Multi-Paradigm Programming

Modern C++ programming

Julien BERNARD

Université de Franche-Comté, France

version 2020.1

Part I

Introduction

Outline

Outline

Hello World

```cpp
#include <iostream>

int main() {
  std::cout << "Hello World\n";
}
```

"There are only two kinds of languages: the ones people complain about and the ones nobody uses."

Bjarne Stroustrup, creator of C++

Objective and scope

Objective

Learn how to use different paradigms in a single application with modern C++

- procedural programming
- object-oriented programming
- functional programming
- generic programming

Scope

- **Not** the basics of programming and/or the basics of C++
- The elements to understand how C++ works
- The idioms and good practices of C++

Organisation

Hours

- Lecture: $6 \times$ 1h30
- Test: $1 \times$ 1h30
- Laboratory: $6 \times$ 3h00

Evaluation

- Multiple choice test (50%)
- Laboratory projects (3) in C++ (50%)

Resources

Online

- C++ Reference: `http://en.cppreference.com/w/cpp`
- C++ FAQ: `http://isocpp.org/faq`
- C++ Core Guidelines: `https://github.com/isocpp/CppCoreGuidelines/`

Books

Bjarne Stroustrup.
The C++ Programming Language.
4th edition, 2013, Addison–Wesley

Outline

C++ History (1/3)

Early C++

Early C++ (1979–1998)

- 1979: "C with Classes", Bjarne Stroustrup, AT&T Bell Labs
- 1983: "C with Classes" $\rightsquigarrow$ C++; CFront 1.0
- 1985: *The C++ Programming Language*, 1st edition, Bjarne Stroustrup
- 1989: *The Annotated C++ Reference Manual*, Bjarne Stroustrup; CFront 2.0
- 1991: First ISO/IEC JTC1/SC22/WG21 meeting; CFront 3.0
- 1992: *Effective C++*, 1st edition, Scott Meyers
- 1993: Standard Template Library, Alexander Stepanov, HP Labs
- 1994: *The Design and Evolution of C++*, Bjarne Stroustrup
- 1998: *Effective C++*, 2nd edition, Scott Meyers

C++ History (2/3)

Standard C++

Standard C++ (1998–2011)

- 1998: ISO standardization, C++98
- 2001: *Modern C++ Design*, Andrei Alexandrescu
- 2003: C++03, minor revision
- 2005: *Effective C++*, 3rd edition, Scott Meyers
- 2006: Performance Technical Report
- 2007: Library Technical Report 1 (TR1)

C++ History (3/3)

Modern C++

Modern C++ (2011–)

- 2011: C++11, major revision
  → "Surprisingly, C++11 feels like a new language"
- 2012: Standard C++ Foundation
- 2014: C++14, minor revision; *Effective Modern C++*, Scott Meyers
- 2015: C++ Core Guidelines, Guidelines Support Library
- 2017: C++17, major revision
- 2020: C++20, major revision of the standard
- 2023: C++23, next major revision of the standard

Outline

Programming paradigm

Definition (Programming paradigm)

A **programming paradigm** is a way to think about the execution and/or the organization of a program. A programming paradigm enables some constructs in a language and forbids other constructs.

Remarks

- There are dozens of programming paradigms.
- Most languages can be classified into multiple paradigms (like C++).

Example (Imperative programming)

Imperative programming is a paradigm that uses a sequence of statements to change the program's state.

Procedural programming

Definition (Procedural programming)

Procedural programming is an imperative programming paradigm based on the concept of *procedure call*.

Procedural programming in C++

C++ is a procedural programming language.

- Modularity through function parameters and return values
- Function call from any other function

$\rightarrow$ C style

Object-oriented programming

Definition (Object-oriented programming)

Object-oriented programming is an imperative programming paradigm based based on the concepts of *objects* (data) and *methods* (code).

Object-oriented programming in C++

C++ is an object-oriented programming language.

- Classes (`class`)
  - → Class-based object-oriented programming (≠ Prototype-based)
- Composition and (multiple) inheritance
- Polymorphism (`virtual` methods, `dynamic_cast`)

Functional programming

Definition (Functional programming)

Functional programming is a programming paradigm based on the concept of *mathematical functions* and forbids side effects (no assignment).

Functional programming in C++

C++ is **not** a functional programming language...

- No currying

...but has elements of a functional programming language.

- Recursion
- Functors and lambda functions[C++11] (closures)
- `std::function`[C++11]
- Partial evaluation (`std::bind`)

Generic programming

Definition (Generic programming)

Generic programming is a programming paradigm where types and algorithms are defined with abstract type parameters.

Generic programming in C++

C++ is a generic programming language.

- Templates
- Standard Template Library (STL): containers, iterators, algorithms
- Concepts[C++20]

Using multiple paradigms

Example

```
void drawAll(const std::vector<Shape*>& shapes) {
  std::for_each(shapes.begin(), shapes.end(),
      [](const Shape *shape) { shape->draw(); }
  );
}
```

- Procedural: `drawAll()`
- Object-oriented: `shape->draw()`
- Functional: `[](const Shape *shape) { }`
- Generic: `std::vector<Shape*>` and `std::for_each`

Outline

C++ program

Definition (C++ program)

A **C++ program** is a sequence of text files (typically header and source files) that contain declarations. They undergo translation to become an executable program, which is executed when the C++ implementation calls its main function.

Entities

Entities

The **entities** of a C++ program are:

- values
- objects
- references
- structured bindings[C++17]
- functions
- enumerators
- types
- class members
- templates
- template specializations
- namespaces
- parameter packs.

Objects

Definition (Object)

An **object** is a *region of storage* with the following properties:

- a size (that can be determined with `sizeof`)
- an alignment requirement (that can be determined with `alignof`)
- a storage duration (automatic, static, dynamic, thread-local)
- a lifetime
- a type
- a value (which may be indeterminate)
- optionally, a name

Outline

Expressions

Expressions

Expressions are characterized by two properties:

- a *type*
- a *value category*

Value categories before C++11

- An **lvalue expression** identifies an object ("locator value")
  - ≈ what is on the left-hand side of an assignment
- A **rvalue expression** is an expression that is not an lvalue expression
  - ≈ what is on the right-hand side of an assignment

Value categories

Overview

Value categories

- Primary categories: lvalue, prvalue, xvalue
- Mixed categories: glvalue, rvalue

$$\begin{tikzcd}
 & & \boxed{\text{expression}} \arrow[dl] \arrow[dr] & & \\
 & \boxed{\text{glvalue}} \arrow[dl] \arrow[dr] & & \boxed{\text{rvalue}} \arrow[dl] \arrow[dr] & \\
\boxed{\text{lvalue}} & & \boxed{\text{xvalue}} & & \boxed{\text{prvalue}}
\end{tikzcd}$$

Value categories

Definition (C++17)

Definitions (Value categories)

- A **glvalue** ("generalized" lvalue) is an expression whose evaluation determines the identity of an object, bit-field, or function

  → *glvalues produce locations*
- A **prvalue** ("pure" rvalue) is an expression whose evaluation either:
  - computes the value of the operand of an operator (has no result object), or
  - initializes an object or a bit-field (has a result object)

  → *prvalues perform initialization*
- An **xvalue** (eXpiring value) is a glvalue that denotes an object or bit-field whose resources can be reused (usually near the end of its lifetime)
- An **lvalue** is a glvalue that is not an xvalue
- An **rvalue** is a prvalue or an xvalue.

Properties of glvalues and rvalues

Properties of a glvalue expression (lvalue or xvalue)

- May be converted to a prvalue
- May be polymorphic (dynamic type $\neq$ static type)
- Can have incomplete type

Properties of a rvalue expression (prvalue or xvalue)

- Has no address
- Can not be used as the left-hand of the built-in assignment operator
- May be used to initialize a const lvalue reference or an rvalue reference
  - $\rightarrow$ In that case, the lifetime of the object identified by the rvalue is extended until the scope of the reference ends

Properties of lvalues and prvalues

Properties of an lvalue expression

- Has an address
- If modifiable, may be used as the left-hand of the built-in assignment operator
- May be used to initialize an lvalue reference

Properties of an prvalue expression

- Can not be polymorphic
- If non-class and non-array, can not be cv-qualified
- Can not have incomplete type
- Can not have abstract class type

Examples of lvalues

Examples (lvalue expressions)

- Name of a variable, function, data member (`std::cin`)
- Function call or overloaded operator expression whose return type is lvalue reference (`std::getline(std::cin, str)`, `std::cout << 1`, `++it`)
- Assignment and compound assignment expressions (`a = b`, `a += b`)
- Builtin pre-increment and pre-decrement expressions (`++i`)
- Builtin indirection expression (`*p`)
- Builtin subscript operator† (`a[n]`)
- String literal (`"Hello"`)
- Member of object expressions† (`a.m`)
- Built-in member of pointer expression† (`a->m`)

†with exceptions

Examples of prvalues

Examples (prvalue expressions)

- Literal[†] (`42`, `true`, `nullptr`)
- Function call or an overloaded operator expression whose return type is non-reference (`str.substr(1, 2)`, `it++`)
- Builtin post-increment and post-decrement expressions (`i++`)
- Builtin arithmetic expressions (`a + b`, `a / b`, `a & b`, `a << b`, `-a`)
- Builtin logical expressions (`a && b`, `a || b`, `!a`)
- Builtin comparison expressions (`a < b`, `a == b`)
- Builtin address-of expression (`&a`)
- The `this` pointer
- Enumerator
- Lambda expression (`[](int x) { return x * x; }`)

[†] with exceptions

Examples of xvalues

Examples (xvalue expressions)

- Function call or an overloaded operator expression, whose return type is rvalue reference to object (`std::move(x)`)
- Expression that designates a temporary object

Determination of the value category

lvalue or rvalue?

1. Determine if the expression is a glvalue or a prvalue
   - glvalues produce locations
   - prvalues perform initialization
2. If glvalue, determine if the expression is a temporary object or not
   - Temporary objects are xvalues
   - Non-temporary objects are lvalues

→ prvalues and xvalues are rvalues, everything else is an lvalue

Outline

`new` expression

`new` expression

A `new` expression allocates and initializes objects with dynamic storage duration.

