Cpp Nuances

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States

Contents

1 Converting char to std::string 2

1.1 Constructor signature 2

1.2 Example 2

2 std::string::npos 3

2.1 Example 3

3 Narrowing 4

4 Aggregate types 5

4.1 Aggregate initialization 5

4.2 Narrowing is not allowed in aggregate-initialization from an initializer list . 6

4.3 implicit conversion using a single-argument constructor 6

5 Floating point literals 8

6 Size of structs and classes 9

7 When is trailing return type useful 10

7.1 Decltype on template parameters 10

7.2 Nested structs 10

8 Most vexing parse 11

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

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

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,

```
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.

```
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.

```
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

```
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.

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.

```
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

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