

Core C++
2023
June 5-7



Running Away From Computation - An Introduction



Social: [@MichaelShah](#)
Web: [mshah.io](#)
Courses: [courses.mshah.io](#)
YouTube:
[www.youtube.com/c/MikeShah](#)

12:20-13:20, Tue, 6th June 2023

60 minutes | Introductory Audience

Please do not redistribute slides without prior permission.

Your Tour Guide for Today

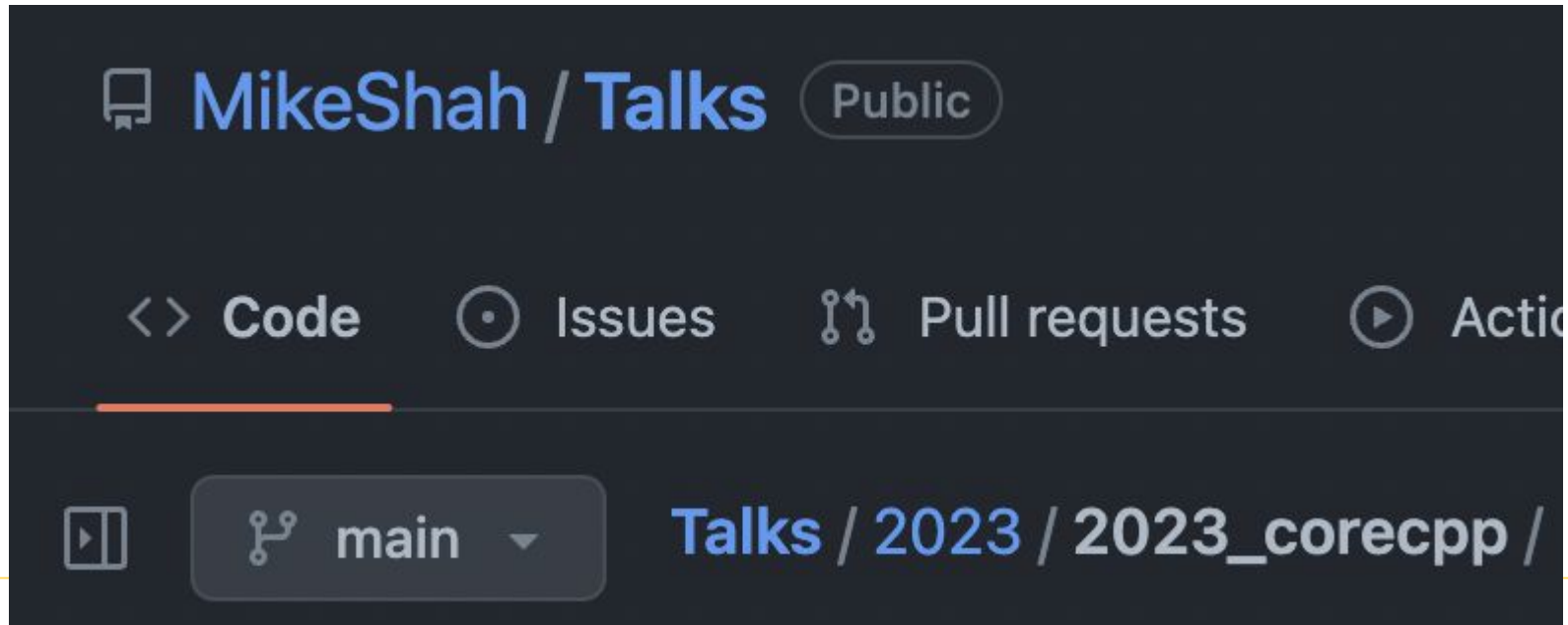
by Mike Shah

- **Associate Teaching Professor** at Northeastern University in Boston, Massachusetts.
 - I teach courses in computer systems, computer graphics, and game engine development.
 - My **research** in program analysis is related to **performance** building static/dynamic analysis and software visualization tools.
- I do **consulting** and **technical training** on modern C++, DLang, Concurrency, OpenGL, and Vulkan projects
 - (Usually graphics or games related)
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of **computer graphics**, visualization, concurrency, and parallelism.
- Contact information and more on: www.mshah.io
- More online training at courses.mshah.io



Code for the talk

- Located here: https://github.com/MikeShah/Talks/tree/main/2023/2023_corecpp



The abstract that you read and enticed
you to join me is here!

Abstract

One of the fun and motivating reasons to use the C++ programming language is the ability to optimize code. One of the best ways to optimize code is to avoid any computation in the first place! In this talk, we are going to learn how to approach the C++ programming language, thinking about compile-time computation (e.g. `constexpr`, `static_assert`, and template meta-programming) and some other tricks to avoid computation at run-time (e.g. short-circuit evaluation, caching, and lazy evaluation). In this talk, participants will learn how these techniques improve performance (with measurements using the `perf` profiler), as well as learn how these techniques also make C++ a safer programming language. This is a beginner level talk

Goal(s) for today

Core C++
2023
June 5-7

So what is this
talk about?

Running Away From Computation - An Introduction



Social: [@MichaelShah](#)
Web: [mshah.io](#)
Courses: [courses.mshah.io](#)
YouTube:
[www.youtube.com/c/MikeShah](#)

12:20-13:20, Tue, 6th June 2022

60 minutes | Introductory Audience

Core C++
2023
June 5-7

Running?



Running Away From Computation - An Introduction



Social: [@MichaelShah](#)
Web: [mshah.io](#)
Courses: [courses.mshah.io](#)
YouTube:
[www.youtube.com/c/MikeShah](#)

12:20-13:20, Tue, 6th June 2022

60 minutes | Introductory Audience

Core C++
2023
June 5-7

Running
Computers
Faster?!

Running Away From Computation - An Introduction



Social: [@MichaelShah](#)
Web: [mshah.io](#)
Courses: [courses.mshah.io](#)
YouTube:
[www.youtube.com/c/MikeShah](#)

12:20-13:20, Tue, 6th June 2022

60 minutes | Introductory Audience

Core C++
2023
June 5-7

What do I mean
“running away”?

Running Away From Computation - An Introduction



Social: [@MichaelShah](#)
Web: [mshah.io](#)
Courses: [courses.mshah.io](#)
YouTube:
[www.youtube.com/c/MikeShah](#)

12:20-13:20, Tue, 6th June 2022

60 minutes | Introductory Audience

What you're going to learn today

- We are going to learn about a fundamental computer science trade-off that C++ offers us
 - Intrigued? Let's continue!
- Audience:
 - Probably more beginner level/student, but perhaps beneficial for mid-level folks to think about.



Pretend these seats are filled :)

<https://pixnio.com/free-images/2017/03/11/2017-03-11-16-47-11-550x413.jpg>

Warning -- this talk does include occasional performance numbers

Please validate on your architecture on data sets relevant to your program

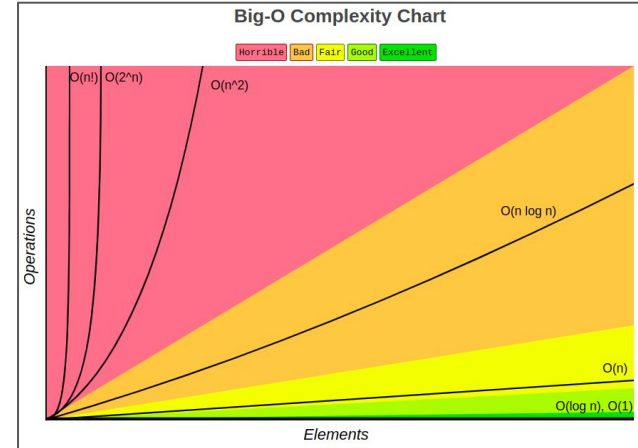
E	Rated 'E' For Everyone!
	(Yup, let's just do our best to make C++ fun for everyone involved)

Question to Audience:

What's the most fundamental trade-off in computer science (in your opinion)?

Trade-off: Time versus Space!

- If you've read an algorithms book, you probably have encountered this topic!
 - Space: meaning memory allocated 'somewhere'
 - Time: meaning, the amount of time to execute a series of statements
- We usually use Big-Oh notation (e.g. $O(n)$, $O(n^2)$, etc.) to describe the space or time of an algorithm or data structure as a function of the number of inputs.
 - That is, 'n' is the number of inputs or size of a collection.
 - Big-O complexity is a tool that might help us estimate or choose an algorithm
 - (In practice we have to measure)
- Let's look at an example however where there is a clear trade-off between space and time.



<https://www.bigocheatsheet.com/>

Space Versus Time Trade-off

A tale of two singly linked lists

Linked List Experiment 1 (1/3)

- I'll show you a little experiment now comparing two singly linked lists
 - The first linked list is a minimal an implementation of a linked list 'LL1'
 - There's a mHead to store the first node
 - Append takes an integer, and constructs a new node that is searched for at the end of the linked list.

```
4 struct Node{
5     Node* next;
6     int data;
7 };
8
9 struct LL1{
10     Node* mHead{nullptr};
11
12     void Append(int _data){
13         Node* newNode = new Node;
14         newNode->data = _data;
15         newNode->next = nullptr;
16
17         if(nullptr == mHead){
18             mHead = newNode;
19         }else{
20             Node* iter = mHead;
21
22             while(nullptr != iter->next){
23                 iter=iter->next;
24             }
25             iter->next = newNode;
26         }
27     }
28
29     void Print(){
30         Node* iter = mHead;
31         while(nullptr != iter){
32             std::cout << iter->data << "\n";
33             iter=iter->next;
34         }
35     }
36 };
```


Linked List Experiment 1 (2/3)

- I'll show you a little experiment now comparing two singly linked lists
 - The first linked list is a minimal an implementation of a linked list 'LL1'
 - There's a mHead to store the first node
 - Append takes an integer, and constructs a new node that is searched for at the end of the linked list.

```
4 struct Node{
5     Node* next;
6     int data;
7 };
8
9 struct LL1{
10     Node* mHead{nullptr};
11
12     void Append(int _data){
13         Node* newNode = new Node;
14         newNode->data = _data;
15         newNode->next = nullptr;
16
17         if(nullptr == mHead){
18             mHead = newNode;
19         }else{
20             Node* iter = mHead;
21
22             while(nullptr != iter->next){
23                 iter=iter->next;
24             }
25             iter->next = newNode;
26         }
27     }
28
29     void Print(){
30         Node* iter = mHead;
31         while(nullptr != iter){
32             std::cout << iter->data << "\n";
33             iter=iter->next;
34         }
35     }
36 };
```

Linked List Experiment 1 (3/3)

