Lecture 07

Smart pointers, move semantics, STL utilities.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

07 Nov 2023

Outline

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

Smart pointers

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<T>`: 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(std::string t) {
    switch (t) {
        case "Triangle":
            return new Triangle{...};
        case "Square":
            return new Square{...};
        default:
            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:
    set_polygon(std::unique_ptr<Polygon> p) {
        polygon = std::move(p);
    }
private:
    std::unique_ptr<Polygon> polygon;
}

std::unique_ptr<Polygon> create_polygon(std::string t) {
    switch (t) {
        case "Triangle":
            return std::make_unique<Triangle>(...); // 'make_unique' available since C++14.
        case "Square":
            return std::make_unique<Square>(...);
        default:
            return std::unique_ptr<Polygon>(); // null pointer.
    }
}
```

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 `if (ptr)` or using it in a boolean context.

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.
- `ptr1.reset(ptr2)`: Equivalent to `ptr1 = std::move(ptr2)`.
- `ptr.get()`: Returns a standard pointer to the handled resource.
- `ptr.release()`: Returns a standard pointer, releasing the resource without deleting it. `ptr` becomes empty.

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

Shared pointers

For instance you have several objects that **refer** to a resource (e.g., a matrix, a shape, ...) that is build dynamically (and maybe is a polymorphic object). 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` do not cause any computational overhead they are just a light wrapper around an ordinary pointer), shared pointers do, so use them only if it is really 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) unique pointer, and you can check if a `std::shared_ptr` is empty by testing `if (ptr)` or using it in a boolean context.

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;
```

`std::weak_ptr`

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

ptr = std::make_shared(5); // Delete managed object, acquires new pointer.
std::weak_ptr<int> weak2 = sptr; // 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;
```

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.

```
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

The problem: swap may be costly

Let's consider this function that swaps the arguments:

```
void swap(Matrix& a, Matrix& b) {
    Matrix tmp{a}; // Make a copy of a.
    a = b;         // Copy assign b to a.
    b = tmp;       // Copy assign tmp to b.
}
```

If `a` and `b` are of big size, this function is very inefficient.

- **Memory inefficient**: we have to store `tmp`.
- **Computationally inefficient**: copy operations imply copying all matrix elements.

In this code, an unintended copy of the matrix occurs during the swap operation.

We need to find a way to prevent these unnecessary copies.

A better swap (before C++11)

Let's assume that `Matrix` stores dynamic data for its elements as a `double* data` (maybe it is better to use a standard vector, but it is not relevant here). Before the introduction of move semantics, I could have solved the problem by writing a special method or a **friend** function. For instance:

```
void swap_with_move(Matrix& a, Matrix& b) {
    // Swap number of rows and columns.

    double* tmp = a.data; // Save the pointer.
    a.data = b.data;      // Copy the pointer.
    b.data = tmp;         // Copy the saved pointer.
}
```

This way I just swap the pointers, saving memory and operations, but **only for this specific situation**. It is not generalizable: I cannot write a function template `swap_with_move<T>` because I need to know how data is stored in `T`, for each case.

Questions to be addressed

To implement move semantics, three questions have to be addressed:

1. How can we identify objects that can be safely *moved* instead of copied, so that the compiler may perform the move automatically, whenever possible?
2. How can I actually implement the move in a uniform and general way?
3. How can I specify that I want to *move*, instead of copying?

Let's give the answer one question at a time. For the first one, we need to introduce value categories.

Categories of values

In C++, a value is characterized by its *type* and its *category*, which expresses how the value can be used.

In C++, we have 4 categories for values: *glvalue*, *prvalue*, *xvalue*, and *lvalue*. Moreover, they can be const or non-const.

To simplify matters (without losing important information), we will only use 2 categories: *lvalue* (which also includes glvalue) and *rvalue* (which includes prvalue and xvalue).

lvalues and rvalues in C

The original definition of lvalues and rvalues from the earliest days of C is as follows:

> An **lvalue** is an expression that may appear on the left and on the right-hand side of an assignment.

