MANCHESTER 1824
The University of Manchester

COMP26020 Programming Languages and Paradigms

Lecture 43: Compile-time Evaluation

Pavlos Petoumenos
pavlos.petoumenos@manchester.ac.uk

Compile-time evaluation

Constant expressions

- Less runtime computation
- Optimisation opportunities (-O1)

```
{
  int arr[] = {0, 1, 2, 3, …, 26}; // Fast
  return arr[21]; // Also fast
}

auto cube = [](int x) {return x * x * x;};
{
  std::vector<int> v(cube(3)); // Slower
  std::iota(v.begin(), v.end(), 0);
  return v[fibonacci(8)]; // Very slow
}
```

```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-112], 0
mov     DWORD PTR [rbp-108], 1
mov     DWORD PTR [rbp-104], 2
mov     DWORD PTR [rbp-100], 3
mov     DWORD PTR [rbp-96], 4
mov     DWORD PTR [rbp-92], 5
mov     DWORD PTR [rbp-88], 6
mov     DWORD PTR [rbp-84], 7
mov     DWORD PTR [rbp-80], 8
mov     DWORD PTR [rbp-76], 9
mov     DWORD PTR [rbp-72], 10
mov     DWORD PTR [rbp-68], 11
mov     DWORD PTR [rbp-64], 12
mov     DWORD PTR [rbp-60], 13
mov     DWORD PTR [rbp-56], 14
mov     DWORD PTR [rbp-52], 15
mov     DWORD PTR [rbp-48], 16
mov     DWORD PTR [rbp-44], 17
mov     DWORD PTR [rbp-40], 18
mov     DWORD PTR [rbp-36], 19
mov     DWORD PTR [rbp-32], 20
mov     DWORD PTR [rbp-28], 21
mov     DWORD PTR [rbp-24], 22
mov     DWORD PTR [rbp-20], 23
mov     DWORD PTR [rbp-16], 24
mov     DWORD PTR [rbp-12], 25
mov     DWORD PTR [rbp-8], 26
mov     eax, DWORD PTR [rbp-28]
pop     rbp
ret
```

Compile-time evaluation

Constant expressions

- Less runtime computation
- Optimisation opportunities (-O1)

```
{
  int arr[] = {0, 1, 2, 3, …, 26}; // Fast
  return arr[21]; // Also fast
}

auto cube = [](int x) {return x * x * x;};
{
  std::vector<int> v(cube(3)); // Slower
  std::iota(v.begin(), v.end(), 0);
  return v[fibonacci(8)]; // Very slow
}
```

```
        mov     eax, 21
        ret
```

```
fibonacci(int):
        push    rbp
        push    rbx
        sub     rsp, 8
        mov     ebx, edi
        test    edi, edi
        je      .L2
        cmp     edi, 1
        je      .L2
        lea     edi, [rdi-1]
        call    fibonacci(int)
        mov     ebp, eax
        lea     edi, [rbx-2]
        call    fibonacci(int)
        lea     ebx, [rbp+0+rax]
.L2:
        mov     eax, ebx
        add     rsp, 8
        pop     rbx
        pop     rbp
        ret
f():
        push    rbp
        push    rbx
        sub     rsp, 8
        mov     edi, 500
        call    operator new(unsigned long)
        mov     rbx, rax
        lea     rdx, [rax+500]
        mov     DWORD PTR [rax], 0
        lea     rax, [rax+4]
.L5:
        mov     DWORD PTR [rax], 0
        add     rax, 4
        cmp     rax, rdx
        jne     .L5
        mov     eax, 0
.L6:
        mov     DWORD PTR [rbx+rax*4], eax
        inc     rax
        cmp     rax, 125
        jne     .L6
        mov     edi, 8
        call    fibonacci(int)
        cdqe
        mov     ebp, DWORD PTR [rbx+rax*4]
        mov     esi, 500
        mov     rdi, rbx
        call    operator delete(void*, unsigned long)
        mov     eax, ebp
        add     rsp, 8
        pop     rbx
        pop     rbp
        ret
```

constexpr

Type modifier for variables and return types

constexpr variables

- const
- Initialisation value constructible at compile-time

constexpr functions

- Developer guarantees that it returns a compile-time constant when fed constants