`new` *type* or `new` *type initializer*

- Attempts to allocate storage and then attempts to construct and initialize either a single unnamed object, or an unnamed array of objects in the allocated storage
- Returns a prvalue pointer to the constructed object or, if an array of objects was constructed, a pointer to the initial element of the array

Example (`new` expression)

```
int *p1 = new int;
int *p2 = new int(42);
double *a = new double[10];
```

Placement `new` expression

Placement `new` expression

A placement `new` expression is used to construct objects in an already allocated storage

`new` (*placement*) *type* or `new` (*placement*) *type* *initializer*

Example (Placement `new` expression)

```
char* ptr = new char[sizeof T]; // allocate memory
T* tptr = new (ptr) T;          // construct in allocated storage
tptr->~T();                     // destruct
delete[] ptr;                   // deallocate memory
```

`delete` expression

`delete` expression

A `delete` expression destroys the object(s) previously allocated by a `new` expression and releases the obtained memory area.

`delete` *expr*

- Destroys one non-array object created by a new-expression
- *expr* can be `nullptr`

`delete []` *expr*

- Destroys an array created by a new[]-expression
- *expr* can be `nullptr`

Remark

Never mix non-array `new`/`delete` and array `new[]`/`delete[]`!

Good practice for memory allocation

Good practice for memory allocation

Never use `new` and `delete`!

Use smart pointers according to ownership semantics:

- Unique object → `std::unique_ptr<T>` and `std::make_unique`[C++14]
- Shared object → `std::shared_ptr<T>` and `std::make_shared`

Example (Smart pointers)

```
std::unique_ptr<int> p0(new int); // OK until C++14
auto p1 = std::make_unique<int>(); // better
auto p2 = std::make_unique<int>(42);
auto a1 = std::make_unique<double[]>(5);

auto p3 = std::make_shared<int>();
auto a2 = std::make_shared<double[]>(5); // C++20
```

Outline

Scoped enumeration

Scoped enumeration

`enum class` *name* { *enumerators* }

- The underlying type is `int`
- The enumerators are contained in the scope of the enumeration
- The enumerators can be accessed using scope resolution operator `::`
- `static_cast` is required to obtain the numeric value of the enumerator

`enum class` *name* : *type* { *enumerators* }

- The underlying type is *type*
- *type* is an integral type

Remark

Always use scoped enumerations!

Scoped enumeration

Example (Scoped enumeration)

```
enum class Suit {
  DIAMOND,
  CLUB,
  HEART,
  SPADE, // extra comma allowed in C++11
};

Suit s = Suit::CLUB;

enum class Altitude : char {
  HIGH = 'h',
  LOW = 'l',
};

char a = static_cast<char>(Altitude::HIGH); // a = 'h'
```

Outline

References

Definitions (References)

A **reference** of type `T` is an alias to an already existing object or function of type `T`. There are two types of references.

- A **lvalue reference**, noted `T&`, can be used to:
  - Alias an lvalue
  - Implement pass-by-reference semantics in function calls
  - Be the return type of a function, the function call is an lvalue expression
- A **rvalue references**, noted `T&&`, can be used to:
  - Extend the lifetime of a temporary object
  - Create overloads for rvalue expressions

Limits of references

Limits of references

- There are no references to void
- A reference is required to be initialized to refer to a valid object or function
  - → A null reference can not exist
- References are not objects (they do not necessarily require storage)
  - → There are no references to references
  - → There are no arrays of references
  - → There are no pointers to references

Reference initialization

Reference initialization

References are initialized in the following situations:

- Variable declarations:
  - Declaration of a named lvalue reference variable with an initializer
  - Declaration of a named rvalue reference variable with an initializer
- Functions (see part 2):
  - Function call expression, when the function parameter has reference type
  - Return statement, when the function returns a reference type
- Objects (see part 3):
  - Initialization of a non-static data member of reference type using a member initializer

References

Examples (References)

```
std::string s1 = "Test";
std::string& r0 = s1;
    // ok: r0 refers to s1
const std::string& r1 = s1;
    // ok: r1 refers to s1 (can not modify s1 through r1)

/* std::string&& r2 = s1; */
    // error: can not bind to lvalue
const std::string& r3 = s1 + s1;
    // okay: lvalue reference to const extends lifetime
/* r3 += "Test"; */
    // error: can not modify through reference to const
std::string&& r4 = s1 + s1;
    // okay: rvalue reference extends lifetime
r4 += "Test";
    // okay: can modify through reference to non-const
std::string& r5 = r4;
    // okay: r4 is an lvalue
```

`std::move`

`std::move`

`std::move` is used to *indicate* that an object may be "moved from", i.e. allowing the efficient transfer of resources from an object to another object.
In particular, `std::move` produces an xvalue expression that identifies its argument.

- A `std::move` is equivalent to a `static_cast` to an rvalue reference type
- If a function accepts rvalue reference parameters, `std::move` can be used to produce an xvalue
- Generally, a moved-from object is in a valid but unspecified state

std::move

Example (std::move)

```
std::vector<std::string> v;
std::string str = "Hello";

v.push_back(str);              // str is copied
v.push_back(std::move(str));   // str is moved
// here str is in an unspecified state, maybe ""
```

Outline

Namespace

Definition (Namespace)

A **namespace** is a named declarative region. The name of the namespace can be used to access the entities declared in that namespace.

Remarks

- Multiple namespace blocks with the same name are allowed.
- The standard library lives in the `std` namespace.

Example (Namespace)

```
namespace foo {
  int f(); // foo::f
  namespace bar {
    int g(); // foo::bar::g
  }
}
```

Inline namespace

Definition (Inline namespace)

An **inline namespace** is a namespace whose members are treated as if they were members of the enclosing namespace.

Example (Inline namespace)

```
namespace foo {
  inline namespace bar {
    int f(); // foo::bar::f or foo::f
    inline namespace baz {
      int g(); // foo::bar::baz::g or foo::bar::g or foo::g
    }
  }
}
```

Inline namespace and versioning

Versioning

Inline namespaces are useful for library versioning.

1. Put your declarations in an inline namespace `v1`
2. In case of ABI breakage, put the old declaration in a non-inline namespace `v1` and the new declaration in an inline namespace `v2` (and so on)
   - The old declaration is still accessible for old users (no need to recompile)
   - The new declaration is the default for new users

Example (Inline namespace and versioning)

```
namespace foo {
  namespace v1 {
    class C; // old declaration of class foo::C
  }
  inline namespace v2 {
    class C; // new declaration of class foo::C
  }
}
```

Unnamed namespace

Definition (Unnamed namespace)

An **unnamed namespace** is a namespace that has no visible name. It is treated as if it was declared with a unique name and its declarations put in the enclosing namespace.

Important remarks

- Any name declared within an unnamed namespace has internal linkage.
- The generated name is unique to the translation unit.

Example (Unnamed namespace)

```
namespace {
  int a = 2;
}
int x = a; // <unique>::a
```

Outline

Exception handling

Definition (Exception handling)

Exception handling provides a way of transferring control and information from some point in the execution of a program to a handler associated with a point previously passed by the execution.

What may throw an exception?

- `throw` expression in case of error
- `dynamic_cast` in case of a bad cast to a reference type
- `typeid` if a null pointer is dereferenced
- `new` expression and allocator function on failure to allocate memory
- Some standard library functions (e.g. `std::vector::at`, `std::string::substr`, etc)

Using exceptions

When to use an exception?

Exceptions are used when no other option for error handling is available:

- Failures to meet the postconditions, such as failing to produce a valid return value object (e.g. when the return type is a reference)
- Failures to meet the preconditions of another function that must be called
- Failures to (re)establish a class invariant (for non-private member functions)

Examples (Use of exceptions)

- When a constructor fails
- When an operator can not produce a result
- When a function has a wide contract and get an unacceptable input

Exception safety

Exception guarantee levels

A function may provide some guarantee regarding exceptions, especially if the function throws an exception.

1. **Nothrow (or nofail) exception guarantee**: it never throws exceptions.
   - Nothrow: errors are reported by other means or concealed (e.g. destructors)
   - Nofail: the function always succeeds (e.g. swaps, move constructors)
2. **Strong exception guarantee**: the state of the program is rolled back to the state just before the function call (e.g. `std::vector::push_back`).
3. **Basic exception guarantee**: the program is in a valid state; it may require cleanup, but all invariants are intact.
4. **No exception guarantee**: the program may not be in a valid state (resource leaks, memory corruption, or other invariant-destroying errors).

Outline

Outline

Range-based for loop

Range-based for loop

`for (`*decl* `:` *range*`)` *loop*

is equivalent to:

```
{
  auto && __range = range ;
  for (auto __begin = b, __end = e; __begin != __end; ++__begin) {
    decl = *__begin;
    loop
  }
}
```

- If *range* is an array, *b* is *range* and *e* is *range+bound*
- If *range* is a class with `begin` and `end` members, *b* is *range*`.begin()`, *e* is *range*`.end()`
- Otherwise, *b* is `begin(`*range*`)` and *e* is `end(`*range*`)`

Range-based for loop

Range-based for loop

`for` (*decl* : *range*) *loop*

is equivalent to[C++17]:

```
{
  auto && __range = range ;
  auto __begin = b ;
  auto __end = e ; // not necessarily the same type as __begin
  for ( ; __begin != __end; ++__begin) {
    decl = *__begin;
    loop
  }
}
```

- If *range* is an array, *b* is *range* and *e* is *range*+*bound*
- If *range* is a class with `begin` and `end` members, *b* is *range*`.begin()`, *e* is *range*`.end()`
- Otherwise, *b* is `begin(`*range*`)` and *e* is `end(`*range*`)`

Range-based `for` loop with initialization[C++20]

Range-based `for` loop with initialization

`for` (*init* ; *decl* : *range*) *loop*

is equivalent to[C++20]:

```
{
  init
  auto && __range = range ;
  auto __begin = b ;
  auto __end = e ;
  for ( ; __begin != __end; ++__begin) {
    decl = *__begin;
    loop
  }
}
```

- If *range* is an array, *b* is *range* and *e* is *range+bound*
- If *range* is a class with `begin` and `end` members, *b* is *range*`.begin()`, *e* is *range*`.end()`
- Otherwise, *b* is `begin(`*range*`)` and *e* is `end(`*range*`)`

Range-based `for`

Example (Range-based `for`)

```
std::vector<int> v;

for (int x : v) {
  std::cout << x << '\n';
}

for (int x : { 1, 2, 3, 4 }) {
  std::cout << x << '\n';
}

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

for (int x : a) {
  std::cout << x << '\n';
}
```

Outline

if statement with initialization[C++17]

if statement with initialization

$$\texttt{if}\ (\textit{init}\,;\textit{cond}\,)\,\textit{stmt}$$

is equivalent to[C++17]:

```
{
  init
  if ( cond ) stmt
}
```

Example (if statement with initialization)

```
std::map<int, std::string> m;

if (auto it = m.find(10); it != m.end()) {
  return it->second.size();
}
```

`switch` statement with initialization[C++17]

`switch` statement with initialization

`switch` (*init* ; *expr*) *stmt*

is equivalent to[C++17]:

```
{
  init
  switch ( expr ) stmt
}
```

Example (`switch` statement with initialization)

```
switch (Device dev = get_device(); dev.state()) {
  case SLEEP: /*...*/ break;
  case READY: /*...*/ break;
  case BAD: /*...*/ break;
}
```

Part II

Procedural programming

Outline

Outline

Outline

Default arguments

Definition (Default arguments)

Default arguments are used in calls where *trailing* arguments are missing.

