stay alive while object my_A is used

```
// New
class B{
    public:
        int this_work();
        bool that_work();
    private:
        int hidden_work();
};

class improved_A{
    public:
        improved_A(B& injected_B);
        int other_work();
        bool work_using_b();
    private:
        B& instance_of_B;
};

int main(int)
{
    B my_B;
    A my_A(B);
}
```

287

287

Dependency Injection and Testing

Going one step further

- Lets create a special version of class B
- Works due to inheritance and references
- Class testB needs all functions used by class improved_A

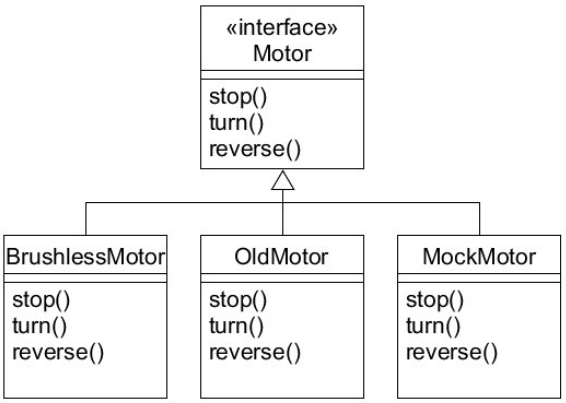
```
class test_B : public B{
    public:
        mockB(int fake_setting);
        int this_work();
        bool that_work();
    private:
        int hidden_work();
}

class improved_A{
    public:
        improved_A(B& injected_B);
        int other_work();
        bool work_using_b();
    private:
        B& instance_of_B;
};

int main(int)
{
    test_B my_B(5);
    A my_A(B);
}
```

288

288

Dependency Injection and Testing

- **Combine this with interface makes it easier to write test code**
- **Advise: For every external interface provide a Mock**



289

289

Service Locator

- **A service locator is an abstraction acting as a registry that maps interfaces to implementations**
- **Service Locator is often considered an antipattern**
  - Moves dependencies from compile time to runtime
  - Makes code harder to test

290

290

Service Locator

- Dependency Injection using a Service Locator

```
class A
{
public:
  A() : _b(ServiceLocator::Instance->CreateInstance<B>()) {}
  void do_work() { _b->do_work(); }
private:
  B* _b;
};
```

291

291

Dependency Injection Frameworks

- A correct design supports dependency injection by itself
- Use of DI frameworks in C++ is uncommon
- Such frameworks can be used to automate DI
  - Google Fruit
  - Boost.DI
  - Hypodermic
  - Autowiring
- DI frameworks are used extensively in Java and C#
  - Java: Spring
  - C# MEF (Managed Extension Framework, part of .NET)

292

292

Dependency Injection and Testing

Exercise: Improve code and add Dependency Injection

- **Use the ‘config_thing’ example code**
- **Create an interface and inject the config class**
- **Split production and test code**
- **Solve the #ifdef construction**
- **Avoid unnecessary code execution**

293

293

12. Compilers and Tooling

294

Compilers and Tooling

- Current Compiler support for C++11/14/17/20
- Compiler versions and differences
- Useful tools

295

295

Compiler Support

- **Several C++ compilers exist in the world, but a few brands have turned out to be dominant:**
  - Microsoft Visual Studio
  - GNU Compiler Collection
  - LLVM - CLang
  - Intel C++ Compiler
- **Historically relevant: Comeau, Borland, Portland group, EDG**
- **Lots of embedded-specific compilers, derived from one of the above**

296

296

Microsoft Visual Studio

- **Typically limited to Windows platform with some idiosyncrasies**
  - Up to vs2008 abysmal C++ support
  - Since vs2010 reasonable C++11 support
  - Since vs2010 available in an Express or Community edition (for free)
- **Nowadays very up-to-date, high quality**
- **Latest release vs 2022**

297

297

GNU Compiler Collection

- **Well-known, historically prominent**
  - Compiler Collection with many more languages: C, Fortran, Ada, Java, Objective-C, Go
  - Support for an enormous number of target architectures
- **Always very up-to-date, high-quality output**
- **Since version 6 or 7 very quick with C++ standard uptake and several experimental parts**
- **Latest release GCC 12.1 (August 2022)**

298

298

LLVM - Clang

- **Relatively recent compiler, LLVM project started in 2000**
  - Low Level Virtual Machine for dynamic compilation techniques
  - Language independent instruction set (IR), JIT compiler
- **Clang frontend started in 2005 at Apple for OpenGL**
  - Mostly compatible with GCC
  - More accessible parse tree and optimization, easier to combine with code editing tools
- **Usually up-to-date with C++ standard and several experimental parts**
- **Latest version Clang 14.0.6 (June 2022)**

