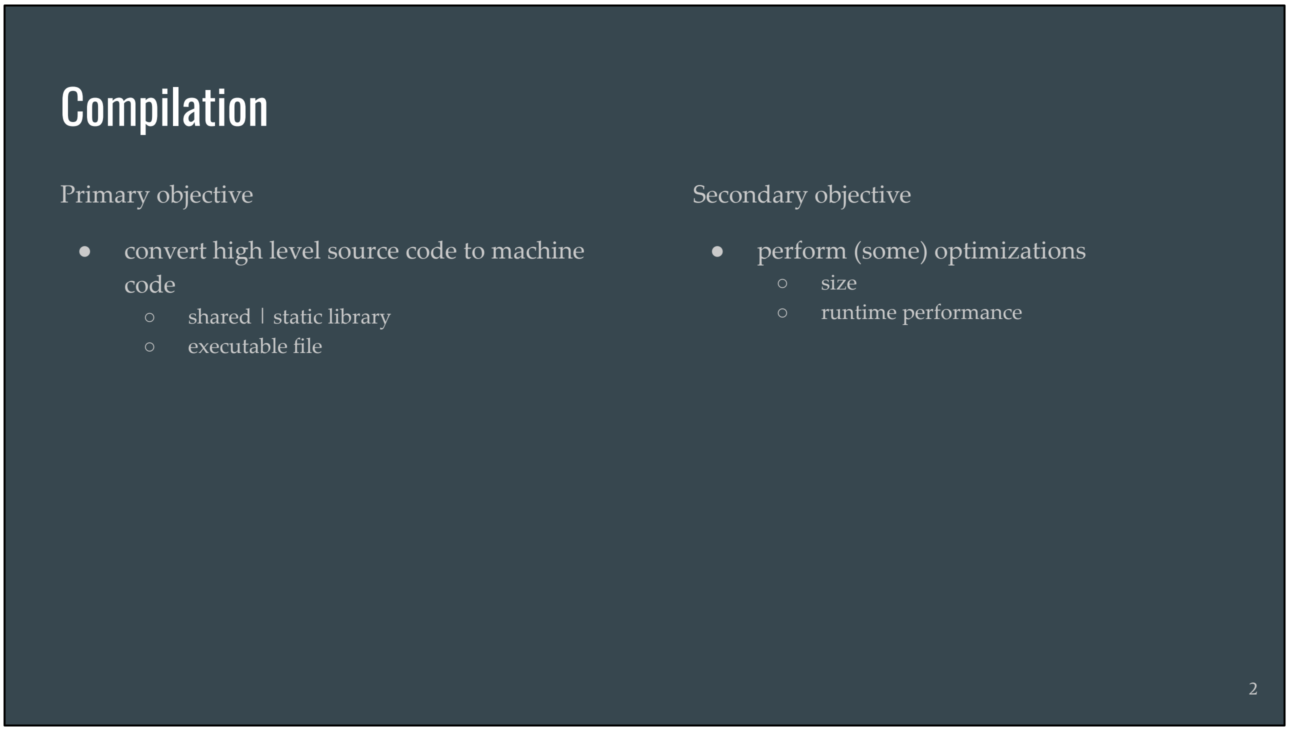
Compiler Optimizations

•••

David Střelák

SitSem, 9.9.2021



Compilation

Primary objective

- convert high level source code to machine code
  - shared | static library
  - executable file

Secondary objective

- perform (some) optimizations
  - size
  - runtime performance

`We will talk about C++ compilation, and we will demonstrate compilation using GCC compiler

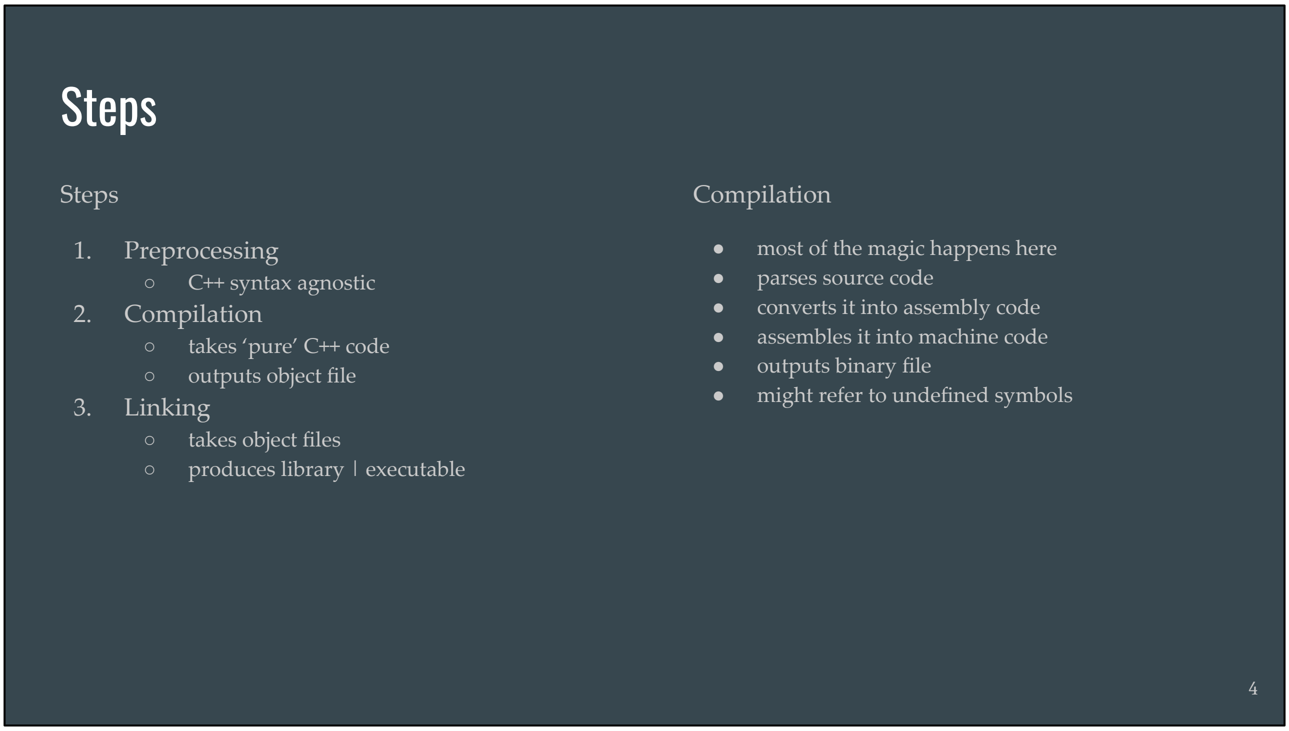
Steps

Steps

1. Preprocessing
   - C++ syntax agnostic
2. Compilation
   - takes ‘pure’ C++ code
   - outputs object file
3. Linking
   - takes object files
   - produces library | executable

