CS100 Lecture 28

Compile-time Computations and C++ Summary

Contents

Example: Binary literals

Binary literals

Built-in binary literals support: since C++14 and C23.

```
switch (code) {
  case 0b0110011: // ...
  case 0b0010011: // ...
  case 0b0000011: // ...
  case 0b0100011: // ...
  // ...
}
```

How do people write binary literals when there is no built-in support?

Runtime solution? No!

This is not satisfactory: The value is computed at run-time!

```
// Convert a "binary number" to a decimal.
int bin2dec(int x) {
  int result = 0, pow_two = 1;
  while (x > 0) {
    result += (x % 10) * pow_two;
    x /= 10;
    pow_two *= 2;
  }
  return result;
}

const int forty_two = bin2dec(101010); // Correct, but slow.
switch (code) {
  case bin2dec(110011): // Error! 'case' label must be compile-time constant!
  // ...
}
```

Preprocessor metaprogramming solution

`#` and `##` operators: Both are used in function-like macros.

`#x`: Stringify `x`.

```
#define SHOW_VALUE(x) std::cout << #x << " == " << x

int ival = 42;
SHOW_VALUE(ival); // std::cout << "ival" << " == " << ival;
```

`a##b`: Concatenate `a` and `b`.

```
#define DECLARE_HANDLER(name) void handler_##name(int err_code)

DECLARE_HANDLER(overflow); // void handler_overflow(int err_code);
```

Preprocessor metaprogramming solution

```
// Four binary digits represent one hexadecimal digit.
#define BX_0000 0
#define BX_0001 1
#define BX_0010 2
// ......
#define BX_1110 E
#define BX_1111 F

#define BIN_A(x) BX_##x

#define BIN_B(x, y) 0x##x##y
#define BIN_C(x, y) BIN_B(x, y)

#define BIN(x, y) BIN_C(BIN_A(x), BIN_A(y))

// Convert a "binary number" (two groups of 4 bits) into a hexadecimal literal.
const int x = BIN(0010, 1010); // BIN_C(BIN_A(0010), BIN_A(1010)) =>
                               // BIN_B(BX_0010, BX_1010) =>
                               // 0x##2##a => 0x2a => 42
```

Template metaprogramming solution

```
// Instantiation of `Binary` converts a "binary number" to a decimal.
// e.g., 111 => ((0 * 2 + 1) * 2 + 1) * 2 + 1 = 7
template <unsigned N> struct Binary {
  static const unsigned value = Binary<N / 10>::value * 2 + (N % 10);
};
template <> struct Binary<0u> { // Specialization for N = 0
  static const unsigned value = 0;
};

const auto x = Binary<101010>::value; // 42
switch (code) {
  case Binary<110011>::value: // OK.
  // ...
}
```

Modern C++: `constexpr` function

Just mark the function `constexpr`, and the compiler will be able to execute it!

```
constexpr int bin2dec(int x) {
  int result = 0, pow_two = 1;
  while (x > 0) {
    result += (x % 10) * pow_two; x /= 10; pow_two *= 2;
  }
  return result;
}

switch (code) {
  case bin2dec(101010): // OK. Since `101010` is a compile-time constant,
                        // the function is executed at compile-time and
                        // produces a compile-time constant.
  // ...
}
```

Compile-time computations

How much work can be done in compile-time?

- Call to numeric functions with compile-time known arguments?
  - e.g., can `std::acos(-1)` be computed in compile-time?
- Even crazier: Compile-time raytracer?!
  - The computations are done entirely in compile-time. At run-time, the only work is to output the image.

Anything can be computed in compile-time, provided that the arguments are compile-time known!

`constexpr` and `consteval`

`constexpr`

Constant expressions: expressions that are evaluated at compile-time.

`constexpr` variable:

```
constexpr double dval = 5.2;
```

By declaring a variable `constexpr`, we mean that its value is compile-time known, and will not change.

- A `constexpr` variable is implicitly `const`.
- It must be initialized from a constant expression. Otherwise, an compile-error.

A `const` variable initialized from a constant expression is also a constant expression.

```
const int ival = 42;
```

`constexpr` functions

`constexpr` functions are **potentially** executed at compile-time:

- When the arguments are constant expressions, it is run at compile-time and produces a constant expression.
- When the arguments are not constant expressions, it is run at run-time just like a normal function.

```cpp
constexpr int add(int a, int b) {
  return a + b;
}
int main() {
  const int x = 10, y = 16;
  constexpr auto result = add(x, y); // OK. The result is a constant expression.
  int n, m; std::cin >> n >> m;
  std::cout << add(n, m) << '\n'; // OK. It is computed at run-time.
}
```

`constexpr` member functions

Member functions may also be `constexpr`. This is particularly useful for some very simple classes:

```
class StringView { // The 'StringView' class in lecture 27.
  const char *mStart = nullptr;
  std::size_t mLength = 0;
public:
  // Constructors
  constexpr StringView(const char *cstr);
  constexpr StringView(const std::string &str);
  // Length
  constexpr std::size_t size() const { return mLength; }
  constexpr bool empty() const { return mStart; }
  // Searching
  constexpr std::size_t find(char c, std::size_t pos = 0) const;
  // ...
};
```

Evolution of `constexpr` functions

`constexpr` was first introduced in C++11, with many restrictions:

- A single `return` statement only. No loops or branches.
- `constexpr` member functions are implicitly `const`: They cannot modify the data members.
- `virtual` functions cannot be `constexpr`.
- Very little standard library support.
- ......