299

299

Intel C++ compiler

- **Heavy focus on generating optimized code for Intel and non-Intel processors**
  - Default flags favor performance over accuracy
  - Automatic support for vectorization: SSE, AVX
  - Mostly compatible with GCC
  - Mostly compatible with Visual Studio
- **Reasonably up-to-date, but tends to lag behind MSVC, GCC, Clang**
- **Latest version Intel C++ 2022.1 (March 2022)**

300

300

Compiler differences

- **Question: Who has mixed and matched different compiler version in one project?**

301

301

Compiler differences

- **C++ on Windows is typically NOT binary compatible between versions**
  - Intel promises to be compatible with Visual Studio, but not across versions
- **C++ on Linux is likely binary compatible between versions**
  - Since GCC 5.1, it supports a Dual ABI
  - Necessary C++ standard changes in std::list and std::string forced a break in ABI compatibility

302

302

Compiler differences

- **General Recommendation: Build all libraries with the same compiler**
- **If this is not feasible:**
  - On Windows: Wrap everything in a DLL behind a C interface
  - On GCC: Gamble if the project contains no libraries and versions built between GCC 4.2 and GCC 5.0 ánd does not use std::string, std::list, etc across shared object boundaries.
  - On GCC: Wrap everything in a SO behind a C interface

303

303

Useful Tools

- **Poll: Which tools and libraries are generally recommended to use with C++?**
- **Build systems**
- **Test frameworks**
- **Debuggers**
- **Profilers**
- **Code formatting**
- **Static Analysis**
- **Something else?**

304

304

Build Systems

- **Windows-only: MSBUILD (part of Visual Studio)**
- **Linux/Unix: Make, Gnu autotools**
- **Cross-platform: CMake**
- **Others: Ninja, SCons, Meson,…**

305

305

Test Frameworks

- **Unit-testing is not built in in C++**
  - An external library is often necessary
  - Note: Visual Studio nowadays includes `Microsoft::VisualStudio::CppUnitTestFramework`
  - Building your own is not really recommended
- **Well-known: Boost::test, Google Test, CppUnit, CppTest, CxxTest**
  - Most require building a binary.
- **Honorable Mention: Catch2**
  - Completely header-only unit testing library

306

306

Debuggers

- **A debugger lets you step through your code line by line**
- **Inspect and modify data**
- **Set breakpoints on code and data**
- **Debuggers are closely coupled to compilers**
  - Visual Studio Debugger
  - GDB
  - LLDB
- **Some (e.g., GDB and LLDB) can be used from the command line**
- **Third-party GUIs act as wrappers (DDD, Emacs, VS Code, etc.)**

307

307

Debuggers

- **Ways to use a debugger**
  - Interactively
  - Attaching to a running process
  - Remotely
  - Post mortem (using a core dump)
  - As a virtual machine (e.g., Valgrind)

308

308

Profilers

- **Profilers collect performance statistics of running code**
  - Call graphs annotated with execution counts and time
  - Memory usage over time
- **Help identify *hotspots* in code and guide optimization**
- **Usually require instrumentation of (source or object) code**
- **GUI tools are used to present statistics**

309

309

Profilers

- **Profilers**
  - Visual Studio Profiler
  - GNU gprof
  - Llvm-profdata
  - Intel Vtune
  - Callgrind (part of Valgrind)

310

310

Code Formatting

- **Most modern IDEs support several styles and can reformat a file if necessary**
- **Sometimes you would like to enforce this pre- or post-commit**

311

311

Code Formatting

- **Uncrustify: Relatively simple and free formatting tool**
  - Has default configs: free-bsd, gnu-indent, kr-indent, mono, msvc, etc.
  - Styles are easily exported and reconfigured:

```
uncrustify --show-config > default.cfg
```

- **Reasonably easy to configure and use:**

```
uncrustify -f main.cpp -c default.cfg > main-uncristified.cpp
```

312

312

Code Formatting

- **Clang-format: Automatically included with LLVM – Clang**
  - Supports several standard styles: LLVM, Google, Mozilla, Microsoft
  - Custom styles are available and configurable

```
clang-format style=microsoft main.cpp > main-ms.cpp
clang-format style=llvm main.cpp > main-llvm.cpp
```

