Back to Basics: const and constexpr

RAINER GRIMM
Many Flavors of Constness

**Flavors**

- `const`
- `const_cast`
- `constexpr`
- `consteval`
- `constinit`
- `is_constant_evaluated`

**Differences**

- Function Execution
- Variable Initialization
Many Flavors of Constness

**Flavors**
- `const`
- `const_cast`
- `constexpr`
- `consteval`
- `constinit`
- `is_constant_evaluated`

**Differences**
- Function Execution
- Variable Initialization
const

**const correctness**: Use the keyword `const` to prevent `const` objects from getting mutated. **const** is a quality attribute of your program.

**const objects**
- must be initialized.
- cannot be modified.
- cannot be victims of data races.
- can only invoke `const` member functions.
const

- **const** member functions cannot change the object.
  ```cpp
  struct Immutable {
    int val{12};
    void canNotModify() const {
      val = 13;    // ERROR
    }
  };
  ```

- Distinguish physical and logical constness of an object.
  - Physical constness: The object is `const` and cannot be changed.
  - Logical constness: The object is `const` but could be changed.

- Declare members that can be changed in `const` member functions as `mutable`.

threadSafeCounter.cpp
**const**

- **By default, pass pointers and references to** `const`

  ```
  voidgetCString(const char* cStr);
  voidgetCppString(const std::string& cppStr);
  ```

- **Semantic:**
  - Pointer and references do not pass ownership → they borrow the resource from the caller
  - A pointer can be a null pointer → you have to check it

- **Exception for non-**`const`** pointers and references**

  ```
  voidmodifyCString(char* cStr);
  voidmodifyCppString(std::string& cppStr);
  ```

  → in/out parameter
const

The pointer and the pointee can be const.

- `const char* cStr`:
  - `cStr points to a char that is const`
  - The pointee cannot be modified, but the pointer can.
- `char* const cStr`:
  - `cStr is a const pointer to char`
  - The pointer cannot be modified, but the pointee can.
- `const char* const cStr`:
  - `cStr is a const pointer to a char that is const`
  - Neither the pointer nor the pointee can be modified.

Read the expressions from right to left.
Many Flavors of Constness

**Flavors**

- `const`
- `const_cast`
- `constexpr`
- `consteval`
- `constinit`
- `is_constant_evaluated`

**Differences**

- Function Execution
- Variable Initialization
const_cast allows it to remove or add the const or volatile qualifier to a variable.

Modifying a const declared object by removing its constness is undefined behavior.

Don’t use a C-cast (`int i = (int) myValue;`), because is applies eventually a series of casts:

```
static_cast const_cast reinterpret_cast
```

modifyingConst.cpp
constCast.cpp
Many Flavors of Constness

### Flavors

- `const`
- `const_cast`
- `constexpr`
- `consteval`
- `constinit`
- `is_constant_evaluated`

### Differences

- Function Execution
- Variable Initialization
Constant expressions
- can be evaluated at compile time.
- give the compiler deep insight.
- are implicit thread-safe.

- Variables
  
  ```cpp
  constexpr double myDouble = 5.2;
  const int myInt = 5;
  ```

  - are implicit `const`.
  - are implicit thread-safe. A data race requires shared mutable state.
  - `const` variables are implicit `constexpr` when initialized with a constant expression.

`const/constexpr` variables make it easy to reason about your concurrent program.
### Functions

```cpp
constexpr int gcd(int a, int b) {
    while (b != 0) {
        auto t = b;
        b = a % b;
        a = t;
    }
    return a;
}
```

- must resolve each dependency at compile time.
- can have variables that must be initialized by constant expressions.
- cannot have `static` and `thread_local` variables.

- have the potential to run at compile time. Must run at compile time when used in a constant expression.
- are pure.
Pure Functions (Mathematical functions)
- Produce the same result when given the same arguments (referential transparency).
- Have no side-effects.
- Don’t change the state of the program.

Advantages
- Easy to test and to refactor
- The call sequence of functions can be changed
- Automatically parallelizable
- Results can be cached
constexpr

### User-defined types

```cpp
struct MyDouble {
  double myVal;
  constexpr MyDouble(double v): myVal(v){}
  constexpr double getVal(){return myVal;}
};
```

- must have at least one `constexpr` constructor.
- can have `constexpr` and non-`constexpr` member functions.
- `constexpr` objects can only invoke `constexpr` member functions.
C++20 supports the `constexpr containers` `std::vector` and `std::string`.

Memory allocated at compile time must also be released at compile time. Transient allocation

- The more than 100 [algorithms of the STL](#) are declared as `constexpr` in C++20.

If possible, declare user-defined types or functions as `constexpr`.