> An **rvalue** is an expression that *can only appear on the right hand side of an assignment*.

Example

```
double fun(); // A function returning a double.

3.14 = a; // Wrong: a literal expression is an rvalue!
fun() = 5; // Wrong: returning an object generates an rvalue!
```

lvalues and rvalues in C++

User-defined types, `const`, and operator overloading make the definition of rvalues/lvalues rather complicated in C++. We avoid the formal definition contained in the standard (very technical). We give a simple definition, correct in most cases:

> An **lvalue** is an expression that refers to a memory location and allows us to take its address via the `&` operator.

> An **rvalue** is an expression that is not an lvalue.

For this reason, lvalue is nowadays interpreted as *locator-value* and no more left-value. It is still true that *a (non-const) rvalue can only be at the right-hand side of an assignment*.

Examples of lvalues

The value held in a variable (i.e., a value with a name) is *always* an lvalue. Even if it is `const` or a `constexpr`, since we can take its address.

```
double a;
int const b = 10;
double* pa = &a; // Address of a.
int const* pb = &b; // Address of b.
```

If a function returns an *lvalue* reference (`&`), the returned value is an lvalue.

```
double& f(double & x) { x *= 3; return x; }

double y = 8.0;
f(y) = 3.0;
double* px = &(f(y)); // Address of y.
```

Examples of rvalues

The value returned by a function is an rvalue.

```
double fun(double x) { ... }
```

Here, `&fun` is a pointer to the function, *not* to the returned value. I cannot take the address of the returned value; it's a temporary object.

Non-string literals are rvalues.

```
double* pd = &(10.5); // Error (taking the address of a temporary doesn't make sense).
```

Compilers are free not to store them in memory, so no address may be taken (and it does not make sense to take it).

Strings, however, are lvalues.

How can we identify objects that can be safely *moved* instead of copied?

Non-const rvalues are eligible for "automatic moving". Indeed, if we cannot take the address, it means that they exist only to be stored somewhere.

So we have the answer to the first question: *rvalues are movable*. In particular, values returned by a function are movable.

How can I actually implement the move in a uniform and general way?

To answer the second question, let's look at how *references* bind according to the category of the bound values.

We consider ordinary references first, from now on called *lvalue references*. A non-const lvalue reference *cannot bind to rvalues*, while both lvalues and rvalues can be bound to const lvalue references.

Reference binding

```
double & pi = 3.14; // Wrong: A literal expression is an rvalue.
double const & another_pi = 3.14; // Ok!

int foo(); // Return an rvalue.
int & foo(int & a); // Return a reference, thus an lvalue.
int goo(const int & a); // Returns an rvalue.

auto p = foo(); // Ok: p is an int.
int & c = foo(p); // Ok: the function returns an lvalue here!
int & d = foo(3); // NO! 3 is an rvalue and cannot be bound to an (lvalue) reference.
auto & x = goo(foo()); // NO! as above.
const int & a = goo(foo()); // Ok, an rvalue binds to a const lvalue reference.
```

Reference binding in overloaded functions

The interplay between reference types and binding is clear (and important) when looking at function overloading.

```
void foo(int & a);
void foo(const int & a);
void goo(const int & a);
void zoo(int & a);

int g;
const int b = 10;

foo(5); // Calls foo(const int &)
foo(g); // Calls foo(int &)
goo(g); // Calls goo(const int &);
foo(b); // Calls foo(const int &)
goo(b); // Calls goo(const int &);
zoo(b); // Error: a const lvalue can bind only to a const lvalue reference.
```

Conclusion on lvalue reference binding

- A non-const lvalue reference can bind only, and preferably, to non-const lvalues.
- A const lvalue reference binds both to lvalues and rvalues, const and non-const alike.