- **Generally very easy to use and configure**
  - Has useful hooks, bindings and scripts in %LLVM%\share\clang
  - For example clang-format-diff.py: reformats a prepared patch or commit

313

313

Code Analysis

- **Static Analysis: Historically done with a “linter”**
  - Lint inspects C code for common mistakes
- **Analyzing C++ is much more difficult than analyzing C**
- **Commercial tools are available with mixed quality**
  - SonarQube: well-known
  - Coverity: Very good heuristics
  - Parasoft

314

314

Code Analysis

- **Visual Studio has it built-in: cl /analyze**
- **Clang has it built-in: clang-analyze**
- **GCC does not have it built-in but supports lots and lots of warning flags:**
  - -Wall, -Wpedantic, -Weffc++
  - But don't forget: -Wduplicated-cond, -Wnull-dereference, -Wold-style-cast, -Wno-unused-parameter, -Wno-unused-function, -Wno-unused-label, -Wsign-compare, etc.
  - -fstack-protector, -fvisibility=hidden
  - -Wl,-z,defs -Wl,-z,now

315

315

Code Analysis

- **CppCheck is a free static analysis tool, available for Windows, Linux, Mac**

```
cppcheck main.cpp
```

316

316

Header file: include what you use

- **Question: What is the recommended order of inclusion of header files?**
- **Include-what-you-use is a small clang-based tool which verifies if you only include what you need.**
  - Reduce the number of useless inclusions

317

317

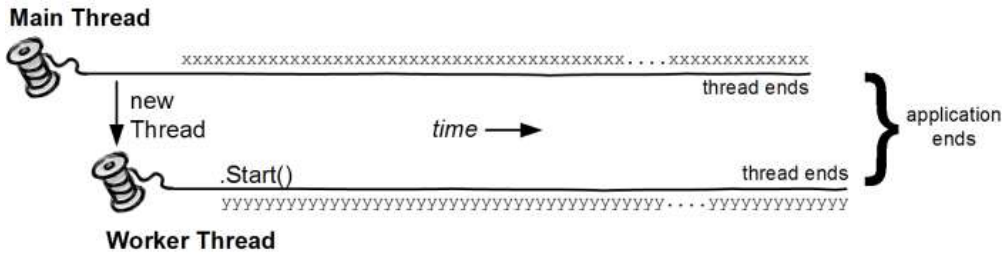
13. Threading

318

Multithreading

Introduction

- **Multithreading is the ability to run multiple "threads of execution" concurrently**



319

319

Multithreading

Advantages

- **Keep the UI responsive: offload heavy work to other thread**
- **Efficiently use a blocked CPU. For example while one thread is waiting on hardware, another thread can do work**
- **Parallel programming. Potentially large speedup by distributing work over many cores**
- **Simultaneous processing of multiple requests**

320

320

Multithreading

Disadvantages

- **Increased complexity, because of the much larger possible execution paths (indeterminate)**
- **Concurrent use of shared resources may lead to race conditions, deadlocks, starvation**
- **Overuse of threads may actually cost more than the expected gain in parallelization**

321

321

Multithreading

- **Threading Memory Model in the standard (C++98)**
  - Before 2011, the C++ language knew nothing about threads
  - Multiple threads of execution did not exist
  - Concurrent or shared data access never happened

322

322

Multithreading

- **Threading Memory Model in practice**
  - Everyone was doing multithreading
  - Win32 threads on Windows
  - Posix threads on Unix, Linux
  - Boost::thread or QThread cross-platform
- **Odd scenario where everyone is multithreading, but rules and guidelines are different per platform and the language knows nothing about it**

323

323

Multithreading

- **Threading Memory Model in the C++11 standard**
  - Standardized memory model in the C++ language
  - Allows multiple threads to co-exist
  - Defining when access to the same memory location is allowed and when updates become visible to other threads
  - New storage duration: thread_local
  - And library support for threads, thread interactions and atomics

324

324

Multithreading

Thread local storage

- Use the storage class `thread_local` for a per-thread global variable:

```
thread_local int x; // one x for each thread (static implied)
```

- Each `thread_local` is allocated when the thread starts and deallocated when the thread ends
- `thread_local` must be static (default) or extern

325

325

Multithreading

std::thread

- Represents a single thread, launched by constructing an std::thread with a function object

```
void f( );
void g( double x, double y, double z );

int main( )
{
   std::thread t1 = std::thread( f );
   std::thread t2 = std::thread( g, 2., 3., 4. ); // pass three args

   t1.join(); // join, otherwise the threads are “destroyed” on exit
   t2.join(); // while they are running. Join waits until done.
}
```