Example (Default arguments)

```
void f(int x, int y = 42, int z = 69);

f(1, 2, 3); // calls f(1, 2, 3)
f(1, 2);    // calls f(1, 2, 69)
f(1);       // calls f(1, 42, 69)
/* f(); */  // error
```

Outline

Variadic arguments

Definition (Variadic arguments)

Variadic arguments are used when a function accepts any number of arguments. They are indicated by a parameter of the form `...` (ellipsis) that must appear last in the parameter list of the function declaration.

Remarks

- The ellipsis can be noted `...` (C++ style) or `, ...` (C style)

```
int f(int x...);     // C++ style
int f(int x, ...);   // C style
```

- To be able to access the arguments, the function must have at least one named parameter before the ellipsis

Accessing variadic arguments

Accessing variadic arguments

The `<cstdarg>` header provides facilities for accessing the variadic arguments:

- The `va_list` type holds the information for accessing the variadic arguments
- The `va_start` macro enables access to the variable arguments
- The `va_arg` macro accesses the next variadic function argument
- The `va_end` macro ends traversal of the variadic function arguments

Default argument promotions

- `nullptr_t` is converted to `void*`
- `float` is converted to `double`
- `bool`, `char`, `short` are converted to `int`

Variadic arguments

Example (Variadic arguments)

```
int add_nums(int count, ...) {
  int result = 0;
  va_list args;
  va_start(args, count);
  for (int i = 0; i < count; ++i) {
    result += va_arg(args, int);
  }
  va_end(args);
  return result;
}

int x = add_nums(4, 25, 25, 50, 50); // x == 150
```

Outline

Outline

Function overloading

Definition (Function overloading)

Function overloading is the ability to create multiple functions with the same name with different implementations. Two overloaded function must differ by the type of their parameters.

Overload resolution

When a function is overloaded, overload resolution is the process to select the function that will be called.

- Determination of the set of candidate functions after name lookup and template argument deduction
- Determination of the set of viable functions after examining arguments and parameters
- Choice of the best viable function

Overload resolution may fail and result in a compilation error.

Function overloading

Example (Function overloading)

```
void f(int x);
void f(double x);
void f(int *x);

template<typename T>
void f(T x);

f(1);       // calls f(int)
f(1.0);     // calls f(double)
f(true);    // calls f(T) with T = bool
f(1.0f);    // calls f(T) with T = float
f(nullptr); // calls f(T) with T = std::nullptr_t
```

Outline

Operator overloading

Operator overloading

Operator overloading is a special case of function overloading. Operators that can be overloaded are:

- Arithmetic operators: `+`, `-`, `*`, `/`, `%`, `^`, `&`, `|`, `~`, `<<`, `>>`
- Increment and decrement operators: `++`, `--`
- Logical operators: `!`, `&&`, `||`
- Assignment operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `&=`, `|=`, `<<=`, `>>=`
- Comparisons operators: `==`, `!=`, `<`, `>`, `<=`, `>=`, `<=>`**[C++20]**
- Access operators: `->`, `->*`, `[]`
- Special operators: `,` (comma), `()` (call)

Overloaded operators

Overloaded operators

Expression	Member function	Free function	Example
`@a`	`(a).operator@()`	`operator@(a)`	`!std::cin` → `std::cin.operator!()`
`a @ b`	`(a).operator@(b)`	`operator@(a,b)`	`std::cout << 42` → `std::cout.operator<<(42)`
`a = b`	`(a).operator=(b)`		`std::string s; s = "abc";` → `std::string.operator=("abc")`
`a(b)`	`(a).operator()(b)`		`std::random_device r; auto n = r();` → `r.operator()()`
`a[b]`	`(a).operator[](b)`		`std::map<int, int> m; m[1] = 2;` → `m.operator[](1)`
`a->`	`(a).operator->()`		`auto p = std::make_unique<S>(); p->bar()` → `p.operator->()`
`a@`	`(a).operator@(0)`	`operator@(a,0)`	`auto i = v.begin(); i++` → `i.operator++(0)`

Restrictions on overloaded operators

Restrictions on overloaded operators

- The following operators can **not** be overloaded:
  - `::` (scope resolution)
  - `.` (member access)
  - `.*` (member access through pointer to member)
  - `?:` (ternary conditional)
- New operators such as `**`, `<>`, or `&|` can **not** be created
- The overloads of operators `&&` and `||` lose short-circuit evaluation
- The overload of operator `->` must either return a raw pointer, or return an object (by reference or by value) for which operator `->` is in turn overloaded
- It is **not** possible to change:
  - the precedence of operators
  - the grouping of operators
  - the number of operands of operators

Operator overloading

Example (Operator overloading)

```
struct X {
  X& operator++();    // prefix ++
  X  operator++(int); // postfix ++
};

X operator+(const X& lhs, const X& rhs);
```

Outline

Properties of types

Properties of types

Types may be of three kinds regarding their ease of use in functions:

1. Easy types:
   - Cheap to copy: all fundamental types, `std::tuple<bool, int*>`, ...
   - Impossible to copy: `std::unique_ptr<T>`, ...
2. Medium types:
   - Cheap to move: `std::string`, `std::vector<T>`, ...
   - Quite cheap to move: `BigData`, `std::array<std::vector<T>,N>`, ...
   - Unknown cost: `unknown::Type`, ...
3. Hard types:
   - Expensive to move: `std::array<BigData,N>`, ...

Output only values

Output only values

For output only values, prefer:

- Return values for easy and medium types
  `X f();`
- Output parameter (reference to non-const) for hard types
  `void f(X& out);`

Example (Output only values)

```
std::vector<int> f1();                      // OK: cheap to move
void f2(std::vector<int>& out);             // KO: already empty?

int f3();                                   // OK: cheap to copy
void f4(int& out);                          // KO: can be read

std::array<int, 4096> f5();                 // KO: too large
void f6(std::array<int, 4096>& out);        // OK
```

Input only values

Input only values

For input only values, prefer:

- Passing by value for easy types
  `void f(T in);`
- Passing by const reference for medium and hard types
  `void f(const T& in);`

Example (Input only values)

```
void f1(const std::string& s);  // OK: always cheap
void f2(std::string s);         // KO: maybe expensive to copy

void f3(int x);                 // OK: unbeatable
void f4(const int& x);          // KO: overhead

void sink(std::unique_ptr<widget> w); // transfer ownership
```

Input/Output values

Input/Output values

For input/output values, prefer:

- Passing by reference to non-const for any type
  `void f(T& inout);`

Example (Input/Output values)

```
void update(Record& r);
```

Special case: copy of a parameter

Copy of a parameter

In the case a function wants a copy of a parameter of medium type, the function can be overloaded to handle two situations:

- `void f(const T& in);`
  - The function will be used for lvalues
  - The function will make a deep copy of the argument
- `void f(T&& in);`
  - The function will be used for rvalues
  - The function will move from the argument

Outline

Argument-dependent lookup

Definition (Argument-dependent lookup)

Argument-dependent lookup (ADL) is the set of rules for looking up the unqualified function names in function-call expressions, including implicit function calls to overloaded operators. These function names are looked up in the namespaces of their arguments in addition to the scopes and namespaces considered by the usual unqualified name lookup.

Argument-dependent lookup

Example (Argument-dependent lookup)

```
std::cout << "Test\n";
// There is no operator<< in global namespace, but ADL
// examines std namespace because the left argument is in
// std and finds std::operator<<(std::ostream&, const char*)

operator<<(std::cout, "Test\n");
// same, using function call notation

endl(std::cout);
// OK: this is a function call: ADL examines std namespace
// because the argument of endl is in std, and finds std::endl
```

Outline

constexpr functions

constexpr functions

A `constexpr` functions must satisfy the following requirements[C++14]:

- Its return type must be a literal type
- Each of its parameter must be a literal type
- The function body must **not** contain:
  - an `asm` declaration
  - a `goto` statement
  - a statement with a label other than `case` and `default`
  - a definition of a variable of non-literal type
  - a definition of a variable of static or thread storage duration
  - a definition of a variable for which no initialization is performed

A `constexpr` function can be used to compute values at compile time.

Remark

In C++11, a `constexpr` function could only have exactly one return statement!

constexpr functions

Example (constexpr functions)

```
constexpr int factorial(int n) {
  if (n <= 1) {
    return 1;
  }

  return n * factorial(n - 1);
}

constexpr int x = factorial(10);
int array[factorial(5)];
```

Outline

Literal suffix

Literal suffix

A **literal suffix** is appended to a literal to specify a type for the literal:

- For integers: `u`/`U` for `unsigned`, `l`/`L` for `long` or `ll`/`LL` for `long long`
- For floats: `f`/`F` for `float`, `l`/`L` for `long double`

Example (Literal suffix)

```
42;   // int
42u;  // unsigned
42l;  // long
42LU; // unsigned long
3.14;   // double
3.14f;  // float
3.14l;  // long double
```

User-defined literals

User-defined literals

User-defined literals are literals with a user-defined suffix. They allow integer, floating-point, character, and string literals to produce objects of user-defined type by defining a user-defined suffix.