Here, "preferably" means that it will be chosen in case there is a choice.

This is **before** C++11. In fact, it is still true if we just use lvalue references.

The consequence is that with just lvalue references, we cannot distinguish lvalues from rvalues.

Relation with moving

Let's examine the following code

```
Matrix foo(); // A function returning a large object.

Matrix a;
a = foo();
```

The return value of `foo` could be moved into `a` safely! (Indeed, the Return Value Optimization already does that for constructors).

It would be beneficial to have an "adornment" that acts like a reference, while ensuring that **it binds exclusively to rvalues and preferably to rvalues**. This way, we can overload the assignment operator as follows:

```
Matrix & operator=(const Matrix & a); // Ordinary copy.
Matrix & operator=(Matrix "new adorn" a); // Move!
```

rvalue reference

Indeed, C++11 has introduced a new kind of adornment, called **rvalue reference**, indicated by `&&`.

It **exclusively and preferably binds to rvalues**. Preferably means that, if given the choice, an rvalue binds to an rvalue reference.

An important thing to remember is that **rvalue references bind rvalues and only rvalues**.

Categories of values

We resume some rules:

- If a function returns a value, that value is considered an **rvalue**.
- If a function returns an lvalue reference (const or non-const), that value is considered an lvalue.
- If a function returns an rvalue reference, that value is an rvalue.
- A (named) variable is **always an lvalue**.

This is fundamental for move semantics.

How is move semantics implemented?

We are now able to answer the second question. The key is **the move constructor and the move assignment operators**.

This is the standard signature of move operations for a class named `Matrix`:

```
Matrix(Matrix&&); // Move constructor.
Matrix & operator=(Matrix&&); // Move assignment operator.
```

Remember that unless you have defined some other constructors or the copy assignment, the compiler provides a synthetic move constructor and move assignment operator automatically, which apply the corresponding moving operation on the non-static data members of the class.

Move semantics for `Matrix` (1/2)

Let's go back to `Matrix`. Assume that `Matrix` stores the data as a pointer to `double`. A possible copy-constructor and copy-assignment take the form:

```
Matrix(const Matrix & rhs) : nr(rhs.nr), nc(rhs.nc), data(new double[nr * nc]) {
    // Make a deep copy.
    for (i = 0; i < rhs.nr * rhs.nc; ++i)
        data[i] = rhs.data[i];
}

Matrix & operator=(const Matrix & rhs) {
    // Release current resource.
    delete[] this->data;
    // Get a new data buffer.
    data = new double[rhs.nr * rhs.nc];
    // Make a deep copy.
    for the i = 0; i < rhs.nr * rhs.nc; ++i)
        data[i] = rhs.data[i];
}
```

Move semantics for `Matrix` (2/2)

The corresponding move operator could be:

```
Matrix(Matrix&& rhs) : nr(rhs.nr), nc(rhs.nc), data(rhs.data) {
    // Fix rhs so it is a valid empty matrix.
    rhs.data = nullptr;
    rhs.nr = rhs.nc = 0;
}

Matrix & operator=(Matrix&& rhs) {
    delete[] this->data; // Release the resource.
    data = rhs.data; // Shallow copy.
    // Fix rhs so it is a valid empty matrix.
    rhs.data = nullptr;
    rhs.nr = rhs.nc = 0;
}
```

I just grab the resource and leave an empty matrix! **It is important to ensure that the moved object can be deleted correctly!**

The consequence

```
Matrix foo();
// ...
Matrix a;
a = foo(); // A move assignment is called.
```

We can say that a class implements move semantics when the move operators are defined, even if they are automatically by the compiler.

Move semantics and perfect forwarding

Now, let's address the third question: **How can I explicitly instruct the compiler to perform a move instead of a copy operation when move semantics are implemented (possibly with the synthesized move operators)?** This question can be divided into two parts:

Move: How to explicitly tell the compiler to replace a copying operation with a move if move semantics are implemented (perhaps with the synthesized move operators).

