Cpp Nuances

Nathan Warner

Computer Science
Northern Illinois University
United States

Contents

Converting char to std::string

Suppose we have a char variable, and we need to ”convert” it to a string. To do this we use the string class constructor, which has two parameters, the size of the string to create, and the character to use as the fill.

1.1 Constructor signature

```
string(size_t n, char x)
```

1.2 Example

```
char c = 'a';
string s(1,c);
```

std::string::npos

Concept 1: In C++, std::string::npos is a static member constant value with the greatest possible value for an element of type size_t. This value, when used as the length in string operations, typically represents "until the end of the string." It is often used in string manipulation functions to specify that the operation should proceed from the starting position to the end of the string, or until no more characters are found.
□

2.1 Example

```
string infix = buffer.substr(index + 2, string::npos);
```

std::string::npos is defined as the maximum value representable by the type size_t. This value is typically used to signify an error condition or a not-found condition when working with strings and other sequence types. However, when used as a length argument in methods like std::string::substr, it effectively becomes a directive to process characters until the end of the string. This is because any attempt to access beyond the end of the string would exceed the string's length, and the methods are designed to stop processing at that point.

Narrowing

Narrowing happens when:

- A value of a larger or more precise type is converted to a smaller or less precise type (e.g., double to int, int to char).
- A floating-point value is converted to an integer type.
- An integer value is converted to a smaller integer type (e.g., int to short) and does not fit within the destination type's range.

Aggregate types

Aggregate types in C++ are simple data structures that hold collections of values and have minimal additional behavior. They are essentially ”plain old data” structures that are easy to initialize and manipulate. The term ”aggregate” is formally defined by the C++ standard.

A class, struct, or union is considered an aggregate if it satisfies all of the following conditions:

1. No User-Defined Constructors: It must not have any explicitly declared constructors (including default, copy, or move constructors).
2. No Private or Protected Non-Static Data Members: All non-static data members must be public.
3. No Virtual Functions: It must not have any virtual functions.
4. No Base Classes: It must not inherit from another class or struct.
5. No Virtual Base Classes: It must not use virtual inheritance.

For example

4.1 Aggregate initialization

Aggregate initialization in C++ refers to a special form of initialization for aggregate types using an initializer list enclosed in {} braces. This allows you to directly specify values for the members of an aggregate type in the order they are declared.

Each member of the aggregate is initialized with the corresponding value provided in the initializer list. The order of values in the initializer list must match the declaration order of members in the aggregate.

If fewer values are provided in the initializer list than there are members in the aggregate, the remaining members are value-initialized (e.g., zero-initialized for fundamental types).

If the type is not an aggregate (e.g., has a user-defined constructor, private members, or virtual functions), aggregate initialization cannot be used.

```
struct S {
    int x,y;
};

S s = {1,2};
```

Consider the next example,

```cpp
struct S {
    int x,y;

    S(int x, int y) : x(x), y(y) {}
};

S s = {1,2}; // Works fine
```

Since the class S has a user-defined constructor, it is no longer an aggregate type.

Aggregate initialization is not applicable here. Instead, C++ checks for constructors that match the initializer list {1, 2}.

Since it found one, the compiler interprets it as calling the constructor S(int x, int y) with the arguments 1 and 2.

In modern C++ (C++11 and later), brace-enclosed initialization is often used for uniform initialization.

If a constructor is available that matches the initializer list, it is called.

4.2 Narrowing is not allowed in aggregate-initialization from an initializer list

The error "narrowing is not allowed in aggregate-initialization from an initializer list" occurs in C++ when you try to initialize an aggregate type (e.g., structs, arrays, or classes with no user-defined constructors) using values in an initializer list, but the values undergo an implicit narrowing conversion that could lose information or precision.

```cpp
struct A {
    int x;
    float y;
};

A a = {1, 3.14}; // OK, no narrowing
A b = {1, 3.14f}; // OK, no narrowing (3.14f is a float literal)

A c = {1, 3.14}; // ERROR: narrowing from `double` to `float`
```

4.3 implicit conversion using a single-argument constructor

A constructor that takes one argument can be used for implicit conversion of that argument type to the class type.

```cpp
struct s {
    int x;
    s(int value) : x(value) {} // Single-argument constructor
};
s s1 = 1;
```

The integer 1 is implicitly converted to an object of type s using the s(int value) constructor.

By default, a single-argument constructor allows implicit conversions. If you want to prevent implicit conversions and require explicit construction, you can use the explicit keyword

```cpp
struct s {
    int x;
    explicit s(int value) : x(value) {}
};

int main() {
    s s1 = 1; // ERROR: Explicit constructor prevents implicit conversion
    s s2(1);  // OK: Direct initialization
}
```

Floating point literals

Floating point literals in c++ will be of type double. For example,

```
cout << typeid(4.09).name(); // d
```

Append *f* to the literal to make it a float

```
cout << typeid(4.09f).name(); // d
```

Size of structs and classes

Consider the code

```
struct s { }
cout << sizeof(s) << endl; // 1

s s1;
cout << sizeof(s1) << endl; // 1
```

Notice that empty structs do not have a size of zero. Empty structs have a size of one byte.

A size of zero for a struct would mean that it occupies no memory. If multiple instances of such a struct are created, they would have no unique memory location to occupy. As a result, the compiler would assign the same memory address to all instances, which would violate fundamental rules of object-oriented programming in C++.

Furthermore, giving fields to the struct or class increases the size of that struct or class by size of the type.

```
struct s { int x; }
cout << sizeof(s); // 4
```

Notice that the size is **not** 5. Creating functions and creating variables in those functions does not add to the size

```
struct s {
    void f()  {
        int  x;
    }
}
cout << sizeof(s); // 1
```

Creating structs inside structs and even adding fields to the inner structs does not increase the size

```
struct s {
    struct k { int x; };
};

cout << sizeof(s); // 1
cout << sizeof(s::k); // 4
```

Lastly, constructors and destructors do not increase the size.

It seems only fields increase the size.

6.1 Padding

Consider the following struct

```
struct s {
    char c; // 1 byte
    int x; // 4 bytes
};
cout << sizeof(s) << endl; // 8
```

Notice that we do not get a size of 5. In C++, padding is introduced by the compiler to ensure proper memory alignment of data members. This improves performance on most hardware architectures. The first member, char c, occupies 1 byte. The next member, int x, requires 4 bytes and should be aligned to a 4-byte boundary (on most systems). To achieve this alignment, 3 bytes of padding are added after c before x starts.

When is trailing return type useful

7.1 Decltype on template parameters

Suppose we had two template types T, U, and a function that accepts Ta, Ub. Suppose we wanted the return type to be the type of $T + U$, we could of course just write

```
template<typename T, typename U>
decltype(T{} + U{}) f(T a, U b) {
    return a + b;
}
```

Or, we could utilize a trailing return

```
template<typename T, typename U>
auto f(T a, U b) -> decltype(a+b) {
    return a + b;
}
```

7.2 Nested structs

Consider the code

```
struct A {
    struct B {};

    B f() const;
};

A::B A::f() const { }
```

Instead of having to use the scope resolution operator, we could use a trailing return type.

```
auto A::f() const -> B {

}
```

By using a trailing return type, we are essentially inside the scope of A by the time we specify the return type.

Most vexing parse

The most vexing parse is a phenomenon in C++ where a line of code that looks like a variable declaration is instead interpreted by the compiler as a function declaration. This often happens because of C++'s ambiguous grammar for declarations.

```cpp
std::string s(); // Treated as a function declaration
```

This will not be treated as a default constructed string by the C++ compiler, but instead as a function declaration. To fix this issue, we instead use brace initialization

```cpp
std::string s{} // A default string variable
```

Notes about copy constructors

The copy constructor in C++ typically takes a const reference (const T&) to ensure correctness and efficiency. If the copy constructor took its parameter by value, this would require making a copy of other before calling the constructor itself. But that copy itself would require calling the copy constructor, leading to infinite recursion until a stack overflow.

If the copy constructor took a non-const reference, this would not allow copying from const objects.

Since a copy operation should not modify the original object, using const ensures the copy constructor can be called for const objects.

Recall that functions (and methods) that take a const reference insure that the function can be called on both const objects and non-const objects. Functions that take non-const references can only accept non-const objects.

As a quick side note, since we are on the topic of const correctness, recall that constant member functions cannot call non-const member functions.

Exceptions during a call to new

When using raw pointers, an exception occurring after a new allocation leads to a resource leak if the allocated memory is not properly managed. This happens because new dynamically allocates memory on the heap, but if an exception interrupts execution before delete is called, the allocated memory remains unreachable and never freed.

To prevent this, use smart pointers (std::unique_ptr or std::shared_ptr), which automatically manage memory and clean up even if an exception occurs.

However, it is true that using std::unique_ptr and std::shared_ptr does not guarantee that there will be no resource leaks if they are not used properly. Specifically, passing new directly to their constructors can still lead to resource leaks in certain cases. This is why std::make_unique and std::make_shared are preferred

make_shared performs a single allocation that stores both the object and the control block together. If an exception occurs, no memory leak happens because allocation and ownership setup happen in one step.