- The suffix is an identifier
- The suffix introduced by a program must begin with an underscore
- The standard library suffixes do not begin with underscores

Example (User-defined literals)

```
1.2_km
0.5_Pa
'c'_X
"abc"_i18n
```

Literal operator

Literal operator

The function called by a user-defined literal is known as **literal operator**. It is a function like any other with a name of the form:

`operator "" `*id*

Only the following parameter lists are allowed on literal operators:

- `(const char *)`
- `(unsigned long long int)`
- `(long double)`
- (*character*)
  - where *character* is one of `char`, `char8_t`[C++20], `char16_t`, `char32_t`
- (`const` *character* `*, std::size_t`)
  - where *character* is one of `char`, `char8_t`[C++20], `char16_t`, `char32_t`

Literal operator

Example (Literal operator)

```
void operator "" _km(long double);

std::string operator "" _i18n(const char*, std::size_t);

constexpr long double operator"" _deg(long double deg) {
    return deg * 3.141592 / 180;
}
```

Literal operators in the standard library

Literal operators in the standard library

- `operator""if`, `operator""i`, `operator""il` for `std::complex` [C++14]
- `operator""h`, `operator""min`, `operator""s`, `operator""ms`, `operator""us`, `operator""ns` for `std::chrono::duration` [C++14]
- `operator""y` for `std::chrono::year` [C++20]
- `operator""d` for `std::chrono::day` [C++20]
- `operator""s` for `std::string` [C++14]
- `operator""sv` for `std::string_view` [C++17]

They are all defined in inline namespaces inside `std::literals`

Outline

Example: `string_id`

Specifications for `string_id`

- Define a `string_id` that is an integer that represents a string
    - Need for a hash function: FNV-1a
- Have a literal operator for creating a `string_id` from a string
- Possibly do all this at compile-time

Implementation of string_id

Example (string_id)

```
using string_id = uint64_t;

constexpr string_id fnv1a_hash(const char *str, std::size_t sz) {
  string_id value = UINT64_C(0xcbf29ce484222325);

  for (std::size_t i = 0; i < sz; ++i) {
    value ^= static_cast<unsigned char>(str[i]);
    value *= UINT64_C(0x100000001b3);
  }

  return value;
}

string_id fnv1a_hash(const std::string& str) {
  return fnv1a_hash(str.data(), str.size());
}

constexpr string_id operator "" _id(const char *str, std::size_t sz) {
  return fnv1a_hash(str, sz);
}
```

Use of string_id

Example (string_id)

```
constexpr string_id id = "Foobar"_id;

/* ... */

switch (fnv1a_hash(str)) {
  case "Toto"_id:
    std::cout << "Toto\n";
    break;

  case "Titi"_id:
    std::cout << "Titi\n";
    break;

  default:
    std::cout << "Someone else\n";
    break;
}
```

Part III

Object-Oriented Programming (1/2)

Outline

Outline

Class

Definition (Class)

A **class** is a user-defined type whose declaration begins with `class` or `struct`.

Remark

`struct` and `class` are indistinguishable in C++, except that the default access mode and default inheritance mode are `public` if class declaration uses `struct` and private if the class declaration uses `class`.

Class members

Class members

A class can have the following kinds of members:

1. Data members
   - Non-static data members, including bit fields
   - Static data members (`static`)
2. Member functions
   - Non-static member functions
   - Static member functions (`static`)
3. Nested types
   - Nested classes and enumerations defined within the class definition
   - Aliases of existing types, defined with `typedef` or type alias declarations
4. Enumerators from all unscoped enumerations defined within the class
5. Member templates (variable[C++14], class or function templates)

Class members

Example (Class members)

```
class A {
public:                 // access specifier
  void f();             // non-static member function
  static void g();      // static member function

  int x;                // non-static data member
  static bool y;        // static data member

  struct B {            // nested class
  };

  using C = double;     // type alias declaration

  template<typename T>
  struct D {            // member class template
  };

  static_assert(sizeof(A) == sizeof(int), "!"); // static_assert
};
```

Outline

Outline

Constructor

Definition (Constructor)

A **constructor** is a special non-static member function of a class that is used to initialize objects of its class type.

Remarks

- Constructors have no names and cannot be called directly
- They are invoked when initialization takes place

Special kinds of constructors

- Converting constructors (without `explicit` specifier)
- Default constructor (without arguments)
- Copy constructor and move constructor (with another object of the same type as the argument)

Initialization order

Initialization order

1. Initialization of virtual bases
   - Only in the most derived class of the object that's being constructed
   - In depth-first left-to-right traversal of the base class declarations
2. Initialization of direct bases
   - In left-to-right order
3. Initialization of non-static data members
   - In order of declaration in the class definition
4. Execution of the body of the constructor

Member initializer list

Definition (Member initializer list)

In the definition of a constructor of a class, a **member initializer list** specifies the *non-default* initializers for direct and virtual base subobjects and non-static data members.

Remark

The list may not be in the same order as the members but compilers issue a warning in that case.

Member initializer list

Example (Member initializer list)

```
struct A {
  A(int i)
  : a(i)  // initialize X::a to the value of the parameter i
  , b(a)  // initialize X::b to refer to X::a
  , i(i)  // initialize X::i to the value of the parameter i
  {

  }

  int a;
  int& b;
  int i;
  int j; // j is not initialized in the constructor A(int)
};
```

Outline

Converting constructor and explicit constructors

Definition (Converting constructor)

A **converting constructor** is not declared with the specifier `explicit`

Converting constructors and explicit constructors

- Both kind of constructors are considered in direct initialization
- Converting constructors are also considered in copy initialization (conversion)

Converting constructor

Example (Converting constructor)

```
struct A {
    A() { }         // converting constructor (since C++11)
    A(int) { }      // converting constructor
    A(int, int) { } // converting constructor (since C++11)
};
int main() {
  A a1 = 1;
      // OK: copy-initialization selects A::A(int)
  A a2(2);
      // OK: direct-initialization selects A::A(int)
  A a3{4, 5};
      // OK: direct-list-initialization selects A::A(int, int)
  A a4 = {4, 5};
      // OK: copy-list-initialization selects A::A(int, int)
  A a5 = static_cast<A>(1);
      // OK: static_cast, direct-initialization
}
```

Explicit constructor

Example (Explicit constructor)

```
struct B {
    explicit B() { }
    explicit B(int) { }
    explicit B(int, int) { }
};
int main() {
/* B b1 = 1; */
      // error: copy-initialization does not consider B::B(int)
  B b2(2);
      // OK: direct-initialization selects B::B(int)
  B b3{4, 5};
      // OK: direct-list-initialization selects B::B(int, int)
/* B b4 = {4, 5}; */ // error: explicit constructor B::B(int, int)
  B b5 = static_cast<B>(1);
      // OK: static_cast, direct-initialization
  B b6;   // OK, default-initialization
  B b7{}; // OK, direct-list-initialization
/* B b8 = {}; */ // error: explicit constructor B::B()
}
```

Outline

Default constructor

Definition (Default constructor)

A **default constructor** is a constructor that can be called with no arguments (either defined with an empty parameter list, or with default arguments provided for every parameter).

Default constructor

- If no user-declared constructors of any kind are provided for a class, the compiler will always declare a default constructor
- A default constructor can be:
  - Explicitely declared with `= default` after its declaration
  - Explicitely deleted with `= delete` after its declaration

Default constructor

Example (Default constructor)

```
struct A {
  A() {  } // user-defined default constructor
};

struct B {
  B(int i = 0) {  } // user-defined default constructor
};

struct C {
  // implicitly defined default constructor
};

struct D {
  D(int i) {  }
  // no implicitly defined default constructor
};
```

Outline

Copy constructor

Definition (Copy constructor)

A **copy constructor** of class `T` is a non-template constructor whose parameter is (generally) `const T&`

Copy constructor

- If a copy constructor is not declared, it is implicitly declared, if possible or it is implicitly deleted otherwise
- A copy constructor can be:
  - Explicitely declared and defaulted with `= default` after its declaration
  - Explicitely deleted with `= delete` after its declaration
- A copy constructor is used in the following cases (`a` and `b` are of type `T`):
  - Initialization: `T a = b;` or `T a(b);`
  - Function argument passing: `f(a);`, where `f` is `void f(T t)`
  - Function return: `return a;` inside a function such as `T f()`, where type `T` has no move constructor

Move constructor

Definition (Move constructor)

A **move constructor** of class `T` is a non-template constructor whose parameter is (generally) `T&&`

Move constructor

- If a move constructor is not declared, it is implicitly declared, if possible or it is implicitly deleted otherwise
- A move constructor can be:
  - Explicitely declared and defaulted with `= default` after its declaration
  - Explicitely deleted with `= delete` after its declaration
- A move constructor is used in the following cases (`a` and `b` are of type `T`):
  - Initialization: `T a = std::move(b);` or `T a(std::move(b));`
  - Function argument passing: `f(std::move(a));`, where `f` is `void f(T t)`
  - Function return: `return a;` inside a function such as `T f()`, where type `T` has a move constructor

Copy and move constructors

Example (Copy and move constructors)

```
struct A {
  std::string s;
  int k;

  A()
  : s("test"), k(-1)
  { }

  A(const A& o)
  : s(o.s), k(o.k)
  { }

  A(A&& o) noexcept
  : s(std::move(o.s))         // explicit move (class type)
  , k(std::exchange(o.k, 0)) // explicit move (non-class type)
  { }
};
```

Copy assignment

Definition (Copy assignment)

A **copy assignment operator** of class `T` is a non-template non-static member function with the name `operator=` and whose parameter is (generally) `const T&` and whose return type is `T&`

Copy assignment

- If a copy assignment is not declared, it is implicitly declared, if possible or it is implicitly deleted otherwise
- A copy assignment can be:
  - Explicitely declared and defaulted with `= default` after its declaration
  - Explicitely deleted with `= delete` after its declaration

Move assignment

Definition (Move assignment)

A **move assignment operator** of class `T` is a non-template non-static member function with the name `operator=` and whose parameter is (generally) `T&&` and whose return type is `T&`

Move assignment

- If a move assignment is not declared, it is implicitly declared, if possible or it is implicitly deleted otherwise
- A move assignment can be:
  - Explicitely declared and defaulted with `= default` after its declaration
  - Explicitely deleted with `= delete` after its declaration

Copy and move assignment

Copy and move assignment

If both copy and move assignment operators are provided, overload resolution selects:

- the move assignment if the argument is an *rvalue* (either a *prvalue* such as a nameless temporary or an *xvalue* such as the result of `std::move`),
- the copy assignment if the argument is an *lvalue* (named object or a function/operator returning lvalue reference)

If only the copy assignment is provided, all argument categories select it (as long as it takes its argument by value or as reference to const, since rvalues can bind to const references), which makes copy assignment the fallback for move assignment, when move is unavailable.

Copy and move assignments

Example (Copy and move assignments)

```
struct A {
  std::string s;
  int k;

  /* ... */

  A& operator=(const A& other) {
    s = other.s;
    k = other.h;
    return *this;
  }

  A& operator=(A&& other) {
    std::swap(s, other.s);
    std::swap(k, other.k);
    return *this;
  }
};
```

Outline

Outline

Destructor

Definition (Destructor)

A **destructor** is a special member function that is called when the lifetime of an object ends. The purpose of the destructor is to free the resources that the object may have acquired during its lifetime.

$$\sim ClassName()$$

Destruction sequence

1. Execution of the body of the destructor
2. Destruction of non-static data members
   - In reverse order of declaration in the class definition
3. Destruction of direct bases
   - In reverse order of construction, i.e. right-to-left
4. Destruction of virtual bases
   - Only in the most derived class of the object that's being destroyed

Destructor call

Destructor call

The destructor is called whenever an object's lifetime ends, which includes:

- program termination, for objects with static storage duration
- thread exit, for objects with thread-local storage duration
- end of scope, for objects with automatic storage duration and for temporaries whose life was extended by binding to a reference
- delete expression, for objects with dynamic storage duration
- end of the full expression, for nameless temporaries
- stack unwinding, for objects with automatic storage duration when an exception escapes their block, uncaught

$\rightarrow$ Most of the time, the destructor is called automatically!

