Lecture 07

Smart pointers, move semantics, exceptions, STL utilities.

Advanced Programming - SISSA, UniTS, 2025-2026

Pasquale Claudio Africa

05 Nov 2025

Outline

1. Smart pointers and reference wrappers
2. Move semantics
3. Exceptions
4. STL utilities
    - I/O streams
    - Random numbers
    - Time measuring
    - Filesystem

Smart pointers and reference wrappers

RAII: Resource Acquisition Is Initialization

RAII, short for Resource Acquisition Is Initialization, plays a significant role in C++. It essentially means that an object should be responsible for both the creation and destruction of the resources it owns.

Not RAII compliant:

```
double *p = new double[10];
```

Who is responsible for destroying the resources pointed to by `p`?

RAII compliant:

```
std::array<double, 10> p;
```

The variable `p` takes care of creating 10 doubles and destroying them.

In C++, smart pointers are important tools to implement RAII.

Pointers in modern C++

In modern C++, we use different types of pointers:

- **Standard pointers:** Use them only to watch (and operate on) an object (resource) whose lifespan is independent of that of the pointer (but not shorter).
- **Owning pointers (Smart pointers):** They control the lifespan of the resource they point to. There are three kinds:
  - `std::unique_ptr`: With unique ownership of the resource. The owned resource is destroyed when the pointer goes out of scope.
  - `std::shared_ptr`: With shared ownership of a resource. The resource is destroyed when the **last** pointer owning it is destroyed.
  - `std::weak_ptr`: A non-owning pointer to a shared resource, reserved for special use cases.

Smart pointers implement the RAII concept. For simply addressing a resource, possibly polymorphically, use ordinary pointers.

Example: the need for `std::unique_ptr` (1/4)

```
class MyClass {
public:
    void set_polygon(Polygon *p) {
        polygon = p;
    }
private:
    Polygon *polygon; // Polymorphic object.
};

Polygon* create_polygon(const std::string& t) {
    if (t == "Triangle") {
        return new Triangle{...};
    } else if (t == "Square") {
        return new Square{...};
    } else {
        return nullptr;
    }
}
```

Example: the need for `std::unique_ptr` (2/4)

```
MyClass a;
a.set_polygon(create_polygon("Triangle"));
```

⚠️ **This design is error-prone, requiring careful handling of resources, leading to potential memory leaks and dangling pointers.**

Example: the need for `std::unique_ptr` (3/4)

```
class MyClass {
public:
    void set_polygon(std::unique_ptr<Polygon> p) {
        polygon = std::move(p); // Transfer ownership (move semantics explained later).
    }
private:
    std::unique_ptr<Polygon> polygon;
};

std::unique_ptr<Polygon> create_polygon(const std::string& t) {
    if (t == "Triangle") {
        return std::make_unique<Triangle>(...); // 'make_unique' available since C++14.
    } else if (t == "Square") {
        return std::make_unique<Square>(...);
    } else {
        return nullptr;
    }
}
```

Example: the need for `std::unique_ptr` (4/4)

```
MyClass a;
a.set_polygon(create_polygon("Triangle"));
```

✅ **This version with `std::unique_ptr` is RAII-compliant, improving resource management.**

How a `std::unique_ptr` works

A `std::unique_ptr<T>` serves as a unique owner of the object of type `T` it refers to. The object is destroyed automatically when the `std::unique_ptr` gets destroyed.

It implements the `*` and `->` dereferencing operators, so it can be used as a normal pointer. However, it can be initialized to a pointer only through the constructor.

The default constructor produces an empty (null) unique pointer, and you can check if a `std::unique_ptr` is empty by testing it in boolean conditions such as `if (ptr)`.

Main methods and utilities of `std::unique_ptr`

- `std::swap(ptr1, ptr2)`: Swaps ownership.
- `ptr1 = std::move(ptr2)`: By definition, **unique** pointers cannot be copied, but their ownership can be transferred using the `std::move` utility. Moves resources from `ptr2` to `ptr1`. The previous resource of `ptr1` is deleted, and `ptr2` remains empty.
- `ptr.reset()`: Deletes the resource, making `ptr` empty.
- `ptr.release()`: Returns a standard pointer, releasing the resource without deleting it. `ptr` becomes empty.
- `ptr1.reset(ptr2.release())`: Transfers ownership from `ptr2` to `ptr1` (alternative to `std::move`).
- `ptr.get()`: Returns a standard pointer to the handled resource.

`std::unique_ptr` instances can be stored in standard containers, such as vectors.

Shared pointers

For instance, assume having several objects that **refer** to a resource (e.g., a matrix, a shape, ...) that is built dynamically (and maybe polymorphically). You want to keep track of all the references in such a way that when (and only when) the last one gets destroyed the resource is also destroyed.

To this purpose you need a `std::shared_ptr<T>`. It implements the semantic of *clean it up when the resource is no longer used*.

While `std::unique_ptr` does not cause any computational overhead (it is just a light wrapper around an ordinary pointer), shared pointers incur overhead due to reference counting (typically requiring two allocations: one for the object and one for the control block). Use them only when shared ownership is truly necessary.