326

326

Multithreading

std::thread

```
void f( );
struct F {
   operator()( int a, int b );
}
int main( )
{
   std::thread t1 = std::thread( f ); // free function
   std::thread t2 = std::thread( F(), -1, 2 ); // functor
   std::thread t3 = std::thread(
      []()
      { std::cout << std::this_thread::get_id() << std::endl; }
   ); // lambda
   ...
}
```

327

327

Multithreading

Sharing data: mutex

- Protect shared data with a std::mutex
- Ensures “mutual exclusion”

```
std::mutex m;
int sh; // shared data
// ...
m.lock();
// ... manipulate shared data:
sh += 1;
m.unlock();
```

328

328

Multithreading

Sharing data: mutex

- Protect shared data with a std::mutex
- Ensures “mutual exclusion”

```
std::mutex m;
int sh; // shared data
// ...
m.lock();
// ... manipulate shared data:
sh += 1;
m.unlock();
```

- Has a try_lock method

329

329

Multithreading

Sharing data: mutex

- std::mutex has a try_lock

- Alternatives exist:
- std::recursive_mutex
- std::timed_mutex

330

330

Multithreading

Sharing data: locks

- **Beware of exceptions; do no use a mutex directly**

```
std::mutex m;
int sh; // shared data
// ...
m.lock();
// ... manipulate shared data: what if an exception is thrown?
sh += 1;
m.unlock();
```

- **Instead, use locks or lock_guards**

331

331

Multithreading

Sharing data: locks

- **Use std::lock_guard**

```
std::mutex m;
int sh; // shared data
// ...
{
   std::lock_guard guard(m); // locked from here
   // ... manipulate shared data: what if an exception is thrown?
   sh += 1;
} // automatically unlocked
```

332

332

Multithreading

Exercise: How to write a simple threadsafe counter

- Write a relatively simple concurrent counter
- Multiple threads count the number of sevens in several random lists
- Print every time they find one.
- Print the total count at the end

333

333

Multithreading

Sharing data: locks

- More control over locking with std::unique_lock
- std::unique_lock provides manual control on when to actually acquire the lock:
- std::defer_lock: post-pone until calling lock()
- std::try_lock: immediately try to lock()
- std::adopt: assume already locked

334

334

Multithreading

Sharing data: locks

- Multiple locks with std::scoped_lock (C++17)
- Allows locking multiple mutexes in a defined order
- Avoids deadlock

335

335

Multithreading

Condition Variables: notify and wait

- Allow threads to notify each other that something is ready
- Allow threads to wait until another thread modifies something
- Prevents busy waiting, polling

336

336

Multithreading

Condition Variables: notify and wait

1. Always acquire a lock on a mutex.
2. Then call wait, wait_for or wait_until
3. On wakeup or notify, the mutex is automatically acquired. Check the condition (again for spurious wakeup)

337

337

Multithreading

Condition Variables: notify and wait

1. Always acquire a lock on a mutex.
2. Then call wait, wait_for or wait_until

```
std::condition_variable cv;
// thread 1
std::unique_lock<std::mutex> lk(mut); // First acquire lock
cv.wait(lk); // Release temporarily until notify
... // when wait returns, the lock will be acquired again.

// thread 2
cv.notify(lk);  // Notify from other thread
```

338

338

Multithreading

Condition Variables: notify and wait

1. Always acquire a lock on a mutex.
2. Then call wait, wait_for or wait_until

```
std::condition_variable cv;
// thread 1
std::unique_lock<std::mutex> lk(mut); // First acquire lock
while( keepwaiting ) {
   cv.wait(lk); // Wait releases internally. On return, lock is
};              // reacquired. Condition needs to be verified again
...             // do actual work

// thread 2
cv.notify_one();  // Notify from other thread
```

339

339

Multithreading

Condition Variables: notify and wait

1. Always acquire a lock on a mutex.
2. Then call wait, wait_for or wait_until

```
std::condition_variable cv;
// thread 1
std::unique_lock<std::mutex> lk(mut); // First acquire lock
cv.wait( lk, !keepwaiting);
...             // do work

// thread 2
cv.notify_one();  // Notify from other thread
```

340

340

Multithreading

Exercise 2: Producer Consumer

- Write a relatively simple single producer consumer queue
- Consumer waits when the queue is empty