Same is true for make_unique

10.1 Exceptions during constructors

If a call to new results in an exception, the object is never allocated. When you use new, the operator first tries to allocate memory by calling operator new(size), which is similar to malloc.

If memory allocation is successful, the constructor of the object is called.

If there isn't enough memory, operator new will throw a std::bad_alloc exception (unless you use nothrow new, which returns nullptr)

If memory allocation succeeds but the constructor of the object throws an exception, the allocated memory is automatically freed, so there is no memory leak. If an exception is thrown in the constructor after memory has been allocated but before construction is complete, the C++ runtime ensures that the allocated memory is automatically freed. This is done by:

- Catching the exception inside the new operator.
- Calling operator delete(ptr) to deallocate the memory.

Notes about polymorphism

Consider the code

```
struct base {
    virtual void print() const {
        cout << "Base" << endl;
    }

    virtual ~base() {
        cout << "Cleaned up base" << endl;
    }
};
struct derived : base {
    void print() const override {
        cout << "Derived" << endl;
    }

    ~derived() {
        cout << "Cleaned up derived" << endl;
    }
};

void f(const base* o) {
    o->print();
}

void f(const base& o) {
    o.print();
}

derived d1;
base& b1 = d1;

f(b1);

base* d = new derived();
f(d);

delete d;
```

We get the output

```
Derived
Cleaned up derived
Cleaned up base
```

The reason both destructor messages (”Cleaned up derived” and ”Cleaned up base”) are printed when you call delete d; is due to polymorphic destruction.

The base class has a virtual destructor (virtual ∼base()). This ensures that when an object is deleted through a pointer to base, the destructor of the derived class will also be invoked before the base destructor.

d is deleted, and because base has a virtual destructor, the destructor call correctly cascades down the inheritance chain:

- First, ∼derived() runs and prints ”Cleaned up derived”.
- Then, ∼base() runs and prints ”Cleaned up base”.

If the destructor in base wasn't virtual, deleting a derived object through a base* pointer would cause undefined behavior (likely only ∼base() would be called, leading to memory leaks).

If base has fields (member variables), whether directly inherited by derived or not, the base destructor must be called to clean up those fields properly when a derived object is deleted.

11.1 Polymorphism with arrays and slicing

11.1.1 Object Slicing

Object slicing occurs when an object of a derived class is assigned to a variable of a base class type, causing the derived part of the object to be ”sliced off”, leaving only the base class portion.

Slicing occurs when

- A derived object is assigned to a base class object (not a pointer or reference).
- A derived object is stored in a container of base class objects (e.g., an array of base objects).

```
struct base {
    int x{1};

};
struct derived : base {
    int y{5};
};

base b = derived{};
// Error, no member named y in base
cout << b.x << endl << b.y;
```

11.1.2 Polymorphism with arrays

Consider the following code

```cpp
struct base {
    virtual void print() const {
        cout << "Base" << endl;
    }

    virtual ~base() {
        cout << "Cleaned up base" << endl;
    }
};

struct derived : base {
    void print() const override {
        cout << "Derived" << endl;
    }

    ~derived() {
        cout << "Cleaned up derived" << endl;
    }
};

void f(const base& o) {
    o.print();
}

auto main(int argc, const char* argv[]) -> int {
    base* barr = new derived[5];

    delete[] barr;

    return EXIT_SUCCESS;
}
```

barr is declared as base*, but the memory actually holds derived objects. When delete[] barr; is called, C++ treats barr as an array of base objects, not derived objects. This breaks proper destructor calls, leading to undefined behavior.

Even though base has a virtual base(), the problem is not about virtual dispatch. Instead, it's about how C++ tracks array allocations.

The array of derived objects was allocated using new derived[5], but delete[] barr; doesn't have enough information to correctly call derived destructors.

When delete[] barr; is called, C++ sees a base* pointer and assumes it points to an array of base objects.

Instead we either,

```cpp
derived* darr = new derived[n]
```

```cpp
vector<unique_ptr<base>> v(5);
for (auto& item : v) {
    item = make_unique<derived>();
}

// Or the standard approach
vector<base*> v2(5);
for (auto& item : v2) {
    item = new derived();
}

for (auto& item : v2) {
    delete item;
}
```

11.2 Runtime polymorphism, dynamic dispatch, dynamic binding, and the vtable

Runtime polymorphism in C++ is achieved through function overriding and is implemented using virtual functions in an inheritance hierarchy. It allows a derived class to provide a specific implementation of a function that is already defined in its base class.

11.2.1 Dynamic dispatch and dynamic binding

Dynamic dispatch is a mechanism where the function to be executed is determined at run-time, rather than at compile-time. This is a key feature of runtime polymorphism and is enabled by virtual functions in C++

When a virtual function is declared in a base class and overridden in a derived class, C++ does not resolve function calls at compile-time.

Instead, when a function is called using a base class pointer/reference, C++ performs a runtime lookup to determine which function to execute.

This lookup is performed using the VTable (Virtual Table) and VPtr (Virtual Pointer) mechanism.

As a side note, the following will not work

```cpp
struct foo {
    consteval virtual int print() const {
        return 12;
    }

    virtual ~foo() {}
};

struct bar : foo {
    consteval int print() const override {
        return 0;
    }
};

foo* f = new bar{};
constexpr int x = f.print();
```

The issue is that since the print functions are determined and called at runtime, it cannot be a compile time constant expression. We get the error ”The value of f is not usable in a constant expression”

Dynamic binding (also called late binding) is the underlying mechanism that allows dynamic dispatch to work. It refers to the process of determining which function implementation should be executed at runtime, rather than at compile time.

Binding refers to associating a function call with a function definition. Dynamic binding means this association happens at runtime, based on the actual type of the object.

Dynamic binding ensures that when a function is called on a base class pointer/reference, the correct overridden function from the derived class is invoked at runtime. Dynamic dispatch is the result of dynamic binding

11.2.2 The Vtable

C++ implements runtime polymorphism using a VTable (Virtual Table) and a VPointer (VPtr)

A VTable is a table of function pointers maintained per class. It stores pointers to the virtual functions defined in a class. Each class with virtual functions has a single VTable.

Each object of a class that has virtual functions contains a hidden pointer, called VPtr (Virtual Pointer). VPtr points to the VTable of that particular class.

During runtime, when a virtual function is called via a base class pointer, the VPtr is used to look up the correct function implementation in the VTable.

When the program starts, the compiler creates a VTable for every class that has virtual functions. Every object of such a class gets a hidden VPtr, which points to the corresponding VTable.

When a virtual function is called using a base class pointer, the call is resolved dynamically by checking the VTable.

The VTable (Virtual Table) is created at compile time, but the VPtr (Virtual Pointer) is assigned and used at runtime.

The VTable itself is constructed at compile time, meaning the compiler generates and lays out the function pointers in the table before the program runs.

The VPtr is assigned at runtime, when an object of the class is created.

Because having virtual functions in our struct / class requires this VPtr to be created, having virtual functions therefore will increase the size of the struct and the objects created by the size of a pointer (8 bytes on 64-bit systems or 4 bytes on 32-bit systems).

each class in the inheritance hierarchy that has at least one virtual function has its own vtable.

- A base class with virtual functions has a vtable that stores pointers to its virtual functions.
- A derived class that overrides any virtual functions gets its own vtable, which replaces the base class's function pointers with the derived class's implementations.
- The vtable is associated with a class, not individual objects.
- Each object of a class with virtual functions has a vptr (virtual table pointer) that points to the vtable of its actual type.

Consider the code

```
struct foo {
    virtual void print() const {
        cout << "Foo" << endl;
    }

    virtual ~foo() {}
};

struct bar : foo {
    void print() const override {
        cout << "Bar" << endl;
    }
};
```

Since foo has virtual functions (print and the destructor), it will have a vtable.

Index	Function Pointer
0	foo::print()
1	foo::~foo() (destructor)

Every instance of foo has a vptr (virtual table pointer) pointing to this table.

Since bar overrides print(), but still inherits the virtual destructor from foo, its vtable will have the overridden version of print().

Index	Function Pointer
0	bar::print()
1	foo::~foo() (inherited destructor)

Each instance of bar will have its vptr pointing to vtable for bar.

```
int main() {
    foo* obj = new bar();
    obj->print();  // Calls bar::print() because vptr points to
    ↪ bar's vtable
    delete obj;    // Calls foo::~foo() due to virtual destructor
}
```

11.3 Can you call pure virtual methods

You cannot instantiate abstract classes. Thus, you cannot call pure virtual methods

11.4 Do the signatures need to exactly match on overridden functions

Yes, the function signature must exactly match the base class function (including return type, parameters, and const qualifiers). If the signature differs in any way, the function in the derived class will be considered a new, independent function rather than an override.

11.5 What happens if you forget the keyword virtual on base class methods

Consider

```
struct s{
    void print() const & {cout << "Base" << endl;}
};

struct k : s{
    virtual void print() const & override {cout << "Derived" <<
    ↪ endl;}
};
```

We get a compiler error *print marked override but does not override any member functions*

11.6 Do you need the keyword virtual on overridden methods?

In the derived class, you do not need to explicitly write virtual again when overriding, though you can include it for clarity.