Example: the need for `std::shared_ptr` (1/2)

```
class Data { ... };

class Preprocessor {
public:
    Preprocessor(const std::shared_ptr<Data> &data, ...) : data(data) {}
private:
    std::shared_ptr<Data> data;
};

class NumericalSolver {
public:
    NumericalSolver(const std::shared_ptr<Data> &data, ...) : data(data) {}
private:
    std::shared_ptr<Data> data;
};
```

Example: the need for `std::shared_ptr` (2/2)

```
std::shared_ptr<Data> shared_data = std::make_shared<Data>(...);

Preprocessor preprocessor(shared_data, ...);
preprocessor.preprocess();

// shared_data will still be used by other resources, hence it cannot be destroyed here.

NumericalSolver solver(shared_data, ...);
solver.solve();
```

How a `std::shared_ptr` works

`std::shared_ptr` allows shared ownership of dynamically allocated objects. It keeps track of the number of shared references to an object through reference counting. When the reference count reaches zero, the object is automatically deallocated, preventing memory leaks.
`std::shared_ptr` is thread-safe, making it suitable for concurrent access. It can also be used for managing resources beyond memory and can be equipped with custom deleters.

It implements the `*` and `->` dereferencing operators as well, so it can be used as a normal pointer. Moreover, it provides copy constructors and assignment operators.

The default constructor produces an empty (null) shared pointer, and you can check if a `std::shared_ptr` is empty by testing it in boolean conditions such as `if (ptr)`.

We can swap, move, get, and release a `std::shared_ptr` just as we do with `std::unique_ptr`.

Example: shared pointers

```
// Create a shared_ptr to a dynamically allocated object.
std::shared_ptr<MyClass> shared_ptr = std::make_shared<MyClass>(42);

// Access the object through the shared_ptr.
shared_ptr->print();

// Create another shared_ptr that shares ownership
std::shared_ptr<MyClass> another_shared_ptr = shared_ptr;

// Check the use count (number of shared_ptrs owning the object).
std::cout << "Use count: " << shared_ptr.use_count() << std::endl;

// Create a new shared_ptr.
std::shared_ptr<MyClass> new_shared_ptr = std::make_shared<MyClass>(55);

// The old one goes out of scope, but is still referenced by 'another_shared_ptr'.
shared_ptr = new_shared_ptr;

// Check the use count again.
std::cout << "Use count: " << shared_ptr.use_count() << std::endl;
```

Cyclic references and `std::weak_ptr`

When using `std::shared_ptr`, be aware of cyclic reference problems:

```
class Node {
public:
    std::shared_ptr<Node> next;
    std::shared_ptr<Node> prev; // ⚠️ Creates cycles! Memory leak.
};

auto node1 = std::make_shared<Node>{};
auto node2 = std::make_shared<Node>{};
node1->next = node2;  // node2's refcount = 2
node2->prev = node1;  // node1's refcount = 2
```

What goes wrong:

- When `node1` and `node2` go out of scope, their refcounts only decrease to 1
- They keep each other alive! Neither can be destroyed → memory leak
- The cycle: `node1` owns `node2`, and `node2` owns `node1`

The solution: `std::weak_ptr`

Key insight: Not all references need to be *owning* references!

```
class Node {
public:
    std::shared_ptr<Node> next;  // Owns the next node.
    std::weak_ptr<Node> prev;    // Observes prev node (doesn't own it).
};
```

What is `std::weak_ptr`?

- A non-owning smart pointer that does not increase reference count
- Can observe an object managed by `shared_ptr` without keeping it alive
- Must be converted to `shared_ptr` (via `weak_ptr::lock()`) before accessing the object

Result: The cycle is broken! When `node1` goes out of scope, its refcount becomes 0 and it's properly destroyed.

Weak pointers

The `std::weak_ptr` is a smart pointer that holds a non-owning (*weak*) reference to an object managed by a `std::shared_ptr`. It must be converted to `std::shared_ptr` to access the referenced object.

```
std::shared_ptr<int> ptr = std::make_shared<int>(10);
std::weak_ptr<int> weak1 = ptr; // Get pointer to data without taking ownership.

ptr = std::make_shared<int>(5); // Delete managed object, acquires new pointer. weak1 expires.
std::weak_ptr<int> weak2 = ptr; // Get pointer to new data without taking ownership.

auto tmp1 = weak1.lock();  // tmp1 is nullptr, as weak1 is expired!
auto tmp2 = weak2.lock();  // tmp2 is a shared_ptr to new data (5).
std::cout << "weak2 value is " << *tmp2 << std::endl;
```

Smart pointers: common pitfalls

Don't mix raw and smart pointers:

```
auto ptr = std::make_unique<int>(42);
int* raw = ptr.get();
delete raw; // ❌ Double deletion! Undefined behavior.
```

Don't create multiple `shared_ptr` from the same raw pointer:

```
int* raw = new int(42);
std::shared_ptr<int> ptr1(raw);
std::shared_ptr<int> ptr2(raw); // ❌ Double deletion!
// Instead: use make_shared or share ownership from existing shared_ptr.
```

Always prefer `make_unique` / `make_shared` :

```
auto ptr = std::make_unique<MyClass>(args);      // ✅ Recommended.
std::unique_ptr<MyClass> ptr(new MyClass(args)); // ❌ Avoid.
```

Reference wrappers

References create aliases to existing objects and must be initialized. It's crucial to be cautious with references to temporary objects. A const reference prolongs the life of a temporary object.

Standard containers can hold only *first-class* objects, but not references. However, you can use `std::reference_wrapper` from the `<functional>` header to store objects with reference-like semantics in a container. Moreover, `std::reference_wrapper` can be assigned **after construction**!

```
int a = 10, b = 20, c = 30;

std::vector<std::reference_wrapper<int>> ref_vector = {a, b, c};

// Modify the original values through the reference wrappers.
for (std::reference_wrapper<int> ref : ref_vector) {
    ref.get() += 5;
}
```

Move semantics

Why do we need move semantics?

Consider returning a large object from a function:

```
std::vector<int> create_large_vector() {
    std::vector<int> result(1'000'000);
    // ... fill with data ...
    return result;  // ⚠️ Does this copy 1 million integers?
}

std::vector<int> v = create_large_vector();
```

Question: Is this efficient or wasteful?