Perfect forwarding: How to write function templates that accept arbitrary arguments and forward them to other functions in a way that the target functions receive the values with the same category they were passed to the forwarding function. *This topic will not be covered in this course but you can find a good explanation here.*

Forcing a move: `std::move`

Well, first of all, `std::move` doesn't move anything. They have chosen a wrong name; they should have called it `std::movable` instead. But we have to live with it.

`std::move(expr)` unconditionally casts `expr` to an rvalue. So it makes it available to be moved.

You use it to indicate to the compiler that you want something to be moved, even if it is an lvalue. It is actually moved if move semantics has been implemented for that type. If not, it will be copied.

A new (generic) version of `swap`

Now we are able to write our `swap`, and in a generic way!

```
template<class T>
void swap(T& a, T& b) {
  T tmp{std::move(a)}; // Move constructor.
  a = std::move(b);    // Move assignment operator.
  b = std::move(tmp);  // Move assignment operator.
}
// Or, even simpler:
std::swap(a, b);
```

⚠️ If your class stores its dynamic and potentially large data in standard containers, you just need the synthetic move operators (which means that you have move semantics for free!). Another good reason to use standard containers.

⚠️ If type `T` implements move semantic, the swap is made using the move operators, and, if implemented correctly, with less memory requirement. If not, we have the usual copy.

Once more: variables are *always* lvalues

Named variables are always lvalues! Even if they are declared as rvalue references. In fact, you can take their address!

In particular, **function parameters (of any function, including constructors) are lvalues**, even if their type is an rvalue reference.

Inside the scope of this function:

```
void f(Matrix&& m) {
  // ...
}
```

`m` is an **lvalue**.

The solution

You have to force the move:

```
class Foo {
public:
  Foo(Matrix&& m) : my_m{std::move(m)} {}
  // ...
private:
  Matrix my_m;
}
```

Now, `my_m{std::move(m)}` calls the **move constructor**, and `m` is moved into `my_m`.

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 supports 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.

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 met, the error message is printed to the standard error 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 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 = 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 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;
    }

    double get_balance() const {
        return balance;
    }

private:
    double balance;
};
```

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

```
Bank_account 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::terminate`, `std::abort`, and, `std::exit`.

Exception handling is increasingly important in code that must be integrated into a broader workflow or graphical interface. However, it's worth noting that the `try-catch` mechanism introduces some inefficiencies since it checks for exceptions every time a function is called. High-performance code often minimizes the use of exception handling.

In practical contexts where exception handling is necessary, the `noexcept` declaration 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.

There are ways, not covered in this course, to properly handle FPEs.

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`**: The base class for input and output streams. It is derived from `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.
`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.

```
std::ifstream file("example.txt", open_mode);

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.

```
std::ofstream file("output.txt", open_mode);

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 work with strings as if they were input and output streams. You can use stringstream for parsing and formatting strings.

```
// 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.

```
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.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: 3.141593e+00
Width 10 with left alignment: 3.14e+00  ;
Width 10 with right alignment: **3.14e+00
```

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.

```
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 random 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`
- You can create custom distributions by subclassing the `std::random_distribution` class and providing your own probability distribution function.

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).

`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`: Represents a high-resolution clock with the smallest possible tick duration. It's often used for precise timing.

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;
```

STL utilities: Filesystem

Filesystem

Since C++17, a full set of utilities to manipulate files, directories, etc. in a filesystem is available.

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

if (std::filesystem::exists(big_file_path)) {
  const auto big_file_size{std::filesystem::file_size(big_file_path)};

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

  if (std::filesystem::space(tmp_path).available > big_file_size) {
    std::filesystem::create_directory(tmp_path.append("example"));
    std::filesystem::copy_file(big_file_path, tmp_path.append("new_file"));
  }
}
```

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.