11.7 What happens if you forget the override

forgetting to put the override keyword when overriding a base class member function has the following consequences

The code will still compile, but if the function signature does not exactly match the base class function, you might accidentally create a new function in the derived class instead of overriding the base function.

If the function signature is incorrect (e.g., different return type, wrong parameters, or missing const qualifier), the base class function is hidden instead of being overridden. This means

- Calls to the function through a base class pointer/reference will not invoke the derived class function as intended.
- Instead, the base class implementation will be called, leading to unexpected behavior.
- If the function in the derived class has a different parameter list, the base class function can only be accessed using Base::functionName(...).

The override keyword forces the compiler to check whether the function actually overrides a virtual function from the base class. Without it, if the base class function signature changes, the derived class function may silently stop overriding it, leading to subtle runtime errors.

```
struct A {
    virtual void print() const {cout <<  "A"; }
};

struct B : A {
    virtual void print() {cout << "B";}
};

A* b = new B;
b->print(); // A

delete b;
```

We forgot the override keyword, which would be fine as long as the function signatures still matched exactly, but notice that we also forgot the mark B's print const. Thus, B's print method does not override A's, and it is an independent function. Therefore, B's VTable remains

Index	fn
0	A::print()

Ie it does not get overridden. If we marked B's print with override, it would give a compiler error since the signatures don't match

11.8 Overrides with default args

Consider the code

```cpp
struct A {
    virtual void print(int x=5) const {cout <<  "A" << x; }
};

struct B : A {
    virtual void print(int x=10) const override {cout << "B" << x;}
};

A* b = new B;
b->print(); // B5

delete b;
```

So B's print gets called, but we get the default arg from A?

Since print() is virtual, dynamic dispatch occurs. The runtime determines that B::print() should be executed, not A::print().

However, default arguments are resolved at compile time, based on the static type of the pointer.

The call b->print(); is interpreted at compile time as

```cpp
b->print(5); // Because `b` is an `A*`, it uses A's default argument
```

11.9 Private and public with polymorphism

Recall that for a struct S

```cpp
struct S {
private:
    void print() {}
};

S s;
s.print() // Error: Print is private
```

We cannot call private methods through an object. However, consider the code

```cpp
struct base {
    virtual void print() const {cout << "Base" << endl;}
};

struct derived : base{
    private:
    void print() const override {cout << "Derived" << endl;}
};
base* b = new derived{};
b->print(); // "Derived"
```

So why does this work? Why are we able to call this private method?

In C++, access control (private, protected, public) is enforced at compile time and is based on the class from which the call is made. However, when using dynamic dispatch (i.e., calling a virtual function through a base class pointer), the actual function that gets invoked is determined at runtime via the vtable mechanism, which is independent of access control.

11.10 Virtual methods are implicitly inline

In C++, virtual methods are implicitly considered inline by the compiler, but this doesn't mean they are always inlined at runtime.

The inline keyword in C++ does not necessarily mean "replace the function call with its body" (though that is one possible effect). Instead, it has two key meanings:

- **Allows multiple definitions across translation units**:
  - Normally, a function or method definition must appear in only one .cpp file (ODR - One Definition Rule).
  - An inline function can be defined in multiple translation units as long as all definitions are identical.
- **Hints for compiler optimization**: The compiler may replace calls to an inline function with its body (but this is not guaranteed).

Virtual Methods Are Typically Defined in Header Files

- Virtual methods must be known at compile-time to construct the vtable (virtual table) correctly.
- Because virtual methods are often declared in a class definition (inside a header file), they must be allowed in multiple translation units.
- Making them implicitly inline prevents ODR violations when the class is included in multiple .cpp files.

The primary reason for marking functions inline to avoid ODR (One Definition Rule) violations applies to functions defined in header files that are included in multiple translation units.

Normally, if you define a function in a header file, and that header is included in multiple .cpp files, you would get multiple definitions of the same function when linking.

Marking the function inline tells the compiler that all definitions of the function in different translation units are identical and should be treated as a single definition.

If a function is defined only in a .cpp file, it will not be included in multiple translation units, so inline is unnecessary.

The existence of structs and classes

Does a Struct Exist in Memory Before an Object is Created? Yes and No. It depends on what part of the struct you're referring to:

- **Struct Definition (Type Information) – Exists at Compile Time**: The struct itself is just a blueprint (like a class). It does not occupy memory on its own.

  Only when you create an object of the struct does memory get allocated for its members.

- **VTable (for Virtual Functions) – Exists in Memory, Even Without Objects**: If the struct contains virtual functions, the compiler generates a VTable at compile time. The VTable itself is stored in static memory (not per object). Even if no object is created, the struct's VTable exists somewhere in memory.

- **Object Instances – Exist in Memory When Created**: When you instantiate an object of the struct, memory is allocated for that object's data members. If the struct has virtual functions, each object has a hidden VPtr (Virtual Pointer), which increases its size.

If A is a struct, why does sizeof(A) have a size even if no object is created? When you define a struct in C++, the compiler determines the size of its layout at compile time. This means that even if you don't create an object of the struct, sizeof(A) can still return a valid size.

The compiler does not assign fixed memory addresses to struct members. Instead, it determines the memory layout (size, alignment, and padding) at compile time. Actual memory addresses are assigned at runtime when an object is created.

Note about heap allocated memory in vectors

If your std::vector contains raw pointers (e.g., std::vector<int*>), the vector will only destroy the pointers themselves but not the dynamically allocated memory they point to. This will cause a memory leak if you don't manually delete each allocated object before the vector goes out of scope.

To avoid leaks, manually delete the elements before clearing the vector

Function that takes a const reference can accept rvalues

In C++, a function that takes a const reference can accept rvalues because const references extend the lifetime of temporary (rvalue) objects.

```cpp
template <typename T>
void f(const T& x) {}
```

A const T& (a reference to a const object of type T) can bind to both lvalues and rvalues, because:

1. Lvalues are naturally bindable to references.
2. Rvalues (temporaries) can bind to const T& because:
   - The const qualifier guarantees that the temporary will not be modified.
   - C++ extends the lifetime of the temporary to match the lifetime of the reference.

```cpp
const int& ref = 100; // OK: binds to temporary, lifetime
↪  extended
std::cout << ref << std::endl; // Prints 100

int& ref = 100; // ERROR: Cannot bind non-const lvalue reference
↪  to an rvalue
```

- If an lvalue is passed → T deduces to int&, making T&& collapse to int&.
- If an rvalue is passed → T deduces to int, making T&& remain int&&.

Notes about c++ casts

First, recall

- static_cast is used for safe and well-defined type conversions that are checked at compile-time.
- dynamic_cast is used only with polymorphic types (i.e., classes with at least one virtual function). It performs runtime type checking and is mainly used for safe downcasting.
- const_cast is used to add or remove const or volatile qualifiers from a variable. It is the only cast that can remove const, allowing modifications to otherwise constant data.
- reinterpret_cast is the most dangerous cast—it converts between completely unrelated types. It does not perform type checking and is used for low-level pointer manipulation.

15.1 Static_cast

When to Use static_cast:

- Converting between numeric types (e.g., int to double).
- Converting between pointers of related classes (e.g., upcasting in inheritance).
- Converting between explicitly defined conversion operators.

What It CANNOT Do:

- It does not check for validity at runtime.
- It cannot cast between unrelated types (use reinterpret_cast for that).
- It cannot remove const or volatile qualifiers (use const_cast for that).

15.2 Dynamic_cast

Dynamic casting in C++ is a feature provided by the language to safely convert pointers or references of base class types to pointers or references of derived class types at runtime. This is particularly useful in scenarios involving polymorphism, where you have a base class pointer or reference pointing to an object of a derived class, and you need to access derived class-specific members or methods.

Dynamic casting is used with pointers or references in class hierarchies that involve polymorphism (i.e., classes with at least one virtual function).

```
dynamic_cast<new_type>(expression)
```

Dynamic casting performs a runtime check to ensure the cast is valid. If the cast is not possible, it returns nullptr for pointers or throws a std::bad_cast exception for references.

Dynamic casting relies on Run-Time Type Information (RTTI), which must be enabled in your compiler.

15.2.1 When to Use Dynamic Casting

- When you need to safely downcast in a polymorphic hierarchy.
- When you are unsure of the actual type of the object at runtime and need to check it.

Dynamic casting incurs a runtime overhead due to the type checking. It only works with polymorphic types (classes with at least one virtual function).

Overuse of dynamic casting can indicate a design flaw; prefer virtual functions and polymorphism where possible.

15.2.2 RTTI

RTTI stands for Run-Time Type Information. It is a feature in C++ that provides mechanisms to determine the type of an object at runtime. RTTI is particularly useful in scenarios involving polymorphism, where you need to identify the actual type of an object pointed to by a base class pointer or reference.

RTTI relies on metadata stored by the compiler for polymorphic types (classes with at least one virtual function). This metadata includes:

- A vtable (virtual table) for each polymorphic class, which contains pointers to its virtual functions.
- A type_info object for each class, which stores information about the class's type.

When you use typeid or dynamic_cast, the compiler generates code to access this metadata at runtime to determine the object's type.