Preprocessing

- macro definitions (#define)
- conditional inclusions (#ifdef)
- error directive (#error)
- source file inclusion (#include)



Steps

Steps

1. Preprocessing
    - C++ syntax agnostic
2. Compilation
    - takes ‘pure’ C++ code
    - outputs object file
3. Linking
    - takes object files
    - produces library | executable

Compilation

- most of the magic happens here
- parses source code
- converts it into assembly code
- assembles it into machine code
- outputs binary file
- might refer to undefined symbols

- optimization is done on level of basic blocks
    - piece of code with single entry, single exit and no branches between

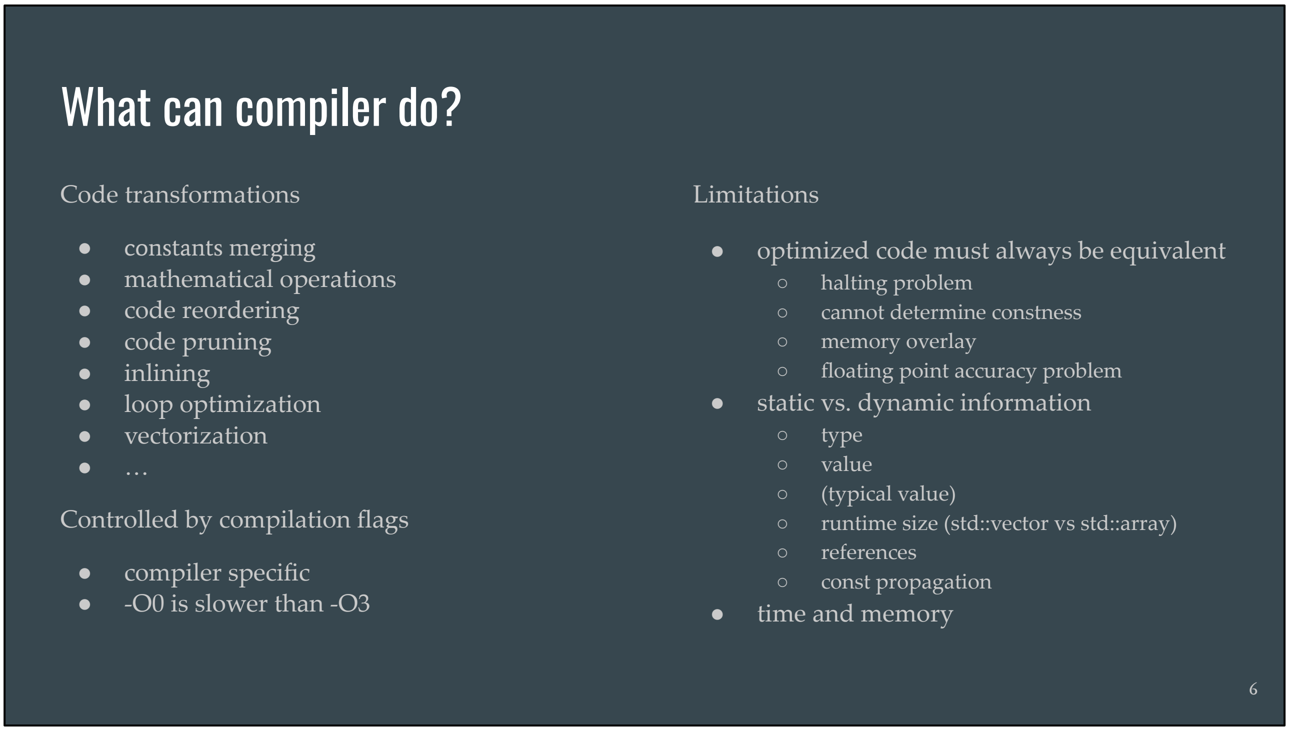
Steps

Steps

1. Preprocessing
   - C++ syntax agnostic
2. Compilation
   - takes ‘pure’ C++ code
   - outputs object file
3. Linking
   - takes object files
   - produces library | executable

Linking

- some magic can happen here
- can create shared library or executable
- links object files by replacing the references with the correct addresses



What can compiler do?

Code transformations

- constants merging
- mathematical operations
- code reordering
- code pruning
- inlining
- loop optimization
- vectorization
- …

Controlled by compilation flags

- compiler specific
- -O0 is slower than -O3

Limitations

- optimized code must always be equivalent
  - halting problem
  - cannot determine constness
  - memory overlay
  - floating point accuracy problem
- static vs. dynamic information
  - type
  - value
  - (typical value)
  - runtime size (std::vector vs std::array)
  - references
  - const propagation
- time and memory

Halting problem

- we cannot solve all problems at compile time, because we might not know if the program will finish

Constness

- all data accessed via pointer or reference or global variables might change
- in praxis, C++ assumes single-thread execution
  - might case nasty surprises
- One thing is propagation of const value, another one is ‘const’ keyword (which is mostly for programmers)

Memory overlay

- say that we want to sum two arrays and store result to the first one. Unless we know that the second one is not part of the first one, we might have a problem