- I'll show you a little experiment now comparing two singly linked lists
 - The first linked list is a minimal an implementation of a linked list 'LL1'
 - There's a mHead to store the first node
 - Append takes an integer, and constructs a new node that is searched for at the end of the linked list.

```
4 struct Node{
5     Node* next;
6     int data;
7 };
8
9 struct LL1{
10     Node* mHead{nullptr};
11
12     void Append(int _data){
13         Node* newNode = new Node;
14         newNode->data = _data;
15         newNode->next = nullptr;
16
17         if(nullptr == mHead){
18             mHead = newNode;
19         }else{
20             Node* iter = mHead;
21
22             while(nullptr != iter->next){
23                 iter=iter->next;
24             }
25             iter->next = newNode;
26         }
27     }
28
29     void Print(){
30         Node* iter = mHead;
31         while(nullptr != iter){
32             std::cout << iter->data << "\n";
33             iter=iter->next;
34         }
35     }
36 };
```

Linked List Experiment 2 (1/3)

- Here is a second implementation of a linked list 'LL2'
 - We are going to pay some storage and add a 'mTail' Node that keeps track of the end of the linked list
 - Observe that our Append() function removes the need for a loop
 - (It also becomes simpler to implement!)

```
9 struct LL2{
10     Node* mHead{nullptr};
11     Node* mTail{nullptr};
12
13     void Append(int _data){
14         Node* newNode = new Node;
15         newNode->data = _data;
16         newNode->next = nullptr;
17
18         if(nullptr == mHead){
19             mHead = newNode;
20             mTail = mHead;
21         }else{
22             mTail->next = newNode;
23             mTail = newNode;
24         }
25     }
26
27     void Print(){
28         Node* iter = mHead;
29         while(nullptr != iter){
30             std::cout << iter->data << "\n";
31             iter=iter->next;
32         }
33     }
34 };
```

Linked List Experiment 2 (2/3)

- Here is a second implementation of a linked list 'LL2'
 - We are going to pay some storage and add a 'mTail' Node that keeps track of the end of the linked list
 - Observe that our Append() function removes the need for a loop
 - (It also becomes simpler to implement!)

```
9 struct LL2{
10     Node* mHead{nullptr};
11     Node* mTail{nullptr};
12
13     void Append(int _data){
14         Node* newNode = new Node;
15         newNode->data = _data;
16         newNode->next = nullptr;
17
18         if(nullptr == mHead){
19             mHead = newNode;
20             mTail = mHead;
21         }else{
22             mTail->next = newNode;
23             mTail = newNode;
24         }
25     }
26
27     void Print(){
28         Node* iter = mHead;
29         while(nullptr != iter){
30             std::cout << iter->data << "\n";
31             iter=iter->next;
32         }
33     }
34 };
```

Linked List Experiment 2 (3/3)

- Here is a second implementation of a linked list 'LL2'
 - We are going to pay some storage and add a 'mTail' Node that keeps track of the end of the linked list
 - Observe that our Append() function removes the need for a loop
 - (It also becomes simpler to implement!)

```
9 struct LL2{
10     Node* mHead{nullptr};
11     Node* mTail{nullptr};
12
13     void Append(int _data){
14         Node* newNode = new Node;
15         newNode->data = _data;
16         newNode->next = nullptr;
17
18         if(nullptr == mHead){
19             mHead = newNode;
20             mTail = mHead;
21         }else{
22             mTail->next = newNode;
23             mTail = newNode;
24         }
25     }
26
27     void Print(){
28         Node* iter = mHead;
29         while(nullptr != iter){
30             std::cout << iter->data << "\n";
31             iter=iter->next;
32         }
33     }
34 };
```

Question to Audience:

- Looking at the Big-O of each linked list -- which do you expect to be faster?
 - Ans: (next slide)

Linked List 1 -- $O(n)$ Append

```
if(nullptr == mHead){
    mHead = newNode;
}else{
    Node* iter = mHead;

    while(nullptr != iter->next){
        iter=iter->next;
    }
    iter->next = newNode;
}
```

Linked List 2 -- $O(1)$ Append

```
if(nullptr == mHead){
    mHead = newNode;
    mTail = mHead;
}else{
    mTail->next = newNode;
    mTail = newNode;
}
```

LL2 is way faster than LL1

- From a 'Big-O' standpoint, we have gone from a $O(n)$ implementation on LL1 to an $O(1)$ time operation for Append on LL2.
 - The empirical measurement here is that we're going from 14.5 seconds down to .013 seconds
 - Wow -- ~1457.2x (~145720%) speed improvement!

LL1
mike@Michaels-MacBook-Air 2023_corecpp % time ./prog 1 100000
experiment size:100000
./prog 1 100000 12.62s user 0.02s system 86% **cpu 14.572** total

LL2
mike@Michaels-MacBook-Air 2023_corecpp % time ./prog 2 100000
experiment size:100000
./prog 2 100000 0.01s user 0.00s system 78% **cpu 0.013** total

1457.2x (145720%) Improvement! (1/2)

- Another way to think of this improvement, is that in order to improve the speed, I reduced the number of instructions needed to get a result at run-time.
 - Simple enough -- do less work while achieving the same result means we'll likely yield a performance improvement

Linked List 1 -- O(n) Append

```
if(nullptr == mHead){
    mHead = newNode;
}else{
    Node* iter = mHead;

    while(nullptr != iter->next){
        iter=iter->next;
    }
    iter->next = newNode;
}
```

Linked List 2 -- O(1) Append

```
if(nullptr == mHead){
    mHead = newNode;
    mTail = mHead;
}else{
    mTail->next = newNode;
    mTail = newNode;
}
```


1457.2x (145720%) Improvement! (2/2)

- Another way to think of this improvement, is that in order to improve the speed, I reduced the number of instructions needed to get a result at **run-time**.
 - Simple enough -- do less work while achieving the same result means likely yield a performance improvement

Linked List 1 -- O(n) Append

```
if(nullptr == mHead){
    mHead = newNode;
}else{
    Node* iter = mHead;

    while(nullptr != iter->next){
        iter=iter->next;
    }
    iter->next = newNode;
}
```

This is a classic example of a 'run-time' optimization (i.e. choosing a better algorithm).

The types of things some of us love to find as performance engineers!

```
if(nu
r
}else{
    mTail->next = newNode;
    mTail = newNode;
}
```

Recap (1/2)

- We made a significant performance improvement at run-time
 - i.e. We reduced the amount of time to complete the task by using a better data structure and implementation
 - The cost for us (the change in space complexity) was one additional pointer
 - 8-bytes of data on a 64-bit system per linked list data structure
 - i.e. $O(1)$ space complexity -- still constant!
 - The benefit
 - We reduce our 'append' to an $O(1)$ operation versus the previous $O(n)$
- And this space trade-off appears very good to me!
 - (especially if we are certain our linked lists will use 'append' frequently)

(Note: I could have 'cheated' and run 'LL1' on a much faster machine to perhaps get it faster -- but let's assume our experiments are run on the same machine in as close of a run-time environment as possible)

Recap (2/2)

- We made a significant performance improvement at run-time
 - i.e. We reduced the amount of time to complete the task by using a better data structure and implementation
 - The cost for us (the change in space complexity) was one additional pointer
 - 8-bytes of data on a 64-bit system per linked list data structure
 - i.e. $O(1)$ space complexity -- still constant!
 - The benefit
 - We reduce our 'append' to an $O(1)$ operation versus the previous $O(n)$
- And this space trade-off appears very good to me!
 - (especially if we are certain our linked lists will use 'append' frequently)

And this leads me to another fundamental trade-off, but specific to C++
(and other compiled languages)

(Note: I could have 'cheated' and run 'LL1' on a much faster machine to perhaps get it faster -- but let's assume our experiments are run on the same machine in as close of a run-time environment as possible)

So this leads into another fundamental trade-off in C++ we can make and that is ...

So this leads into another fundamental trade-off in C++ we can make and that is ...

Compile-Time versus Run-Time

Compile-Time Versus Run-Time

- In C++ we really have two places where we can trade space for time*
 - Compile-Time and run-time
- Something that I often tell my students when they first start programming in C++ is that we can think about computation at compile-time and run-time
 - If they're coming from a background programming in interpreted languages this is something new
 - And in some ways, we do 'pay' for the 'cognitive overhead' initially with the language (thinking about run-time and compile-time operations),
 - i.e. Having to remember to edit->save->compile-run.
 - (Though tools, IDEs, etc. lower some of this cognitive burden).

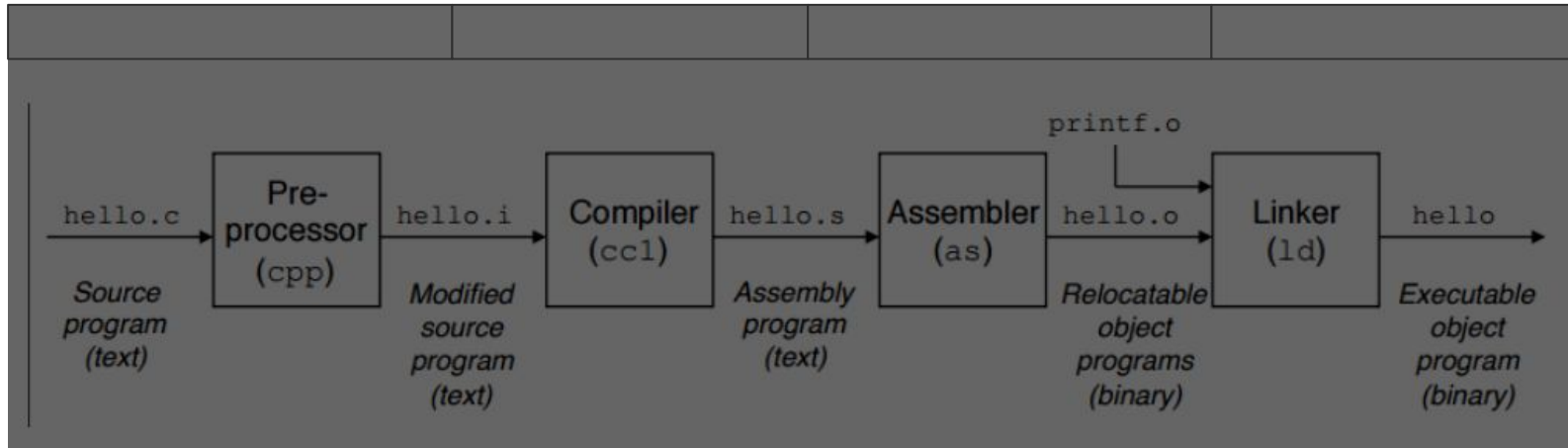
*okay maybe 3, it might be fair to add link time (the static and dynamic linker never gets enough respect huh) -- and we can further subdivide from there, but I like 2 for this talk

Compilation Process

A quick look

C++ Compilation (using g++, clang++, msvc, etc.)