Destructor

Example (Destructor)

```
struct A {
  int i;
  A(int i) : i(i) { std::cout << "c" << i << ' '; }
  ~A() { std::cout << "d" << i << ' '; }
};

A a0(0);

int main() {
  A a1(1);
  A* p;
  { // nested scope
    A a2(2);
    p = new A(3);
  } // a2 out of scope
  delete p; // calls the destructor of a3
}
// prints: c0 c1 c2 c3 d2 d3 d1 d0
```

Outline

Rule of Three

Definition (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.

Reason

The implicitly-defined special member functions are typically incorrect if the class is managing a resource whose handle is an object of non-class type (raw pointer, POSIX file descriptor, etc), whose destructor does nothing and copy constructor/assignment operator performs a "shallow copy" (copy the value of the handle, without duplicating the underlying resource).

Rule of Three

Example (Rule of Three)

```
class three {
  const char *m_str;
  three(const char *str, std::size_t n) : m_str(new char[n])
  { std::memcpy(m_str, str, n); }
public:
  three(const char* s = "") : three(s, std::strlen(s) + 1) { }
  ~three() { delete [] m_str; }
  three(const three& other) // copy constructor
  : three(other.m_str) { }
  three& operator=(const three& other) { // copy assignment
    std::size_t n = strlen(other.m_str) + 1;
    auto str = new char[n];
    std::memcpy(str, other.m_str, n);
    delete [] m_str;
    m_str = str;
    return *this;
  }
};
```

Rule of Five

Definition (Rule of Five)

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

Remark

Unlike Rule of Three, failing to provide move constructor and move assignment is usually not an error, but a missed optimization opportunity.

Rule of Five

Example (Rule of Five)

```
class five {
  const char *m_str;
  five(const char *str, std::size_t n) : m_str(new char[n])
  { std::memcpy(m_str, str, n); }
public:
  five(const char* s = "") : five(s, std::strlen(s) + 1) { }
  ~five() { delete [] m_str; }
  five(const five& other) // copy constructor
  : five(other.m_str) { }
  five(five&& other) noexcept // move constructor
  : m_str(std::exchange(other.m_str, nullptr)) { }
  five& operator=(const five& other) { // copy assignment
    return *this = five(other);
  }
  five& operator=(five&& other) noexcept { // move assignment
    std::swap(m_str, other.m_str);
    return *this;
  }
};
```

Rule of Zero

Definition (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.

Rule of Zero

Example (Rule of Zero)

```
class zero {
  std::string m_str;
public:
  zero(std::string str)
  : m_str(std::move(str))
  { }
};
```

Outline

Resource Acquisition Is Initialization (RAII)

Definition (Resource Acquisition Is Initialization (RAII))

Resource Acquisition Is Initialization (RAII), is a C++ programming technique which binds the life cycle of a resource that must be acquired before use to the lifetime of an object.

Examples (Resources where RAII could be used)

- Allocated heap memory
- Thread of execution
- Open socket
- Open file
- Locked mutex
- Disk space
- Database connection

RAII in Practice

RAII in Practice

- Encapsulate each resource into a class, where
  - the constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done
  - the destructor releases the resource and never throws exceptions
- Always use the resource via an instance of a RAII-class that either
  - has automatic storage duration or temporary lifetime itself, or
  - has lifetime bounded by the lifetime of an automatic or temporary object

Remark

Classes with `open()`/`close()`, `lock()`/`unlock()`, or `init()`/`destroy()` member functions are typical examples of non-RAII classes

Resource Acquisition Is Initialization

Example (Resource Acquisition Is Initialization)

```
std::mutex m;

void bad() {
  m.lock();                    // acquire the mutex
  f();                         // f() may throw, KO!
  if(!everything_ok()) return; // early return, KO!
  m.unlock();                  // the mutex is released
}

void good() {
  std::lock_guard<std::mutex> lk(m); // RAII class
  f();                               // f() may throw, OK!
  if(!everything_ok()) return;       // early return, OK!
}                                    // the mutex is released
```

Part IV

Object-Oriented Programming (2/2)

Outline

Outline

Outline

Derived Classes

Derived Classes

Any class may be declared as *derived* from one or more *base classes* which, in turn, may be derived from their own base classes, forming an inheritance hierarchy. Each base class:

- may have an access specifer: `public`, `protected`, `private`
- may be declared `virtual`

Remark

If the access specifier is omitted, it defaults to:

- `public` for `struct`
- `private` for `class`

Derived Classes

Example (Derived Classes)

```
struct Base {
    int a, b, c;
};

// every object of type Derived includes Base as a subobject
struct Derived : Base {
    int b;
};

// every object of type DerivedAgain includes
// Derived and Base as subobjects
struct DerivedAgain : Derived {
    int c;
};
```

Inheritance and member access

Inheritance and member access

`struct B { }; struct D :` *specifier* `B { };`

Inheritance	Member in B		
specifier	`public`	`protected`	`private`
`public`	`public`	`protected`	-
`protected`	`protected`	`protected`	-
`private`	`private`	`private`	-

Member access in D

Outline

Virtual base class

Virtual base class

For each distinct base class that is specified `virtual`, the most derived object contains only one base class subobject of that type, even if the class appears many times in the inheritance hierarchy (as long as it is inherited `virtual` every time).

Example (Famous example of virtual base class)

$$\begin{tikzcd}
 & \texttt{std::iostream} \arrow[dl] \arrow[dr] & \\
\texttt{std::istream} \arrow[dr] & & \texttt{std::otream} \arrow[dl] \\
 & \texttt{std::ios} &
\end{tikzcd}$$

Virtual base class

Example (Virtual base class)

```
struct B { int n; };
class X : public virtual B { };
class Y : virtual public B { };
class Z : public B {};
// every object of type AA has one X, one Y, one Z, and two B's:
// one that is the base of Z and one that is shared by X and Y
struct AA : X, Y, Z {
  AA() {
    X::n = 1; // modifies the virtual B subobject's member
    Y::n = 2; // modifies the same virtual B subobject's member
    Z::n = 3; // modifies the non-virtual B subobject's member
    std::cout << X::n << Y::n << Z::n << '\n'; // prints 223
  }
};
```

Outline

Empty Base Optimization (EBO)

Empty Base Optimization (EBO)

Empty Base Optimization (EBO) allows the size of an empty base subobject to be zero

Explanation

- The size of any object or member subobject is required to be at least 1 even if the type is an empty class, in order to be able to guarantee that the addresses of distinct objects of the same type are always distinct.
- However, base class subobjects are not so constrained, and can be completely optimized out from the object layout

Empty Base Optimization

Example (Empty Base Optimization)

```
struct Base { }; // empty class

struct Derived : Base {
  int i;
};

// the size of any object of empty class type is at least 1
static_assert(sizeof(Base) >= 1, "!");

// empty base optimization applies
static_assert(sizeof(Derived) == sizeof(int), "!");
```

Outline

Outline

Virtual functions

Definition (Virtual function)

A **virtual function** is a non-static member function with the `virtual` specifier. A virtual function supports dynamic dispatch, i.e. its behavior can be overridden in derived classes.

Virtual functions override

A function that overrides a virtual function in a derived class must have the same:

- name
- parameter type list (but not the return type)
- cv-qualifiers and ref qualifiers

Remark

A virtual function does not need to be visible (`public` or `protected`) to be overridden.

Virtual functions

Example (Virtual functions)

```
struct Base {}
   virtual void f() { std::cout << "base\n"; }
};
struct Derived : Base {
  void f() override { std::cout << "derived\n"; }
};
int main() {
    Base b;
    Derived d;

    Base& br = b; // the type of br is Base&
    Base& dr = d; // the type of dr is Base& as  well
    br.f(); // prints "base"
    dr.f(); // prints "derived"
    Base* bp = &b; // the type of bp is Base*
    Base* dp = &d; // the type of dp is Base* as  well
    bp->f(); // prints "base"
    dp->f(); // prints "derived"
}
```

`override` and `final`

`override` and `final`

- If a function is declared with the specifier `override`, but does not override a virtual function, the program is ill-formed
- If a function is declared with the specifier `final`, and another function attempts to override it, the program is ill-formed

override and final

Example (override and final)

```
struct B {
  virtual void f(int);
  virtual void g(int);
};

struct D : B {
  virtual void f(int) override; // D::f(int) overrides B::f(int)
  /* virtual void f(long) override; */ // Error
  void g(int) final;
};

struct A : D {
  /* void g(int); */ // Error
};
```

Outline

Virtual destructor

Virtual destructor

Even though destructors are not inherited, if a base class declares its destructor virtual, the derived destructor always overrides it. This makes it possible to delete dynamically allocated objects of polymorphic type through pointers to base.

Important remark

If a class is polymorphic (declares or inherits at least one virtual function), it must declare its destructor private.

Virtual destructor

Example (Virtual destructor)

```
class Base {
public:
  virtual ~Base() { /* releases Base's resources */ }
};

class Derived : public Base {
  ~Derived() { /* releases Derived's resources */ }
};

int main() {
  Base* b = new Derived;

  delete b;
  // Makes a virtual function call to Base::~Base()
  // since it is virtual, it calls Derived::~Derived() which can
  // release resources of the derived class, and then calls
  // Base::~Base() following the usual order of destruction
}
```

Outline

Pure virtual functions and abstract classes

Definition (Pure virtual function)

A **pure virtual function** is a virtual function with the *pure specifier* noted `= 0`.

Definition (Abstract class)

An **abstract class** is a class that either defines or inherits at least one pure virtual function.

Remarks

- No objects of an abstract class can be created
- Abstract classes are used as base classes for concrete classes

Pure virtual functions and abstract classes

Example (Abstract class)

```
struct Abstract {
  virtual void f() = 0;
};

struct Concrete : Abstract {
  void f() override;
};
```

Outline

Virtual table and virtual table pointer

Definition (Virtual table)

A **virtual table** (or vtable) is a per-class table containing function pointers to virtual functions. It is generated by the compiler.

