Welcome back to C++ - Modern C++

Article • 11/07/2022

Since its creation, C++ has become one of the most widely used programming languages in the world. Well-written C++ programs are fast and efficient. The language is more flexible than other languages: It can work at the highest levels of abstraction, and down at the level of the silicon. C++ supplies highly optimized standard libraries. It enables access to low-level hardware features, to maximize speed and minimize memory requirements. C++ can create almost any kind of program: Games, device drivers, HPC, cloud, desktop, embedded, and mobile apps, and much more. Even libraries and compilers for other programming languages get written in C++.

One of the original requirements for C++ was backward compatibility with the C language. As a result, C++ has always permitted C-style programming, with raw pointers, arrays, null-terminated character strings, and other features. They may enable great performance, but can also spawn bugs and complexity. The evolution of C++ has emphasized features that greatly reduce the need to use C-style idioms. The old C-programming facilities are still there when you need them. However, in modern C++ code you should need them less and less. Modern C++ code is simpler, safer, more elegant, and still as fast as ever.

The following sections provide an overview of the main features of modern C++. Unless noted otherwise, the features listed here are available in C++11 and later. In the Microsoft C++ compiler, you can set the /std compiler option to specify which version of the standard to use for your project.

Resources and smart pointers

One of the major classes of bugs in C-style programming is the *memory leak*. Leaks are often caused by a failure to call `delete` for memory that was allocated with `new`. Modern C++ emphasizes the principle of *resource acquisition is initialization* (RAII). The idea is simple. Resources (heap memory, file handles, sockets, and so on) should be *owned* by an object. That object creates, or receives, the newly allocated resource in its constructor, and deletes it in its destructor. The principle of RAII guarantees that all resources get properly returned to the operating system when the owning object goes out of scope.

To support easy adoption of RAII principles, the C++ Standard Library provides three smart pointer types: std::unique_ptr, std::shared_ptr, and std::weak_ptr. A smart pointer handles the allocation and deletion of the memory it owns. The following example shows a class with an array member that is allocated on the heap in the call to `make_unique()`. The calls to `new` and `delete` are encapsulated by the `unique_ptr` class. When a `widget` object goes out of scope, the unique_ptr destructor will be invoked and it will release the memory that was allocated for the array.

C++

```cpp
#include <memory>
class widget
{
private:
    std::unique_ptr<int[]> data;
public:
    widget(const int size) { data = std::make_unique<int[]>(size); }
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000);  // lifetime automatically tied to enclosing scope
                        // constructs w, including the w.data gadget member
    // ...
    w.do_something();
    // ...
} // automatic destruction and deallocation for w and w.data
```

Whenever possible, use a smart pointer to manage heap memory. If you must use the **`new`** and **`delete`** operators explicitly, follow the principle of RAII. For more information, see Object lifetime and resource management (RAII).

`std::string` and `std::string_view`

C-style strings are another major source of bugs. By using std::string and std::wstring, you can eliminate virtually all the errors associated with C-style strings. You also gain the benefit of member functions for searching, appending, prepending, and so on. Both are highly optimized for speed. When passing a string to a function that requires only read-only access, in C++17 you can use std::string_view for even greater performance benefit.

`std::vector` and other Standard Library containers

The standard library containers all follow the principle of RAII. They provide iterators for safe traversal of elements. And, they're highly optimized for performance and have been thoroughly tested for correctness. By using these containers, you eliminate the potential for bugs or inefficiencies that might be introduced in custom data structures. Instead of raw arrays, use vector as a sequential container in C++.

```C++
vector<string> apples;
apples.push_back("Granny Smith");
```

Use map (not `unordered_map`) as the default associative container. Use set, multimap, and multiset for degenerate and multi cases.

```C++
map<string, string> apple_color;
// ...
apple_color["Granny Smith"] = "Green";
```

When performance optimization is needed, consider using:

- Unordered associative containers such as unordered_map. These have lower per-element overhead and constant-time lookup, but they can be harder to use correctly and efficiently.
- Sorted `vector`. For more information, see Algorithms.

Don't use C-style arrays. For older APIs that need direct access to the data, use accessor methods such as `f(vec.data(), vec.size());` instead. For more information about containers, see C++ Standard Library Containers.

Standard Library algorithms

Before you assume that you need to write a custom algorithm for your program, first review the C++ Standard Library algorithms. The Standard Library contains an ever-growing assortment of algorithms for many common operations such as searching, sorting, filtering, and randomizing. The math library is extensive. In C++17 and later, parallel versions of many algorithms are provided.

Here are some important examples:

- `for_each`, the default traversal algorithm (along with range-based `for` loops).
- `transform`, for not-in-place modification of container elements
- `find_if`, the default search algorithm.

- `sort`, `lower_bound`, and the other default sorting and searching algorithms.

To write a comparator, use strict `<` and use *named lambdas* when you can.