15.3 const_cast

const_cast is only used to remove const or volatile qualifiers from a variable. It cannot be used to add const.

```cpp
#include <iostream>

void modify(int* ptr) {
    *ptr = 42;
}

int main() {
    const int x = 10;

    // Removing const
    int* ptr = const_cast<int*>(&x);

    modify(ptr); // Undefined behavior if `x` was originally a `const` object

    std::cout << "x: " << x << std::endl; // This may not reflect the change due to UB
}
```

This is unsafe if x was originally declared as const int x = 10;, because modifying x leads to undefined behavior. However, if x was originally non-const and then cast to const, modifying it later using const_cast is safe.

If you want to add const, you should use

- Implicit conversion
- static_cast
- Declaring a const reference or pointer to a non-const object

```
int a = 5;
const int* ptr = &a; // Adding const implicitly
const int& ref = a;  // Adding const implicitly
```

const_cast is useful in a few specific cases where you need to work around const qualifiers safely.

15.3.1 Modifying a non-const object that was passed as const

If a function receives a const parameter but you know that the actual object is non-const, you can safely cast away const and modify it.

15.3.2 Removing const to use overloaded functions

Sometimes you have an overloaded function where one version accepts const and another modifies the object. const_cast allows selecting the modifying version when needed.

```
#include <iostream>

class Example {
    public:
    void print() {
        std::cout << "Non-const print" << std::endl;
    }

    void print() const {
        std::cout << "Const print" << std::endl;
    }
};

void forceModify(const Example& obj) {
    const_cast<Example&>(obj).print();  // Calls non-const
    ↪ version
}

int main() {
    Example e;
    forceModify(e);  // Calls non-const print()
}
```

Note: Const cast is a runtime operation and does not perform any checks

15.4 reinterpret_cast

reinterpret_cast is a type of casting operator in C++ that is used to convert one pointer type to another, even if the types are entirely unrelated. It performs a low-level reinterpretation of the underlying binary representation of the data.

Note: reinterpret_cast is a runtime operation and does not perform any checks

15.5 Why are c casts unsafe?

C-style casting is unsafe and ambiguous because:

- It can perform multiple types of conversions at once, including:
  - static_cast
  - reinterpret_cast
  - const_cast
  - Even dynamic_cast (if a class has virtual functions)
- It lacks compile-time safety—you might unintentionally use an invalid cast.
- It is hard to search and debug since (Type) doesn't indicate what kind of conversion is being performed.

The compiler and functions

The compiler handles functions through several stages, from parsing the source code to generating machine code. Here's an overview of how the compiler deals with functions

1. **Parsing and Syntax Analysis:** The compiler reads the source code and identifies function declarations and definitions.

   It checks the syntax of the function, such as the return type, function name, parameter list, and body.

2. **Semantic Analysis:** The compiler checks the meaning of the function, such as

   - Whether the function is declared before use (or has a prototype).
   - Whether the function parameters and return type match the function calls.
   - Whether the function body adheres to type rules (e.g., no invalid operations on types).

3. **Function Overloading Resolution:** If multiple functions with the same name exist (function overloading), the compiler determines which function to call based on the arguments provided.

4. **Code Generation:** The compiler generates intermediate or machine code for the function body.

   It allocates memory for local variables and parameters.

   It generates instructions for the function's logic, such as arithmetic operations, loops, and conditionals.

5. **Function Calls:** When a function is called, the compiler generates code to:

   - Push the arguments onto the stack (or pass them via registers, depending on the calling convention).
   - Transfer control to the function's code.
   - Save the return address so the program knows where to continue after the function finishes.

6. **Inlining (Optional):** If a function is marked with the inline keyword or the compiler determines it is beneficial, the compiler may replace the function call with the actual function body to avoid the overhead of a function call.

7. **Linkage:** If a function is declared in one translation unit (source file) and defined in another, the compiler ensures the function is properly linked.

   The compiler generates symbols for functions, which the linker resolves during the linking phase.

8. **Optimization:** The compiler may optimize functions to improve performance, such as:

   - Removing unused code (dead code elimination).
   - Unrolling loops.
   - Inlining small functions.
   - Optimizing tail-recursive functions.

When a function is called, the compiler (and the runtime environment) manages memory for local variables in a specific way

Local variables in a function are typically stored in the stack, a region of memory that is managed automatically by the compiler and runtime environment.

When a function is called, the compiler allocates memory on the stack for all of its local variables.

This memory is only valid for the duration of the function call. Once the function returns, the memory is deallocated (freed).

Each function call creates a stack frame (also called an activation record), which contains:

- The function's local variables.
- The return address (where the program should continue after the function finishes).
- The function's parameters (if any).

The stack grows downward in memory, and each new function call adds a new stack frame on top of the previous one.

When a function returns, its stack frame is deallocated, and the memory becomes available for reuse.

If the same function is called again, a new stack frame is created, and the local variables are allocated in the same memory region (which may have been overwritten by other function calls in the meantime).

Local variables have automatic storage duration, meaning they are created when the function is called and destroyed when the function returns.

This means that the values of local variables are not preserved between function calls.

constexpr variables are treated differently because they are compile-time constants.

The compiler evaluates constexpr variables at compile time and replaces their uses with their computed values.

No memory is allocated for constexpr variables at runtime—they are essentially ”baked into” the code.

If a local variable is declared static, it has static storage duration, meaning it is allocated memory once and persists across function calls.

Static local variables are not stored on the stack but in a separate region of memory (typically the data segment).

16.1 Important: compile time constant variables and memory (constant propagation/constant folding)

When the compiler encounters a compile-time constant (e.g., a constexpr variable), it evaluates the constant expression at compile time and replaces all uses of the variable with the computed value. This means:

- **No Memory Allocation for the Variable:** Since the value of the compile-time constant is known at compile time, the compiler does not allocate memory for the variable at runtime.

  Instead, the variable is treated like a literal value (e.g., 10, 3.14, etc.), and its value is "baked into" the generated code wherever it is used.

- **Replacement of Uses:** The compiler replaces every occurrence of the compile-time constant variable with its computed value. This process is called constant propagation or constant folding.

The introduction of the nullptr

nullptr was introduced in C++11 to address several issues and improve type safety when dealing with null pointers. Prior to C++11, the standard way to represent a null pointer was to use the macro NULL, which is typically defined as 0 or ((void*)0) in C++. However, this approach had several drawbacks:

17.1 Ambiguity in Overloaded Functions

In C++, NULL is typically defined as 0, which is an integer. This can lead to ambiguity when overloading functions that take both integer and pointer arguments. For example

```
void foo(int);
void foo(char*);

foo(NULL); // Which foo is called? Ambiguous, as NULL is an integer.
```

nullptr resolves this ambiguity because it has its own type, std::nullptr_t, which is implicitly convertible to any pointer type but not to integral types. Thus:

```
foo(nullptr); // Unambiguously calls foo(char*)
```

17.2 Type safety

Using 0 or NULL for null pointers can lead to type safety issues because 0 is an integer literal, and it can be implicitly converted to other types, such as bool or int. This can result in unintended behavior or bugs.

nullptr is a keyword that explicitly represents a null pointer, and it cannot be implicitly converted to an integer. This makes the code more type-safe and less prone to errors.

17.3 The type of nullptr

nullptr is of type nullptr_t

```
auto ptr = nullptr // has type nullptr\_t
```

17.4 nullptr vs void*

nullptr is used to explicitly indicate that a pointer does not point to any valid memory location.

It has its own type, std::nullptr_t, which is implicitly convertible to any pointer type (e.g., int*, char*, etc.) but not to other types like integers.

void* is a generic pointer type that can point to any data type.

Brace initialization vs Parenthesized Initialization

Brace initialization (also called uniform initialization or list initialization) was introduced in C++11 and has stricter rules compared to parenthesized initialization. Specifically, consider a uniform initialization of the form `type{expr}`

- **Prohibits narrowing conversions:** Brace initialization does not allow implicit narrowing conversions. If expr cannot be converted to Type without losing information (e.g., converting a double to an int), the compiler will emit an error.
- **Prevents most vexing parse:** Brace initialization avoids ambiguity with function declarations, which can occur with parenthesized initialization.
- **Calls constructors explicitly:** If Type has a constructor that takes an std::initializer_list, brace initialization will prefer that constructor.

If an std::initializer_list constructor exists, it will be chosen over other constructors, even if another constructor is a better match.

```
struct s {
    s(int x) {cout << "1" << endl;}
    s(initializer_list<int> x) {cout << "2" << endl;}
};
s s1{2}; // 2
s s2(2); // 1
```

Parenthesized initialization (also called direct initialization) is more permissive and allows narrowing conversions. It behaves like a function call, where the compiler attempts to convert expr to Type using implicit conversions, even if narrowing occurs.

Note that parenthesized initialization does call constructors

Since c++20, direct initialization will work for aggregate types

$$structlatticeintx, y;; latticep(10, 20); //OKsincec + +20$$

Are chars unsigned?

In C++, the char type is neither inherently signed nor unsigned. It's implementation-defined behavior.