The problem: In pre-C++11, this would potentially make expensive copies of temporary objects that are about to be destroyed anyway.

The solution: Move semantics allows us to *steal* resources from temporaries instead of copying them!

Return Value Optimization (RVO)

Good news: Modern compilers can already optimize away many copies automatically!

- **RVO** (Return Value Optimization): When a function returns a newly-created object, the compiler can construct it directly in the caller's memory location, avoiding any copy or move.
- **NRVO** (Named Return Value Optimization): Similar to RVO, but for named local variables that are returned.

Modern compilers use **RVO/NRVO** to construct the returned object directly at the destination - no copy or move at all! This is even mandatory since C++17 for cases like above.

So why do we need move semantics? Move semantics is essential for cases where RVO doesn't apply: swaps, container operations (`push_back`, `insert`), conditional returns, and explicit ownership transfers between existing objects.

A concrete example: the costly swap

Let's see a real performance problem:

```
void swap(std::vector<int>& a, std::vector<int>& b) {
    std::vector<int> tmp = a; // Copy all of a (expensive!)
    a = b;                    // Copy all of b (expensive!)
    b = tmp;                  // Copy all of tmp (expensive!)
}

std::vector<int> vec1(1'000'000); // 1 million elements.
std::vector<int> vec2(1'000'000); // 1 million elements.
swap(vec1, vec2); // Three allocations + copy 3 million integers! 😱
```

Insight: We're copying millions of integers just to swap two vectors, when we could simply swap the internal pointers!

The solution: *moving* instead of copying

What if we could just swap the pointers inside the vectors?

```
void swap(std::vector<int>& a, std::vector<int>& b) {
    // Conceptually: just swap internal pointers.
    // No copying of elements needed!
}
```

This is what move semantics enables!

Modern C++11 swap with move semantics:

```
std::vector<int> tmp = std::move(a);  // "Steal" a's data.
a = std::move(b);                     // "Steal" b's data.
b = std::move(tmp);                   // "Steal" tmp's data.
// Result: O(1) instead of O(n) - just pointer swaps, no element copies!
```

Key idea: When we know an object is about to die (temporary), we can safely *steal* its resources instead of copying them.

When can we safely *move* an object?

The fundamental question: How does the compiler know when it's safe to steal resources instead of copying?

Answer: We need to distinguish between:

- **Objects we still need** → must copy
- **Objects we don't need anymore** → can move (steal resources)

This is where **value categories** come in: `lvalue` vs `rvalue`

Value categories: lvalue (locator value)

- Has a name and a memory address
- Lives beyond a single expression
- Examples: variables, function parameters

```
int x = 10;                    // x is an lvalue.
std::string name = "Alice"; // name is an lvalue.
```

Value categories: rvalue (*right value* or temporary)

- Doesn't have a name (temporary)
- About to be destroyed
- Examples: literals, function return values, temporary objects

```
42                  // rvalue (literal).
x + 10              // rvalue (temporary result).
std::string("hi")   // rvalue (temporary object).
create_vector()     // rvalue (return value).
```

Key insight: rvalues are safe to move from because they're about to disappear anyway!

Quick test: lvalue or rvalue?

```
int x = 10;
int y = 20;

x               // lvalue (has a name, we might use it again).
42              // rvalue (temporary literal).
x + y           // rvalue (temporary result).
std::min(x, y) // rvalue (temporary return value).
&x              // OK: can take address of lvalue.
&42             // ERROR: can't take address of rvalue.
```

Rule of thumb: If you can take its address with `&`, it's an lvalue!

Rvalue references: the key to move semantics

The challenge: We need a way to overload functions based on whether an argument is temporary (movable) or not.

Solution: C++11 introduced **rvalue references** (`T &&`)

```
void process(std::string& s);   // #1: Called for lvalues.
void process(std::string&& s);  // #2: Called for rvalues (temporaries).

std::string name = "Alice";
process(name);                  // Calls #1 (lvalue).
process("Bob");                 // Calls #2 (rvalue - temporary).
process(std::string("Eve")); // Calls #2 (rvalue - temporary).
```

Key property: `T&&` binds **only** to rvalues (temporaries we can safely steal from)!

Implementing move: the copy constructor

```
class Matrix {
private:
    size_t rows, cols;
    double* data;

public:
    // Copy constructor (deep copy - expensive).
    Matrix(const Matrix& other)
        : rows(other.rows), cols(other.cols),
          data(new double[rows * cols]) {
        std::copy(other.data, other.data + rows*cols, data); // Copy all elements.
    }
};
```

Copy: Allocate new memory + copy all data 🐢

Implementing move: the move constructor

Now we can implement efficient *moving* by providing a **move constructor**:

```
class Matrix {
private:
    size_t rows, cols;
    double* data;

public:
    // Move constructor (steal resources - cheap!).
    Matrix(Matrix&& other)
        : rows(other.rows), cols(other.cols),
          data(other.data) { // Steal the pointer!
        other.data = nullptr; // Leave source in valid state.
        other.rows = other.cols = 0;
    }
};
```

