

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

# Optimization Design Patterns

Gold Sponsors

think-cell

QUBE

intel  
software  
Bloomberg

Engineering

10:00-11:00, Fri, 4th Aug. 2023

60 minutes | Introductory Audience



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

# Optimization ~~Design~~ ~~Patterns~~ Strategies

Gold Sponsors

think-cell

QUBE

intel  
software  
Bloomberg

Engineering

10:00-11:00, Fri, 4th Aug. 2023

60 minutes | Introductory Audience



Please do not redistribute slides without prior permission.



# Your Tour Guide for Today

by Mike Shah

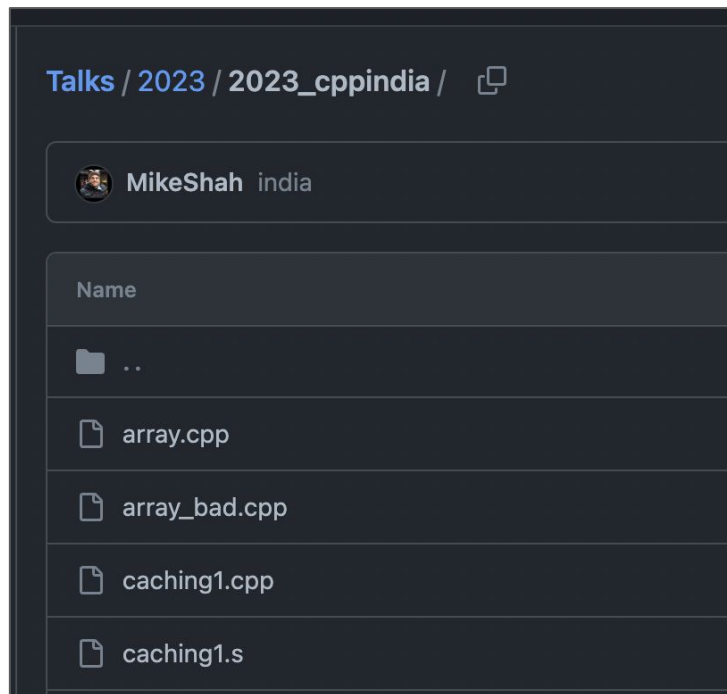
- **Associate Teaching Professor** at Northeastern University in Boston, Massachusetts.
  - I teach courses in computer systems, computer graphics, and game engine development.
  - My **research** in program analysis is related to **performance** building static/dynamic analysis and software visualization 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](http://www.mshah.io)
- More online training at [courses.mshah.io](http://courses.mshah.io)





# Code for the talk

- Located here: [https://github.com/MikeShah/Talks/tree/main/2023/2023\\_cppindia](https://github.com/MikeShah/Talks/tree/main/2023/2023_cppindia)



Name
..
array.cpp
array_bad.cpp
aching1.cpp
aching1.s



..



array.cpp



array\_bad.cpp



aching1.cpp



aching1.s



## Abstract

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

"Premature optimization is the root of all evil" is a saying credited to Donald Knuth that speaks to many programmers with experience -- now anecdotally I have observed folks overlooking the next sentence stating: "Yet we should not pass up our opportunities in that critical 3%". In this talk, the audience will be introduced to some common optimization design patterns. I will discuss precomputation, lazy versus eager evaluation, batching, caching, specialization, hinting, hashing, and using your compiler among 'optimization design patterns' that every programmer should be aware of. Examples will be demonstrated in Modern C++, and the goal is for the audience to leave feeling comfortable implementing each optimization design pattern to improve performance of their code.



## Question to Audience:

How many of you have heard this phrase?  
(On the next slide...)



“premature optimization is the  
root of all evil [or at least most  
of it in programming].” -  
Donald Knuth





## Question to Audience:

How many of you have read Knuth's Paper in which this is quoted?



## Structured Programming with go to Statements (1/3)

- The original paper is filled with lots of gems (including the famous quoted statement)

### Structured Programming with go to Statements

DONALD E. KNUTH

*Stanford University, Stanford, California 94305*

A consideration of several different examples sheds new light on the problem of creating reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without **go to** statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code. The discussion brings out of posing points of view about whether or not **go to** statements should be abolished; some merit is found on both sides of this question. Finally, an attempt is made to define the true nature of structured programming, and to recommend fruitful directions for further study.