`constexprVector.cpp`
Many Flavors of Constness

**Flavors**
- `const`
- `const_cast`
- `constexpr`
- `consteval`
- `constinit`
- `is_constant_evaluated`

**Differences**
- Function Execution
- Variable Initialization
consteval

cconsteval generates an immediate function.

- Every call of an immediate function generates a constant expression that is executed at compile time.

consteval

- cannot be applied to destructors.
- has the same requirements as a constexpr function.

```cpp
consteval int sqr(int n) {
    return n * n;
}
constexpr int r = sqr(100);  // OK

int x = 100;
int r2 = sqr(x);            // Error
```
Many Flavors of Constness

**Flavors**
- `const`
- `const_cast`
- `constexpr`
- `consteval`
- `constinit`
- `is_constant_evaluated`

**Differences**
- Function Execution
- Variable Initialization
constinit guarantees that a variable with static storage duration is initialized at compile time. This variable is still mutable.

- Global objects, or objects declared with `static` or `extern`, have static storage duration.
- Objects with a static storage duration are allocated at the program start and deallocated at its end.
Static Initialization Order Fiasco: The initialization order of static variables between translation units is not specified.

- Initialization of static happens in two steps.
  - Compile time. Statics that are not const-initialized are zero-initialized.
  - Run-time: The zero-initialized statics are dynamic initialized at run time.

constinit solves the static initialization order fiasco.
```cpp
// sourceSIOF1.cpp
int square(int n) {
    return n * n;
}
auto staticA = square(5);

// mainSIOF1.cpp
#include <iostream>
extern int staticA;
auto staticB = staticA;

int main() {
    std::cout << "staticB: " << staticB;
}
```
// sourceSOIF3.cpp
constexpr int quad(int n) {
    return n * n;
}

constinit auto staticA = quad(5);

int main() {
    std::cout << "staticB: " << staticB;
}
Many Flavors of Constness

**Flavors**
- `const`
- `const_cast`
- `constexpr`
- `consteval`
- `constinit`
- `is_constant_evaluated`

**Differences**
- Function Execution
- Variable Initialization
`std::is_constant_evaluated` determines whether the function is executed at compile time or run time.

```cpp
constexpr double power(double b, int x) {
    if (std::is_constant_evaluated() && !(b == 0.0 && x < 0)) {
        if (x == 0) return 1.0;
        double r = 1.0, p = x > 0 ? b : 1.0 / b;
        auto u = unsigned(x > 0 ? x : -x);
        while (u != 0) {
            if (u & 1) r *= p;
            u /= 2;
            p *= p;
        }
        return r;
    } else return std::pow(b, double(x)); // not declared constexpr
}
```

[https://en.cppreference.com/w/cpp/types/is_constant_evaluated](https://en.cppreference.com/w/cpp/types/is_constant_evaluated)
Many Flavors of Constness

Flavors
- `const`
- `const_cast`
- `constexpr`
- `consteval`
- `constinit`
- `is_constant_evaluated`

Differences
- Function Execution
- Variable Initialization
Function Execution

```
#include <iostream>

int sqrRunTime(int n) { return n * n; }
consteval int sqrCompileTime(int n) { return n * n; }
constexpr int sqrRunOrCompileTime(int n) { return n * n; }

int main() {
    constexpr int prod1 = sqrRunTime(100); // ERROR
    constexpr int prod2 = sqrCompileTime(100);
    constexpr int prod3 = sqrRunOrCompileTime(100);

    int x = 100;
    int prod4 = sqrRunTime(x);
    int prod5 = sqrCompileTime(x); // ERROR
    int prod6 = sqrRunOrCompileTime(x);
}
```
Many Flavors of Constness

**Flavors**

- `const`
- `const_cast`
- `constexpr`
- `consteval`
- `constinit`
- `is_constant_evaluated`

**Differences**

- Function Execution
- Variable Initialization
Variable Initialization

#include <iostream>

constexpr int constexprVal = 1000;
constinit int constinitVal = 1000;

int main() {
    auto val = 1000;
    const auto res = ++val;

    std::cout << "res: " << ++res << '\n';           // ERROR
    std::cout << "++constexprVal: " << ++constexprVal << '\n'; // ERROR
    std::cout << "++constinitVal: " << ++constinitVal << '\n';

    constexpr auto localConstexpr = 1000;
    constinit auto localConstinit = 1000;                // ERROR
}

constexprConstinit.cpp
Variable Initialization

Initialization of a local non-cost variable at compile time.

```cpp
consteval auto doubleMe(auto val) {
    return 2 * val;
}

int main() {
    auto res = doubleMe(1010);  // compile-time initialization
    ++res;                   // 2021    // non-const
}
```

compileTimeInitializationLocal.cpp
Many Flavors of Constness

**Flavors**
- `const`
- `const_cast`
- `constexpr`
- `consteval`
- `constinit`
- `is_constant_evaluated`

**Differences**
- Function Execution
- Variable Initialization