5.4 — Constant expressions and compile-time optimization

The as-if rule

In C++, compilers are given a lot of leeway to optimize programs. The **as-if rule** says that the compiler can modify a program however it likes in order to produce more optimized code, so long as those modifications do not affect a program's "observable behavior".

For advanced readers

There is one exception to the as-if rule: unnecessary calls to a copy constructor can be elided (omitted) even if those copy constructors have observable behavior. We cover this topic in lesson 14.15 -- Class initialization and copy elision.

Exactly how a compiler optimizes a given program is up to the compiler itself. However, there are things we can do to help the compiler optimize better.

An optimization opportunity

Consider the following short program:

```
#include <iostream>

int main()
{
	int x { 3 + 4 };
	std::cout << x << '\n';

	return 0;
}
```

The output is straightforward:

```
7
```

However, there's an interesting optimization possibility hidden within.

If this program were compiled exactly as it was written (with no optimizations), the compiler would generate an executable that calculates the result of `3 + 4` at runtime (when the program is run). If the program were executed a million times, `3 + 4` would be evaluated a million times, and the resulting value of `7` produced a million times.

Because the result of `3 + 4` never changes (it is always `7`), re-calculating this result every time the program is run is wasteful.

Compile-time evaluation of expressions

Modern C++ compilers are able to evaluate some expressions at compile-time. When this occurs, the compiler can replace the expression with the result of the expression.

For example, the compiler could optimize the above example to this:

```
#include <iostream>

int main()
{
	int x { 7 };
	std::cout << x << '\n';

	return 0;
}
```

This program produces the same output (`7`) as the prior version, but the resulting executable no longer needs to spend CPU cycles calculating `3 + 4` at runtime! Even better, we don't need to do anything to enable this behavior (besides have optimizations turned on).

Key insight

Such optimizations make our compilation take longer (because the compiler has to do more work), but because expressions only need to be evaluated once at compile-time (rather than every time the program is run) the resulting executables are faster and use less memory.

The ability for C++ to perform compile-time evaluation is one of the most important and evolving areas of modern C++.

Constant expressions

One kind of expression that can always be evaluated at compile time is called a "constant expression". The precise definition of a constant expression is complicated, so we'll take a simplified view: A **constant expression** is an expression that contains only compile-time constants and operators/functions that support compile-time evaluation.

A **compile-time constant** is a constant whose value *must be* known at compile time. This includes:

- Literals (e.g. '5', '1.2')
- Constexpr variables (we discuss these shortly in lesson 5.5 -- Constexpr variables)
- Const integral variables with a constant expression initializer (e.g. `const int x { 5 };`). This is a historical exception -- in modern C++, constexpr variables are preferred.

- Non-type template parameters (see 11.9 -- Non-type template parameters).
- Enumerators (see 13.2 -- Unscoped enumerations).

Const variables that are not compile-time constants are sometimes called **runtime constants**. Runtime constants cannot be used in a constant expression.

Tip

Const non-integral variables are always runtime constants (even if they have a constant expression initializer). If you need such variables to be compile-time constants, define them as constexpr variables instead (see lesson 5.5 -- Constexpr variables).

The most common type of operators and functions that support compile-time evaluation include:

- Arithmetic operators with operands that are compile-time constants (e.g. `1 + 2`)
- Constexpr and consteval functions (we'll discuss these later in the chapter)

In the following example, we identify the constant expressions and non-constant expressions. We also identify which variables are non-constant, runtime constant, or compile-time constant.

```
#include <iostream>

int getNumber()
{
    std::cout << "Enter a number: ";
    int y{};
    std::cin >> y;

    return y;
}

int main()
{
    // Non-const variables:
    int a { 5 };                // 5 is a constant expression
    double b { 1.2 + 3.4 };     // 1.2 + 3.4 is a constant expression

    // Const integral variables with a constant expression initializer
    // are compile-time constants:
    const int c { 5 };          // 5 is a constant expression
    const int d { c };          // c is a constant expression
    const long e { c + 2 };     // c + 2 is a constant expression

    // Other const variables are runtime constants:
    const int f { a };          // a is not a constant expression
    const int g { a + 1 };      // a + 1 is not a constant expression
    const long h { a + c };     // a + c is not a constant expression
    const int i { getNumber() }; // getNumber() is not a constant expression

    const double j { b };       // b is not a constant expression
    const double k { 1.2 };     // 1.2 is a constant expression

    return 0;
}
```

An expression that is not a constant expression is sometimes called a **runtime expression**. For example, `std::cout << x << '\n'` is a runtime expression, both because `x` is not a compile-time constant, and because `operator<<` doesn't support compile-time evaluation when used for output (since output can't be done at compile-time).

Why we care about constant expressions

Constant expressions are useful for two reasons:

- Constant expressions are *always* eligible for compile-time evaluation, meaning they are more likely to be optimized at compile-time.
- Certain C++ features require constant expressions.

For advanced readers

A few common cases where a constant expression is required:

- The initializer of a constexpr variable (5.5 -- Constexpr variables).
- A non-type template argument (11.9 -- Non-type template parameters).
- The defined length of a `std::array` (17.1 -- Introduction to std::array) or a C-style array (17.7 -- Introduction to C-style arrays).

When are constant expressions evaluated?