**Keywords and phrases:** structured programming, **go to** statements, language design, event indicators, recursion, Boolean variables, iteration, optimization of programs, program transformations, program manipulation systems searching, Quicksort, efficiency

**CR categories:** 4.0, 4.10, 4.20, 5.20, 5.5, 6.1 (5.23, 5.24, 5.25, 5.27)

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3 %. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified. It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail. After working with such tools for seven years, I've become convinced that all compilers written from now on should be designed to provide all programmers with feedback indicating what parts of their programs are costing the most; indeed, this feedback should be supplied automatically unless it has been specifically turned off.



## Structured Programming with go to Statements (2/3)

- There's also this one too right after!

### Structured Programming with go to Statements

DONALD E. KNUTH

*Stanford University, Stanford, California 94305*

A consideration of several different examples sheds new light on the problem of creating reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without **go to** statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code. The discussion brings out opposing points of view about whether or not **go to** statements should be abolished; some merit is found on both sides of this question. Finally, an attempt is made to define the true nature of structured programming, and to recommend fruitful directions for further study.

**Keywords and phrases:** structured programming, **go to** statements, language design, event indicators, recursion, Boolean variables, iteration, optimization of programs, program transformations, program manipulation systems searching, Quicksort, efficiency

**CR categories:** 4.0, 4.10, 4.20, 5.20, 5.5, 6.1 (5.23, 5.24, 5.25, 5.27)

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3 %.** A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified. It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail. After working with such tools for seven years, I've become convinced that all compilers written from now on should be designed to provide all programmers with feedback indicating what parts of their programs are costing the most; indeed, this feedback should be supplied automatically unless it has been specifically turned off.



## Structured Programming with go to Statements (3/3)

- And where exactly to optimize

### Structured Programming with go to Statements

DONALD E. KNUTH

*Stanford University, Stanford, California 94305*

A consideration of several different examples sheds new light on the problem of creating reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without **go to** statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code. The discussion brings out opposing points of view about whether or not **go to** statements should be abolished; some merit is found on both sides of this question. Finally, an attempt is made to define the true nature of structured programming, and to recommend fruitful directions for further study.

**Keywords and phrases:** structured programming, **go to** statements, language design, event indicators, recursion, Boolean variables, iteration, optimization of programs, program transformations, program manipulation systems searching, Quicksort, efficiency

**CR categories:** 4.0, 4.10, 4.20, 5.20, 5.5, 6.1 (5.23, 5.24, 5.25, 5.27)

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3 %.** A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified. **It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail.** After working with such tools for seven years, I've become convinced that all compilers written from now on should be designed to provide all programmers with feedback indicating what parts of their programs are costing the most; indeed, this feedback should be supplied automatically unless it has been specifically turned off.



(potentially bad if facing a new challenge)

“**premature** optimization is the  
root of all evil” ‘But never  
optimizing when the  
opportunity is available is also  
evil’

- This is how I paraphrase  
Knuth to my students

(\*Again, Knuth is not saying to never optimize)





# Optimization is Tricky

(You're going to see in my examples!)



# More from Knuth [[Original Paper link](#)] (1/4)

(From Knuth's paper)

## Structured Programming with `go to` Statements

DONALD E. KNUTH

Stanford University, Stanford, California 94305

A consideration of several different examples sheds new light on the problem of creating reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without `go to` statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code. The discussion brings out opposing points of view about whether or not `go to` statements should be abolished; some merit is found on both sides of this question. Finally, an attempt is made to define the true nature of structured programming, and to recommend fruitful directions for further study.