- char, signed char, and unsigned char are distinct types.
- The default behavior of char depends on the compiler. Some compilers treat char as signed char, while others treat it as unsigned char.
- To ensure consistent behavior, use signed char or unsigned char explicitly.

Size of types

- **bool**: 1 Byte
- **char**: 1 Byte
- **short**: 2 Bytes
- **int**: 4 Bytes
- **long**: 4 or 8 bytes
- **long long**: 8 bytes
- **float**: 4 bytes
- **double**: 8 bytes
- **long double**: 16 bytes
- **string**: 32 bytes
- **Pointers**: 8 bytes

Note that the size of primitive types are usually implementation defined, and adding signed or unsigned does not change the size.

Also, references are the size of the type they refer to.

Trivially copyable

A trivially copyable type in C++ is a type that can be copied efficiently using memcpy or similar low-level operations without breaking its correctness. This means it does not require custom copy/move constructors or destructors.

A class or struct is trivially copyable if:

- Can be copied with memcpy without breaking program semantics.
- Has no user-defined copy/move constructors, assignment operators, or destructors.
- Only contains trivially copyable members.
- Does not have virtual functions or virtual base classes.

POD types (Plain old data)

POD (Plain Old Data) refers to a type in C++ that is simple, compatible with C structures, and has well-defined memory layouts. A POD type behaves like a C-style struct and lacks modern C++ features such as constructors, destructors, virtual functions, and non-trivial member functions.

A type is considered POD if:

- It is a trivial type (trivial constructor, destructor, copy/move operations).
- It is a standard-layout type (data layout matches C structs).

22.1 Trivial type

A type is trivial if:

- It has a trivial default constructor (compiler-generated, does nothing).
- It has a trivial copy/move constructor and assignment (member-wise copying).
- It has a trivial destructor (compiler-generated, does nothing).

22.2 Standard layout

A type is standard-layout if:

- It has no virtual functions or virtual base classes.
- It has only standard-layout base classes.
- All non-static data members have the same access control (public vs private matters).
- It does not inherit from multiple base classes with different access specifiers.

For a C++ structure (struct or class) to match a C struct's layout, it must

- Store its members in the same order as declared.
- Not have hidden padding or unexpected compiler transformations.
- Not have virtual functions or virtual base classes.
- Not use complex features like multiple inheritance.
- Use only standard-layout types for its members.

If a type is both trivial and standard-layout, it is POD.

Notes about initialization

23.1 How many names does brace initialization have?

23.1.1 Uniform Initialization (General Term)

Introduced in C++11, {} initialization is often referred to as uniform initialization because it provides a consistent syntax for initializing objects of any type.

Note that uniform initialization is an informal term

23.1.2 List Initialization

List initialization is the official term in the C++ standard for using {} to initialize objects.

23.1.3 Aggregate Initialization

When {} is used to initialize aggregates (structs, arrays, or classes with public members and no user-defined constructors), it's called aggregate initialization.

```
struct Point {
    int x, y;
};
Point p = {1, 2}; // Aggregate initialization
```

23.1.4 Value Initialization

If {} is used without any elements, it results in value initialization.

```
struct  point {
    int x{},y{}; // Default value initialized
};
```

Note that we can also do

```
struct point {
    int x,y
};
point p{}; // Construct p and default value initialize x,y
```

23.1.5 Direct List Initialization

When {} is used with a constructor that takes a list, it is called direct list initialization.

```
std::vector<int> v{1, 2, 3}; // Direct list initialization
```

23.1.6 Copy List Initialization

When {} is used on the right-hand side of an assignment or in variable initialization without explicit construction, it's called copy list initialization.

```
std::vector<int> v = {1, 2, 3}; // Copy list initialization
```

23.1.7 Narrowing Prevention Initialization

{} initialization prevents narrowing conversions, meaning it does not allow implicit type conversions that lose information.

```
int x = 3.5;  // Allowed (implicit conversion)
int y{3.5};   // Error (narrowing conversion)
```

23.1.8 Default Member Initialization

{} can be used to initialize class members with default values.

```
struct Example {
    int a = {};  // Value-initialized to 0
};
```

23.1.9 Copy list initialization and direct list initialization in aggregate types

We note that for an aggregate type, copy list initialization and direct list initialization behave the same

```
struct point {
    int x,y
};
point p1 = {1,2};
point p1{1,2};
```

For aggregates, both = (copy list initialization) and {} (direct list initialization) behave identically because C++ applies aggregate initialization in both cases.

However, if point had a user-defined constructor, then they could behave differently.

If we defined a constructor that takes an initializer_list, both the above objects would call that constructor

```
struct  point {
    int x{},y{}; // Default value initialized

    point(std::initializer_list<int> l) {
        cout << "called init list " << endl;
    }
};

point p1 = {1,2}; // called init list
point p2{1,2}; // called init list
```

Note that if we define a virtual method, our type becomes polymorphic and is no longer aggregate. Therefore, we cannot use either initialization above. Note however that c++ will still implicitly give us the default constructors, so the following will still work

```
struct point {
    int x,y;

    virtual void f();
};
point p1{1,2} // Error
point p2 = {1,2} // Error
point p3{};
```

In this case, we need to explicitly define our constructor

```
struct point {
    int x,y;

    point(int x, int y) : x(x), y(y) {}

    virtual void f();
};
point p1{1,2}
point p2 = {1,2}
point p3{};
```

23.2 Other forms of initialization and their names

23.2.1 Direct Initialization

Uses the parenthesis () syntax

```
int x(10);   // Direct initialization
std::string s("hello");
```

Note that aggregate initialization logic will not work with this syntax

```
struct point{
    int x,y
};
point p(1,2) // Error! Must define the constructor.
```

23.2.2 Copy Initialization

Uses assignment =

```
int x = 10;
```

Creates a temporary object and copies/moves it to initialize the variable. May involve implicit conversions. Calls copy constructor if the type has one.

23.2.3 Default Initialization

No explicit initializer

```
int x;      // Default initialization (uninitialized in local scope)
std::string s; // Calls default constructor (empty string)
```

23.3 Excess elements in scalar initialization

Consider the code

```
int x{1,2,3}
```

Results in a compilation error ”excess elements in scalar initialization”

Further consider

```
template<typename ... Args>
struct foo {
    int var;

    foo(Args... args) : var(args...) {}
};
X x(1); // fine
X y(1,2,3); // Excess elements in scalar initialization
```

Notes about constructors and destructors

24.1 Value initialized vs default initialized

Consider the code

```
struct s1 {
    string s{};
    int x{};
};

struct s2 {
    string s;
    int x;
};
```

Whats the difference? In s1, the members are explicitly initialized with {}

```
struct s1 {
    std::string s{};  // Initializes to an empty string
    int x{};          // Initializes to 0
};
```

This ensures that when an instance of s1 is created, s will be initialized to an empty string (""), and x will be initialized to 0.

In s2, the members are not explicitly initialized:

```
struct s2 {
    std::string s;  // Default constructor of std::string
    ↪ initializes it to ""
    int x;          // Uninitialized, contains garbage value if
    ↪ not explicitly set
};
```

s is fine because std::string has a default constructor that initializes it to "".

x, however, remains uninitialized when a default-constructed object of s2 is created, leading to an indeterminate value.

24.2 What exactly is =default

the = default specifier is used to explicitly declare that a special member function should be automatically generated by the compiler with its default behavior.

```cpp
struct s {
    char c;
    int x;

    s() = default;
};
```

This declares a default constructor (s()) explicitly, but it tells the compiler to generate it using the default implementation.

By default, the compiler-generated constructor does not initialize member variables. So, this:

```cpp
s() = default;
```

is equivalent to:

```cpp
s() {}  // Default constructor, does nothing
```

which means the members (char c; int x;) remain uninitialized when an object of s is created

24.2.1 Non-POD fields

=default construct (default constructor) calls default constructors of non-POD (plain-old-data) members.

```cpp
struct S {
    std::string str; // std::string has a default constructor
    int x;

    S() = default; // Compiler generates: S() {} (but calls std::string's constructor)
};
```

The std::string member is properly initialized (since std::string has a default constructor). The int x is uninitialized.

24.2.2 With copy constructors

```cpp
struct S {
    std::string str;
    int x;

    S() = default;
    S(const S&) = default;  // Compiler generates: S(const S& other) : str(other.str), x(other.x) {}
};
```

Generates a copy constructor that copies each member individually using their copy constructors.

24.2.3 Copy Assignment

Generates an assignment operator that copies each member individually.

```cpp
struct S {
    std::string str;
    int x;

    S& operator=(const S&) = default;
};
```

Equivalent to

```cpp
S& operator=(const S& other) {
    str = other.str; // Calls std::string's assignment operator
    x = other.x;     // Simply copies x
    return *this;
}
```

24.2.4 Move constructor

Generates a move constructor that moves each member individually.

```cpp
struct S {
    std::string str;
    int x;

    S(S&&) = default;  // Compiler generates: S(S&& other) : str(std::move(other.str)), x(other.x) {}
};
```

This allows efficient moving:

```cpp
S obj1;
S obj2 = std::move(obj1);  // Moves `str`, but leaves `x` as a copy
```

the move constructor does not ”move” the int x. Instead, it simply copies x from the source. This is because int is a trivially copyable type, and moving it is no different from copying.