Move: Just copy a pointer 🚀

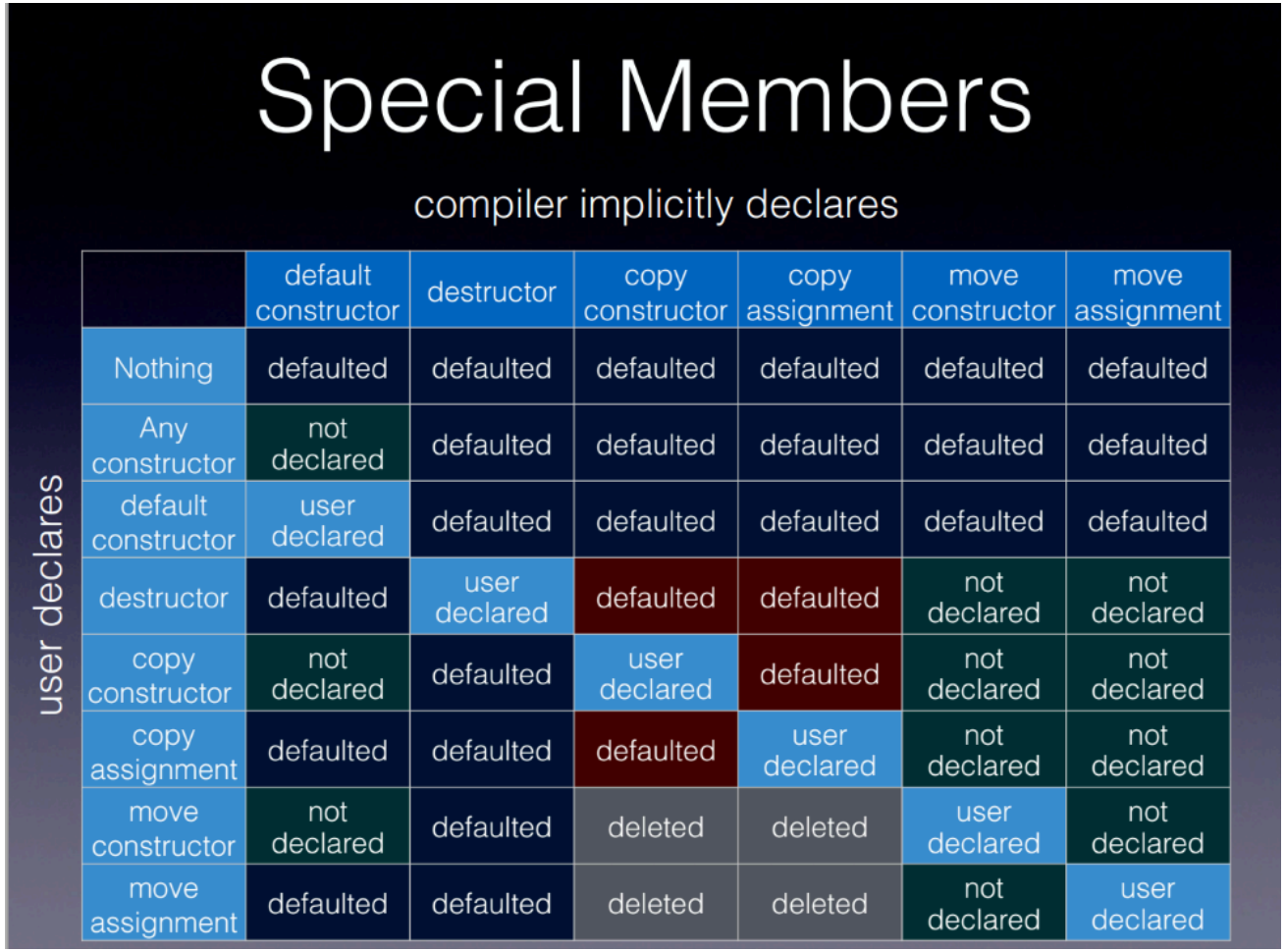
Move assignment operator

Similarly, we need a **move assignment operator**:

```
class Matrix {
public:
    /* ... */

    // Move assignment (steal resources).
    Matrix& operator=(Matrix&& other) {
        if (this != &other) {
            delete[] data;     // Clean up old data.
            rows = other.rows;
            cols = other.cols;
            data = other.data; // Steal the pointer.
            other.data = nullptr; // Leave source valid.
            other.rows = other.cols = 0;
        }
        return *this;
    }
};
```

Constructors and destructor synthetically generated



Special Members

compiler implicitly declares

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

Source

When does moving happen automatically?

The compiler automatically calls move operations in these situations:

```
Matrix create_matrix() {
    Matrix m(100, 100);
    // ... fill with data ...
    return m; // Automatic move (m is about to be destroyed).
}

Matrix a = create_matrix(); // Move constructor called.

Matrix b(200, 200);
b = create_matrix(); // Move assignment called.

std::vector<Matrix> vec;
vec.push_back(create_matrix()); // Move constructor called.
```

No need for `std::move` here - the compiler knows these are temporaries!

Forcing a move with `std::move`

What if we want to move from an lvalue (named variable)?

```
Matrix a(100, 100);
Matrix b = a; // Copy (a is an lvalue, we might use it again).
```

Use `std::move` to indicate *"I'm done with this object"*:

```
Matrix a(100, 100);
Matrix b = std::move(a); // Move! (Explicitly saying: I don't need 'a' anymore).
// ⚠️ 'a' is now in a valid but unspecified state (typically empty).
```

Important: `std::move` doesn't actually move anything - it just casts an lvalue to an rvalue!

```
// std::move is roughly equivalent to:
static_cast<Matrix&&>(a)
```

Practical example: the efficient swap

Now we can write an efficient, generic swap:

```
template<typename T>
void swap(T& a, T& b) {
    T tmp = std::move(a); // Move construct tmp from a.
    a = std::move(b);     // Move assign b to a.
    b = std::move(tmp);   // Move assign tmp to b.
}
```

Performance comparison for `std::vector<int>(1'000'000)`:

- Old (copy-based): ~30ms + allocate 12MB temporary memory!
- New (move-based): ~0.00001ms + no extra allocation!