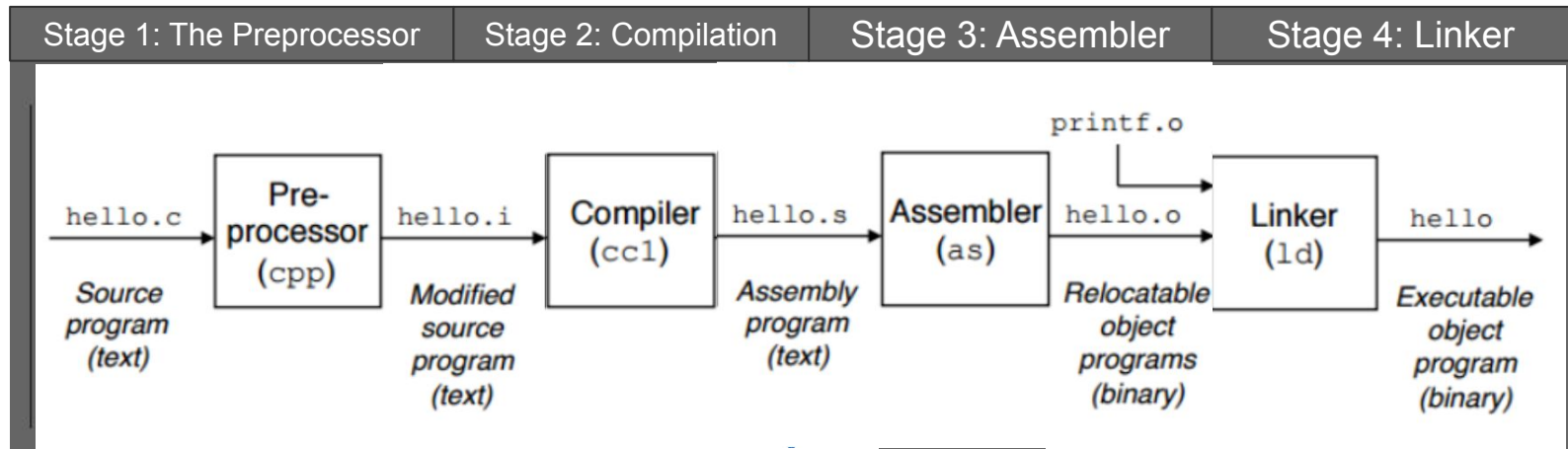
- I think it's fair to classify C++ as an example of a 'compiled language' [1]
- Compilers take our source code (.cpp files) and eventually transform our code to assembly and ultimately machine code.
 - (Eventually that assembly code is turned into an executable object file
 - (i.e. The thing you can just double click on to run or type ./program).)



[1] C++ interpreters do exist however! e.g. <https://root.cern/cling/>

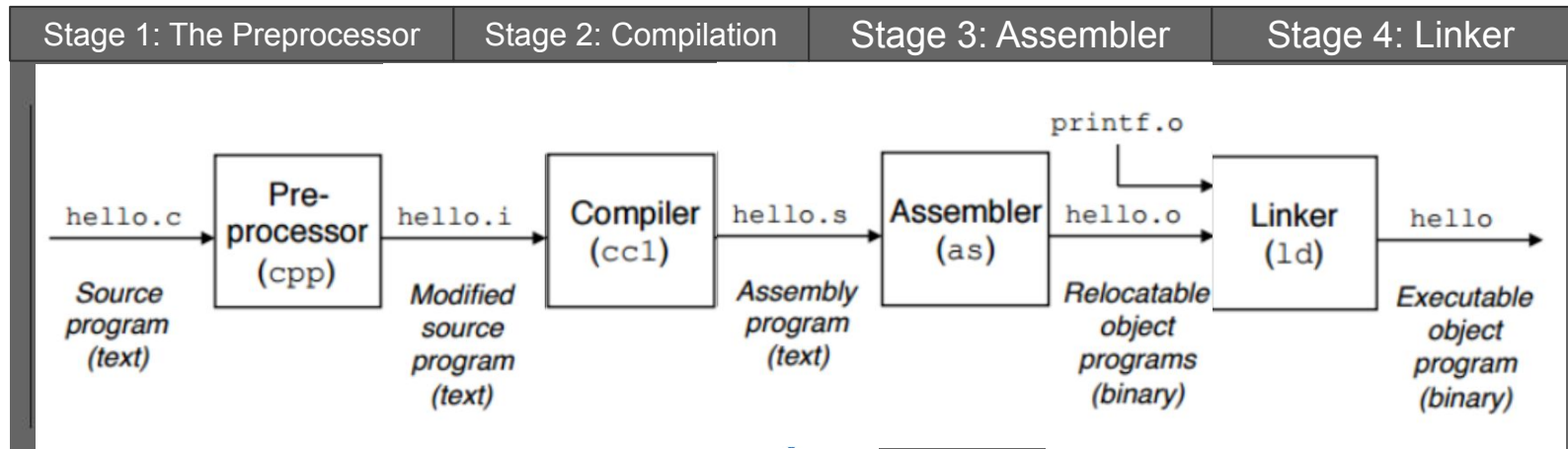
Stages of a C++ Program

- A high level view of the compilation process of source code is shown:
 - Observe that there is 'computation' going on during these stages
 - Much of it is computation to transform C++ syntax into assembly
 - But we can actually use these stages to perform useful computations during **compile-time**.



Compile-Time versus Run-time (1/3)

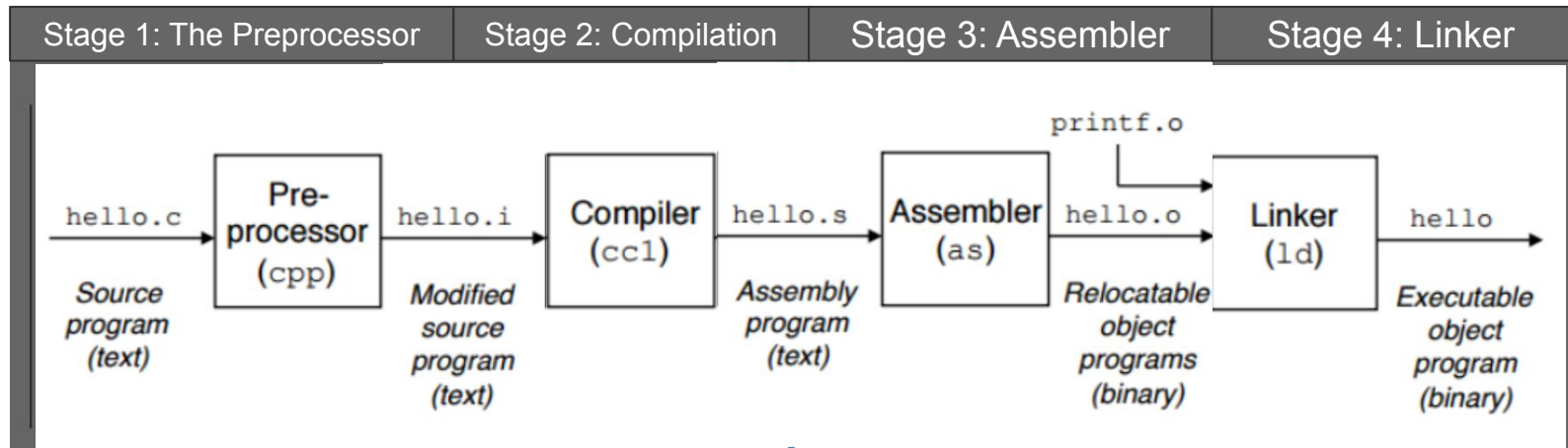
- So **compile-time** means we think about what operations and computations that happen before we execute a program
- Run-time means we are concerned with the actual execution of the program



[1] C++ interpreters do exist however! e.g. <https://root.cern/cling/>

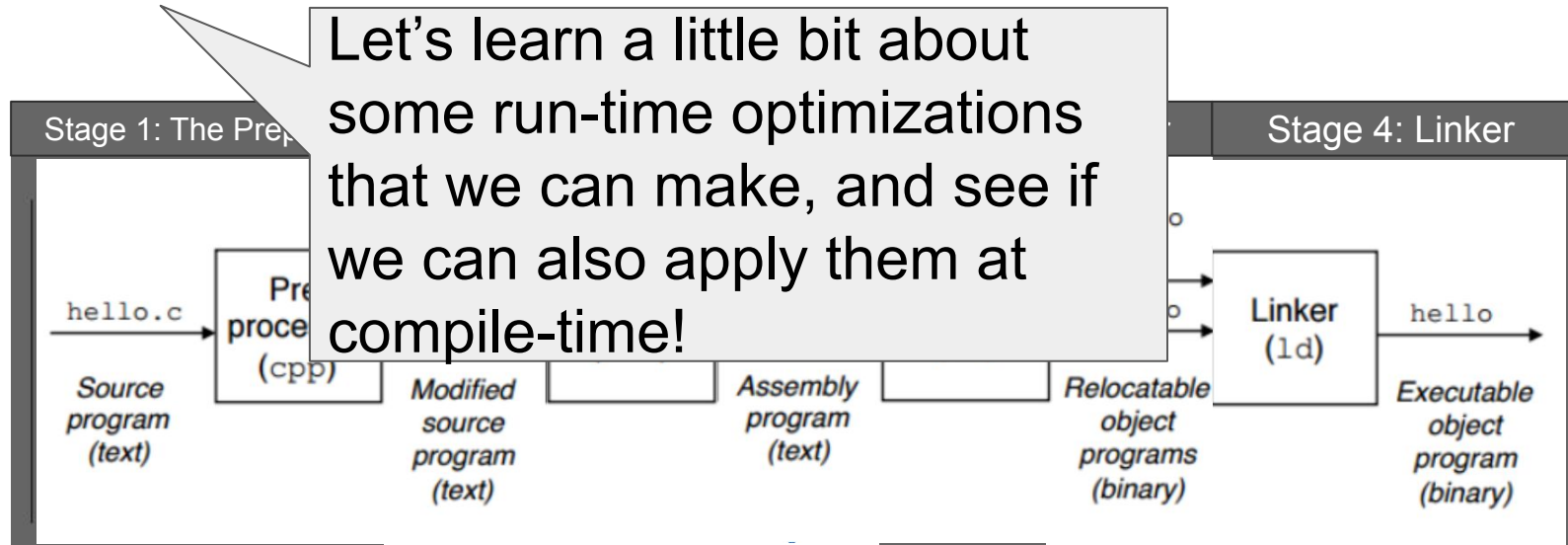
Compile-Time versus **Run-time** (2/3)

- So compile-time means we think about what operations and computations that happen before we execute a program
- **Run-time** means we are concerned with the actual execution of the program



Compile-Time versus Run-time (3/3)

- So compile-time means we think about what operations and computations that happen before we execute a program
- **Run-time** means we are concerned with the actual execution of the program



Running away from computation at **run-time**

Run-time Optimization is fun and... (1/2)

- It's a good place to run the scientific method:
 - Ask a Question
 - (“e.g. Why is my code slow”)
 - Do some Research
 - (e.g. Watch some Corecpp talks on performance, read some C++ books, etc.)
 - Construct a hypothesis
 - (e.g. “I think this is slow because of XYZ”)
 - Test your hypothesis with an experiment
 - (e.g. “Run your code with a profiler”)
 - Analyze your data
 - (i.e. Look at where you are spending time in your profile)
 - Communicate your results
 - (e.g. “Hey team, merge my pull request, my program is 145720 times faster!”)