Floating point accuracy

- a * a * a * a is not the same as (a * a) * (a * a) due to final representation

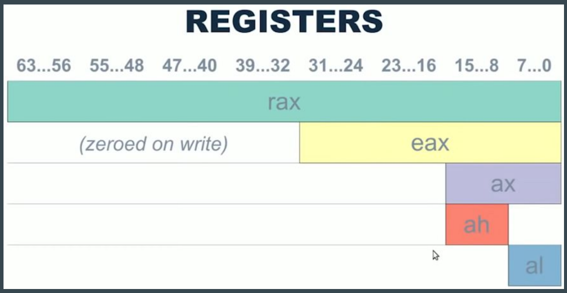
x86 Assembly 101

Registers

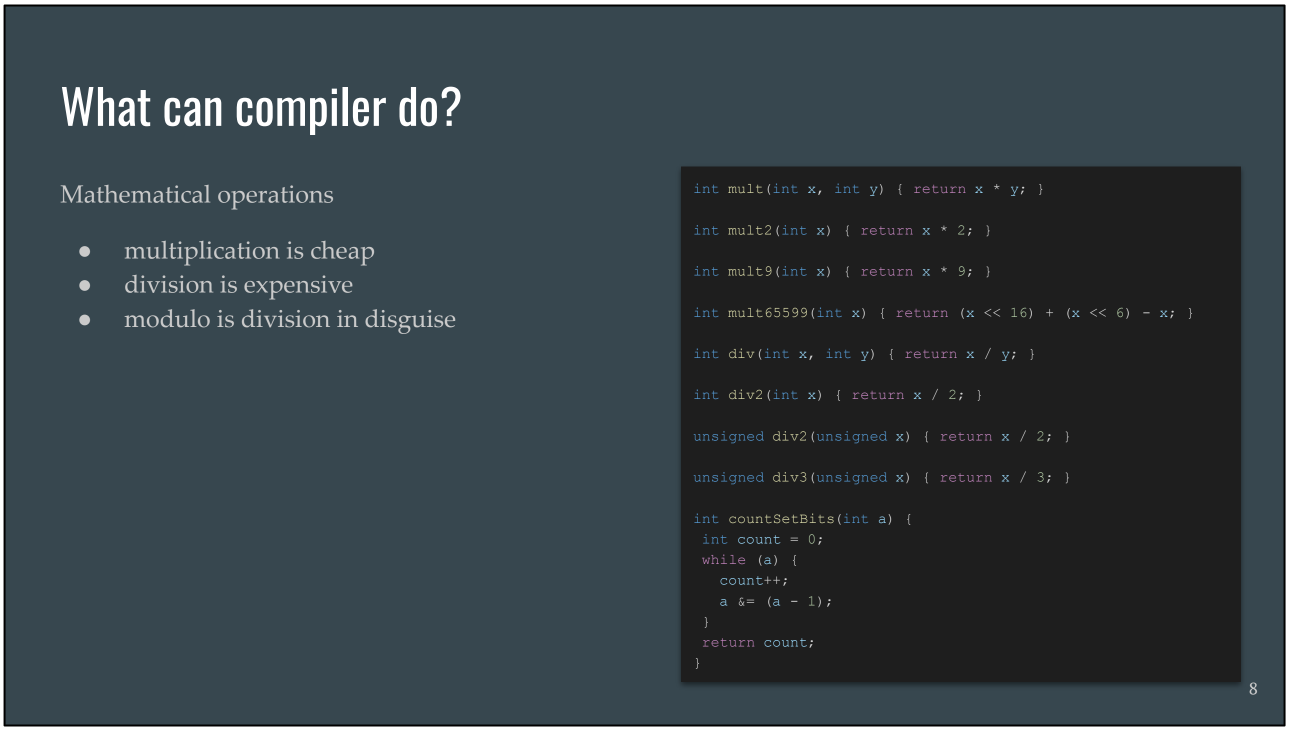
- 64b values, 16 in total
- rax, rbx, rcx, rdx, rsp, rbp, rsi, rdi, r8-r15
- rdi, rsi, rdx … are arguments
- rax is return value from function

Instructions (Intel asm syntax)

- op
- op dest
- op dest, src
- op dest, src1, src2



Notice that processor uses register renaming, there are many more physical registers.



What can compiler do?

Mathematical operations

- multiplication is cheap
- division is expensive
- modulo is division in disguise

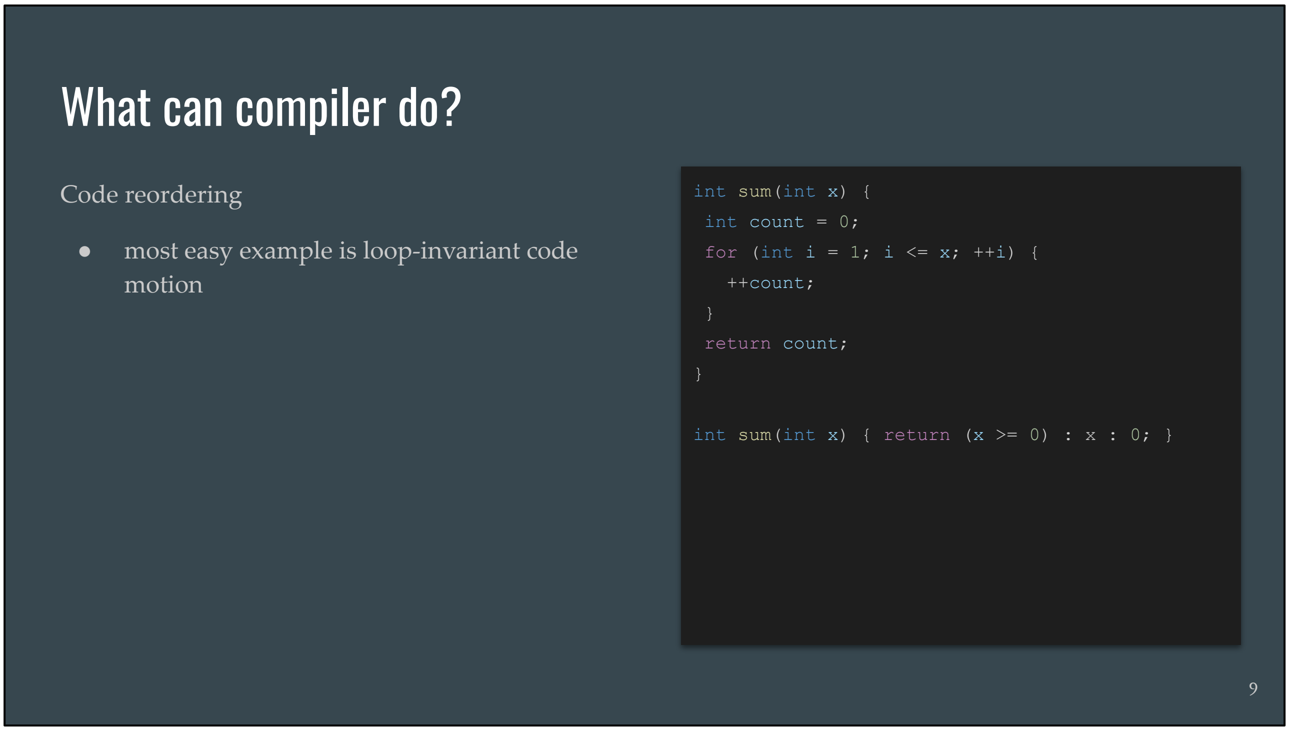
```
int mult(int x, int y) { return x * y; }

int mult2(int x) { return x * 2; }

int mult9(int x) { return x * 9; }

int mult65599(int x) { return (x << 16) + (x << 6) - x; }

int div(int x, int y) { return x / y; }

int div2(int x) { return x / 2; }

unsigned div2(unsigned x) { return x / 2; }

unsigned div3(unsigned x) { return x / 3; }

int countSetBits(int a) {
 int count = 0;
 while (a) {
   count++;
   a &= (a - 1);
 }
 return count;
}
```

math_ops.cpp
gcc 10.3 -O1 -march=haswell

- mult by 2, 4, 8 is done by load effective address (because it’s faster than shift and you don’t need to bother with sign)
- haswell 32 bit multiply in 4 cycles (one cycle is ⅓ of a nanosecond)
  - multiple multipliers per core
  - they are pipelined (you can start one multiplication each cycle)
- haswell 32 bit divide takes 22-29 cycles
  - and there’s one divider per core
  - not pipelined
- modulus and division are the same operation
  - slow hash map

gcc 10.3 -O3 -march=haswell

- bitcount special operation



What can compiler do?