**Keywords and phrases:** structured programming, `go to` statements, language design, event indicators, recursion, Boolean variables, iteration, optimization of programs, program transformations, program manipulation systems searching, Quicksort, efficiency

**CR categories:** 4.0, 4.10, 4.20, 5.20, 5.5, 6.1 (5.23, 5.24, 5.25, 5.27)

*This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without `go to` statements; (b) **a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code.***



# More from Knuth [[Original Paper link](#)] (2/4)

- Optimization *might* result in you making trade-offs beyond space and time
  - e.g. readability, maintainability, and sometimes even correctness/precision

## Structured Programming with `go to` Statements

DONALD E. KNUTH

Stanford University, Stanford, California 94305

A consideration of several different examples sheds new light on the problem of creating reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without `go to` statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code. The discussion brings out opposing points of view about whether or not `go to` statements should be abolished; some merit is found on both sides of this question. Finally, an attempt is made to define the true nature of structured programming, and to recommend fruitful directions for further study.

**Keywords and phrases:** structured programming, `go to` statements, language design, event indicators, recursion, Boolean variables, iteration, optimization of programs, program transformations, program manipulation systems searching, Quicksort, efficiency

**CR categories:** 4.0, 4.10, 4.20, 5.20, 5.5, 6.1 (5.23, 5.24, 5.25, 5.27)

*This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without `go to` statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code.*



# More from Knuth [[Original Paper link](#)] (3/4)

- However, I might add, sometimes the simplest code is the most optimized!
  - It's easiest for the hardware to predict -- so we really have to know the whole software and hardware stack!

## Structured Programming with go to Statements

DONALD E. KNUTH

Stanford University, Stanford, California 94305

A consideration of several different examples sheds new light on the problem of creating reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without go to statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code. The discussion brings out opposing points of view about whether or not go to statements should be abolished; some merit is found on both sides of this question. Finally, an attempt is made to define the true nature of structured programming, and to recommend fruitful directions for further study.

**Keywords and phrases:** structured programming, go to statements, language design, event indicators, recursion, Boolean variables, iteration, optimization of programs, program transformations, program manipulation systems searching, Quicksort, efficiency

**CR categories:** 4.0, 4.10, 4.20, 5.20, 5.5, 6.1 (5.23, 5.24, 5.25, 5.27)

*This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without go to statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code.*



# More from Knuth [Original]

- However, I might add, sometimes the simplest code is the most optimized!
  - It's easiest for the hardware to predict -- so we really have to know the whole software and hardware stack!

## Structured Programming with go to Statements

DONALD E. KNUTH

Stanford University, Stanford, California 94305

A consideration of several different examples sheds new light on the problem of creating reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without go to statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code. The discussion brings out opposing points of view about whether or not go to statements should be abolished; some merit is found on both sides of this question. Finally, an attempt is made to define the true nature of structured programming, and to recommend fruitful directions for further study.

**Keywords and phrases:** structured programming, go to statements, language design, event indicators, recursion, Boolean variables, iteration, optimization of programs, program transformations, program manipulation systems searching, Quicksort, efficiency

**CR categories:** 4.0, 4.10, 4.20, 5.20, 5.5, 6.1 (5.23, 5.24, 5.25, 5.27)

- Will some patterns that I demonstrate obfuscate and make your code harder to read?

- Maybe (though they are simple for today's introduction)
- But hopefully you'll become familiar with some tools to help you choose the right optimization strategy.

*statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code.*



# (Aside) CPU, Hard drive, and general Architecture

- This talk is too short to discuss how hardware works -- BUT there are some great talks you could watch to get up to speed and are also performance related
  - code::dive conference 2014 - Scott Meyers: Cpu Caches and Why You Care
    - <https://www.youtube.com/watch?v=WDIkqP4JbkE>
  - CppCon 2014: Mike Acton "Data-Oriented Design and C++"
    - <https://www.youtube.com/watch?v=rX0ltVEVjHc>
  - CppCon 2016: Timur Doumler "Want fast C++? Know your hardware!"
    - <https://www.youtube.com/watch?v=BP6NxVxDQIs>
  - CppCast Episode 287: Trading Systems with Carl Cook
    - <https://youtu.be/nmIJqiOtWSs?t=948> (Specifically on the challenges)
  - "Performance Matters" by Emery Berger
    - <https://www.youtube.com/watch?v=r-TLSBdHe1A>
  - CppCon 2016: Chandler Carruth "High Performance Code 201: Hybrid Data Structures"
    - <https://www.youtube.com/watch?v=vElZc6zSIXM>



# (Aside) Compiler Optimizations

- Compilers aren't really a pattern but a great place to look for 'themes' in how to write fast code.
  - It's good to be familiar with compiler optimizations so you know these themes.
    - (It will help you hand tune code as well)
  - It's good to be familiar with compiler optimizations so you know what they will do with certainty for you
  - Run the different optimization levels is a good skill for new programmers to know about.