24.2.5 Move Assignment

Generates a move assignment operator that moves each member individually.

```cpp
struct S {
    std::string str;
    int x;

    S& operator=(S&&) = default;
};
```

Equivalent to:

```cpp
S& operator=(S&& other) {
    str = std::move(other.str);  // Moves the string instead of copying
    x = other.x;  // Simply copies x (since int doesn't benefit from move)
    return *this;
}
```

24.2.6 Destructor

Generates a destructor that:

- Does nothing for fundamental types.
- Calls the destructors of member objects.

```cpp
struct S {
    std::string str;
    int x;

    ~S() = default;  // Compiler generates: ~S() {} (but calls std::string's destructor)
};
```

24.2.7 Do you need to write them?

In C++, the compiler will automatically generate default implementations for special member functions only if they are needed and not explicitly declared.

However, if you declare any of them (without defining them), the compiler will not automatically generate them. This is where = default comes in—it explicitly tells the compiler to generate the default implementation.

24.3 =delete

=delete does the opposite of =default... It deletes the default implementation

24.4 Rule of five

In C++, the Rule of Five states that if you define or explicitly delete any of the following five special member functions, you should likely define or delete all five

- Destructor
- copy constructor
- copy assignment operator
- move constructor
- move assignment operator

```
struct s {
    string str{};

    s(const s& other) {
        cout << "Copy constructor called" << endl;
        str = other.str;
    }
};
s s1;
```

In this case, we get an error. When you explicitly define a copy constructor in C++, the compiler does not automatically generate the default constructor for you. This is due to the Rule of Five

The C++ standard states that if you declare any of the above special member functions, the compiler will not generate a default constructor for you:

Once you define any of these functions, the compiler assumes you want full control over object creation and copying, so it does not provide the default constructor.

24.5 A confused compiler

Consider the code

```cpp
struct  point {
    int x{};

    point (int x) : x(x) {}

    point(int&& other) {
        cout << "called second" << endl;
        x = other;
    }
};
point p = 20; // Error
```

We get ”error: conversion from ‘int’ to ‘point’ is ambiguous”.

happens because there are multiple constructors that can accept an int, and the compiler doesn’t know which one to pick.

- point(int) is a direct match for int.
- point(int&&) can accept an int as an rvalue reference.

The compiler does not know which one to pick, so it gives an error.

If we mark the first constructor explicit, we then prevent implicit conversion from int to point, and the second constructor will be the one called.

```cpp
struct  point {
    int x{};

    explicit point (int x) : x(x) {}

    point(int&& other) {
        cout << "called second" << endl;
        x = other;
    }
};
point p = 20;
```

24.6 Which constructor will be called

Suppose we have

```cpp
struct  point {
    int x{};

    point(const int& other) {
        cout << "called first" << endl;
        x = other;
    }

    point(int&& other) {
        cout << "called second" << endl;
        x = other;
    }
};
point p = 20;
```

Which constructor gets called? The second constructor (point(int&& other)) is called in this case because the literal 20 is an rvalue.

We have two suitable constructors, and the compiler must determine which constructor to call.

The compiler deems the second as a better match than the first, because 20 is a pure rvalue. Note that if we removed the second constructor, the first would work. A constructor that takes a const reference can bind to both rvalues and lvalues.

24.7 Constructors are implicitly inline constexpr

constructors are implicitly constexpr under certain conditions. This means they can be evaluated at compile time if all their operations are constexpr. The main reason for this behavior is to make objects usable in constant expressions without requiring explicit constexpr annotations.

24.8 Destructors are implicitly noexcept

In C++, destructors are implicitly declared as noexcept unless a potentially throwing operation is explicitly present in the destructor's definition. This behavior exists to improve performance, exception safety, and compatibility with standard library features.

When a function is marked as noexcept, the compiler can generate more efficient code:

If destructors could throw by default, objects in RAII (Resource Acquisition Is Initialization) and smart pointers (std::unique$_ptr, std :: shared_ptr)wouldleadtoundefinedbehaviororterminatetheprogramwhenade$

```
struct B {
    ~B() { throw std::runtime_error("Error!"); } // Dangerous!
};

void func() {
    try {
        B b;
        throw std::runtime_error("Oops");
    } catch (...) {
        // Stack unwinding will terminate the program if ~B()
    ↪ throws
    }
}
```

To prevent such issues, destructors are noexcept by default, meaning they do not propagate exceptions. If an exception occurs in a noexcept destructor, std::terminate() is called.

24.9 When is an object "Fully constructed"

In C++, an object is considered fully constructed when its constructor has finished executing successfully. Specifically:

- **For a non-inherited class (without base classes)**: The object is fully constructed after its constructor runs to completion.
- **For a class with base classes**: The object is fully constructed after all base class subobjects and non-static data members have been successfully initialized.
- **For a class with member variables**: Each member is constructed in the order they are declared in the class definition, before the body of the constructor executes. If a member's constructor throws an exception, the object is not fully constructed.
- **For a derived class**: A derived class object is fully constructed only after
  - The base class constructor(s) finish execution.
  - All member variables of the derived class are constructed.
  - The body of the derived class constructor executes successfully.

24.10 When is an object not fully constructed

If a base class constructor throws, the derived class never completes construction.

If a member variable's constructor throws, the object never becomes fully constructed.

If a constructor exits via an exception, the destructor never runs, because the object was never fully formed.

Consider the code

```
struct B {
    B() { throw std::runtime_error("Error"); }
    ~B() { cout << "Destructor called" << endl; }
};

try {
    B b;
} catch (...) {

}
```

Here we get no output, the constructor never finished, which means the object was not fully constructed, which means the destructor will not get called.

Further consider

```
struct B {
    B() {}
    B(int n) : B() { throw std::runtime_error("Error"); }
    ~B() { cout << "Destructor called" << endl; }
};

try {
    B b;
} catch (...) {

}
```

In this example, we do get ”destructor called”. The destructor is called because the default constructor B() successfully constructs an object, even though the other constructor (B(int)) throws an exception.

Dividing by zero

Consider the code

```
int a = 5, b = 0;
int c = a/b;
```

Here we get *terminated by signal SIGFPE (Floating point exception)*

Notes about inheritance

26.1 Constructor chain

In C++ inheritance, the constructor chain (or constructor delegation) refers to the sequence in which constructors of base and derived classes are called when an object of the derived class is created.

When an object of a derived class is created, the constructor of the base class runs before the constructor of the derived class.

If the base class has multiple levels, constructors are called in top-to-bottom order (from base to most derived).

If the base class does not have a user-defined constructor, the compiler provides a default constructor that is automatically invoked.

If the base class has a parameterized constructor, you must explicitly call it in the initializer list of the derived class.

26.2 Destructor chain

When a class has a virtual destructor, the vtable includes a pointer to the most derived destructor. However, destructors are special because they are split into two parts in the vtable:

- **Complete Object Destructor:** Calls all destructors in the chain (Child → Dad → Grandfather).
- **Base Object Destructor:** Only relevant when the object is part of another object (not directly deleted).

When Child overrides Dad (which overrides Grandfather), the vtable for Child includes a pointer to the most derived destructor (∼Child), but that destructor is responsible for calling the entire chain.

If Grandfather's destructor is not virtual, the vtable will not be consulted during delete obj. Instead, the compiler resolves delete obj at compile-time and calls Grandfather::∼Grandfather() directly, ignoring the destructors of Dad and Child

Destructors execute in reverse order of constructors (i.e., derived class destructor runs first, then base class destructor).

26.3 So who needs a virtual destructor?

Consider the situation

```cpp
struct grandfather {
    virtual ~grandfather() = default;
};


struct dad : grandfather {
};

struct child : dad {
};
```

When a destructor is declared virtual in a base class (grandfather), all derived classes (dad and child) automatically have virtual destructors, even if they don't explicitly declare them

Since dad and child inherit from grandfather, they do not need to explicitly mark their destructors as virtual—they already are.

```cpp
struct grandfather {
    virtual ~grandfather() { std::cout << "grandfather destroyed\n"; }
};

struct mom : public virtual grandfather {
    ~mom() { std::cout << "mom destroyed\n"; } // Implicitly virtual
};

struct dad : public virtual grandfather {
    ~dad() { std::cout << "dad destroyed\n"; } // Implicitly virtual
};

struct child : public mom, public dad {
    ~child() { std::cout << "child destroyed\n"; } // Implicitly virtual
};
grandfather* g = new child();
delete g;
/*
child destroyed
dad destroyed
mom destroyed
grandfather destroyed
*/
```

26.4 Default access modifier

Consider the code

```
class A {};
class B : A {};
```

The default access modifier is private for class inheritance, and public for structs.

Booleans under the hood: Adding two booleans

In C++, bool is typically represented under the hood as an integral type, often stored as a single byte (char-sized) in memory. However, when used in expressions, bool values implicitly promote to int.

In C++, bool is effectively an integer type but only holds 0 (false) or 1 (true).

When used in an arithmetic expression, a bool value is promoted to an int.