Code reordering

- most easy example is loop-invariant code motion

```
int sum(int x) {
 int count = 0;
 for (int i = 1; i <= x; ++i) {
   ++count;
 }
 return count;
}

int sum(int x) { return (x >= 0) : x : 0; }
```

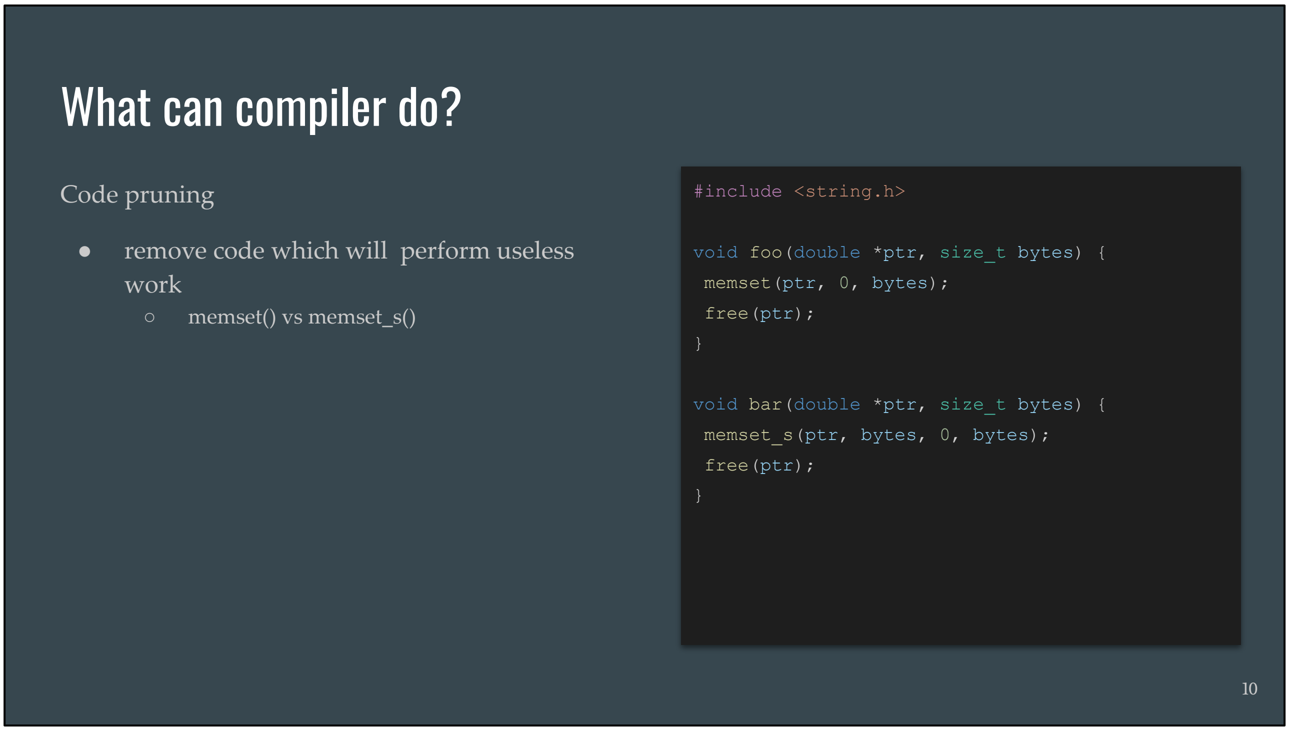
9

code_reordering.cpp
gcc 10.3 -O1

```
sum(int):
        test    edi, edi // edi is 32b rdi ;  check if x is 0
        jle     .L4
        lea     edx, [rdi+1] // set edx to x
        mov     eax, 1 // set result to 1 (holds our i)
.L3:
        add     eax, 1 // add 1 to i
        cmp     edx, eax // if i equals x
        jne     .L3
.L2:
        mov     eax, edi // set result to edi, that is to x
        ret
.L4:
        mov     edi, 0
        jmp     .L2
```

gcc 10.3 -O2

- creates expected assembly



What can compiler do?

Code pruning

- remove code which will perform useless work
  - memset() vs memset_s()

