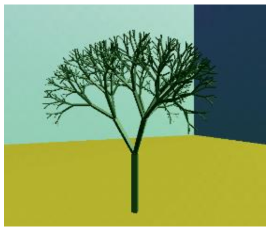

GATS Companion

C++: const(…) = speed & safety

Author: Garth Santor
Editors: Trinh Hān
Version/Copyright: 1.0.0 (2024-01-20)

Overview

Constants in C and C++ are often misunderstood, at times appearing magical often disappearing in the machine code. In this document I discuss the foundational concepts, the purpose of the const language elements, best practices, and the future.

For now, I've intentionally left off discussing `const`… in relation to classes. I'll cover that sometime in the future.

Concepts

Mathematical constant

A fixed and well-defined number such as *zero*, *one*, *pi*, *e*, *i*, *the golden ratio*, etc.

The value of a mathematical constant is determined by theorems and bear names reflecting their purpose or the mathematician that discovered the value.

The five most important mathematical constants are:

- **Zero** – the additive identity.
- **One** – the multiplicative identity.
- **Pi** or π – the ratio of a circle's circumference to its diameter.
- ***e*** – Euler's number is the base of the natural logarithms.
- ***i*** – the imaginary number, such that $i^2 = -1$.

Physical constant

Physical constants are like a mathematical constant but not explainable by a theorem. They are determined by experimentation.

Some of the most famous physical constants are:

- the gravitational constant *G*.
- the speed of light *c*.
- the Planck constant *h*.

Constant functions

A mathematical function whose output is the same for every input.

Example: *y(x)=42* is a function that defines a horizontal line with a y-intercept of 42.

Literal

A textual representation of a value (in source code).

C++ Examples:

'c' is a character literal
"hello" is a string literal
42 is an integer literal
3.14 is a real number literal

Constants in programming

A value that is not altered by the program during normal execution.

Literals are unnamed constants.

Named constants have an identifier associated with the value.

Examples:

`constexpr int num_cores{32};`

`std::numbers::pi`

Variable in programming

An abstract storage location with a symbolic name.

e.g., `int age{42};`

In this example the symbolic name 'age' is associated with a storage location that has been abstracted by the compiler. In this case, either in global static memory or the process stack. Either way, the programmer doesn't need to be concerned with the specific location.

Constant expression/Constant folding

An expression that can be evaluated at compile-time as opposed to runtime (i.e., the resulting value of the expression never changes).

Example: 42 / 2 always evaluates to 21 and is therefore a constant expression.

The process of identifying and evaluating constant expressions is called 'constant folding'. Constant folding has been a feature of compilers going back to Fortran and COBOL during the 1950s.

C++ language support

'const'

A type-qualifier that indicates that the named data is read-only. This restriction only applies after a value is assigned to that memory location.

A variable tagged as const could be evaluated at compile-time if its value comes from a literal or constant expression.

Example: `const int max_cores{ std::thread::hardware_concurrency() };`

'constexpr' variables

'constexpr' is a type-qualifier that indicates that the named data is read-only and will be assigned its value at compile time.

'constexpr' variables can acquire their values from *literals*, other *constexpr variables*, *constexpr functions*, or *consteval functions*.

Example: `constexpr int gigabyte{ 1024 * 1024 * 1024 };`

'constinit' variables

'constinit' is a type-qualifier that indicates that the named data will be assigned its value at compile time but that it is still a variable that can be modified later at runtime.

Example: `constinit static size_t cache_size{ std::numeric_limits<size_t>::max() / 5 };`

'constexpr' functions

'constexpr' preceding a function definition indicates that the function is capable of evaluation at either runtime or at compile time.

Compile time evaluation occurs with the function parameters can be resolved at compile time (i.e., they are literals or constexpr variables). If the parameter values cannot be resolved until runtime, a constexpr function will be executed at runtime.

Example:

```
constexpr int square(int n) { return n * n; }
int z;
cin >> z;

int x = square(5);        // compile-time
int y = square(z);        // runtime
```

‘consteval’ functions

‘consteval’ preceding a function definition indicates that the function should only be evaluated at compile time.

The arguments to a consteval function must be literals, constexpr variables, or the result of a consteval function (or constexpr function that can be evaluated at compile-time)

Example:

```
consteval int square(int n) { return n * n; }
int z;
cin >> z;

int x = square(5);        // compile-time
int y = square(z);        // error
```

‘std::is_constant_evaulated()’

Is a C++ 20 function that returns true if the enclosing function is evaluated at compile time but returns false if it is evaluated at runtime. The function is found in the `<type_traits>` library.

Example:

```
if (std::is_constant_evaluated())
    return square(n);           // as defined above...
else
    return n*n;
```

What is the point?

What is the point to all of this? My program works without them.

As the title to this document indicates, its all about speed and safety.

Safety first!

Readability

Constants with clear names make it easy to determine that the computation is correct.

Example:

```
c = 6.28318530727958647692528676655 * radius;      // circumference
```

vs.

```
c = 2 * pi * radius;
```

The first one is incorrect; it should have been:

```
c = 6.28318530717958647692528676655 * radius;
```

Logic errors become syntax errors.

Typographical errors can be caught by the compiler.

Example:

```
void foo(int n) {
    if (n = 42)
        do_this();
    else
```

```
        do_that();
}
```

This incorrect code always executes *do_this()* since the if condition is an assignment, not the intended comparison.

The correct line should have read, "**if** (n == 42)"

This logic error becomes a syntax error by adding a const to the parameter.

```
void foo(const int n) {
    if (n = 42)                 // error: cannot assign to constant 'n'
        do_this();
    else
        do_that();
}
```

Speed

Yes, sometimes you can have your cake, and eat it too!

All of the constant tags provide information to the compiler's optimizer that allows it to safely take shortcuts.

- Constant folding reduces the number of operations required to complete a computation.
- Const tells the compiler that the value will not change and therefore does not need to be repeatedly accessed from memory (it would be safe to leave the value in a CPU register).
- Constexpr tells the compiler that the value of the expression will never change, so go ahead and pre-calculate the result. The compiler can often avoid a separate read operation to acquire the value, instead bundling the value with the operation that will use it (the value is loaded when the machine code that will use it is loaded).
- Constinit can guarantee that global and static variables will have their initial values when the program loads, as they would be stored in the executable file as opposed to being computed after the program loads. This makes many initializations effectively 'free'.
- Consteval guarantees that a computation that a programmer believes to be compile-time computable is in-fact computed at compile-time.

Best Practices

1. Declare mathematical and physical constants as *constexpr variables*. Mathematical constants either never change or rarely. If they do, you can make a single change that would update every reference to that constant.
2. Limit *const* to variables whose values are determined at runtime but don't change after input or after being passed to a function.
3. Write any function as a *constexpr function* if possible.
4. Use 'constinit' to help variables load faster (where you can).
5. Use 'consteval' to help force compile-time execution where you know this is possible.
6. Use 'is_constant_evaluated()' to provide the flexible options: compile-time where you can, runtime where you have to.

In modern C++ (which starts with C++ 11), best practices #1 – 3 should become routine and applied wherever you can.

Practices #4 – 6 are restricted to C++ 20 and later and tend to be used by library developers and experts that are performance tuning their code.

Future

'consteval if'

With C++ 23, if-statements can be tagged with 'consteval' or 'not consteval' which will allow a function to choose between a runtime or compile time solution as required.

Full math library support

With C++ 26, the floating-point functions will become constexpr functions.

Document History

Version	Date	Activity
1.0.0	2024-01-20	Document created.