Run-time Optimization is fun and... (2)

In this talk, I'll provide a few examples and see if we can find some themes

- It's a good place to run the scientific method:
 - Ask a Question
 - (“e.g. Why is my code slow”)
 - Do some Research
 - (e.g. Watch some Corecpp talks on performance, read some C++ books, etc.)
 - Construct a hypothesis
 - (e.g. “I think this is slow because of XYZ”)
 - Test your hypothesis with an experiment
 - (e.g. “Run your code with a profiler”)
 - Analyze your data
 - (i.e. Look at where you are spending time in your profile)
 - Communicate your results
 - (e.g. “Hey team, merge my pull request, my program is 145720 times faster!”)

#1 Use a better algorithm (for your use case)

- We saw this with the linked list example so I hope that is clear.
 - In order to improve run-time performance, we often trade space for time.
 - More storage in our linked list example unlocked a better algorithm
 - It's a good starting point to try to reduce the amount of 'recomputation' that you have perform.

Linked List 1 -- $O(n)$ Append

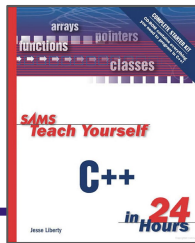
```
if(nullptr == mHead){
    mHead = newNode;
}else{
    Node* iter = mHead;

    while(nullptr != iter->next){
        iter=iter->next;
    }
    iter->next = newNode;
}
```

Linked List 2 -- $O(1)$ Append

```
if(nullptr == mHead){
    mHead = newNode;
    mTail = mHead;
}else{
    mTail->next = newNode;
    mTail = newNode;
}
```

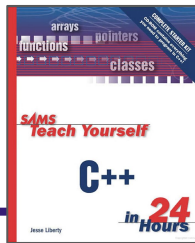

#2 Do Less Work (1/4)



- Sometimes you don't need to trade space for time to reduce the number of computations you perform
- Here's an example I probably learned in a book like above -- basic but important!
 - Short-circuit evaluation

```
1 // g++ -g -Wall -std=c++20 shortcircuit.cpp -o prog
2 #include <iostream>
3
4 bool Is_ExpensiveToComputeFunction(){
5     bool result = false;
6     /*
7         ... lots of code ...
8         ... result maybe true or false ...
9     */
10    std::cout << "I take a really loooooong time!\n";
11    return result;
12 }
13
14 // Entry point to program
15 int main(int argc, char* argv[]){
16
17     bool flag = false;
18
19     // Short-circuit evaluation
20     if (flag && Is_ExpensiveToComputeFunction()){
21
22     }
23
24     // Not short-circuit evaluated
25     if (flag & Is_ExpensiveToComputeFunction()){
26
27     }
28
29     return 0;
30 }
```

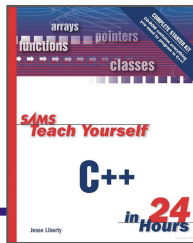
#2 Do Less Work (2/4)



- Sometimes you don't need to trade space for time to reduce the number of computations you perform
- Here's an example I probably learned in a book like above -- basic but important!
 - Short-circuit evaluation
 - Observe that the first example allows me an 'early exit without evaluating the entire condition

```
1 // g++ -g -Wall -std=c++20 shortcircuit.cpp -o prog
2 #include <iostream>
3
4 bool Is_ExpensiveToComputeFunction(){
5     bool result = false;
6     /*
7         ... lots of code ...
8         ... result maybe true or false ...
9     */
10    std::cout << "I take a really loooooong time!\n";
11    return result;
12 }
13
14 // Entry point to program
15 int main(int argc, char* argv[]){
16
17     bool flag = false;
18
19     // Short-circuit evaluation
20     if (flag && Is_ExpensiveToComputeFunction()){
21
22     }
23
24     // Not short-circuit evaluated
25     if (flag & Is_ExpensiveToComputeFunction()){
26
27     }
28
29     return 0;
30 }
```

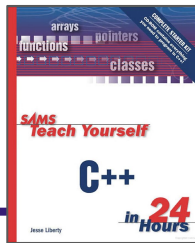
#2 Do Less Work (3/4)



- If I naively flip the operation for the more expensive operation to occur first -- that could be costly!
 - It *may* make sense to do the cheap thing first!
 - (i.e. put 'flag' first like in the previous example)
- Order of evaluation can sometimes matter!

```
1 //g++ -g -Wall -std=c++20 shortcircuit_worse.cpp -o prog
2 #include <iostream>
3
4 bool Is_ExpensiveToComputeFunction(){
5     bool result = false;
6     /*
7         ... lots of code ...
8         ... result maybe true or false ...
9     */
10    std::cout << "I take a really loooooong time!\n";
11    return result;
12 }
13
14 // Entry point to program
15 int main(int argc, char* argv[]){
16
17     bool flag = false;
18
19     // Short-circuit evaluation
20     if (Is_ExpensiveToComputeFunction() && flag){
21
22     }
23
24     // Not short-circuit evaluated
25     if (Is_ExpensiveToComputeFunction() & flag){
26
27     }
28
29     return 0;
30 }
```

#2 Do Less Work (4/4)



- If I naively flip the operation for the more expensive operation to occur first -- that could be costly!
 - It *may* make sense to do the cheap thing first!
 - (i.e. put 'flag' first like in the previous example)
- Order of evaluation can sometimes matter!
 - There that's better!

```
1 // g++ -g -Wall -std=c++20 shortcircuit.cpp -o prog
2 #include <iostream>
3
4 bool Is_ExpensiveToComputeFunction(){
5     bool result = false;
6     /*
7     ... lots of code ...
8     ... result maybe true or false ...
9     */
10    std::cout << "I take a really loooooong time!\n";
11    return result;
12 }
13
14 // Entry point to program
15 int main(int argc, char* argv[]){
16
17     bool flag = false;
18
19     // Short-circuit evaluation
20     if (flag && Is_ExpensiveToComputeFunction()){
21
22     }
23
24     // Not short-circuit evaluated
25     if (flag & Is_ExpensiveToComputeFunction()){
26
27     }
28
29     return 0;
30 }
```

Micro versus macro decisions (1/2)

- And I think this is an interesting junction point when it comes to run-time optimizations
 - **Turn Left:** We can look at more little optimizations that add up (and this can often be in very meaningful ways!)
 - e.g. short-circuiting
 - **Turn Right:** Continue writing better algorithms and data structures
 - e.g. Choose a better data structure

Let's take a left turn and look!

1 // g++ -g
2 #include
3
4 bool Is_E
5 bool
6 /*
7
8
9 */
10 std::
11 return
12 }
13
14 // Entry point to program
15 int main(int argc, char* argv[]){
16
17 bool flag = false;
18
19 // Short-circuit evaluation
20 if (flag && Is_ExpensiveToComputeFunction()){
21
22 }
23
24 // Not short-circuit evaluated
25 if (flag & Is_ExpensiveToComputeFunction()){
26
27 }
28
29 return 0;
30 }

Turn Left
Micro
optimize

Turn Right
Revisit Better
algorithms,
better data
structures



45

Micro versus macro decisions (2/2)

- And I think this is an interesting junction point when it comes to run-time optimizations
 - **Turn Left:** We can look at more little optimizations that add up (and this can often be in very meaningful ways!)
 - e.g. short-circuiting
 - **Turn Right:** Continue writing better algorithms and data structures
 - e.g. Choose a better data structure

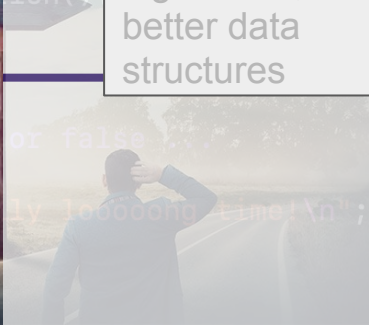
Let's take a left turn and look!

1 // g++ -g
2 #include
3
4 bool Is_ExpensiveToComputeFunction()
5 bool
6 /*
7
8
9 */
10 std::cout << "Is_ExpensiveToComputeFunction() is true!\n";
11 return true;
12 }
13
14 // Entry point to program
15 int main(int argc, char* argv[]){
16
17 bool flag = false;
18
19 // Short-circuit evaluation
20 if (flag && Is_ExpensiveToComputeFunction()){
21
22 }
23
24 // Not short-circuit evaluated
25 if (flag & Is_ExpensiveToComputeFunction()){
26
27 }
28
29 return 0;
30 }

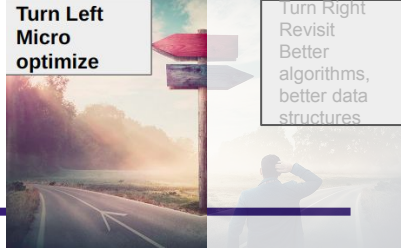
Turn Left
Micro
optimize



Turn Right
Revisit Better
algorithms,
better data
structures

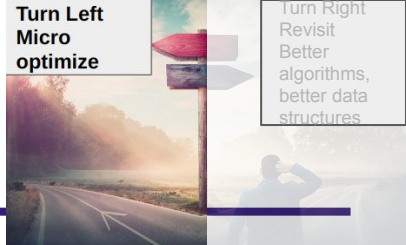


46



#3 Hand tune optimizations

- This is getting into the fun world of really hand tuning our code
 - Find code we are not using and eliminate it
 - i.e. dead code elimination
 - Common Subexpression elimination
 - i.e. Compute an expression one time and cache a value
 - Unrolling loops
 - i.e. Help our processor out by manually unrolling loops
 - inlining functions
 - i.e. avoiding function call overhead and enabling other optimizations
 - Strength Reduction/Instruction Selection
 - i.e. choosing better instructions
- I love cleaning up my code with some of these optimizations
 - (Spoiler alert: We'll revisit these at 'compile-time' as many of these listed at compile-time!)