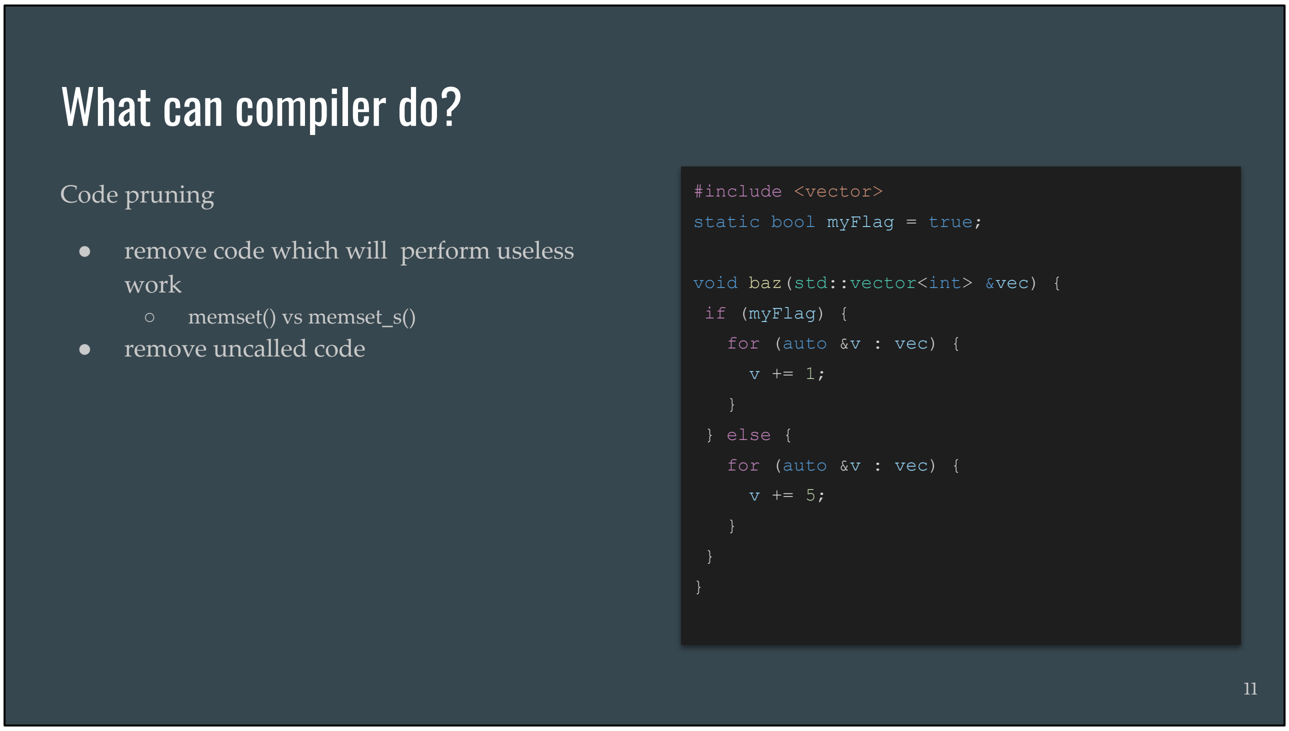
```
#include <string.h>

void foo(double *ptr, size_t bytes) {
 memset(ptr, 0, bytes);
 free(ptr);
}

void bar(double *ptr, size_t bytes) {
 memset_s(ptr, bytes, 0, bytes);
 free(ptr);
}
```

Memset might be removed if it does not perform any useful operation. In this case, we will free the memory after wiping it.

code_pruning.c
C compiler gcc 10.3 -O1

```
foo:
        sub     rsp, 8 // probably some stack alignment magic
        call    free
        add     rsp, 8
        ret
```

gcc 10.3 -O2 is cleaner



What can compiler do?

Code pruning

- remove code which will perform useless work
    - memset() vs memset_s()
- remove uncalled code

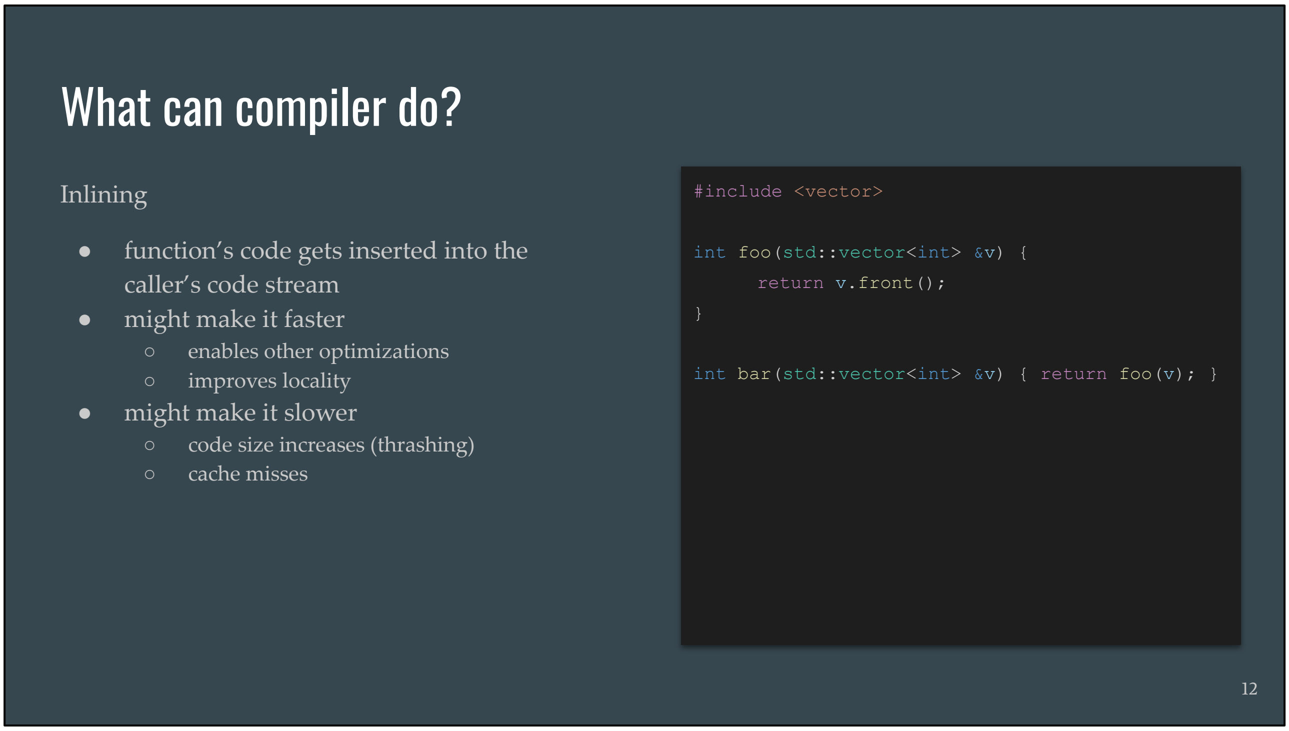
```
#include <vector>
static bool myFlag = true;

void baz(std::vector<int> &vec) {
 if (myFlag) {
   for (auto &v : vec) {
     v += 1;
   }
 } else {
   for (auto &v : vec) {
     v += 5;
   }
 }
}
```

Uncalled code might be impossible to detect

- we will talk later how we can change that
- we're back to the halting problem

code_pruning.cpp
C++ compiler gcc 10.3 -O1

because myFlag is static, it is not visible to another compilation unit [internal linkage], i.e. compiler is sure that it will not change
changing static to const works the same



What can compiler do?