The standard library already provides this: `std::swap(a, b)`

A crucial detail: named rvalue references are lvalues!

Rule: A **named** variable is always an lvalue, even if its type is an rvalue reference!

```
void process(Matrix&& m) { // m has type Matrix&&
    // Inside this function, m is an 'lvalue' (it has a name, you can take &m).

    Matrix other = m;             // ❌ Calls copy constructor (m is lvalue).
    Matrix other = std::move(m);  // ✅ Calls move constructor.
}
```

Why? You might use `m` multiple times in the function - moving from it implicitly would be dangerous:

```
void process(Matrix&& m) {
    use(m); // First use.
    use(m); // Second use - would fail if m was already moved!
}
```

Move semantics: summary (1/2)

The problem: Copying large objects (especially temporaries) is wasteful.

The solution: Move semantics - *steal* resources from objects we don't need anymore.

Key concepts:

- **lvalue** = has a name, might be used again → must copy
- **rvalue** = temporary, about to die → can move
- `T&&` = rvalue reference, binds only to temporaries
- **Move constructor/assignment** = steal resources instead of copying
- `std::move(x)` = cast lvalue to rvalue (says *"I'm done with x"*)

Move semantics: summary (2/2)

When moves happen automatically:

- Returning local variables from functions
- Passing temporaries to functions
- RVO (Return Value Optimization) - even better than moving!

Remember: After `std::move(x)`, don't use `x` again - except to (re)assign/destroy it!

What does move semantics have to do with the STL?

All standard containers support move semantic, and **all standard algorithms** are written so that if the contained type implements move semantics, the creation of unnecessary temporaries can be avoided. All containers also have a `swap()` method that performs swaps intelligently.

Smart pointers support move (but `std::unique_ptr` disallows copy).

For instance, `std::sort()` (which does a lot of swaps) is much more efficient on dynamically sized objects if move semantics are implemented.

Move semantics also make a few (but not all) template metaprogramming techniques now used in some libraries, like Eigen, to avoid unnecessary large size temporaries.

Revisiting `std::unique_ptr` and move semantics

Remember the `set_polygon` example from earlier?

```
void set_polygon(std::unique_ptr<Polygon> p) {
    polygon = std::move(p); // Now we understand why std::move is needed!
}
```

- The parameter `p` is passed by value, creating a new `unique_ptr` that takes ownership
- Inside the function, `p` is a **named variable**, hence an **lvalue**
- To transfer ownership to the member `polygon`, we need `std::move(p)` to cast it to an rvalue
- Without `std::move`, the code wouldn't compile because `unique_ptr` doesn't have a copy assignment operator

This is a common pattern when transferring ownership of unique pointers!

Exceptions

Preconditions, postconditions, and invariants

In software development, a function (or method) can be seen as a mapping from input data to output data. The software developer specifies the conditions under which the input data is considered valid; this specification is called a **precondition**. The developer also guarantees that the expected output, called a **postcondition**, is provided when the input adheres to the precondition. Failure to meet these conditions is considered a **fault** or **bug** in the code.

An **invariant** of a class is a condition that must be satisfied by the state of an object at any point in time, except for transient situations like the object's construction process. An object is considered to be in an **inconsistent state** if the invariants are not met.

The verification of preconditions, postconditions, and invariants is an integral part of **code verification** during the development phase.

An example

Consider a function in C++:

```
Matrix cholesky(const Matrix& m);
```

- This function has a **precondition** that requires the input matrix `m` to be symmetric positive definite.
- The **postcondition** is that the output matrix is a lower triangular matrix representing the Cholesky factorization of `m`.
- An **invariant** of a symmetric matrix `m` is that `m(i,j) = m(j,i)` for all matrix elements.

Run-time assertions

Example

```
double calculate(double operand1, double operand2) {
    assert(operand2 != 0 && "Operand2 cannot be zero.");

    const double result = /* ... */;

    assert(result >= 0 && "Negative result!");

    return result;
}
```

For improved efficiency, all assertions can be disabled (i.e. the argument to `assert()` will be **ignored**) by defining the `NDEBUG` preprocessor macro, for instance:

```
g++ -DNDEBUG main.cpp -o main
```

Compile-time assertions

Example

```
template <typename T, int N>
class MyClass {
public:
    MyClass() {
        // Here goes a condition that can be evaluated at compile-time, such as constexpr.
        static_assert(std::is_arithmetic_v<T> && N > 0, "Invalid template arguments.");
        // ...
    }
};
```

If the condition is not met, the error message is printed to the error stream and compilation will fail.

Exceptions

An **exception** is an anomalous condition that disrupts the normal flow of a program's execution when left unhandled. It is not the result of incorrect coding but rather arises from challenging or unpredictable circumstances.

Examples of exceptions include running out of memory after a `new` operation, failing to open a file due to insufficient privileges, or encountering an invalid floating-point operation (floating-point exception or FPE) that cannot be easily predicted.

It's essential to note that an **incorrect behavior** (e.g., failure to meet a postcondition for correct input data) stemming from incorrect coding is **not** an exception; it is a bug that should be debugged.

Why handling exceptions

Historically, in scientific computing, exceptions were often not handled at all or led to program termination with an error message. However, the rise of graphical interfaces and more complex software systems has made exception handling more critical. An algorithm's failure should not lead to the termination of the entire program.

There is a growing need to perform *recovery* operations when exceptions occur.

Exception handling in C++

C++ provides an effective mechanism to handle exceptions. The basic structure consists of:

- Using the `throw` command to indicate that an exception has occurred. You can throw an object containing information about the exception.
- Employing the `try-catch` blocks to catch and handle exceptions. If an exception is not caught, it will propagate up the call stack and might lead to program termination.