#3 Hand tune optimizations - Dead Code Elimination

- Dead Code Elimination is the process of removing unused variables or unreachable code from our project
 - No need to include computation (and storage) that we are not going to make use of!
 - (And it's just less to manage during a code review!)

```
int foo(void) {  
    int a = 24;  
    int b = 25; /* Assignment to dead variable */  
    int c;  
    c = a * 4;  
    return c;  
    b = 24; /* Unreachable code */  
    return 0;  
}
```

https://en.wikipedia.org/wiki/Dead-code_elimination

#3 Hand tune optimizations - Common Subexpression Elimination (1)

- We can compute a value once and 'cache' the value to avoid recomputing it
 - Observe on the right that the value 'b*c' is computed more than once.
 - Thus, we can cache that value
 - More expensive operations (e.g. trig function, certain divisions, etc.) may have more benefit.

Example [\[edit \]](#)

In the following code:

```
a = b * c + g;  
d = b * c * e;
```

it may be worth transforming the code to:

```
tmp = b * c;  
a = tmp + g;  
d = tmp * e;
```

if the cost of storing and retrieving `tmp` is less than the cost of calculating `b * c` an extra time.

https://en.wikipedia.org/wiki/Common_subexpression_elimination

#3 Hand tune optimizations - Common

- That little idea of ‘caching’ might also help you avoid recomputing sneaky function calls
 - Compare the two loops
 - Perhaps ‘size’ of a graph has to be walked each loop
 - If we know the size won’t change in the loop, no need to have the overhead!

Example

In the following

```
a = b *  
d = b *
```

it may be wo

```
tmp = b  
a = tmp  
d = tmp
```

if the cost of

https://en.cppreference.com/w/cpp/string/basic_string_view

```
1 // g++ -g -Wall -std=c++20 size.cpp -o prog  
2 #include <cstdlib>  
3  
4 struct Graph{  
5     /*  
6     */  
7     */  
8     size_t size() const{  
9         return 42;  
10    }  
11 };  
12  
13  
14 // Entry point to program  
15 int main(int argc, char* argv[]){  
16  
17     Graph g;  
18  
19     for(size_t i=0; i < g.size(); i++){  
20         /* Do some work */  
21     }  
22  
23     // Observe that we 'cache' the size here  
24     // so we only have to compute it once.  
25     const size_t graph_size = g.size();  
26     for(size_t i=0; i < graph_size; i++){  
27         /* Do some work */  
28     }  
29  
30     return 0;  
31 }
```

*Note: Pretend that size actually does some meaningful traversal and does not just return the integer ‘42’.

#3 Hand tune optimizations - Strength

- And here's one final hand tuned optimization that may make a difference when it comes to instruction selection
 - Observe ++i versus i++
 - Observe '<' versus '!='
- These types of 'strength reduction' optimizations may result in assembly instructions that take fewer cycles
 - Less computation--great!
 - (And in this case, probably exactly what we want with the algorithm,
 - != for example is more intentional of the intent)

Example

In the following

```
a = b *  
d = b *
```

it may be worth

```
tmp = b  
a = tmp  
d = tmp
```

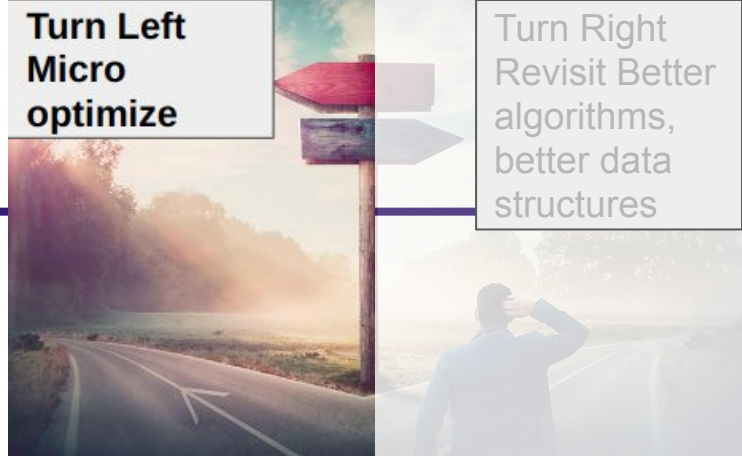
if the cost of

https://en.cppreference.com/w/cpp/string/basic_string_view

```
1 // g++ -g -Wall -std=c++20 size.cpp -o prog  
2 #include <cstdlib>  
3  
4 struct Graph{  
5     /*  
6     */  
7     */  
8     size_t size() const{  
9         return 42;  
10    }  
11 };  
12  
13  
14 // Entry point to program  
15 int main(int argc, char* argv[]){  
16  
17     // Observe that we 'cache' the size here  
18     // so we only have to compute it once.  
19     const size_t graph_size = g.size();  
20     for(size_t i=0; i != graph_size; ++i){  
21         /* Do some work */  
22     }  
23  
24     // Observe that we 'cache' the size here  
25     // so we only have to compute it once.  
26     const size_t graph_size = g.size();  
27     for(size_t i=0; i < graph_size; i++){  
28         /* Do some work */  
29     }  
30  
31     return 0;  
32 }
```

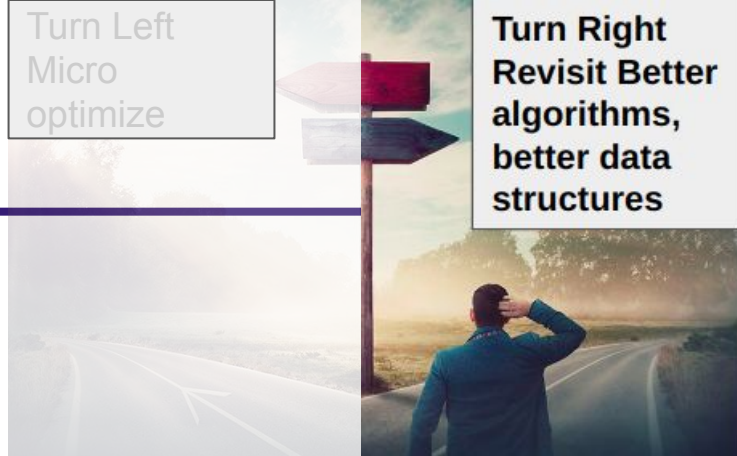
And we can continue further... (1/3)

- Hand optimizing code is fun (to me)
 - And going through the process makes us often think carefully line by line of how much and what are we computing
- But let's try to take that other road -- let's revisit some algorithms and data structures



And we can continue further... (2/3)

- Hand optimizing code is fun (to me)
 - And going through the process makes us often think carefully line by line of how much and what are we computing
- But let's try to take that other road -- let's revisit some algorithms and data structures

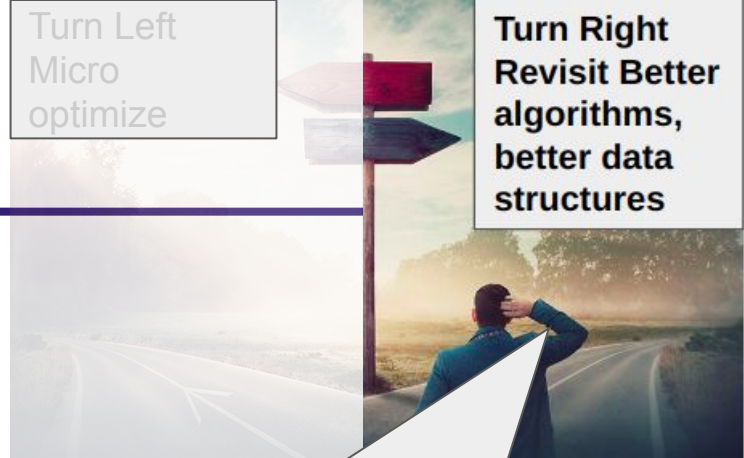


And we can continue further... (3/3)

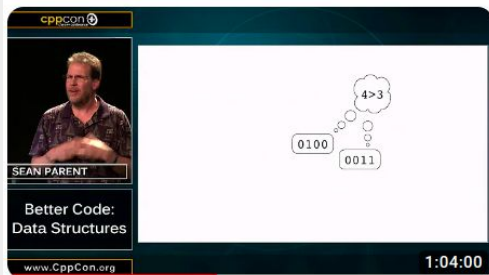
- Hand optimizing code is fun (to me)
 - And going through the process makes us often think carefully line by line of how much and what are we computing
- But let's try to take that other road -- let's revisit some algorithms and data structures

Turn Left
Micro
optimize

Turn Right
Revisit Better
algorithms,
better data
structures



Yes -- I did think of the Sean Parent talk when labeling this text



CppCon 2015: Sean Parent "Better Code: Data Structures"

97K views · 7 years ago



CppCon

<http://www.cppcon.org> -- The standard library containers are often both misused and underused. Instead of creating new ...



Introduction | Goal | What is a data structure | What is a structure | Dimensionless space | Four bits... 25 chapters ▾

Turn Left
Micro
optimize

Turn Right
Revisit Better
algorithms,
better data
structures

#4 Better Algorithms, Better Data Structures (1/3)

- A C++ STL example might be `map` versus `unordered_map`
- **Question to the Audience:** What is the main difference between these two data structures?
 - Answer: next slide

#4 Better Algorithms, Better Data Structures (2/3)

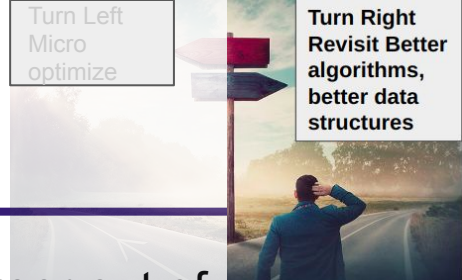
- A C++ STL example might be `map` versus `unordered_map`
- **Question to the Audience:** What is the main difference between these two data structures?
 - Answer: next slide
 - `map` is sorted
 - This means we usually need a balanced tree structure (e.g. rb-tree)
 - $(\log_2(n))$ operations)
 - `unordered_map` is not sorted
 - This means we usually use a hash table
 - $(O(1))$ average case operations)
 - If you don't need your data sorted, then `unordered_map` can be much more efficient at insertion/deletion/update.
 - Don't pay for something that you won't need at run-time

Turn Left
Micro
optimize

Turn Right
Revisit Better
algorithms,
better data
structures

#4 Better Algorithms, Better Data Structures (3/3)

- ^ We can do a few semesters worth or otherwise make a career out of learning more data structures and algorithms
 - So this is this the end of our run-time optimizations strategies?
 - i.e. Just keep learning (or maybe one day inventing yourself!) new algorithms/data structures for the problem you solve?