Inlining

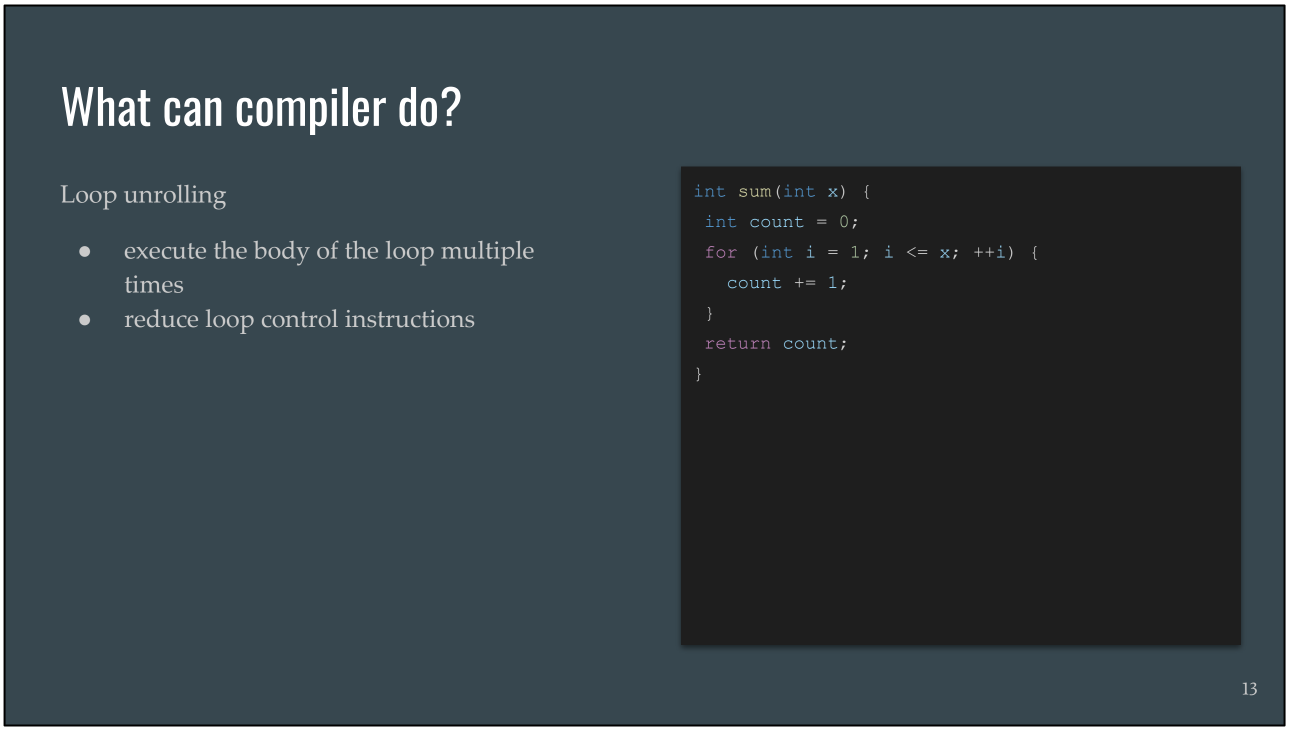
- function’s code gets inserted into the caller’s code stream
- might make it faster
  - enables other optimizations
  - improves locality
- might make it slower
  - code size increases (thrashing)
  - cache misses

```
#include <vector>

int foo(std::vector<int> &v) {
      return v.front();
}

int bar(std::vector<int> &v) { return foo(v); }
```

inline.cpp

gcc 10.3 -O1

notice that both foo() and bar() are equivalent



What can compiler do?

Loop unrolling

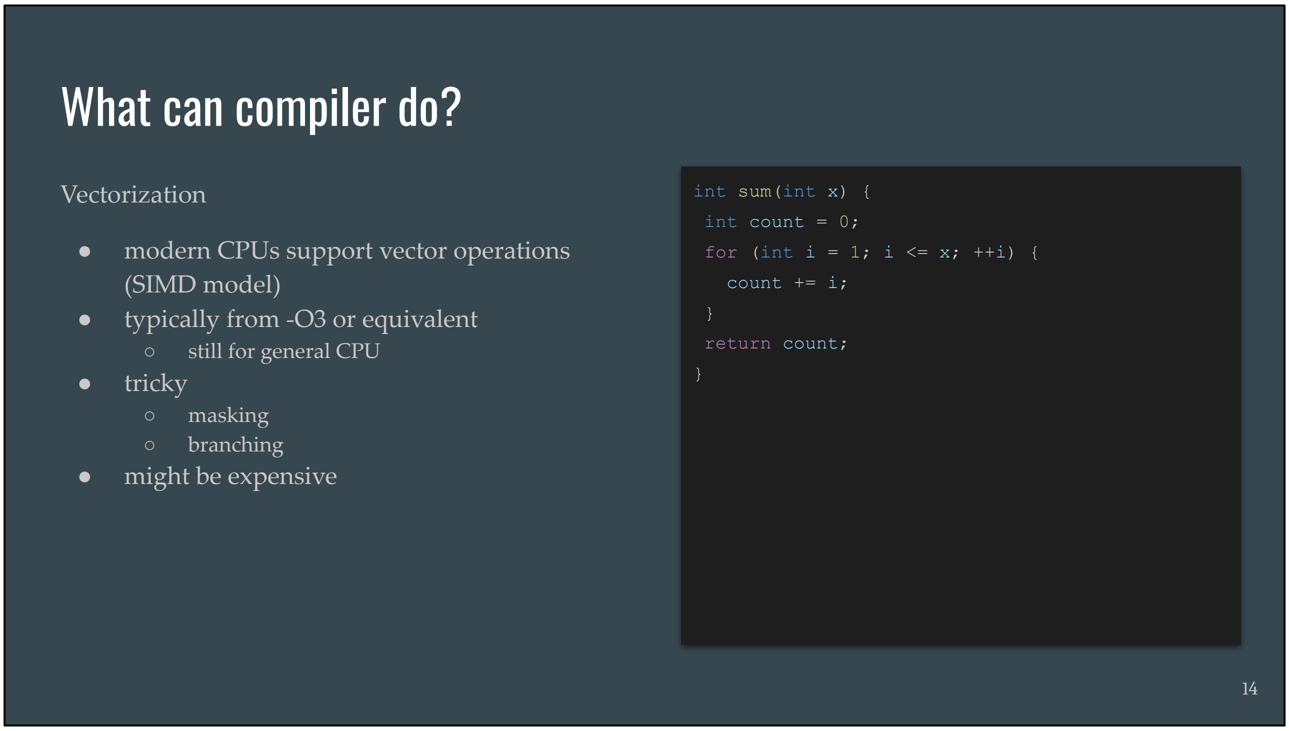
- execute the body of the loop multiple times
- reduce loop control instructions

```
int sum(int x) {
 int count = 0;
 for (int i = 1; i <= x; ++i) {
   count += 1;
 }
 return count;
}
```

13

at the end of a loop, there are two possibilities. Either you go back to the top, or continue on. The processor makes an educated guess on which is going to happen. If it gets it right, everything is good. If not, it has to flush the pipeline and stall for a bit while it prepares for taking the other branch.
Unrolling a loop eliminates branches and the potential for those stalls, especially in cases where the odds are against a guess.
Processors are loading the instructions from the executable into the instruction cache. Since its cache, it holds a limited amount of instructions which can be accessed quickly.

loop_unrolling.cpp
gcc 10.3 -O1
 -O1 -funroll-loops



What can compiler do?