341

341

Multithreading

Atomic variables

- Guaranteed atomic access
- Well defined behavior if one thread reads and another thread writes.
- Locking not necessary

```
std::atomic< bool > flag;
std::atomic< int  > value; // Has atomic arithmetic
struct TrivialStruct;
std::atomic< TrivialStruct >; // Has atomic arithmetic
```

342

342

Multithreading

Chrono

- Several threading classes can wait for a specified time: wait_until, try_lock_until
- The time can be specified with std::chrono
- std::chrono::duration for a time interval
- std::timepoint for a point in time
- std::chrono also has several clocks useful for measuring time and performance: wall_clock, steady_clock, high resolution clock

343

343

Multithreading

std::async

- Manual thread management is complicated
- std::async abstracts thread management away

- Synchronously

```
auto result = task( args ); // Call task( args )and get result
```

- Asynchronously

```
auto handle = std::async( task, args ); // Create a “future”
auto result = handle.get( ); // Block until result is ready
```

344

344

Multithreading

std::async

- std::launch controls when/where task is executed:
- std::launch::async: launches a new thread

- std::launch::deferred: runs in requesting thread (lazy)
- This could be another thread.

- No arguments means unspecified: The task may or may not run on a separate thread

345

345

Multithreading

std::async

- More control with futures, promises and packaged_task.

- Question: What do we actually need to communicate between async tasks?

346

346

Multithreading

std::async

- **More control with futures, promises and packaged_task.**
- **Question: What do we actually need to communicate between async tasks?**
  - A place where the producer can “write” its result
  - A place where the consumer can “check” if the result is there
  - A mechanism to communicate failure, to store an exception

347

347

Multithreading

Exercise 3: Partial Sum

- **Given a big vector of numbers (from 1…N)**
- **Make a function that computes the sum**
- **By distributing it over four async calls**

348

348

Conclusion

349

Conclusion

Conclusion

- **C++ is a flexible language. You don't have to use everything that is possible. Use the things you and your team are comfortable with.**
- **Try to write code that is clear and correct, rather than code that is smart.**
- **There are still many things we haven't talked about. Nevertheless, I hope you have gained an insight in the range of possibilities and control that C++ provides.**

350

350

Recommended reading

Recommended Reading and Reference

- The C++ Programming Language by Bjarne Stroustrup
- The Design and Evolution of C++ by Bjarne Stroustrup
- Effective C++ series by Scott Meyers
- The C++ Standard Library by Nicolai Josuttis
- https://www.cppreference.com, online reference
- https://herbsutter.com/elements-of-modern-c-style/
- https://herbsutter.com/gotw/
- http://www.parashift.com/c++-faq-lite: FAQ
- https://github.com/isocpp/CppCoreGuidelines

351

351

Extra Exercises

352

Extra exercises

Exercise 1: rvalue references (1/3)

- Besides normal references, with C++11 we also have rvalue references, denoted by &&.
- These refer to objects that are no longer needed elsewhere.
- They are for example used for std::unique_ptr, but can also be used to increase efficiency.
- This often involves the move constructor Object(Object&& source) and move assignment operator Object & operator=(Object&& source).

353

353

Extra exercises

Exercise 1: rvalue references (2/3)

- Implement for your stack class a default constructor, copy constructor and assignment operator.
- Print a debugging message in all of these.
- Give your stack class a “<“ operator that compares the size of the stacks.

354

354

Extra exercises

Exercise 1: rvalue references (3/3)

- **Test the efficiency of your stack class for:**
  - Constructing a vector of stacks
  - Sorting that vector of stacks
- **Add a move constructor and a move assignment operator that also print a debugging message.**
- **Test the efficiency again.**

355

355

Extra exercises

Exercise 2: Fixed-size matrix class

Create a Matrix class that:

- **Has template arguments for the number of columns and rows. Use static_assert to guarantee that they are both 6 or less.**
- **Allows for all the usual arithmetic operations, including matrix multiplication.**
- **Test that code that is mathematically incorrect does not compile (e.g. multiplying incompatible matrices)**
- **Write unit tests for your class (you can also work in pairs)**

356

356

Reverse Polish notation

- In a regular calculator one would type this:

  12 + 20 =

  42 - 4 * 5 =

- In reverse Polish notation, the operator follows their operands.

  12 20 +

  42 4 5 * -

- Numbers are added on top of a stack
- Operators take their operands from the stack, result back on top

357

357

Assignment #1

- The following set of command seems to work:
- 0 + reset

- Broken:
- 12 + reset

358

358

Assignment #2

- The following set of command is accepted
- “12” “3” “store/”
- Use the debugger to figure out what and why this is accepted.

359

359