Definition (Virtual table pointer)

A **virtual table pointer** (or vptr) is a per-object pointer to a virtual table added by the constructor. The vptr is part of the object.

Call to a virtual function

1. Fetch the virtual table pointer of the object
2. Call the function declared in the table pointed by the virtual table pointer

Consequence: calling a virtual function requires an indirection!

Virtual table and virtual table pointer

Example (Virtual table and virtual table pointer)

```
struct A {
  virtual void f();
  virtual void g();
}; // vtable is [ A::f, A::g ]

struct B : A {
  void f() override;
}; // vtable is [ B::f, A::g ]

struct C : A {
  void g() override;
}; // vtable is [ A::f, C::g ]

int main() {
  A a; // vptr -> A::vtable
  B b; // vptr -> B::vtable
  A* c = new C // vptr -> C::vtable
  c->g();
}
```

Outline

dynamic_cast

dynamic_cast

A `dynamic_cast` safely converts pointers and references to classes up, down, and sideways along the inheritance hierarchy.

`dynamic_cast`<*type*>(*expr*)

expr is:

- A glvalue of a complete class type if *type* is a reference
- A prvalue of a pointer to a complete class type if *type* is a pointer

Result of a dynamic_cast

- If the cast is successful, `dynamic_cast` returns a value of type *type*.
- If the cast fails and *type* is a pointer type, it returns a null pointer.
- If the cast fails and *type* is a reference type, it throws an exception that matches a handler of type `std::bad_cast`.

dynamic_cast

Example (dynamic_cast)

```
struct V {
    virtual void f() { };
};
struct A : virtual V { };
struct B : virtual V { };
struct D : A, B { };
struct E : V { };

int main() {
    D d;          // the most derived object
    A* a = &d;  // upcast, dynamic_cast may be used
    D* new_d = dynamic_cast<D*>(a); // downcast
    B* new_b = dynamic_cast<B*>(a); // sidecast
    E* e = dynamic_cast<E*>(e);     // = nullptr
}
```

typeid

typeid

The `typeid` operator queries information of a type and returns an object of type `const std::type_info&`. It can be used:

- with a type
- with an expression

typeid with an expression

- If the expression is a glvalue expression that identifies an object of a polymorphic type, the `typeid` expression evaluates the expression and then refers to the `std::type_info` object that represents the dynamic type of the expression
- If expression is not a glvalue expression of polymorphic type, `typeid` does not evaluate the expression, and the `std::type_info` object it identifies represents the static type of the expression

typeid

Example (typeid)

```
struct Base { virtual void foo() {} }; // polymorphic
struct Derived : Base {};

int main() {
  std::string str;
  std::cout << typeid(str).name() << '\n';
  // NSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
  // with c++filt -t: std::__cxx11::basic_string<char,
  //    std::char_traits<char>, std::allocator<char>>

  int a = 0;
  std::cout << typeid(a = a + 1) << '\n'; // i
  std::cout << a << '\n'; // 0

  Derived d;
  Base& b = d;
  std::cout << typeid(b).name() << '\n'; // 7Derived
}
```

Outline

Outline

PImpl: Pointer to Implementation

PImpl: Pointer to Implementation

PImpl (Pointer to Implementation) (a.k.a. d-pointer or opaque pointer) is a technique to hide implemntation details of a class in order to achieve ABI stability.

PImpl in practice

```
class Foo {
public:
  Foo();
private:
  struct Impl;
  std::unique_ptr<Impl> m_impl;
};

/* ... */

struct Foo::Impl {
  // ...
};
```

PImpl or not PImpl

Advantages of PImpl

- Compilation firewall: a modification in `Impl` does not require a recompilcation of all classes that use `Foo`
- Move-friendly: a `std::unique_ptr` is easy to move, especially in containers

Drawbacks of PImpl

- Access overhead: an access to a member requires an indirection through a pointer
- Space overhead: the object requires an additional pointer
- Lifetime management: the implementation object is allocated on the heap
- Maintenance overhead: all the implementation has to be in a dedicated file

Outline

`const` correctness

`const` correctness

`const` correctness is a set of rules to offer a guarantee about the (non-)mutability of object:

- Choose reference to `const` or pointer to `const` for parameters if you do not intend to modify the parameters
- Make a member function `const` if the function does not modify the object
- If a member function is `const` and returns a reference to a member, the reference should be `const` or the function should return by value

Benefits of `const` correctness

- The caller has the guarantee that a variable or an object will not be modified
- The compiler ensures that the callee respects the contract by emitting an error in case of modification of a `const` object
- The `const` property is spread through member function calls

const correctness

Example (const correctness)

```
class Person {
public:
  const std::string& getName() const; // OK!
  std::string& getNameEvil() const;   // KO!
  int getAge() const;                 // OK!

  std::string& getName();             // OK!
  // ...
};

void kaboom(const Person& p) {
  p.getNameEvil() = "Igor";           // KO!
}
```

Part V

Functional programming

Outline

Outline

Functional programming

Functional programming concepts

- First-class and high order functions → `std::function`
- Pure functions → `constexpr`
- Recursion → OK
- Strict or lazy evaluation → strict!
- Type system → KO!
- Referential transparency → KO! (`operator=`)

→ C++ is not really a functional programming language

Outline

Function object

Definition (Function object)

A **function object** is an object that can be used on the left of the function call operator

Function objects

- Function pointer
- Object of a class with an overloaded `operator()`

Call operator

Call operator

The call operator `operator()` can be overloaded in any class

- it can take any number of arguments
- it can return anything

Remark

A function object can have a state

Function object

Example (Function object)

```
struct Compare {
  bool operator()(int a, int b) {
    return a < b;
  }
};

struct AddTo {
  int constant;
  AddTo(int value) : constant(value) { }
  int operator()(int x) {
    return x + constant;
  }
};

AddTo adder(10);
int x = adder(1); // 11
```

Outline

Lambda expressions

Definition (Lambda expressions)

A **lambda expression** is a prvalue of unique class type (closure type) that provides an unamed function object capable of capturing variables in scope (closure).

Lambda expressions

[*captures*] (*params*) { *body* } or [*captures*] (*params*) -> *ret* { *body* }

- *captures* is a comma-separated list of captures
- *params* is the list of parameters, as any function
  - The type of a parameter can be `auto`[C++14] → generic lambda
- *ret* is the return type, if not present it is implied by the function return statements (or `void` if it does not return any value)

Captures

Captures

The *captures* is a comma-separated list of zero or more captures, optionally beginning with the capture default. The only capture defaults are:

- `&` (implicitly capture the used automatic variables by reference)
- `=` (implicitly capture the used automatic variables by copy).

An individual capture can be:

- *identifier* (simple by-copy capture)
- *identifier* = *initializer* (by-copy capture with initializer[C++14])
- &*identifier* (simple by-reference capture)
- &*identifier* = *initializer* (by-reference capture with initializer[C++14])
- `this` (simple by-reference capture of the current object)
- `*this` (simple by-copy capture of the current object[C++17])

Lambda expressions

Example (Lambda expressions)

```
auto cmp = [](int a, int b) {
  return a < b;
};

int constant  = 10;
auto adder = [constant](int x) { return x + constant };

int x = adder(1); // 11
```

Lambda expressions with capture default

Example (Lambda expressions with capture default)

```
void S::f(int i) {
  [&]() {};           // OK: by-reference capture default
  [&, i]() {};        // OK: by-reference capture, except i by copy
  [&, &i]() {};       // Error: by-reference is already the default
  [&, this]() {};     // OK, equivalent to [&]
  [&, this, i]() {}; // OK, equivalent to [&, i]

  [=]() {};           // OK: by-copy capture default
  [=, &i]() {};       // OK: by-copy capture, except i by reference
  [=, *this]() {};    // until C++17: Error: invalid syntax
                      // since c++17: OK: captures S by copy
  [=, this]() {};     // until C++20: Error: this when = is default
                      // since C++20: OK, same as [=]
}
```

Conversion of a lambda expression

Conversion of a lambda expression

A capture-less lambda expression can be converted to a function pointer.

Example (Lambda expressions)

```
auto l = [](int a) -> int { return a; }

using FuncType = int (*)(int);
FuncType f = l; // OK!
```

Outline

Callable type

Definition (Callable type)

A **callable type** is a type `T` for which the `INVOKE` operation is applicable

Callable types

Given `f` an object of type `T`, `INVOKE(f,` t_1`,` t_2`,...,` t_N`)` is equivalent to:

- If `f` is a pointer to member function of class `T`:
  - If t_1 is of type `T`, `(`t_1`.*f)(`t_2`,...,`t_N`)`
  - If t_1 is a specialization of `std::reference_wrapper`, `(`t_1`.get().*f)(`t_2`,...,`t_N`)` [C++17]
  - Otherwise, `((*`t_1`).*f)(`t_2`,...,`t_N`)`
- If `f` is a pointer to data member of class `T` and $N = 1$:
  - If t_1 is of type `T`, t_1`.*f`
  - If t_1 is a specialization of `std::reference_wrapper`, t_1`.get().*f` [C++17]
  - Otherwise, `(*`t_1`).*f`
- If `f` is a function object, `f(`t_1`,` t_2`,...,` t_N`)`

Pointer to member

Pointer to member access

The member access operator expressions through pointers to members have one of the two forms:

- *lhs*`.*`*rhs* if *lhs* is of class type `T`
- *lhs*`->*`*rhs* if *lhs* is of type pointer to class type `T`

rhs is an expression of type pointer to member (data or function)

Pointer to member

A pointer to non-static member `x` (data or function) which is a member of class `C` can be initialized with the expression `&C::x` exactly.

Pointer to member

Example (Pointer to member)

```
struct C {
  int m;
  void f(int n) { std::cout << n << '\n'; }
};
int main() {
  int C::* p = &C::m;
  C c = {7};
  std::cout << c.*p << '\n';   // prints 7
  C* cp = &c;
  cp->m = 10;
  std::cout << cp->*p << '\n'; // prints 10

  void (C::* p)(int) = &C::f;
  C c;
  (c.*p)(1);                  // prints 1
  C* cp = &c;
  (cp->*p)(2);                // prints 2
}
```

Callable in the standard library

Callable in the standard library

- Types:
  - `std::function` can store any callable
  - `std::reference_wrapper` can store a reference to any callable
  - `std::packaged_task` can store any callable (invoked asynchronously)
- Functions that accepts any callable:
  - `std::bind`
  - `std::thread::thread`
  - `std::call_once`
  - `std::async`

std::function

Example (std::function)

```
void print_num(int i) {
  std::cout << i << '\n';
}
struct PrintNum {
  void operator()(int i) const { std::cout << i << '\n'; }
};
int main() {
  // store a free function
  std::function<void(int)> f_display = print_num;
  f_display(-9);
  // store a lambda
  std::function<void(int)> f_display_lambda = [](int i) {
      print_num(i);
  };
  f_display_lambda(42);
  // store a call to a function object
  std::function<void(int)> f_display_obj = PrintNum();
  f_display_obj(18);
}
```

Partial application

Partial application

`std::bind` can be used for partial application of a function. More precisely, `std::bind(f,`a_1`,...,`a_n`)` generates a forwarding call wrapper for a callable object `f`. Calling this wrapper is equivalent to invoking `f` with some of its arguments bound to the arguments of `std::bind` and unbound arguments replaced by the placeholders `_1`, `_2`, `_3`, etc. of namespace `std::placeholders`.