This is part of the integral promotion rules in C++.

```cpp
bool b1 = 1, b2 = 1;
cout << typeid(b1 + b2).name();
```

Type Promotion

Type promotion in C++ refers to the implicit conversion of smaller or lower-ranked types to larger or higher-ranked types in expressions. This ensures consistent operations without data loss.

- **Boolean Promotion:** bool converts to int (false → 0, true → 1) when used in arithmetic or bitwise operations.
- **Integral Promotion:** Types smaller than int (char, short, bool) promote to int if int can represent all values; otherwise, they promote to unsigned int.

If operands have different types, the one with the lower rank is converted to the higher-ranked type.

28.1 Ranking

Standard conversion sequences are categorized in one of three ranks. The ranks are listed in order from best to worst:

- **Exact match:** This rank includes the following conversions:
  - Identity conversions
  - Lvalue-to-rvalue conversions
  - Array-to-pointer conversions
  - Qualification conversions
- **Promotion:** This rank includes integral and floating point promotions.
- **Conversion:** This rank includes the following conversions:
  - Integral and floating-point conversions
  - Floating-integral conversions
  - Pointer conversions
  - Pointer-to-member conversions
  - Boolean conversions

Consider

```
void f(double x) { }
void f(unsigned x) { }
f(1.0); // Error, call to f is ambiguous
```

This situation is ambiguous because the integer literal 1.0 can be converted to either int or unsigned int, and both conversions have the same rank in the standard conversion sequence (Floating-integral conversions)

Further, consider

```cpp
void f(int x) { cout << "1"; }
void f(unsigned x) { cout << "2"; }
f((short)5); // 1
```

Although both conversions have rank *promotion*, the compiler chooses the first

The compiler looks for the best function match among the available overloads

- short → int is an integral promotion.
- short → unsigned int is also a promotion, but only considered if int is not available.

Since integral promotions have higher precedence than standard conversions, f(int) is chosen because it's a direct promotion, whereas f(unsigned) is a conversion if int is available.

Consider

```cpp
void f(char x) { cout << "1"; }
void f(short x) { cout << "2"; }
f((bool)5); // Error: call to f is ambiguous
```

The function call f((bool)5); is ambiguous because both void f(char) and void f(short) are equally valid candidates based on integral promotions, and neither is strictly better than the other

This situation falls under conversion rank because bool is promoted to int first, and then int must be converted to either char or short, both of which have equal ranking in standard conversions.

promotions are a subset of standard conversions that always take precedence over regular conversions in overload resolution. A promotion only occurs when converting a smaller type to a larger type of the same category (integral or floating-point), without changing its fundamental nature.

Integral promotions occur when a smaller integer type is promoted to at least int or unsigned int. These promotions are preferred over standard conversions and are commonly used in arithmetic expressions, function calls, and overload resolution.

Notes about static

29.1 Static variables inside member functions

Static variables inside member functions behave the same as static variables in regular functions

29.2 Using instances to call static methods

We note that we can use instances to call static methods using the dot notation

```
struct s{
    int x = 20;
    static void f() {
        cout << "static method" << endl;
    }
};
s::f();

s S;
S.f();
```

29.3 Can you make pure virtual methods static?

No, you cannot

29.4 Inline Static methods?

Consider the code

```
struct s{
    static void print() {} // Implicitly inline... Means
    static inline void print() {} // Exact same
};
```

However, defining static methods outside the class does not make it implicitly inline

```
struct s  {
    static void print();
};
void s::print() {} // Not inline
```

The inline keyword suggests that the function can be defined inside the header file (or within the struct/class definition) without violating the one-definition rule (ODR).

It is a hint to the compiler that function calls may be replaced with the actual function body to reduce function call overhead (though modern compilers decide this automatically).

More importantly, in this case, inline ensures that if the function is defined in a header file and included in multiple translation units, it does not cause multiple definition errors.

We can of course still explicitly make the second version inline..

```
struct s  {
    static void print();
};
inline void s::print() {} // Not inline
```

29.5 Static and extern (linkage)

Global variables without the use of the keyword static have external linkage. In other translation units that are compiled with the file that has the global variable, we use the keyword `extern` to access it

```
// File1.cpp
int x = 20;

// File2.cpp
extern int x; // Refers to x from file1.cpp
```

Declaring a global variable as static restricts its visibility to the current translation unit

If we declare a global variable static in on unit, we can get a separate copy in a different unit by declaring the same variable (with the same signature) and also using the keyword static

```
// File1.cpp
static int x = 20;

// File2.cpp
static int x = 20; // File2 gets its own copy of x
```

29.5.1 Using extern to declare a variable with the same name as a static variable from another translation unit

If you use extern to declare a variable with the same name as a static variable from another translation unit, the linker will not find the definition of that variable, leading to a linker error (undefined reference). This is because static variables have internal linkage, meaning they are restricted to their translation unit and cannot be accessed from another file.

```
// File1.cpp
static int x = 20;

// File2.cpp
extern int x; // Linker error
```

29.5.2 Forgetting to use extern

If you forget to put extern when declaring a global variable in another translation unit, the compiler will treat it as a new, separate definition rather than referring to an existing one. This can lead to multiple definition errors at the linking stage

```
//  File1.cpp
int x = 20;

// File2.cpp
int x; // Liner error: Multiple definition
```

29.5.3 Forgetting to use keyword static

```
//  File1.cpp
static int x = 20;

// File2.cpp
int x = 20;
```

No errors for two units, file2 will simply make its own version of x with external linkage. If you add a third unit and forget to put static on that x as well, we would then get multiple definition errors.

29.5.4 Constinit with extern

constinit is only required in the definition (where the variable is actually allocated memory).

When using extern to refer to that variable in another translation unit, constinit is not necessary because extern simply tells the compiler that the variable is defined elsewhere.

```
// File1.cpp
constinit int x = 50;

// File2.cpp
extern int x;
```

Trying to move const objects

Consider the code

```
struct s {
    string str{};

    s() = default;

    s(const string& other) {
        cout << "Copy constructor called" << endl;
        str = other;
    }

    s(string&& other) {
        cout << "Move constructor called " << endl;

    }
};
const string str = "Hello";
s s1(std::move(str));
```

Something interesting happens, we get *"Copy constructor called"*. A constant object cannot be moved because moving modifies the source object.

The move constructor requires an argument of type std::string&& (a temporary, non-const string).

However, std::move(str) produces a const std::string&& (rvalue reference to const), which cannot be passed to the move constructor because the move constructor expects a non-const rvalue reference (string&&).

```
s(const string& other) {   // (1) Copy constructor
    cout << "Copy constructor called" << endl;
    str = other;
}

s(string&& other) {        // (2) Move constructor
    cout << "Move constructor called" << endl;
}
```

Since std::move(str) produces const std::string&&, it cannot bind to (2) s(string&&), because the move constructor does not accept const.

Instead, s(const string&) (the copy constructor) is selected, because it can accept a const std::string&.

This is one reason why we have const in the copy constructor, if the copy constructors argument was not const, we would get an error trying to move the const object.

Smart pointers in conjunction with raw ptrs

31.1 Shared_ptr with raw ptrs

Consider the code

```
int x = 50;
std::shared_ptr<int> ptr = std::make_shared<int>(50);
```

What happens if we do

```
int* ptr2 = &(*ptr);
```

Will it increase the reference count? No it does not

```
cout << ptr.use_count() << endl; // 1
```

31.2 Unique_ptr with raw ptrs

What if we do

```
std::unique_ptr<int> ptr = std::make_unique<int>(50);
int* ptr2 = &(*ptr);
```

Does this even work? Yes, this works perfectly fine. *ptr dereferences the std::unique_ptr<int>, retrieving the integer it owns. &(*ptr) takes the address of that integer, effectively obtaining a raw pointer to the managed object.

If ptr2 is used only within the scope where ptr exists and is valid, it's generally fine

If ptr is reset or goes out of scope, ptr2 becomes a dangling pointer, leading to undefined behavior

Calling delete on a nullptr

Calling delete on a nullptr is perfectly safe and has no effect.

The delete operator first checks whether the pointer is nullptr. If the pointer is nullptr, delete does nothing and returns immediately.

This behavior ensures that you don't have to explicitly check for nullptr before calling delete.

Notes about short circuits

Short-circuiting in C++ refers to the behavior of logical operators (and ||) where the second operand is not evaluated if the result of the entire expression can be determined from the first operand alone.

33.1 With ands

Consider the following code

```
bool foo() {
    cout << "Foo" << endl;
    return false;
}

bool bar() {
    cout << "bar" << endl;
    return true;
}

if (foo() && bar()) // "Foo"
```

The bar function never gets called because when the foo function returns, the if statement ”short-circuits” and exits. If we want to prevent short circuiting, we use a bitwise and (&)

```
if (foo() & bar()) // Foo\n bar
```

33.2 With ors

Similarly,

```
bool foo() {
    cout << "Foo" << endl;
    return true;
}

bool bar() {
    cout << "bar" << endl;
    return true;
}

if (foo()  bar()) // Foo
```

We again just get ”Foo”. When the foo function returns true, the if statement is automatically true and again we get a short circuit, which leads to bar never getting called. If we want to prevent short circuiting with ors, we use bitwise or (|)

```
if (foo() | bar()) // Foo\n bar
```

Notes about types

- bool
- char (unsigned and signed)
- short (unsigned and signed)
- int (unsigned and signed)
- long (unsigned and signed)
- long long (unsigned and signed)

Notes about inline functions

Recall that an inline function in C++ is a function that is expanded in place where it is called, rather than executing a traditional function call. This is a compiler directive that can reduce function call overhead and potentially improve performance.