Vectorization

- modern CPUs support vector operations (SIMD model)
- typically from -O3 or equivalent
  - still for general CPU
- tricky
  - masking
  - branching
- might be expensive

```
int sum(int x) {
 int count = 0;
 for (int i = 1; i <= x; ++i) {
   count += i;
 }
 return count;
}
```

vectorization.cpp
gcc 10.3 -O1
gcc 10.3 -O3

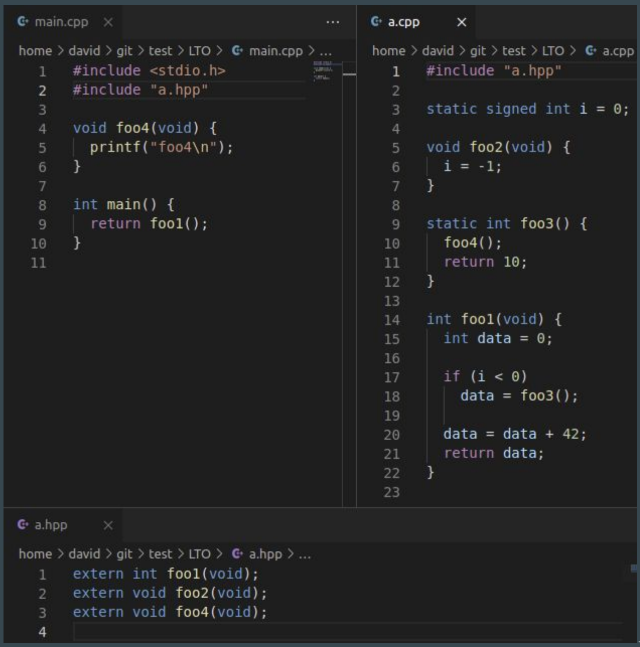
notice the xmmX registers - streaming simd extension: 128b registers, since version two able to hold four int numbers

.long means long int (i.e. 32b), because standard defines int as at least 16b: https://en.cppreference.com/w/cpp/language/types

-O3 -march=haswell
notice that now it uses ymmX registers - those are from haswell's advanced vector extension 2 instruction set architecture, and we're using 256b registers now

Link time optimization

- compilation is done at source file level
  - and thus some optimizations cannot be applied
- during linking, more magic can happen


main.cpp

home > david > git > test > LTO > main.cpp > ...

```
#include <stdio.h>
#include "a.hpp"

void foo4(void) {
  printf("foo4\n");
}

int main() {
  return foo1();
}
```

a.cpp

home > david > git > test > LTO > a.cpp

```
#include "a.hpp"

static signed int i = 0;

void foo2(void) {
  i = -1;
}

static int foo3() {
  foo4();
  return 10;
}

int foo1(void) {
  int data = 0;

  if (i < 0)
    data = foo3();

  data = data + 42;
  return data;
}
```

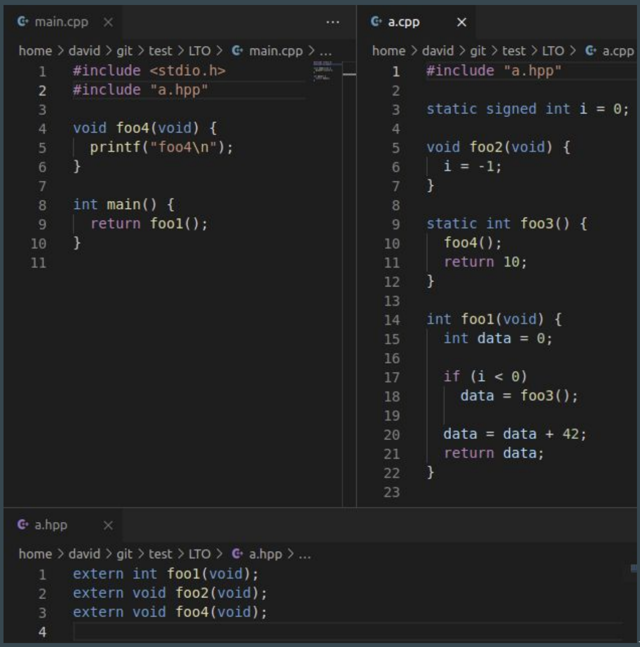
a.hpp

home > david > git > test > LTO > a.hpp > ...

```
extern int foo1(void);
extern void foo2(void);
extern void foo4(void);
```

No LTO

- foo2() cannot be removed
  - because somebody might still need that symbol during linking
- foo3() can be inlined and removed
  - because it's static
- as a result, i can change
  - and foo1() cannot be simplified



main.cpp

home > david > git > test > LTO > main.cpp > ...

```
#include <stdio.h>
#include "a.hpp"

void foo4(void) {
  printf("foo4\n");
}

int main() {
  return foo1();
}
```

a.cpp

home > david > git > test > LTO > a.cpp

```
#include "a.hpp"

static signed int i = 0;

void foo2(void) {
  i = -1;
}

static int foo3() {
  foo4();
  return 10;
}

int foo1(void) {
  int data = 0;

  if (i < 0)
    data = foo3();

  data = data + 42;
  return data;
}
```

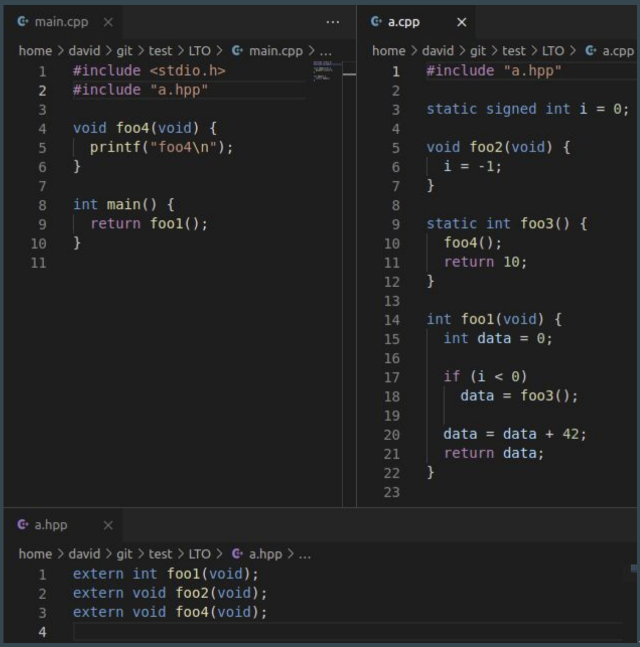
a.hpp

home > david > git > test > LTO > a.hpp > ...

```
extern int foo1(void);
extern void foo2(void);
extern void foo4(void);
```