Evolution of `constexpr` functions

In C++14:

- Multiple statements, loops and branches are allowed.
- `constexpr` member functions are no longer implicitly `const`.
- `constexpr` lambdas are still not yet allowed.

In C++17:

- Much more standard library support: A lot more functions are made `constexpr` since C++17.
- Lambdas are automatically `constexpr` when it can be.

Evolution of `constexpr` functions

C++20: A huge step!

- `constexpr` functions can perform **dynamic memory allocations**!
    - Memory allocated at compile-time must also be released at compile-time.
- **Destructors** can be `constexpr`!
- Standard library **containers** like `std::vector`, `std::string` can be `constexpr`!
- Standard library **algorithms** are `constexpr`!
- `virtual` functions can be `constexpr`!

C++20: `constexpr` support in the standard library

```cpp
#include <vector>
#include <algorithm>

constexpr int find_or_42(const std::vector<int> &vec, int target) {
  auto found = std::ranges::find(vec, target); // Compile-time search
  return found == vec.end() ? 42 : *found;
}

int main() {
  // 'vec' is initialized in compile-time!
  constexpr auto result_1 = find_or_42({1, 4, 2, 8, 5, 7}, 10); // 42
  constexpr auto result_2 = find_or_42({2, 3, 5, 7}, 3);        // 3
  static_assert(result_1 == 42);
  static_assert(result_2 == 3);
}
```

`constexpr` numeric functions

Since C++23, some simple numeric functions in `<cmath>`, like `abs`, `ceil`, `floor`, `trunc`, `round`, ... are `constexpr`.

Since C++26, the **power, square/cubic root, trigonometric, hyperbolic, exponential and logarithmic** functions are all `constexpr`!

`constexpr` functions are pure

Pure functions:

- Produce the same result when given the same arguments.
- Have no side effects. They cannot modify the value of variables outside them.

The following functions are impure:

```cpp
int g = 10; // A global variable

int fun(int x){
  return x + g; // Depend on the global variable `g` and
                // can return different results for the same argument.
}
int foo(int x){
  g++; // Modify the global variable `g`.
  return x;
}
```

`consteval`: Immediate functions

`consteval` generates an *immediate* function.

- An immediate function must be executed at compile-time to produce a constant expression.

`consteval` cannot be applied to destructors.

A `consteval` function has the same requirements as a `constexpr` function.

`consteval`: Immediate functions

- `constexpr`: **potentially** executed at compile-time.
- `consteval`: **must be** executed at compile-time.

```
consteval int sqr(int n) {
  return n * n;
}

constexpr int r = sqr(100); // OK.
int x = 100;      // 'x' is not a constant expression!
int r2 = sqr(x); // Error: 'sqr' must be called with constant expressions.
```

Note: A non-`const` variable is not treated as a constant expression, even if initialized from a constant expression.

C++ Summary

The past

Back to 1979, the Bell Labs: C with Classes made by Bjarne Stroustrup.

- An object-oriented C with the ideas of "class" from Simula (and several other languages).
- Member functions, subclasses, constructors and destructors, protection mechanisms (`public`, `private`, `friend`), ...
- Based on C, with many improvements.

The past

After C with Classes was seen as a "medium success" by Stroustrup, he moved on to make a better new language - C++ was born (1983).

- Virtual functions, overloading, references, `const`, ...
- Templates, exceptions, RTTI, namespaces, STL were added in the 1990s.

By the year 1998, C++ had become matured and standardized with the four major parts (*Effective C++* Item 1):

- C
- Object-Oriented C++
- Template C++
- The STL

Entering Modern C++

A huge step at 2011:

- Rvalue references, move semantics, variadic templates, perfect forwarding
- Better template metaprogramming support
- Smart pointers
- `auto` and `decltype`
- Lambdas, `std::function`
- The concurrency library (`std::thread`, `std::mutex`, `std::atomic`, ...)
- `constexpr`: Support for more straightforward compile-time computations
- ......

Evolution since C++11

More specialized library facilities:

- `optional`, `tuple`
- `string_view`
- `filesystem`: Standardized file system library
- `regex`: The regular expression library

More compile-time computation support:

- More restrictions on `constexpr` functions and `auto` deduction are removed.
- Class Template Argument Deduction (CTAD)

C++20 is historic!

CppCon2021 Talk by Bjarne Stroustrup: C++20: Reaching the aims of C++

C++20 is the first C++ standard that delivers on virtually all the features that Bjarne Stroustrup dreamed of in *The Design and Evolution of C++* in 1994.

- Coroutines (Talk)
- Concepts and requirements (`concept`, `requires`) (Talk)
- Modules (Talk) (Talk on the implementation by MSVC)
- Ranges library
- Formatting library
- Three-way comparison (`operator<=>`, `std::partial_ordering`, ...)

Future

- Static reflection and metaprogramming (Talk) (P2237R0) (P1240R1) (P2320R0)
- Metaclasses: Generative C++ (P0707R3)
- Pattern matching (Talk) (Herb Sutter's `cppfront` project)
- Structured concurrency support (executors) (Talk) (P2300R6)
- Internal representation of C++ code suitable for analysis (Talk) (GitHub Page)
- ……

Goodbye CS100

C++ is so complex! But programming is not only about C++.

- Basic mathematics
- Data structures and Algorithms
- Basic knowledge about computer systems
- Deeper understanding of the specific field of your interest

Goodbye CS100

Skills that you'd better develop:

- A *main* programming language that you master and understand
- Shell scripts
- Git
- Markdown and $\LaTeX$

Goodbye CS100

Good luck and have fun in computer science.