The `try` block contains the code that might **`throw`** an exception, while the `catch` block handles the exception.

Example

```
int safe_divide(int dividend, int divisor) {
    if (divisor == 0) {
        throw std::runtime_error("Division by zero is not allowed.");
    }
    return dividend / divisor;
}

try {
    const int result = safe_divide(10, 0); // Attempt to divide by zero.
    std::cout << "Result: " << result << std::endl;
} catch (const std::exception& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
}
```

Standard exceptions

The Standard Library in C++ provides predefined **exception classes** for common exceptions. They are accessible through the `<exception>` header. These classes derive from `std::exception`, which defines a *virtual* method `what()` to return an exception message.

```
virtual char const * what() const noexcept;
```

These standard exceptions are designed to be used or derived from when creating your own exceptions. This promotes consistency and helps others understand your error handling approach.

An overview of standard exceptions

- `std::exception`: The base class for all standard exceptions. It provides a `what()` method to retrieve an error message.
- `std::runtime_error`: Represents runtime errors.
- `std::logic_error`: Represents logical errors in the program. It includes exceptions like `std::invalid_argument` and `std::domain_error`.
- `std::overflow_error`: Indicates arithmetic overflow errors.
- `std::underflow_error`: Indicates arithmetic underflow errors.
- `std::range_error`: Indicates errors related to out-of-range values.
- `std::bad_alloc`: Used to indicate memory allocation errors.
- `std::bad_cast`: Indicates casting errors during runtime type identification (RTTI).
- `std::bad_typeid`: Used for errors related to the type identification of objects.
- `std::bad_exception`: A placeholder for all unhandled exceptions.

Example: custom exception handling in C++ (1/3)

```
class InsufficientFundsException : public std::exception {
public:
    InsufficientFundsException(double balance, double withdrawal_amount)
        : balance(balance), withdrawal_amount(withdrawal_amount) {}

    const char * what() const noexcept override {
        return "Insufficient funds: Cannot complete the withdrawal.";
    }

    double get_balance() const { return balance; }

    double get_withdrawal_amount() const { return withdrawal_amount; }

private:
    double balance;
    double withdrawal_amount;
};
```

Example: custom exception handling in C++ (2/3)

```
class BankAccount {
public:
    BankAccount(double initial_balance) : balance(initial_balance) {}

    void withdraw(double amount) {
        if (amount <= 0) {
            throw std::range_error("The requested amount is negative.");
        }

        if (amount > balance) {
            throw InsufficientFundsException(balance, amount);
        }

        balance -= amount;
    }

private:
    double balance;
};
```

Example: custom exception handling in C++ (3/3)

```
BankAccount account(1000.0);

try {
    account.withdraw(1500.0); // Or: account.withdraw(-500.0);
} catch (const InsufficientFundsException& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
    std::cerr << "Balance: " << e.get_balance()
              << ", Withdrawal amount: " << e.get_withdrawal_amount() << std::endl;
} catch (const std::range_error& e) {
    std::cerr << "Exception caught: " << e.what() << std::endl;
} catch (...) {
    std::cerr << "Unknown exception caught." << std::endl;
}
```

Old-style error control

In situations where an algorithm's failure is one of its expected outcomes (e.g., the failure of convergence in an iterative method), returning a **status** rather than throwing an exception may be more suitable. Instead of terminating the program, a status variable is used to indicate the outcome, which can be checked by the caller. See also:

- `std::exit` (normal program termination).
- `std::abort` (abnormal program termination)
- `std::terminate` (terminate with unhandled exception, with a customizable handler)

However, it's worth noting that the `try-catch` mechanism introduces overhead only when exceptions are actually thrown. Modern implementations have zero-cost when no exception occurs. High-performance code often minimizes the use of exception handling.

In practical contexts where exception handling is necessary, the `noexcept` operator can help optimize efficiency by indicating functions and methods that do not throw exceptions.

Floating point exceptions

It's important to note that **floating point exceptions** (FPE) are a special type of exception. In IEEE-compliant architectures, invalid arithmetic operations on floating-point numbers do not result in program failure. Instead, they produce special numerical values like `inf` (infinity) or `nan` (not-a-number), and the operations continue.

This unique behavior distinguishes floating point exceptions from traditional exceptions.

FPEs can be controlled using the `<cfenv>` header (`feenableexcept`, etc.). See here for details.

STL utilities

STL utilities: I/O streams

I/O streams

Input/Output (I/O) streams in C++ provide a convenient way to perform input and output operations, allowing you to work with various data sources and destinations, such as files, standard input/output, strings, and more. C++ I/O streams are part of the Standard Library (STL) and are based on the concept of streams. The key components of C++ I/O streams are `iostream`, `ifstream`, `ofstream`, and `stringstream`.

- **`iostream`**: A typedef for input and output streams. The actual base classes are `istream` (for input) and `ostream` (for output). It is used for interacting with the standard input and output streams.

```
int number;
std::cout << "Enter a number: ";
std::cin >> number;
std::cout << "You entered: " << number << std::endl;
```

File streams: open modes

The `std::ios_base` namespace defines the following options to deal with files.

Option	Description
`in`	File open for reading: the internal stream buffer supports input operations.
`out`	File open for writing: the internal stream buffer supports output operations.
`binary`	Operations are performed in binary mode rather than text.
`ate`	The output position starts **at** the **e**nd of the file, even in case of concurrent writing.
`app`	All output operations happen at the end of the file, `app` ending to its existing contents.
`trunc`	Any contents that existed in the file before it is open are truncated/discarded.