LTO

- foo2() is not used
  - is removed
- i cannot be changed
- foo3() cannot be called
  - it is removed
- foo1() is simplified
  - and inlined
  - and removed
- main() returns 42



main.cpp

home > david > git > test > LTO > main.cpp > ...

```cpp
#include <stdio.h>
#include "a.hpp"

void foo4(void) {
  printf("foo4\n");
}

int main() {
  return foo1();
}
```

a.cpp

home > david > git > test > LTO > a.cpp

```cpp
#include "a.hpp"

static signed int i = 0;

void foo2(void) {
  i = -1;
}

static int foo3() {
  foo4();
  return 10;
}

int foo1(void) {
  int data = 0;

  if (i < 0)
    data = foo3();

  data = data + 42;
  return data;
}
```

a.hpp

home > david > git > test > LTO > a.hpp > ...

```cpp
extern int foo1(void);
extern void foo2(void);
extern void foo4(void);
```

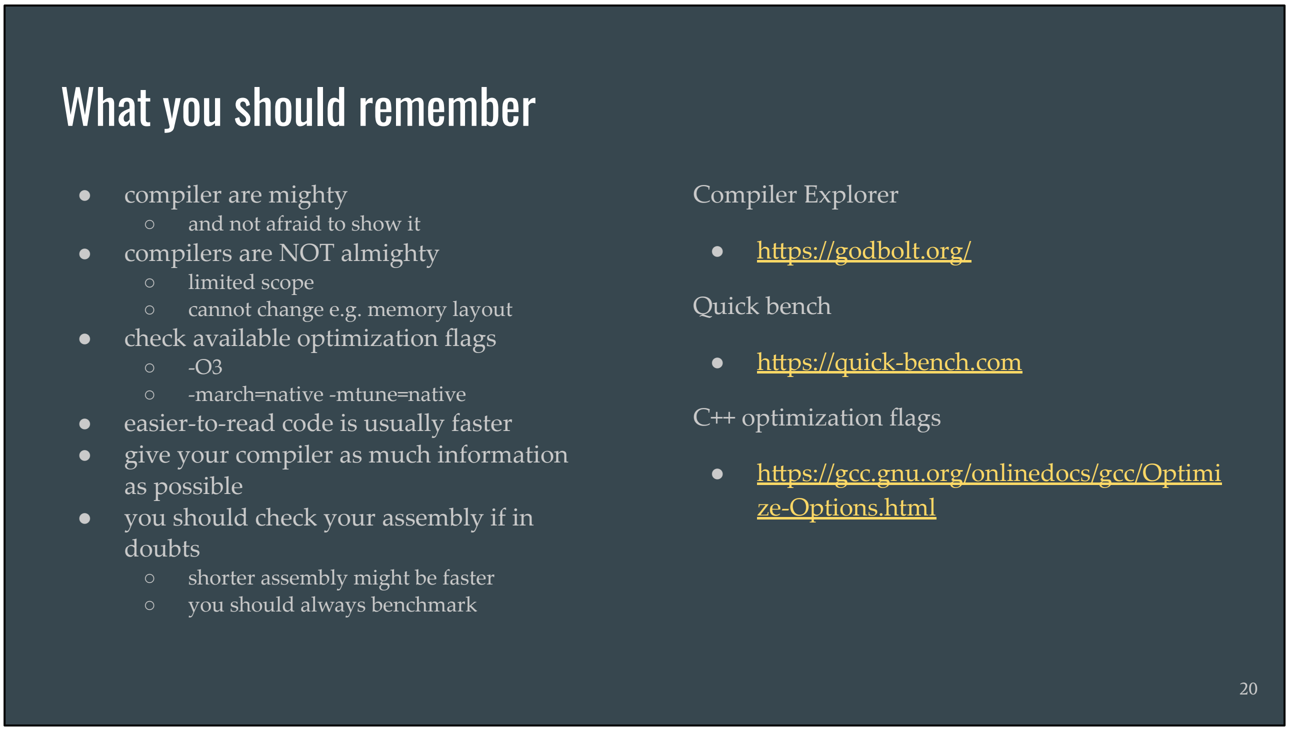
Example of limitations

Pass by reference

```
#include <vector>

void foo(std::vector<int> &vec, bool b) {
 for (auto &v : vec) {
   if (b) {
     v += 1;
   }
 }
}

void bar(std::vector<int> &vec, const bool &b) {
 for (auto &v : vec) {
   if (b) {
     v += 1;
   }
 }
}
```

What we didn’t talk about

- profile guided optimizations
  - you compile your code, run it, and based on the profile, you recompile the code
- compiler intrinsics
  - they might lead to faster code
  - but code is harder to read and less portable
- const(expr)
  - some things might be precomputed at the runtime
- unwanted optimizations
  - problem with security
- vectorization of more complex code
- multithreaded code



What you should remember

- compiler are mighty
  - and not afraid to show it
- compilers are NOT almighty
  - limited scope
  - cannot change e.g. memory layout
- check available optimization flags
  - -O3
  - -march=native -mtune=native
- easier-to-read code is usually faster
- give your compiler as much information as possible
- you should check your assembly if in doubts
  - shorter assembly might be faster
  - you should always benchmark

Compiler Explorer

- https://godbolt.org/

Quick bench

- https://quick-bench.com

C++ optimization flags

- https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Resources

https://stackoverflow.com/a/6264256

http://www.cplusplus.com/doc/tutorial/preprocessor/

https://en.wikipedia.org/wiki/Floating-point_arithmetic#Accuracy_problems

https://llvm.org/docs/LinkTimeOptimization.html

https://isocpp.org/wiki/faq/inline-functions

https://wiki.sei.cmu.edu/confluence/display/c/MSC06-C.+Beware+of+compiler+optimizations

https://youtu.be/bSkpMdDe4g4

http://6.s081.scripts.mit.edu/sp18/x86-64-architecture-guide.html

Q&A