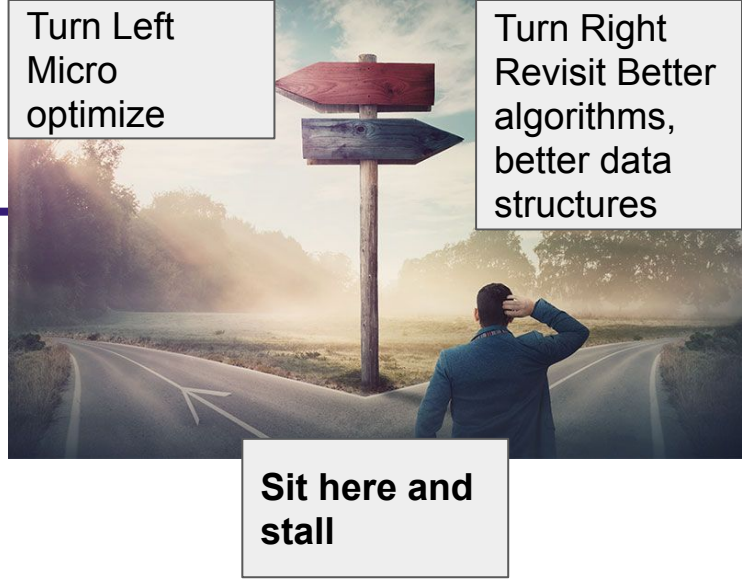
Done with run-time computation?

- Well...
 - I've shown you cute/real tricks to optimize some code
 - I've reminded you that data structures and algorithms matter for efficiency
- The good news is we have more strategies to truly 'run away from computation' -- and make our code potentially more performant



#5 Delay/Stall

- So there's another option instead of making a left turn or a right turn -- let's just sit here and wait
 - i.e. we're going to pick a third route of just stalling or delaying.
 - Let's just stall/delay/defer our computation as long as possible
 - Let's run away from the problem (at least for a while) :)



#5 Delay/Stall - Promise/Future (1/2)

- Here's a mechanism we have available in C++11 using Promises and Futures
 - (You can read the example code later.)
- The key idea here is that we can launch another thread to do some work
 - Then proceed in our program as normal
 - Then block until we have a result
 - (can also use 'myFuture.get')
- This seems to be the 'right idea' for delaying our computation
 - (Though we're still computing somewhere, on some other thread -- so a price is paid!)

```
1 // g++ -g -Wall -std=c++20 promise.cpp -o prog -lpthread
2 #include <iostream>
3 #include <future>
4 #include <thread>
5
6 int ComputeSomethingExpensive(){
7     using namespace std::chrono_literals;
8     /* really expensive computation... */
9     std::this_thread::sleep_for(2000ms);
10    std::cout << "Working asynchronously" << std::endl;
11
12    return 42;
13 }
14
15 // Entry point to program
16 int main(int argc, char* argv[]){
17
18     // std::promise stores a value that will be later acquired
19     // asynchronously. It is a 'promise' that guarantees some
20     // value will be available later.
21     // The promise 'pushes' the result to the future when
22     // we execute it.
23     std::promise<int> myPromise;
24
25     // Future waits for a value (i.e. reads)
26     // Future is associated with a promise.
27     // The type in the promise and future must match (i.e. both are int).
28     std::future<int> myFuture = myPromise.get_future();
29     // The 'get_future' associates the promise with this future
30     // thus 'bundling' them together.
31
32     std::cout << "Do some other work here" << std::endl;
33
34     // Launch a thread which will do some work.
35     // Note: We need to 'capture' myPromise here
36     //       in order to use it in our lambda.
37     // Note: 'myPromise' does not have to be in a thread,
38     //       we could call myPromise.set_value within main, but
39     //       we would wait for the function to execute.
40     std::thread worker([&myPromise]{
41         myPromise.set_value(ComputeSomethingExpensive());
42     });
43
44     std::cout << "continue on with our lives" << std::endl;
45
46     myFuture.wait();
47     // Note: .get effectively calls '.wait' so this
48     //       is redundant, just use 'get' in this case.
49     // 'get blocks' on the future until a result is available.
50     std::cout << "myFuture is: " << myFuture.get() << std::endl;
51
52     // Don't forget to join your thread!
53     // Otherwise, consider using jthread
54     worker.join();
55
56     return 0;
57 }
```

#5 Delay/Stall - Promise/Future (2/2)

- In this example however, we are still doing all of the work in a separate thread
 - So I need to show you something that builds off of this mechanism

```
std::thread worker([&myPromise]{
    myPromise.set_value(ComputeSomethingExpensive());
});

std::cout << "continue on with our lives" << std::endl;

myFuture.wait();
```

```
1 // g++ -g -Wall -std=c++20 promise.cpp -o prog -lpthread
2 #include <iostream>
3 #include <future>
4 #include <thread>
5
6 int ComputeSomethingExpensive(){
7     using namespace std::chrono_literals;
8     /* really expensive computation... */
9     std::this_thread::sleep_for(2000ms);
10    std::cout << "Working asynchronously" << std::endl;
11
12    return 42;
13 }
14
15 // Entry point to program
16 int main(int argc, char* argv[]){
17
18     // std::promise stores a value that will be later acquired
19     // asynchronously. It is a 'promise' that guarantees some
20     // value will be available later.
21     // The promise 'pushes' the result to the future when
22     // we execute it.
23     std::promise<int> myPromise;
24
25     // Future waits for a value (i.e. reads)
26     // Future is associated with a promise.
27     // The type in the promise and future must match (i.e. both are int).
28     std::future<int> myFuture = myPromise.get_future();
29     // The 'get_future' associates the promise with this future
30     // thus 'bundling' them together.
31
32     std::cout << "Do some other work here" << std::endl;
33
34     // Launch a thread which will do some work.
35     // Note: We need to 'capture' myPromise here
36     //       in order to use it in our lambda.
37     // Note: 'myPromise' does not have to be in a thread,
38     //       we could call myPromise.set_value within main, but
39     //       we would wait for the function to execute.
40     std::thread worker([&myPromise]{
41         myPromise.set_value(ComputeSomethingExpensive());
42     });
43
44     std::cout << "continue on with our lives" << std::endl;
45
46     myFuture.wait();
47     // Note: .get effectively calls '.wait' so this
48     //       is redundant, just use 'get' in this case.
49     // 'get blocks' on the future until a result is available.
50     std::cout << "myFuture is: " << myFuture.get() << std::endl;
51
52     // Don't forget to join your thread!
53     // Otherwise, consider using jthread
54     worker.join();
55
56     return 0;
57 }
```

#5 Delay/Stall - std::async

- std::async is a simpler mechanism to defer computation.
 - std::async will return a std::future for us.
- Observe in this example we do not actually compute until we hit '.get'
 - This is due to the std::launch::deferred argument in std::async
 - This is the execution policy
- We call this deferred (a.k.a 'lazy') computations.
 - e.g.
 - See [std::launch::deferred](#)

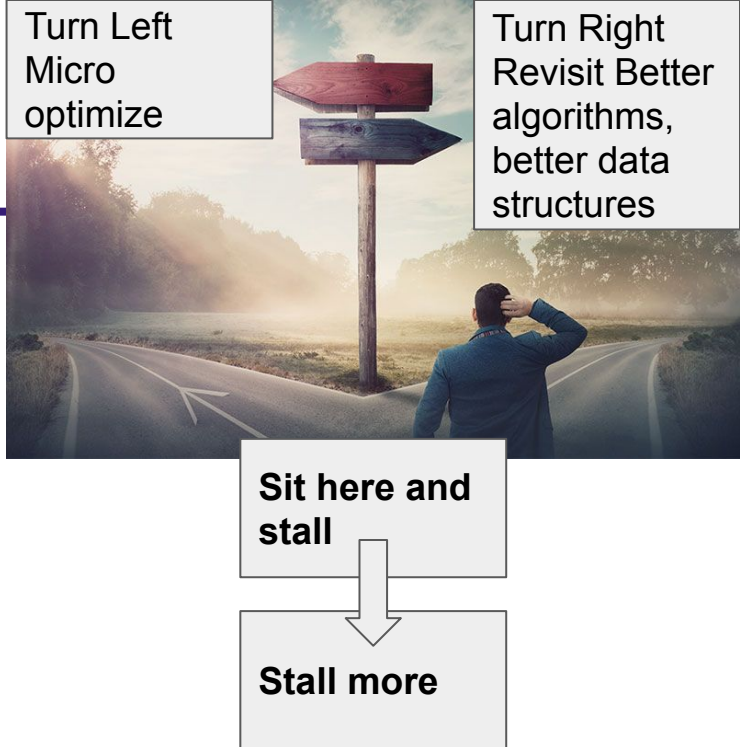
```
1 // g++ -g -Wall -std=c++20 deferred.cpp -o prog -lpthread
2 #include <iostream>
3 #include <thread>
4 #include <future>
5
6 void expensiveComputation(){
7     using namespace std::chrono_literals;
8     /* really expensive computation... */
9     std::this_thread::sleep_for(2000ms);
10
11     std::cout << "Computing something expensive\n";
12 }
13
14 // Entry point to program
15 int main(int argc, char* argv[]){
16
17     // Setup a promise/future
18     auto lazy = std::async(std::launch::deferred, &expensiveComputation);
19
20     std::cout << "Do some work here" << std::endl;
21
22     // Try switching flag to true and false and
23     // see the different behaviors
24     bool flag=false;
25
26     if(flag){
27         // Function not called until we
28         // explicitly ask for result.
29         lazy.get();
30     }
31
32     std::cout << "Continuing on with our lives" << std::endl;
33
34     return 0;
35 }
```

We never execute this block of code so we pay no cost to evaluate 'lazy'

This is lazy evaluation

Stall longer

- So I like this strategy of stalling/delaying computation
 - We can thus avoid some unnecessary computation
- But there also exists more strategies to stall and avoid computation.
 - Let's review one more!