`std::ifstream`

`std::ifstream`: This class is used for reading data from files. You can open a file for input and read data from it.

```
#include <fstream>

std::ifstream file("example.txt", std::ios::in);

if (file.is_open()) {
    std::string line;
    while (std::getline(file, line)) {
      std::cout << line << std::endl;
    }
    file.close();
} else {
    std::cerr << "Failed to open the file." << std::endl;
}
```

`std::ofstream`

`std::ofstream`: This class is used for writing data to files. You can open a file for output and write data to it.

```
#include <fstream>

std::ofstream file("output.txt", std::ios::out);

if (file.is_open()) {
    file << "Hello, World!" << std::endl;
    file.close();
} else {
    std::cerr << "Failed to open the file." << std::endl;
}
```

`std::stringstream`

`std::stringstream`: This class allows you to manipulate strings as if they were input and output streams. You can use it for parsing and formatting strings.

```
#include <sstream>

// Using std::stringstream to format data into a string.
std::stringstream ss;
const int num = 42;
const double pi = 3.14159265359;

ss << "The answer is: " << num << ", and Pi is approximately " << pi;
std::cout << ss.str() << std::endl;

// Parsing data from a string using std::stringstream.
std::string input = "123 45.67";
int parsed_int;
double parsed_double;

std::stringstream(input) >> parsed_int >> parsed_double;
```

I/O formatting

Formatting: I/O streams provide various formatting options to control the appearance of output. For instance, `std::setw`, `std::setprecision`, `std::setfill`, etc., from the `<iomanip>` header, allow setting field width, precision, and fill characters in the output.

```
#include <iomanip>
const double pi = 3.14159265359;
std::cout << "Default: " << pi << std::endl;
std::cout << "Fixed with 2 decimal places: " << std::fixed << std::setprecision(2) << pi << std::endl;
std::cout << "Scientific notation: " << std::scientific << pi << std::endl;
std::cout << std::defaultfloat << std::setprecision(6);
std::cout << "Width 10 with left alignment: " << std::left << std::setw(10) << pi << ";" << std::endl;
std::cout << "Width 10 with right alignment: " << std::right << std::setw(10) << std::setfill('*') << pi << std::endl;
```

Output:

```
Default: 3.14159
Fixed with 2 decimal places: 3.14
Scientific notation: 3.14e+00
Width 10 with left alignment: 3.14159   ;
Width 10 with right alignment: ***3.14159
```

STL utilities: random numbers

Random numbers

The capability of generating random numbers is essential not only for statistical purposes but also for internet communications. But an algorithm is deterministic. However, several techniques have been developed to generate pseudo-random numbers. They are not really random, but they show a low level of auto-correlation.

C++ support for statistical distributions

C++ provides extensive support for (pseudo) random number generators and univariate statistical distributions. You need the header `<random>`. The chosen design is based on two types of objects:

1. **Engines**: They serve as a stateful source of randomness, providing random unsigned integer values uniformly distributed in a range. They are normally used with distributions.
2. **Distributions**: They specify how values generated by the engine have to be transformed to generate a sequence with prescribed statistical properties. The design separates the (pseudo) random number generators from their use to generate a specific distribution.

Engines

Random number engines generate pseudo-random numbers using seed data as an entropy source. Several different classes of pseudo-random number generation algorithms are implemented as templates that can be customized. Some basic engines include:

- `linear_congruential_engine`: Linear congruential algorithm
- `mersenne_twister_engine`: Mersenne twister algorithm
- `subtract_with_carry_engine`: Subtract-with-carry algorithm (a lagged Fibonacci)
- Many more available in the `<random>` header

For simplicity, the library provides predefined engines, such as `std::default_random_engine`, which balances efficiency and quality. There are also non-deterministic engines, like `std::random_device`, which generate non-deterministic random numbers based on hardware data.

Engines

You can generate an object of the chosen class either with the default constructor or by providing a seed (an unsigned integer). If you use the same seed, the sequence of pseudo-random numbers will be the same every time you execute the program.

```
#include <random>

std::default_random_engine rd1;           // With a default-provided seed.
std::default_random_engine rd2{1566770}; // With a user-provided seed.
```

How to use the `random_device`

The `random_device` provides non-deterministic random numbers based on hardware data. However, it is slower than other engines and is often used to generate the seed for another random engine. Here's how to use it:

```
std::random_device rd{};
std::default_random_engine rd3{rd()}; // With a randomly generated seed.
```

Default distributions in the STL

- `std::uniform_int_distribution`, `std::uniform_real_distribution`
- `std::normal_distribution`, `std::lognormal_distribution`, `std::exponential_distribution`
- `std::binomial_distribution`, `std::poisson_distribution`,
- `std::geometric_distribution`, `std::bernoulli_distribution`
- `std::discrete_distribution`
- `std::piecewise_constant_distribution`, `std::piecewise_linear_distribution`

Example: distributions

Distributions are template classes that implement a call operator `()` to transform a random sequence into a specific distribution. You need to pass a random engine to the distribution to generate numbers according to the desired distribution. For example:

```
std::random_device rd{};
std::default_random_engine gen{rd()};
std::uniform_int_distribution<> dice{1, 6};

for (unsigned int n = 0; n < 10; ++n)
    std::cout << dice(gen) << ' ';

std::cout << std::endl;
```

Here, `uniform_int_distribution` provides an integer uniform distribution in the range $[1, 6]$ (both endpoints inclusive).

seed_seq

The utility `std::seed_seq` consumes a sequence of integer-valued data and produces a requested number of unsigned integer values. It provides a way to seed multiple random engines or generators that require a lot of entropy.