Remark

The arguments to `std::bind` are copied or moved, and are never passed by reference unless wrapped in `std::ref` or `std::cref`.

Partial application

Example (Partial application)

```
void f(int n1, int n2, int n3, const int& n4, int n5) {
}
int g(int n) {
  return n;
}
int main() {
  using namespace std::placeholders;  // for _1, _2, _3...
  int n = 7;
  // (_1 and _2 are from std::placeholders, and represent future
  // arguments that will be passed to f1)
  auto f1 = std::bind(f, _2, _1, 42, std::cref(n), n);
  n = 10;
  f1(1, 2, 1001);
  // 1 is bound by _1, 2 is bound by _2, 1001 is unused
  // makes a call to f(2, 1, 42, n, 7)

  auto f2 = std::bind(f, _3, std::bind(g, _3), _3, 4, 5);
  f2(10, 11, 12); // makes a call to f(12, g(12), 12, 4, 5);
}
```

Partial application

Example (Partial application)

```
struct Foo {
  void print_sum(int n1, int n2) {
    std::cout << n1 + n2 << '\n';
  }
  int data = 10;
};
int main() {
  using namespace std::placeholders;  // for _1, _2, _3...

  // bind to a pointer to member function
  Foo foo;
  auto f3 = std::bind(&Foo::print_sum, &foo, 95, _1);
  f3(5); // makes a call to foo.print_sum(95, 5)

  // bind to a pointer to data member
  auto f4 = std::bind(&Foo::data, _1);
  std::cout << f4(foo) << '\n'; // make a 'call' to foo.data
}
```

Outline

High-order functions

Definition (High-order function)

A **high-order function** is a function that does at least one of the following:

- takes one or more functions as arguments
- return a function as its result

High-order functions

The three most well-known high-order functions are:

- `map` → `std::transform`
- `fold` → `std::accumulate`
- `filter` → `std::copy_if`

std::transform

Example (std::transform)

```
std::string rot13(const std::string& in) {
  std::string out;

  std::transform(in.begin(), in.end(), std::back_inserter(out),
    [](char c) -> char {
      if ('a' <= c && c <= 'm') {
        return c + 13;
      }
      if ('n' <= c && c <= 'z') {
        return c - 13;
      }
      return c;
    }
  );

  return out;
}
```

std::accumulate

Example (std::accumulate)

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int sum = std::accumulate(v.begin(), v.end(), 0); // 55

int product = std::accumulate(v.begin(), v.end(), 1,
    std::multiplies<int>()); // 3628800
```

std::copy_if

Example (std::copy_if)

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::vector<int> out;

std::copy_if(v.begin(), v.end(), std::back_inserter(out),
  [](int x) {
    return x % 2 == 0;
  }
);

// out = { 2, 4, 6, 8, 10 }
```

Part VI

Generic programming

Outline

Outline

Outline

Template

Definition (Template)

A **template** is an entity that defines:

- a family of classes (class template), which may be nested classes
- a family of functions (function template), which may be member functions
- an alias to a family of types (alias template)
- a family of variables (variable template)**[C++14]**
- a concept**[C++20]**

Template parameters

Templates are parametrized by one or more template parameters, of three kinds:

- type template parameters
- non-type template parameters
- template template parameters

Template

Example (Template)

```
template<typename T, std::size_t N>
struct Array {
  T data[N];
};

template<template<typename> class T>
std::size_t size(const T& container) {
  return container.size();
}

template<typename T>
using Ptr = T*;

template<class T>
constexpr T pi = T(3.14159265358979323851L); // C++14
```

Template explicit instantiation

Template explicit instantiation

- An **explicit instantiation definition** forces instantiation of a template. It may appear in the program anywhere after the template definition, and for a given argument list, is only allowed to appear once in the entire program.
  - `template class` *name*<*arguments*>; `//class`
  - `template` *name*<*arguments*>(*parameters*); `//function`
  - `template` *name*(*parameters*); `//function (short)`
  - `template` *name*<*arguments*>; `//variable (C++14)`
- An **explicit instantiation declaration** skips implicit instantiation step: the code that would otherwise cause an implicit instantiation instead uses the explicit instantiation definition provided elsewhere.
  - `extern template class` *name*<*arguments*>; `//class`
  - `extern template` *name*<*arguments*>(*parameters*); `//function`
  - `extern template` *name*(*parameters*); `//function (short)`
  - `extern template` *name*<*arguments*>; `//variable (C++14)`

Template explicit instantiation

Example (Template explicit instantiation)

```
// declarations (in header)
extern template struct Array<float, 2>;
extern template std::size_t size<std::string>(
                                        const std::string& container);
extern template std::size_t size(const std::string& container);
extern template float pi<float>;

// definitions (in source)
template struct Array<float, 2>;
template std::size_t size<std::string>(
                                        const std::string& container);
template std::size_t size(const std::string& container);
template float pi<float>;
```

Template specialization

Template specialization

- An **explicit full template specialization** allows customizing the template code for a given set of template arguments. Class templates, function templates or variables templates (and some others) can be fully specialized.

  `template<>` *declaration*;

- A **partial template specialization** allows customizing *class templates* for a given category of template arguments.

  `template<`*parameters*`> class` *name*`<`*arguments*`> {` *members* `};`

Template specialization

Example (Template specialization)

```
template<typename T>   // primary template
struct is_void : std::false_type {
};

template<>  // explicit specialization for T = void
struct is_void<void> : std::true_type {
};
```

Template argument deduction for function template

Template argument deduction for function template

In order to instantiate a function template, every template argument must be known, but not every template argument has to be specified. When possible, the compiler will deduce the missing template arguments from the function arguments.

Example (Template argument deduction for function template)

```
template<typename To, typename From>
To convert(From f);

void g(double d) {
  int i = convert<int>(d);
      // calls convert<int,double>(double)
  char c = convert<char>(d);
      // calls convert<char,double>(double)
  int(*ptr)(float) = convert;
      // instantiates convert<int, float>(float)
}
```

Function overloads vs function specializations

Function overloads vs function specializations

Only non-template and primary template overloads participate in overload resolution. The specializations are not overloads and are not considered. Only after the overload resolution selects the best-matching primary function template, its specializations are examined to see if one is a better match.

Example (Function overloads vs function specializations)

```
template<class T> void f(T);
    // #1: overload for all types
template<>        void f(int*);
    // #2: specialization of #1 for pointers to int
template<class T> void f(T*);
    // #3: overload for all pointer types

f(new int(1)); // calls #3
    // even though specialization of #1 would be a perfect match
```

Parameter pack

Definition (Template parameter pack)

A **template parameter pack** is a template parameter that accepts zero or more template arguments (non-types, types, or templates). It can appear in alias template, class template and function template parameter lists. A template with at least one parameter pack is called a **variadic template**.

- *type* ... *Name*
- `typename` ... *Name*
- `template`<*parameters*> `typename` ... *Name*

Definition (Function parameter pack)

A function parameter pack is a function parameter that accepts zero or more function arguments.

Name ... *name*

Parameter pack

Example (Parameter pack)

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

template<class ... Types> void f(Types ... args);
f();        // OK: args contains no arguments
f(1);       // OK: args contains one argument: int
f(2, 1.0); // OK: args contains two arguments: int and double
```

Parameter pack expansion

Parameter pack expansion

A **parameter pack expansion** occurs when a *pattern* followed by an ellipsis, in which the name of at least one parameter pack appears at least once, is expanded into zero or more comma-separated instantiations of the pattern, where the name of the parameter pack is replaced by each of the elements from the pack, in order.

Example (Parameter pack expansion)

```
template<class ...Us> void f(Us... pargs) {}
template<class ...Ts> void g(Ts... args) {
  f(&args...); // "&args..." is a pack expansion
               // "&args" is its pattern
}

g(1, 0.2, "a");
    // Ts... args expand to int E1, double E2, const char* E3
    // &args... expands to &E1, &E2, &E3
    // Us... pargs expand to int* E1, double* E2, const char** E3
```

Parameter pack

Example (Parameter pack)

```
void tprintf(const char* format) { // base function
  std::cout << format;
}
template<typename T, typename... Targs>
void tprintf(const char* format, T value, Targs... Fargs) {
  for ( ; *format != '\0'; format++ ) {
    if ( *format == '%' ) {
      std::cout << value;
      tprintf(format+1, Fargs...); // recursive call
      return;
    }
    std::cout << *format;
  }
}
int main() {
  tprintf("% world% %\n","Hello",'!',123);
  return 0;
}
```

Dependent name

Definition (Dependent name)

A **dependant name** is a name that depends on a type of type template parameters or a value of non-type template parameters, inside the definition of a template (both class template and function template).

Remark

Name lookup and binding are different for dependent names and non-dependent names.

Disambiguators for dependent name

Disambiguators for dependent name

- In a declaration or a definition of a template, including alias template, a name that is not a member of the *current instantiation* and is dependent on a template parameter is not considered to be a type unless the keyword `typename` is used
  - → The keyword `typename` may only be used in this way before qualified names (e.g. `T::x`), but the names need not be dependent.
- In a template definition, a dependent name that is not a member of the *current instantiation* is not considered to be a template name unless the disambiguation keyword `template` is used
  - → The keyword `template` may only be used in this way after operators `::` (scope resolution), `->` (member access through pointer), and `.` (member access)

Dependent name

Example (Dependent name)

```
template <typename T>
void foo(const std::vector<T> &v) {
  // std::vector<T>::const_iterator is a dependent name,
  typename std::vector<T>::const_iterator it = v.begin(); // OK

  typedef typename std::vector<T>::const_iterator iter_t; // OK
  iter_t * p;
  // iter_t is a dependent name, but it's known to be a type name
}

template<typename T>
struct S {
    template<typename U> void foo() { }
};
template<typename T>
void bar() {
    S<T> s;
    s.template foo<T>(); // OK
}
```

Outline

Substitution Failure Is Not An Error (SFINAE)

Substitution Failure Is Not An Error (SFINAE)

When substituting the explicitly specified or deduced type for the template parameter fails, the specialization is discarded from the overload set instead of causing a compile error.

Example (SFINAE)