C++

```
auto comp = [](const widget& w1, const widget& w2)
     { return w1.weight() < w2.weight(); }

sort( v.begin(), v.end(), comp );

auto i = lower_bound( v.begin(), v.end(), widget{0}, comp );
```

`auto` instead of explicit type names

C++11 introduced the auto keyword for use in variable, function, and template declarations. `auto` tells the compiler to deduce the type of the object so that you don't have to type it explicitly. `auto` is especially useful when the deduced type is a nested template:

C++

```
map<int,list<string>>::iterator i = m.begin(); // C-style
auto i = m.begin(); // modern C++
```

Range-based `for` loops

C-style iteration over arrays and containers is prone to indexing errors and is also tedious to type. To eliminate these errors, and make your code more readable, use range-based `for` loops with both Standard Library containers and raw arrays. For more information, see Range-based for statement.

C++

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v {1,2,3};

    // C-style
    for(int i = 0; i < v.size(); ++i)
    {
        std::cout << v[i];
    }

    // Modern C++:
    for(auto& num : v)
    {
        std::cout << num;
    }
}
```

`constexpr` expressions instead of macros

Macros in C and C++ are tokens that are processed by the preprocessor before compilation. Each instance of a macro token is replaced with its defined value or expression before the file is compiled. Macros are commonly used in C-style programming to define compile-time constant values. However, macros are error-prone and difficult to debug. In modern C++, you should prefer constexpr variables for compile-time constants:

C++

```
#define SIZE 10 // C-style
constexpr int size = 10; // modern C++
```

Uniform initialization

In modern C++, you can use brace initialization for any type. This form of initialization is especially convenient when initializing arrays, vectors, or other containers. In the following example, `v2` is initialized with three instances of `S`. `v3` is initialized with three instances of `S` that are themselves initialized using braces. The compiler infers the type of each element based on the declared type of `v3`.

C++

```
#include <vector>

struct S
{
    std::string name;
    float num;
    S(std::string s, float f) : name(s), num(f) {}
};

int main()
{
    // C-style initialization
    std::vector<S> v;
    S s1("Norah", 2.7);
    S s2("Frank", 3.5);
    S s3("Jeri", 85.9);

    v.push_back(s1);
    v.push_back(s2);
    v.push_back(s3);

    // Modern C++:
    std::vector<S> v2 {s1, s2, s3};

    // or...
    std::vector<S> v3{ {"Norah", 2.7}, {"Frank", 3.5}, {"Jeri", 85.9} };

}
```

For more information, see Brace initialization.

Move semantics

Modern C++ provides *move semantics*, which make it possible to eliminate unnecessary memory copies. In earlier versions of the language, copies were unavoidable in certain situations. A *move* operation transfers ownership of a resource from one object to the next without making a copy. Some classes own resources such as heap memory, file handles, and so on. When you implement a resource-owning class, you can define a *move constructor* and *move assignment operator* for it. The compiler chooses these special members during overload resolution in situations where a copy isn't needed. The Standard Library container types invoke the move constructor on objects if one is defined. For more information, see Move Constructors and Move Assignment Operators (C++).

Lambda expressions

In C-style programming, a function can be passed to another function by using a *function pointer*. Function pointers are inconvenient to maintain and understand. The function they refer to may be defined elsewhere in the source code, far away from the point at which it's invoked. Also, they're not type-safe. Modern C++ provides *function objects*, classes that override the operator()

operator, which enables them to be called like a function. The most convenient way to create function objects is with inline lambda expressions. The following example shows how to use a lambda expression to pass a function object, that the `find_if` function will invoke on each element in the vector:

```C++
std::vector<int> v {1,2,3,4,5};
int x = 2;
int y = 4;
auto result = find_if(begin(v), end(v), [=](int i) { return i > x && i < y; });
```

The lambda expression `[=](int i) { return i > x && i < y; }` can be read as "function that takes a single argument of type `int` and returns a boolean that indicates whether the argument is greater than `x` and less than `y`." Notice that the variables `x` and `y` from the surrounding context can be used in the lambda. The `[=]` specifies that those variables are *captured* by value; in other words, the lambda expression has its own copies of those values.

Exceptions

Modern C++ emphasizes exceptions, not error codes, as the best way to report and handle error conditions. For more information, see Modern C++ best practices for exceptions and error handling.

`std::atomic`

Use the C++ Standard Library std::atomic struct and related types for inter-thread communication mechanisms.

`std::variant` (C++17)

Unions are commonly used in C-style programming to conserve memory by enabling members of different types to occupy the same memory location. However, unions aren't type-safe and are prone to programming errors. C++17 introduces the std::variant class as a more robust and safe alternative to unions. The std::visit function can be used to access the members of a `variant` type in a type-safe manner.

See also

C++ Language Reference

Lambda Expressions

C++ Standard Library

Microsoft C/C++ language conformance