For example, the internal state of the `mt19937` generator is represented by 624 integers, hence the best way to seed it is to fill it with 624 numbers based on a high-entropy source (e.g., the `random_device` provided by the operating system):

```
std::random_device rd{};
std::array<std::uint32_t, 624> seed_data;
std::generate(seed_data.begin(), seed_data.end(), std::ref(rd));
std::seed_seq seq(seed_data.begin(), seed_data.end());

std::mt19937 gen{seq};
```

You can use the generated seeds to feed different random engines.

Shuffling

In C++, you can shuffle a range of elements using the `std::shuffle` utility from the `<algorithm>` header. It shuffles the elements randomly so that each possible permutation has the same probability of appearance. Here's an example:

```
std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::random_device rd{};
std::default_random_engine g{rd()};
std::shuffle(v.begin(), v.end(), g);
```

Every time you run this code, the vector `v` will be shuffled differently.

Sampling

Another useful utility in `<algorithm>` is `std::sample`, which extracts `n` elements from a range without repetition and inserts them into another range. Here's an example:

```
int n = 10;
std::vector<double> p;
// Fill p with more than n values to sample.
std::vector<double> res;
auto seed = std::random_device{}();
std::sample(p.begin(), p.end(), std::back_inserter(res), n, std::mt19937{seed});
```

This code generates a different realization of the sample every time you run it.

STL utilities: Time measuring

Time measuring

C++ provides three common clocks:

- `std::chrono::system_clock`: Represents the system-wide real-time clock. It's suitable for measuring absolute time (can change if the user changes the time on the host machine).
- `std::chrono::steady_clock`: Represents a steady clock that never goes backward. It's suitable for measuring time intervals and performance measurements.
- `std::chrono::high_resolution_clock`: May provide the highest resolution available, but is often just an alias for `steady_clock`. Prefer `steady_clock` for portability.

Example: time measuring

```
void my_function() {
    // Code to measure.
}

auto start = std::chrono::high_resolution_clock::now();
my_function();
auto end = std::chrono::high_resolution_clock::now();

auto duration =
  std::chrono::duration_cast<std::chrono::microseconds>(end - start);

std::cout << "Time taken by function: "
          << duration.count() << " microseconds" << std::endl;
```

Example: benchmarking

```
void my_function() {
    // Code to measure.
}

const int num_iterations = 1000;

auto start = std::chrono::high_resolution_clock::now();
for (int i = 0; i < num_iterations; ++i) {
    my_function();
}
auto end = std::chrono::high_resolution_clock::now();

auto duration =
  std::chrono::duration_cast<std::chrono::microseconds>(end - start);

std::cout << "Average time taken by function: "
          << duration.count() / num_iterations << " microseconds" << std::endl;
```

`std::chrono` duration literals (since C++14)

```cpp
#include <chrono>
using namespace std::chrono_literals;

// Old way (C++11):
auto duration1 = std::chrono::milliseconds(100);
auto duration2 = std::chrono::seconds(2);
auto duration3 = std::chrono::minutes(5);

// New way (C++14):
auto duration1 = 100ms;
auto duration2 = 2s;
auto duration3 = 5min;

auto total = 1h + 30min + 45s;

// Timing with literals:
auto start = std::chrono::high_resolution_clock::now();
my_function();
auto end = std::chrono::high_resolution_clock::now();

auto duration = end - start;
if (duration > 100ms) {
    std::cout << "Function took more than 100ms!" << std::endl;
}
```

STL utilities: Filesystem

Filesystem (since C++17)

A full set of utilities to manipulate files, directories, etc. in a filesystem became available.

```
const auto path{"big/file/to/copy"};

try {
  if (std::filesystem::exists(path)) {
    const auto file_size{std::filesystem::file_size(path)};

    std::filesystem::path tmp_path{"/tmp"};

    if (std::filesystem::space(tmp_path).available > file_size) {
      std::filesystem::path new_dir = tmp_path / "example";
      std::filesystem::create_directory(new_dir);
      std::filesystem::copy_file(path, new_dir / "new_file");
    }
  }
} catch (const std::filesystem::filesystem_error& e) {
  std::cerr << "Filesystem error: " << e.what() << std::endl;
}
```

Best practices (1/2)

Smart pointers:

- Prefer `std::unique_ptr` by default for single ownership
- Use `std::shared_ptr` only when shared ownership is truly needed
- Always use `std::make_unique` / `std::make_shared` for construction
- ❌ **Avoid** raw `new` / `delete` in modern C++
- Use `std::weak_ptr` to break cycles in shared pointer graphs

Move semantics:

- Implement move constructors/assignment for resource-owning classes
- Use `std::move` to explicitly transfer ownership
- Return by value and trust RVO/NRVO (Return Value Optimization)
- Remember: named variables are always lvalues, even rvalue references

Best practices (2/2)

STL utilities:

- Prefer standard exceptions over custom error codes
- Use I/O streams for reading from/writing to files and manipulating formatted strings
- Seed random engines properly for security-critical applications
- Use `std::chrono::steady_clock` for timing measurements
- Leverage `<filesystem>` for portable file operations

A final recommendation

C++ is continuously evolving, and to maintain backward compatibility, new features are added while very few, if any, are eliminated. However, if you adopt a specific programming style, you'll find yourself using only a subset of what C++ has to offer.

The more outdated and cumbersome features that make programming more complex and less elegant will gradually be used less and less.

It's advisable to start incorporating the new features that genuinely assist you in writing cleaner, simpler code. Most of the features illustrated here move in that direction.

But **always remember**: the most important aspect of your code is whether it accomplishes the right task. An elegant code that yields incorrect results is of no use.

Static and shared libraries.