When a function is declared as inline, the compiler replaces the function call with the actual function definition (code) at compile time. This eliminates the overhead of a function call, such as:

- Pushing arguments onto the stack.
- Jumping to the function's memory location.
- Returning to the caller.

Advantages:

- **Eliminates Function Call Overhead**: Since the function code is directly inserted at the call site, there is no function call overhead.
- **Faster Execution**: If the function is small, inlining can make the program faster.
- **Useful for Small, Repeated Functions**: It is particularly useful for small functions that are frequently called (e.g., getter functions in classes).

Disadvantages:

- **Increased Binary Size (Code Bloat)**: If a function is large and used in multiple places, its repeated expansion increases the binary size.
- **Decreased Cache Efficiency**: More inline code means a larger executable, which may lead to poor instruction cache performance.

Note that inlining is a suggestion, not a command, The compiler may ignore the inline keyword if the function is too complex.

35.1 Inline functions in the context of multiple translation units

In C++, a program can consist of multiple translation units, where each source file (.cpp) and its included headers form a separate unit compiled independently before linking. Using inline functions across multiple translation units can help avoid multiple definition errors.

When defining a function in a header file (.h), multiple source files (.cpp) including the header may result in multiple definitions of the function

When compiled separately and linked together, the linker will complain about multiple definitions of square(int) because each translation unit includes its own separate copy.

The inline keyword tells the compiler that multiple identical definitions of the function are allowed across translation units, and they should be merged.

Normally, functions with external linkage (int square(int)) violate the One Definition Rule (ODR) when included in multiple translation units.

Marking the function inline allows the linker to consolidate multiple instances into one.

35.2 Class/Struct methods are implicitly inline

Since it is common to define classes or structs in header files, the c++ compiler makes all methods inline to avoid violation of the one definition rule (ODR)

Notes about object creation

Consider the code

```
struct A{
    A() { cout << 'a'}
    ~A() { cout << 'A'}
};

struct B {
    B() {cout << 'b'}
    ~B() {cout << 'B'}
    A a;
};

int main() {B b;} // abBA
```

Member variables are initialized before the constructor is called. The destructor is called before member variables are destroyed.

What is a ”function signature”

a function signature refers to the combination of components that uniquely identify a function within a given scope. However, different contexts use ”function signature” to mean slightly different things.

When the compiler determines whether two functions are overloaded (i.e., different functions in the same scope), it considers:

- Function name
- Parameter types (including order and const qualifiers on parameters)

37.1 Function Signature (Overriding Perspective)

When dealing with function overriding in a derived class, the function signature must exactly match the base class function’s:

- return types
- Function name
- Parameter types (including order and const/reference qualifiers)
- const qualifier (for member functions)
- Volatile qualifier (volatile if present)
- Ref-qualifier (& or &&)

While return types are not part of the function signature in the overloading perspective, they are in the overriding perspective

```
struct s {
    virtual void print() const {}
};
struct k : s {
    int print() const override {} // Error: Return type must match the overridden function
}
```

37.2 What is NOT Part of a Function Signature?

The following do not contribute to a function’s signature:

- **Return type**: Functions cannot be overloaded solely by differing return types.
- **Default arguments**: Default arguments are resolved at compile time, so they are not part of the function signature.

Note: Even if parameter types are aliased, they resolve to the same type and do not create a different signature.

37.3 noexcept?

The noexcept specifier is part of the function signature for function overloading but not for overriding.

Maximal munch principle

Consider the code

```cpp
int a=5, b=2;
cout << a+++++b; // ERROR
```

Why is this an error? Why is it not simply `a++ + ++b` = 8? The Maximal Munch Principle is a rule used in lexical analysis (tokenization) to determine how to group characters into tokens. It states: *Always consume the longest possible sequence of characters that form a valid token.*

When scanning input text, the lexer tries to form the longest valid token at each step before moving on. This avoids ambiguity in tokenization. So after parsing a++, it is not allowed to just parse +, it has to parse ++. The sequence is thus parsed as:

```cpp
a ++ ++ + b
```

which is ill-formed since post-increment requires a modifiable lvalue but the first post-increment will produce a prvalue

What happens if a noexcept function throws an exception?

If a function is declared noexcept but still throws an exception, the program immediately terminates by calling std::terminate(). This behavior exists to enforce the contract that noexcept functions are not supposed to throw.

noexcept functions promise not to throw. The compiler and standard library rely on this assumption. If a noexcept function throws, continuing execution would violate the contract and lead to undefined behavior. Instead of allowing UB, C++ forces a hard fail by calling std::terminate().

If a noexcept function must perform an operation that might throw, you need to catch exceptions inside the function and handle them before returning.

Exception propogation

When an exception propagates, it means that the exception is not handled in the current function and is instead passed up the call stack until it reaches a function that can handle it (i.e., a function with a try-catch block). If no function catches the exception, the program terminates.

When an exception is thrown using throw, control is immediately transferred to the nearest matching catch block in the call stack.

```cpp
void foo() {
    throw std::runtime_error("Error in foo");
}

void bar() {
    foo();  // No try-catch here, so the exception propagates
}

int main() {
    try {
        bar();  // Calls bar(), which calls foo()
    } catch (const std::runtime_error& e) {
        std::cout << "Caught exception: " << e.what() << '\n';
    }
}
```

The catch all exception handler (...)

In C++, catch (...) is a catch-all exception handler, meaning it catches any exception, regardless of its type. It is useful when you want to handle or log exceptions without knowing their exact type.

```
try {
    someFunction();
} catch (...) {
    std::cerr << "An unknown exception occurred!\n";
}
```

A typedef cannot be a template

We must use a *using* directive instead

Return value optimization

Consider the code

```cpp
struct E
{
    E() { std::cout << "1"; }
    E(const E&) { std::cout << "2"; }
    ~E() { std::cout << "3"; }
};

E f()
{
    return E();
}

int main()
{
    f(); // 13 (no copy constructor called)
}
```

The copy constructor isn't called because of a compiler optimization known as copy elision (specifically, Return Value Optimization or RVO). In this case, the temporary E() object is constructed directly in the location where the return value of the function f() resides, so no copy (or move) constructor call is needed. This optimization is allowed (and in some cases mandated in C++17 and later) by the C++ standard.

Using function templates in multiple files

Consider the following program, which doesn't work correctly:

```cpp
// main.cpp:
#include <iostream>

template <typename T>
T addOne(T x); // function template forward declaration

int main() {
    std::cout << addOne(1) << '\n';
    std::cout << addOne(2.3) << '\n';

    return 0;
}

// add.cpp
template <typename T>
T addOne(T x) // function template definition
{
    return x + 1;
}
```

If addOne were a non-template function, this program would work fine: In main.cpp, the compiler would be satisfied with the forward declaration of addOne, and the linker would connect the call to addOne() in main.cpp to the function definition in add.cpp.

But because addOne is a template, this program doesn't work, and we get a linker error:

In main.cpp, we call addOne<int> and addOne<double>. However, since the compiler can't see the definition for function template addOne, it can't instantiate those functions inside main.cpp. It does see the forward declaration for addOne though, and will assume those functions exist elsewhere and will be linked in later.

When the compiler goes to compile add.cpp, it will see the definition for function template addOne. However, there are no uses of this template in add.cpp, so the compiler will not instantiate anything. The end result is that the linker is unable to connect the calls to addOne<int> and addOne<double> in main.cpp to the actual functions, because those functions were never instantiated.

If add.cpp had instantiated those functions, the program would have compiled and linked just fine. But such solutions are fragile and should be avoided:

The most conventional way to address this issue is to put all your template code in a header (.h) file instead of a source (.cpp) file

```cpp
// add.h
#ifndef ADD_H
#define ADD_H

template <typename T>
T addOne(T x) // function template definition
{
    return x + 1;
}

#endif
// main.cpp
#include "add.h" // import the function template definition
#include <iostream>

int main()
{
    std::cout << addOne(1) << '\n';
    std::cout << addOne(2.3) << '\n';

    return 0;
}
```

That way, any files that need access to the template can include the relevant header, and the template definition will be copied by the preprocessor into the source file. The compiler will then be able to instantiate any functions that are needed.

You may be wondering why this doesn't cause a violation of the one-definition rule (ODR). The ODR says that types, templates, inline functions, and inline variables are allowed to have identical definitions in different files. So there is no problem if the template definition is copied into multiple files (as long as each definition is identical).

But what about the instantiated functions themselves? If a function is instantiated in multiple files, how does that not cause a violation of the ODR? The answer is that functions implicitly instantiated from templates are implicitly inline. And as you know, inline functions can be defined in multiple files, so long as the definition is identical in each.