The compiler is only *required* to evaluate constant expressions at compile-time in contexts that require a constant expression (such as the initializer of a compile-time constant). In contexts that do not require a constant expression, the compiler may choose whether to evaluate a constant expression at compile-time or at runtime.

```
const int x { 3 + 4 }; // constant expression 3 + 4 must be evaluated at compile-time
int y { 3 + 4 };       // constant expression 3 + 4 may be evaluated at compile-time
or runtime
```

Because variable `x` has type `const int` and a constant expression initializer, it is a compile-time constant. Its initializer must be evaluated at compile-time (otherwise the value of `x` wouldn't be known at compile-time, and `x` wouldn't be a compile-time constant).

Because variable `y` does not require a constant expression initializer, the compiler can choose whether to evaluate `3 + 4` at compile-time or runtime.

Even when not required to do so, modern compilers will *usually* evaluate a constant expression at compile-time because it is an easy optimization and more performant to do so.

Partial optimization of constant subexpressions

Now consider the following example:

```
#include <iostream>

int main()
{
        std::cout << 3 + 4 << '\n';

        return 0;
}
```

The full expression `std::cout << 3 + 4 << '\n';` is a runtime expression because output can only be done at runtime. But notice that the full expression contains constant subexpression `3 + 4`.

Related content

We define the terms “full expression” and “subexpression” in lesson 1.10 -- Introduction to expressions.

Compilers have long been able to optimize constant subexpressions, even when the full expression is a runtime expression. This optimization process is called “constant folding”, and is allowed under the as-if rule.

The resulting optimized code would look like this:

```
#include <iostream>

int main()
{
	std::cout << 7 << '\n';

	return 0;
}
```

Optimization of non-constant expressions

Compilers are even capable of optimizing non-constant expressions or subexpressions in certain cases. Let’s revisit a prior example:

```
#include <iostream>

int main()
{
	int x { 7 };            // x is non-const
	std::cout << x << '\n'; // x is a non-constant subexpression

	return 0;
}
```

When `x` is initialized, the value `7` will be stored in the memory allocated for `x`. Then on the next line, the program will go out to memory again to fetch the value `7` so it can be printed.

Even though `x` is non-const, a smart compiler might realize that `x` will always evaluate to `7` in this particular program, and under the as-if rule, optimize the program to this:

```
#include <iostream>

int main()
{
	int x { 7 };
	std::cout << 7 << '\n';

	return 0;
}
```

Since x is no longer used in the program, the compiler could go one step further and optimize the program to this:

```
#include <iostream>

int main()
{
	std::cout << 7 << '\n';

	return 0;
}
```

In this version, the variable x was removed completely (because it was not used, and thus not needed). When a variable is removed from a program, we say the variable has been **optimized out** (or **optimized away**).

However, since x is non-const, such optimizations require the compiler to realize that the value of x actually doesn't change (even though it could). Whether the compiler realizes this comes down to how complex the program is and how sophisticated the compiler's optimization routines are.

Const variables are easier to optimize

Now let's consider this similar example:

```
#include <iostream>

int main()
{
	const int x { 7 }; // x is now const
	std::cout << x << '\n';

	return 0;
}
```

In this version, the compiler no longer has to infer that x won't change. Because x is now const, the compiler now has a guarantee that x can't be changed after initialization. This makes it easier for the compiler to understand that it can safely optimize x out of this program, and therefore it is more likely to do so.

Ranking variables by the likelihood of the compiler being able to optimize them:

- Compile-time constant variables (always eligible to be optimized)
- Runtime constant variables
- Non-const variables (likely optimized in simple cases only)

Key insight

Making a variable constant helps the compiler optimize.

Compile-time constant variables can also be used in constant expressions, which are more likely to be evaluated at compile-time than runtime expressions.

Both of these help make our programs faster and use less memory.

Optimization can make programs harder to debug

When the compiler optimizes a program, variables, expressions, statements, and function calls may be rearranged, altered, replaced with a value, or even removed entirely. Such changes can make it hard to debug a program effectively.

At runtime, it can be hard to debug compiled code that no longer correlates very well with the original source code. For example, if you try to watch a variable that has been optimized out, the debugger won't be able to locate the variable. If you try to step into a function that has been optimized away, the debugger will simply skip over it. So if you are debugging your code and the debugger is behaving strangely, this is the most likely reason.

At compile-time, we have little visibility and few tools to help us understand what the compiler is even doing. If a variable or expression is replaced with a value, and that value is wrong, how do we even go about debugging it? This is an ongoing challenge.

To help minimize such issues, debug builds will typically turn down (or turn off) optimizations, so that the compiled code will more closely match the source code.

Author's note

Compile-time debugging is an underdeveloped area. As of C++23, there are a number of papers under consideration for future language standards (such as this one) that (if approved) will add capabilities to the language that will help.

Quiz time

Question #1

For each statement, identify:

- Whether the initialization expression is a constant expression or non-constant expression.
- Whether the variable is non-constant, a runtime constant, or a compile-time constant

```
char a { 'q' };
```

Show Solution

```
const double b { 5.0 };
```

Show Solution

```
const int c { a * 2 }; // a defined above
```

Show Solution

```
const int d { 0 };
```

Show Solution

```
double e { b + 1.0 }; // b defined above
```

Show Solution

```
const int f { d + 2 }; // d defined above
```

Show Solution

```
const int g { getNumber() }; // getNumber returns an int by value
```

Show Solution

Extra credit:

```
const int h{};
```

Show Solution