```cpp
struct Test {
  typedef int foo;
};
template <typename T>
void f(typename T::foo) { } // Definition #1
template <typename T>
void f(T) { }               // Definition #2
int main() {
  f<Test>(10); // Call #1.
  f<int>(10);  // Call #2. Without error thanks to SFINAE.
}
```

Perfect forwarding

Definition (Forwarding reference)

A **forwarding reference** is:

- a function parameter of a function template declared as rvalue reference to cv-unqualified type template parameter of that same function template
- `auto&&` except when deduced from a brace-enclosed initializer list

Reference collapsing

The **reference collapsing** rule applies when a reference to reference is created through type manipulation (template or typedef).

if . . .	`T&`	`T&&`
`T = X&`	`X&`	`X&`
`T = X&`	`X&`	`X&&`

Perfect forwarding

std::forward

- Forwards lvalues as either lvalues or as rvalues, depending on T
  - → When t is a forwarding reference, this overload forwards the argument to another function with the value category it had when passed to the calling function.
- Forwards rvalues as rvalues and prohibits forwarding of rvalues as lvalues
  - → This overload makes it possible to forward a result of an expression (such as function call), which may be rvalue or lvalue, as the original value category of a forwarding reference argument.

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

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

Perfect forwarding

Example (Perfect forwarding)

```
int f(int x, int& y);
template<typename X, typename Y>
int wrapper_f(X&& x, Y&& y) {
  return f(std::forward<X>(x), std::forward<Y>(y));
}

int x = 3;
wrapper_f(4, x); // OK! X = int,  Y = int&
wrapper_f(x, x); // OK! X = int&, Y = int&
wrapper_f(x, 4); // KO! X = int&, Y = int

template<class T, class... U>
std::unique_ptr<T> make_unique(U&&... u) {
  return std::unique_ptr<T>(new T(std::forward<U>(u)...));
}
```

Templates and Argument Dependent Lookup

Templates and Argument Dependent Lookup

ADL can be used in templates in order to call functions defined in the namespace of the template argument.

Example (Templates and Argument Dependent Lookup)

```
template<typename T>
void maybe_swap(bool b, T& a, T& b) {
  using std::swap;
  if (b) {
    swap(a, b);
  }
}
```

Outline

Automatic type detection

Automatic type detection in C++11 and C++14

`auto` can be used to automatically detect a type in the following situations.

- Variable declaration: the type is determined from the initializer
  `auto x = 1 + 2;` → `int`
- Function declaration: the type is the trailing return type
  `auto func(int x) -> int;` → `int`
- Function declaration: the type is deduced from its return statement[C++14]
  `auto add(int x, double y) { return x + y; }` → `double`
- Variable declaration with `decltype(auto)`: the type is deduced from the initializing expression using the rules for `decltype`[C++14]
- Function declaration with `decltype(auto)`: the type is deduced from the return statement using the rules for `decltype`[C++14]
- Parameter declaration in a lambda expression (generic lambda)[C++14]
  `[](auto x) { return x + 1; }`

Automatic type detection

Automatic type detection in C++17 and C++20

`auto` can be used to automatically detect a type in the following situations.

- Template parameter: the type is deduced from the argument[C++17]
  ```
  template<auto Param> class C { };
  ```
- Structure binding declaration[C++17]
  ```
  auto [it, inserted] = dict.insert("Hello");
  ```
- Function parameter declaration[C++20]
  ```
  int sign(auto x) { return (x > 0) - (x < 0); }
  ```

Type of an entity or expression

Type of an entity

`decltype` can be used to inspect the type of an entity:

- unparenthesized identifier
- unparenthesized class member access expression

Type of an expression

`decltype` can be used to inspect the type of an expression of type `T`:

- if the expression is an xvalue, then `decltype` yields `T&&`
- if the expression is an lvalue, then `decltype` yields `T&`
- if the expression is an prvalue, then `decltype` yields `T`

Warning!

For an identifier `x`, `decltype(x)` and `decltype((x))` are often different types.

Type of an entity or expression

Example (Type of an entity or expression)

```
struct A { double x; }
const A* a;

decltype(a->x) y;
    // type of y is double (declared type)
decltype((a->x)) z = y;
    // type of z is const double& (lvalue expression)
```

When to use automatic type detection?

Two schools

School #1: Only when necessary and/or convenient

- When the type has no name (e.g. lambda)
  `auto f = [](int x) { return x + 1; };`
- When the type is too long (e.g. iterator types)
  `auto it = dict.find("Toto");`
- When the type name is redundant with its initializer
  `auto ptr = std::make_unique<Foo>();`

School #2: Almost Always Auto

Some people advocate for "Almost Always Auto" (AAA), even for simple types (e.g. `auto i = std::size_t{0};` instead of `std::size_t i = 0;`)

→ Do as you want: be consistent, write readable and maintainable code

Outline

Outline

Template meta-programming (TMP)

Definition (Template meta-programming)

Template meta-programming is a technique in which templates are used by a compiler to generate source code, which is merged by the compiler with the rest of the source code and then compiled. The output of these templates include compile-time constants, data structures, and complete functions. The use of templates can be thought of as compile-time execution.

Remark

- In C++, the template language is Turing-complete!
- The template language is a form a functional programming

Template meta-programming

Example (Template meta-programming)

```
template <unsigned n>
struct Factorial {
  static constexpr unsigned value = n * factorial<n - 1>::value;
};

template <>
struct Factorial<0> {
  static constexpr unsigned value = 1;
};

unsigned x = Factorial<10>::value; // 3628800
```

Constexpr If

Constexpr If

The statement that begins with `if constexpr` is known as the constexpr if statement. In a constexpr if statement, the value of the condition must be a contextually converted constant expression of type `bool`.

Remark

`if constexpr` can replace SFINAE and other TMP tricks

Constexpr If

Example (Constexpr If)

```
template <typename T>
auto get_value(T t) {
  if constexpr (std::is_pointer_v<T>) {
    return *t; // deduces return type to int for T = int*
  } else {
    return t;  // deduces return type to int for T = int
  }
}
```

Outline

Tag dispatching

Tag dispatching

Tag dispatching is a way of using function overloading to dispatch based on properties of a type, and is often used hand in hand with traits classes.

- The relation between tag dispatching and traits classes is that the property used for dispatching is often accessed through a traits class.
- A **tag** is simply a class whose only purpose is to convey some property for use in tag dispatching and similar techniques.

Traits

A traits class provides a way of associating information with a compile-time entity (type, integral constant). Examples of traits in the standard library includes:

- `std::iterator_traits`
- `std::numeric_limits`
- All the type traits in `<type_traits>` (`std::is_...`)

Tag dispatch

Example (Tag dispatch)

```
template <typename Iter, typename Distance>
void advance_impl(Iter& it, Distance n, forward_iterator_tag) {
  while (--n >= 0) {
    ++it;
  }
}

template <typename Iter, typename Distance>
void advance_impl(Iter& it, Distance n, random_iterator_tag) {
  it += n;
}

template <typename Iter, typename Distance>
void advance(Iter& it, Distance n) {
  return advance_impl(it, n,
      typename std::iterator_traits<Iter>::iterator_category());
}
```

Outline

Curiously recurring template pattern (CRTP)

Definition (Curiously recurring template pattern (CRTP))

The **curiously recurring template pattern** (CRTP) is an idiom in which a class `X` derives from a class template instantiation using `X` itself as template argument.

Use of CRTP

The typical use of CRTP is *static polymorphism*

Curiously recurring template pattern

Example (Curiously recurring template pattern)

```
template <class T>
struct Base {
  void interface() {
    static_cast<T*>(this)->implementation();
  }

  static void static_func() {
    T::static_sub_func();
  }
};

struct Derived : Base<Derived> { // CRTP is here
  void implementation();
  static void static_sub_func();
};
```

Outline

Policy-based programming

Policy-based programming

Policy-based programming is a design approach based on an idiom known as *policies*. It is a compile-time variant of the strategy pattern.
The central idiom in policy-based design is a class template (called the *host class*), taking several type parameters as input, which are instantiated with types selected by the user (called *policy classes*), each implementing a particular implicit interface (called a *policy*), and encapsulating some orthogonal (or mostly orthogonal) aspect of the behavior of the instantiated host class.

Policy-based programming

Example (Policy-based programming)

```
template <typename LanguagePolicy>
class HelloWorld : private LanguagePolicy {
  using LanguagePolicy::message;
public:
  void run() const { std::cout << message(); }
};
class LanguagePolicyEnglish {
protected:
  std::string message() const { return "Hello, World!"; }
};
class LanguagePolicyFrench {
protected:
  std::string message() const { return "Bonjour, Monde !"; }
};
int main() {
  using HelloWorldEnglish = HelloWorld<LanguagePolicyEnglish>;
  HelloWorldEnglish hello_world;
  hello_world.run(); // prints "Hello, World!"
}
```

Part VII

Extras

Outline

Outline

Design Rules (1/4)

General rules

General rules

1. C++'s evolution must be driven by real problems.
2. Don't get involved in a sterile quest for perfection.
3. C++ must be useful *now*.
4. Every feature must have a reasonably obvious implementation.
5. Always provide a transition path.
6. C++ is a language, not a complete system.
7. Provide comprehensive support for each supported style.
8. Don't try to force people to use a specific programming style.

Design Rules (2/4)

Design support rules

Design support rules

1. Support sound design notions.
2. Provide facilities for program organization.
3. Say what you mean.
4. All features must be affordable.
5. It is more important to allow a useful feature than to prevent every misuse.
6. Support composition of software from separately developed parts.

Design Rules (3/4)

Language-technical rules

Language-technical rules

1. No implicit violations of the static type system.
2. Provide as good support for user-defined types as for built-in types.
3. Locality is good.
4. Avoid order dependencies.
5. If in doubt, pick the variant of a feature that is easiest to teach.
6. Syntax matters (often in perverse ways).
7. Preprocessor usage should be eliminated.

Design Rules (4/4)

Low-level programming support rules

Low-level programming support rules

1. Use traditional (dumb) linkers.
2. No gratuitous incompatibilities with C.
3. Leave no room for a lower-level language below C++ (except assembler).
4. What you don't use, you don't pay for (zero-overhead rule).
5. If in doubt, provide means for manual control.

That's all for now. . .

Questions?