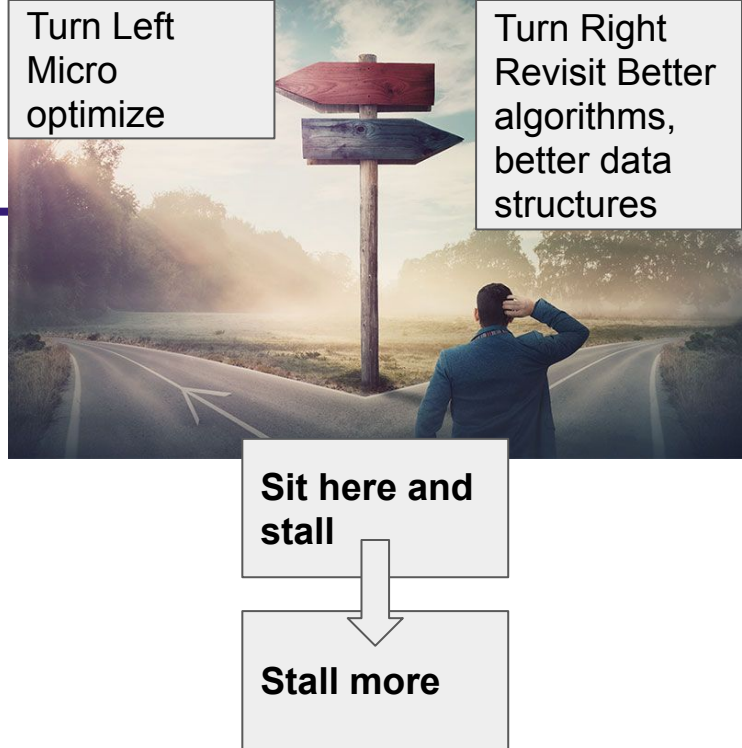


#6 - Copy on Write (CoW)

- Copy-on-Write is just another strategy to defer computation (like our previous ones)
 - The idea is whenever we make a copy of some memory (i.e. a data structure), we don't immediately need to make a fresh copy of that memory
 - For example: if the only operations we are doing is a 'read operation' on our newly copied data, do we really need to update anything?
 - The answer is no -- at least for as long as we can!
- Note:
 - Copy-on-write is also known as 'lazy initialization'

Run-time computation Strategies (1/3)

- So here's a summary of reducing computation
 - #1 Use a Better Algorithm
 - (Often paying storage to enable this)
 - #2 Do less work
 - (Short circuiting, or choosing the least costly operation first if we can terminate early)
 - #3 Hand tune our code
 - (Select better instructions, perform clever code tricks)
 - #4 Better algorithms, better data structures
 - (Kind of the same as number 1 -- just be sure to pay for what you actually need the data structure to do)
 - #5 Delay/Stall
 - `std::async` as an example for deferring computation (Note: We did use a thread to help us)
 - #6 Copy-on-Write
 - Another way to defer



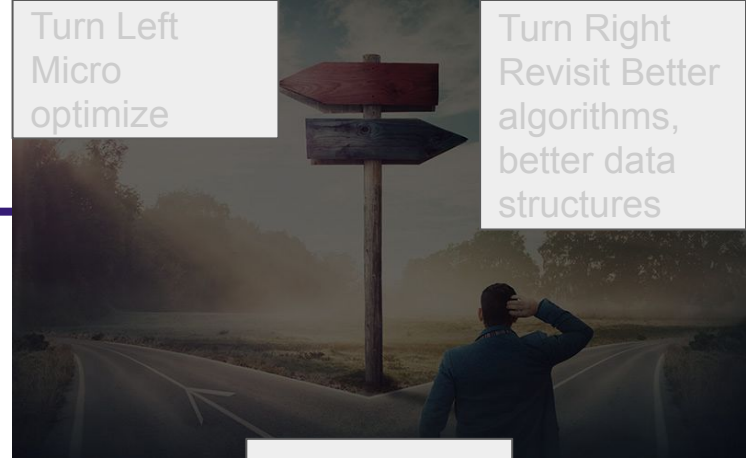
Run-time computation Strategies (2/3)

- So here's a summary of reducing computation

- #1 Use a Better Algorithm
 - (Often paying storage to enable this)
- #2 Do less work
 - (Short circuiting, or choosing the least costly operation first if we can terminate early)
- #3 Hand tune our code
 - (Select better instructions, perform clever code tricks)
- #4 Better algorithms, better data structures
 - (Kind of the same as number 1 -- just make sure to pay for what you actually need the data to do)
- #5 Delay/Stall
 - `std::async` as an example for deferred computation (Note: We did use a thread to handle the computation)
- #6 Copy-on-Write
 - Another way to defer computation

Turn Left
Micro
optimize

Turn Right
Revisit Better
algorithms,
better data
structures



Sit here and
stall

Stall more

- This isn't even close to a complete list of how we can further think about optimizing code (or the art of profiling, understanding cpu/gpu architecture, etc.)
- But hopefully this gives you some model of thinking at run-time!

The banner image is divided into three sections. The left section shows a close-up of a man's face wearing a red cap. The middle section displays a vertical bar chart with values ranging from 0.000000 to 50.000000. The right section features a complex network diagram with numerous nodes and connecting lines.

- This isn't even close to a complete list of how we can further think about optimizing code (or the art of profiling, understanding cpu/gpu architecture, etc.)
- But hopefully this gives you some model of thinking at run-time!

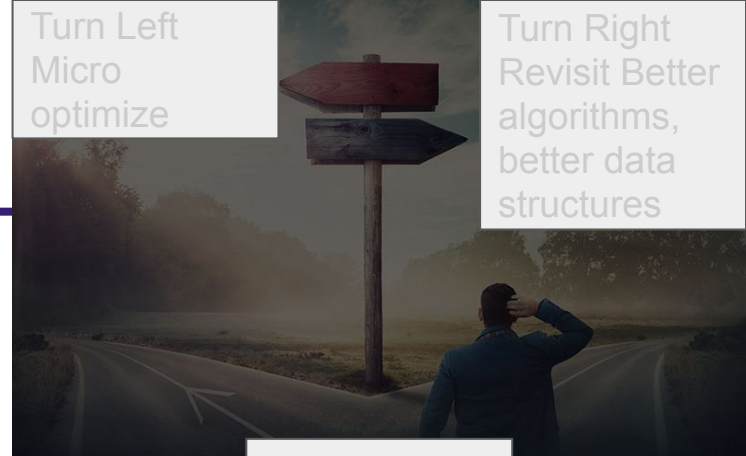


orm clever code

- This

- This isn't even close to a complete list of how we can further think about optimizing code (or the art of profiling, understanding cpu/gpu architecture, etc.)
- But hopefully this gives you some model of thinking at run-time!

Turn Right
Revisit Better
algorithms,
better data
structures



Stall more

Running away from computation at
compile-time

Compile-Time Computation

- So remember, the great thing about a compiled language like C++ is we also get to make choices to execute code at compile-time.
 - At first the idea of 'execute at compile-time' seems weird (and it still is weird to me sometimes)
 - i.e. You might ask -- doesn't the machine need to execute before we can run?
- Let me introduce you to some ideas of computation we can control at compile-time.
 - (Spoiler alert: Many of these items are familiar from our run-time optimizations!)

#1 Let the Compiler Optimize (-02) - Dead Code Elimination

- At compile-time our compiler often has a much better view of the entire source code versus myself at a given time
 - So the compiler is able to remove any provably unused variables, expressions, or unreachable code.

```
int foo(void) {  
    int a = 24;  
    int b = 25; /* Assignment to dead variable */  
    int c;  
    c = a * 4;  
    return c;  
    b = 24; /* Unreachable code */  
    return 0;  
}
```

https://en.wikipedia.org/wiki/Dead-code_elimination

#1 Let the Compiler Optimize (-02) - Common Subexpression Elimination

- Our compiler can similarly use heuristics to otherwise cache subexpressions
 - More expensive operations (e.g. trig function, certain divisions, etc.) may have even more benefit here!
- Aside: These compiler optimizations are good to know -- because then you can write more readable code in your first iterations and focus on correctness before fine tuning.

Example [\[edit\]](#)

In the following code:

```
a = b * c + g;  
d = b * c * e;
```

it may be worth transforming the code to:

```
tmp = b * c;  
a = tmp + g;  
d = tmp * e;
```

if the cost of storing and retrieving `tmp` is less than the cost of calculating `b * c` an extra time.

https://en.wikipedia.org/wiki/Common_subexpression_elimination

Strategy 2 - Question to Audience:

- What (or will there be) is the difference in the assembly code between these two functions?
 - (Answer: next slide)

```
1 // g++ -g -Wall -std=c++20 ref.cpp -o prog
2
3 int global;
4
5 void PassByPointer(const int* p)
6 {
7     global += *p;
8 }
9 void PassByReference(const int& p) {
10     global += p;
11 }
```


#2 Use References [[Core guideline](#)]

- What (or will there be) is the difference in the assembly code between these two functions?
 - Answer:
 - No difference!

```
1 // g++ -g -Wall -std=c++20 ref.cpp -o prog
2
3 int global;
4
5 void PassByPointer(const int* p)
6 {
7     global += *p;
8 }
9 void PassByReference(const int& p) {
10     global += p;
11 }
```

```
PassByPointer(int const*):
  8b 07
401120  mov    (%rdi),%eax
      01 05 04 2f 00 00
401122  add    %eax,0x2f04(%rip)          # 40402c <global>
      c3
401128  ret
      0f 1f 80 00 00 00 00
401129  nopl   0x0(%rax)
PassByReference(int const&):
  8b 07
401130  mov    (%rdi),%eax
      01 05 f4 2e 00 00
401132  add    %eax,0x2ef4(%rip)          # 40402c <global>
      c3
401138  ret
      0f 1f 80 00 00 00 00
401139  nopl   0x0(%rax)
```

- Same instructions, but generally we like 'references' over pointers. Why?
 - Well, I should probably check if that pointer 'p' is a nullptr
 - That's a few extra steps of computation that I have to worry about less often with references.

Question to Audience:

- So thinking about our previous discussion on pointers and references here's a question.
- Question: When is the best time to catch a bug?
 - (ans: next slide)

Question to Audience:

- So thinking about our previous discussion on pointers and references here's a question.
- Question: When is the best time to catch a bug?
 - Ideally before it even occurs!
 - So, if we can catch a bug at compile-time, that is optimal for us as a developer (and of course our end-user)
 - Let's look at some of the features C++ offers us

#3 Use static_assert when possible [[Core Guideline](#)] (1/2)

- In C++11 we added static_assert which does a check at compile-time against values known at compile-time (constexpr coming up!)
 - static_assert is an example of us being explicit in the language of where we want to check something -- in this case the size of an integer
 - (versus assert, which is checked at run-time)
- With static_assert, the user does not pay a cost at run-time

```
1 // g++ -g -Wall -std=c++20 static_assert.cpp -o prog
2 #include <cassert>
3
4 // Entry point to program
5 int main(int argc, char* argv){
6
7     static_assert(sizeof(int)==5091 && "compile-time check");
8     assert(sizeof(int)==7091 && "run-time check!");
9
10    return 0;
11 }
12
```

7,33-35 All

```
mike:2023_corecpp$ g++ -g -Wall -std=c++20 static_assert.cpp -o prog && ./prog
static_assert.cpp: In function 'int main(int, char**)':
static_assert.cpp:7:35: error: static assertion failed
   7 |     static_assert(sizeof(int)==5091 && "compile-time check");
     |
```

#3 Use static_assert when possible [Core Guideline] (2/2)

- In C++11 we added static_assert which does a check at compile-time

```
1 // g++ -g -Wall -std=c++20 static_assert.cpp -o prog
2 #include <cassert>
3
4 // Entry point to program
5 int main(int argc, char* argv){
6
```

P.5: Prefer compile-time checking to run-time checking

Reason Code clarity and performance. You don't need to write error handlers for errors caught at compile time.