Compile-time evaluation

Constant expressions

- Less runtime computation
- Optimisation opportunities (-O1)

```
constexpr int fibonacci(int num)
{
    if (num == 0)
        return 0;
    if (num == 1)
        return 1;
    return fibonacci(num - 1) + fibonacci(num - 2);
}
```

Compile-time evaluation

Constant expressions

- Less runtime computation
- Optimisation opportunities (-O1)

```
{
  int arr[] = {0, 1, 2, 3, …, 26}; // Fast
  return arr[21]; // Also fast
}

auto cube = [](int x) {return x * x * x;};
{
  std::vector<int> v(cube(3)); // Slower
  std::iota(v.begin(), v.end(), 0);
  return v[fibonacci(8)]; // Very slow
}
```

```
        mov     eax, 21
        ret
```

```
fibonacci(int):
        push    rbp
        push    rbx
        sub     rsp, 8
        mov     ebx, edi
        test    edi, edi
        je      .L2
        cmp     edi, 1
        je      .L2
        lea     edi, [rdi-1]
        call    fibonacci(int)
        mov     ebp, eax
        lea     edi, [rbx-2]
        call    fibonacci(int)
        lea     ebx, [rbp+0+rax]
.L2:
        mov     eax, ebx
        add     rsp, 8
        pop     rbx
        pop     rbp
        ret
f():
        push    rbp
        push    rbx
        sub     rsp, 8
        mov     edi, 500
        call    operator new(unsigned long)
        mov     rbx, rax
        lea     rdx, [rax+500]
        mov     DWORD PTR [rax], 0
        lea     rax, [rax+4]
.L5:
        mov     DWORD PTR [rax], 0
        add     rax, 4
        cmp     rax, rdx
        jne     .L5
        mov     eax, 0
.L6:
        mov     DWORD PTR [rbx+rax*4], eax
        inc     rax
        cmp     rax, 125
        jne     .L6
        mov     edi, 8
        call    fibonacci(int)
        cdqe
        mov     ebp, DWORD PTR [rbx+rax*4]
        mov     esi, 500
        mov     rdi, rbx
        call    operator delete(void*, unsigned long)
        mov     eax, ebp
        add     rsp, 8
        pop     rbx
        pop     rbp
        ret
```

Compile-time evaluation

Constant expressions

- Less runtime computation
- Optimisation opportunities (-O1)

```
{
  int arr[] = {0, 1, 2, 3, …, 26}; // Fast
  return arr[21]; // Also fast
}

auto cube = [](int x) {return x * x * x;};
{
  std::vector<int> v(cube(3)); // Slower
  std::iota(v.begin(), v.end(), 0);
  return v[fibonacci(8)]; // Very slow
}
```

```
        mov     eax, 21
        ret
```

```
f():
        push    rbx
        mov     edi, 500
        call    operator new(unsigned long)
        mov     rdi, rax
        lea     rax, [rax+500]
        mov     DWORD PTR [rdi], 0
        lea     rdx, [rdi+4]
.L2:
        mov     DWORD PTR [rdx], 0
        add     rdx, 4
        cmp     rdx, rax
        jne     .L2
        mov     edx, 0
.L3:
        mov     DWORD PTR [rdi+rdx*4], edx
        inc     rdx
        cmp     rdx, 125
        jne     .L3
        mov     ebx, DWORD PTR [rdi+84]
        mov     esi, 500
        call    operator delete(void*, unsigned long)
        mov     eax, ebx
        pop     rbx
        ret
```

constexpr

Variable type or return and argument types must be literal types

- Numbers, pointers, enums, arrays of literals
- Classes with trivial/constexpr destructor, constexpr constructor, literal data members

Non-virtual functions

Functions cannot define non-literal variables

Recap

constexpr

- Value is constant and can be initialised at compile time
- Function returns constant values for constant inputs

Compiler calculates parts of the program at compile time

Less code, faster execution

Up Next

"...a much smaller and clearer language..."

Core Guidelines

Tools