Similar to the references versus pointer discussion -- how nice it is if you can catch bugs at compile-time and avoid writing lots of error handling code!

at run-time)

- With static_assert, the user does not pay a cost at run-time

All
& ./prog

#4 constexpr [[cppreference](#)]

- In C++ we can mark things as ‘constexpr’ to try to evaluate at compile-time
 - Thus, we pay a cost at compile-time (developer) as opposed to run-time (user) to perform a computation
 - ‘constexpr’ also gives us some advantages when you start thinking about undefined behavior -- the behavior must be defined for us to compute a result (otherwise, compile-error or compiler bug!)

constexpr specifier (since C++11)

- **constexpr** - specifies that the value of a variable or function can appear in **constant expressions**

Explanation

The **constexpr** specifier declares that it is possible to evaluate the value of the function or variable at compile time. Such variables and functions can then be used where only compile time **constant expressions** are allowed (provided that appropriate function arguments are given).

A **constexpr** specifier used in an object declaration or non-static member function (until C++14) implies **const**. A **constexpr** specifier used in a function or static data member (since C++17) declaration implies **inline**. If any declaration of a function or function template has a **constexpr** specifier, then every declaration must contain that specifier.

#4 constexpr [[cppreference](#)] - factorial (1/3)

- Here's an example of a 'factorial' evaluated and checked at compile-time using constexpr

```
1 // g++ -g -Wall -std=c++20 compiletime_factorial.cpp -o prog
2 #include <cassert>
3
4 constexpr int fac(int n)
5 {
6     int result = 1;
7     for (int i = 2; i <= n; ++i){
8         result *= i;
9     }
10    return result;
11 }
12
13 // Entry point to program
14 int main(int argc, char* argv){
15
16     static_assert(fac(5) == 120 && "compile-time check");
17
18     return 0;
19 }
20
"factorial.cpp" 20L, 341B written
```

```
mike:2023_corecpp$ g++ -g -Wall -std=c++20 factorial.cpp -o prog && ./prog
```

Compile-time factorial

#4 constexpr [c++reference] -

The version on the left does not make use of constexpr

```
1 // g++ -g -Wall -std=c++20 runtime_factorial.cpp -o prog
2 #include <cassert>
3
4 int fac(int n)
5 {
6     int result = 1;
7     for (int i = 2; i <= n; ++i){
8         result *= i;
9     }
10    return result;
11 }
12
13 // Entry point to program
14 int main(int argc, char* argv[]){
15
16     assert(fac(5) == 120 && "compile-time check");
17
18     return 0;
19 }
20
"runtime_factorial.cpp" 20L, 320B written
```

run-time factorial (no constexpr)

```
1 // g++ -g -Wall -std=c++20 compiletime_factorial.cpp -o prog
2 #include <cassert>
3
4 constexpr int fac(int n)
5 {
6     int result = 1;
7     for (int i = 2; i <= n; ++i){
8         result *= i;
9     }
10    return result;
11 }
12
13 // Entry point to program
14 int main(int argc, char* argv[]){
15
16     static_assert(fac(5) == 120 && "compile-time check");
17
18     return 0;
19 }
20
"factorial.cpp" 20L, 341B written
```

```
mike:2023_corecpp$ g++ -g -Wall -std=c++20 factorial.cpp -o prog && ./prog
```

Compile-time factorial

#4 constexpr [cppreference]

When comparing the assembly, it becomes immediately obvious which program is doing more computation!

```
1 // g++ -g -Wall -std=c++20 runtime_factorial.cpp -o prog
main:
31 c0
xor    %eax,%eax
c3
ret
66 2e 0f 1f 84 00 00 00 00 00
cs nopw 0x0(%rax,%rax,1)
0f 1f 00
nopl   (%rax)
fac(int):
83 ff 01
cmp    $0x1,%edi
7e 23
jle    401140 <fac(int)+0x28>
83 c7 01
add    $0x1,%edi
b8 02 00 00 00
mov    $0x2,%eax
ba 01 00 00 00
mov    $0x1,%edx
66 0f 1f 44 00 00
nopw   0x0(%rax,%rax,1)
0f af d0
imul   %eax,%edx
83 c0 01
add    $0x1,%eax
39 fa
cmp    %edi,%eax
75 f6
jne    401138 <fac(int)+0x18>
e9 d9
mov    %edx,%eax
c3
ret
0f 1f 00
"runtime_factorial.cpp" 20L, 320B written
```

run-time factorial (no constexpr)

```
main:
31 c0
xor    %eax,%eax
c3
ret
66 2e 0f 1f 84 00 00 00 00 00
cs nopw 0x0(%rax,%rax,1)
0f 1f 00
nopl   (%rax)
12
13 // Entry point to program
14 int main(int argc, char* argv[]){
15     static_assert(fac(5) == 120 && "compile-time check");
17     return 0;
19 }
20
"factorial.cpp" 20L, 341B written
mike:2023_corecpp$ g++ -g -Wall -std=c++20 factorial.cpp -o prog && ./prog
```

Compile-time factorial

#5 Static Data (1/2)

- Now what happens when we have lots of data that we want to load?
 - We could pay the cost of loading a file, checking if a file exists, allocating memory, computing results in a table, etc.
 - Or we could just embed the data inside the executable
 - Now we are making a compile-time decision on time versus space trade-off
- Our tool for embedding data is 'static'
 - Note: At compile-time we are paying the cost of our time to insert a precomputed result with each compile if we are making changes

```
1 // g++ -g -Wall -std=c++20 static.cpp -o prog
2 // Dump assembly with:
3 // g++ -g -Wall -std=c++20 static.cpp -S And see static.s
4 #include <cassert>
5
6 static long long lookup_factorials[] = {
7     1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800
8 };
9
10
11 // Entry point to program
12 int main(int argc, char* argv[]){
13
14     assert(lookup_factorials[5] == 720 && "compile-time check");
15
16     return 0;
17 }
```

```
1 .file "static.cpp"
2 .text
3 .Ltext0:
4 .data
5 .align 32
6 .type _ZL17lookup_factorials, @object
7 .size _ZL17lookup_factorials, 80
8 _ZL17lookup_factorials:
9 .quad 1
10 .quad 2
11 .quad 6
12 .quad 24
13 .quad 120
14 .quad 720
15 .quad 5040
16 .quad 40320
17 .quad 362880
18 .quad 3628800
```

#5 Static Data (2/2)

- Observe that I can precompute much of the data I need.
 - In fact, if I want to compute the next factorial -- it's faster to do a lookup from the table (i.e. we have 'memoized' part of the solution) our last index, and proceed forward.

```
1 // g++ -g -Wall -std=c++20 static.cpp -o prog
2 // Dump assembly with:
3 // g++ -g -Wall -std=c++20 static.cpp -S And see static.s
4 #include <cassert>
5
6 static long long lookup_factorials[] = {
7     1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800
8 };
9
10
11 // Entry point to program
12 int main(int argc, char* argv[]){
13
14     assert(lookup_factorials[5] == 720 && "compile-time check");
15
16     return 0;
17 }
```

```
1 .file "static.cpp"
2 .text
3 .Ltext0:
4 .data
5 .align 32
6 .type _ZL17lookup_factorials, @object
7 .size _ZL17lookup_factorials, 80
8 _ZL17lookup_factorials:
9 .quad 1
10 .quad 2
11 .quad 6
12 .quad 24
13 .quad 120
14 .quad 720
15 .quad 5040
16 .quad 40320
17 .quad 362880
18 .quad 3628800
```

(Aside)

- There are some tools like bin2h that may also be useful for storing binary data in header files
 - <https://github.com/rinthel/bin2h>
- Note: I believe C23 has #embed
 - So C++ should have some access as well depending on your compiler support

#6 Template Metaprogramming

- We have further tools that can perform computation at compile-time!
 - Well, really templates are our tool for generating code at compile-time!
 - (We pay in 'space' in this case)
 - But we can use templates to choose at compile-time optimal algorithms
 - (see example to the right)
 - Or, we truly can use templates to compute

```
template <int length>
Vector<length>& Vector<length>::operator+=(const Vector<length>& rhs)
{
    for (int i = 0; i < length; ++i)
        value[i] += rhs.value[i];
    return *this;
}
```

```
template <>
Vector<2>& Vector<2>::operator+=(const Vector<2>& rhs)
{
    value[0] += rhs.value[0];
    value[1] += rhs.value[1];
    return *this;
}
```

Observe in this example that someone made a specialization of a vector class to unroll a loop. At compile-time, we can 'choose' the optimal specialized algorithm

https://en.wikipedia.org/wiki/Template_metaprogramming#Compile-time_code_optimization

Summarizing

- **#1 Utilize your compiler**
 - Enable optimizations (i.e. Using -O1, -O2, -O3 -- That's an uppercase letter 'o')
- **#2 Use References**
 - Generally, prefer them when you don't need a pointer!
- **#3 static_assert**
 - Test at compile-time (and potentially save run-time computation)
- **#4 constexpr**
 - Explicitly request to evaluate at compile-time
- **#5 Utilize static storage**
 - Utilize precomputed data in algorithms
- **#6 Template Metaprogramming**
 - Select at compile-time the optimal algorithm
 - Or, can otherwise generate code

Wrapping Up

Summary

- We've discussed a fundamental trade-off in computer science: Time versus Space
 - And we can now start taking that decision into our C++ code at compile-time and run-time
- Hopefully you're leaving with a few introductory tricks on how to compute (or avoid computation) at run-time and/or compile-time
 - The theme of this talk could've been 'moving computation: trade-offs' perhaps
 - But again understand that a key advantage of C++, is our ability to choose where, when, and how much we compute.
 - So just remember, a subset of the 'time and space' trade-off is: 'compile-time versus run-time computation trade-off'

Further resources and training materials

- Sean Parent: Better Code Better Data Structures
 - <https://www.youtube.com/watch?v=sWgDk-o-6ZE>
- API Design for C++ by Martin Reddy
 - See chapter on Copy-on-Write for implementation
- History of Time: Asynchronous C++ - Steven Simpson [ACCU 2017]
 - <https://www.youtube.com/watch?v=Z8tbjyZFAVQ>
- Compiler optimizations
 - <https://compileroptimizations.com/>

A Homework Assignment for Students

- Try computing factorial multiple ways
 - at run-time
 - at compile-time using a precomputed table
 - using templates
- Measure the space of the final binaries for each
- Measure the run-time executing each program
- Measure the time to compile each program
 - What if you split up some of the files? Does that change the compile-time?

Thank you Core C++!



Running Away From Computation - An Introduction

Social: [@MichaelShah](#)
Web: [mshah.io](#)
Courses: [courses.mshah.io](#)
YouTube: [www.youtube.com/c/MikeShah](#)

12:20-13:20, Tue, 6th June 2023

60 minutes | Introductory Audience

Thank you!

Extra