## Types of optimization

Factors affecting optimization

Common themes

✓ Specific techniques

Loop optimizations

Data-flow optimizations

SSA-based optimizations

Code generator optimizations

Functional language optimizations

Other optimizations

Interprocedural optimizations

Practical considerations

[https://en.wikipedia.org/wiki/Optimizing\\_compiler](https://en.wikipedia.org/wiki/Optimizing_compiler)



Goal(s) for today



# What you're going to learn today

- Today this talk is a 'grab-bag' of optimization design strategies that may (or may not) improve the performance of your code.
  - At the least, you'll know a few strategies that exist and that you can try to apply to your code today!



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.  
They are very small 'microbenchmarks' for learning.

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

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



# Optimization Patterns

(Or really strategies/trade-offs)



# Optimization Patterns/Strategies/Trade-offs (1/2)

- 'Patterns' are 'blueprints' or 'recipes' that might help solve a problem
- When it comes to optimizations, I think there are a few strategies that can be useful
  - It's probably more accurate to however describe these as 'strategies' or 'trade-offs' for obtaining more of something (where 'more' today is usually faster execution).
- How I determine a pattern, needs further academic formalization -- I'm not necessarily looking for bit hacks (e.g.  $a * 2$  versus  $a \ll 2$ )
  - But rather opportunities where I am trading **space** for **time**.



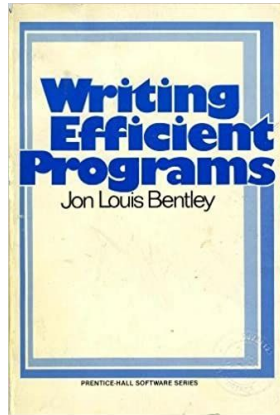
## Optimization Patterns/Strategies/Trade-offs (2/2)

- 'Patterns' are '... problem
- When it comes to strategies that can be useful
  - It's probably obtaining mo
- How I determine or 'trade-offs' for (cution).  
necessarily loc ation -- I'm not
  - But rather op

**Let me summarize this for you in the next slide(s)**

**(Note: I'll run through the next 20 or so slides quickly and you can review them in detail later)**





# From John Bentley's Rules of Performance

Next few slides based off of MIT's Performance Engineering course and my 2020 Performance Engineering course


- [Space-for-Time Rules](#)
  1. [Data Structure Augmentation](#)
  2. [Store Precomputed Results](#)
  3. [Caching](#)
- [Time-for-Space Rules](#)
  1. [Packing](#)
  2. [Interpreters](#)
- [Loop Rules](#)
  1. [Code Motion Out of Loops](#)
  2. [Combining Tests](#)
  3. [Loop Unrolling](#)
  4. [Transfer-Driven Loop Unrolling](#)
  5. [Unconditional Branch Removal](#)
  6. [Loop Fusion](#)
- [Logic Rules](#)
  1. [Exploit Algebraic Identities](#)
  2. [Short-Circuit Monotone Functions](#)
  3. [Reorder Tests](#)
  4. [Precompute Logical Functions](#)
  5. [Control Variable Elimination](#)
- [Procedure Design Rules](#)
  1. [Collapse Procedure Hierarchies](#)
  2. [Exploit Common Cases](#)
  3. [Use Coroutines](#)
  4. [Transform Recursive Procedures](#)
  5. [Use Parallelism](#)
- [Expression Rules](#)
  1. [Initialize Data Before Runtime](#)
  2. [Exploit Algebraic Identities](#)
  3. [Eliminate Common Subexpressions](#)
  4. [Combine Paired Computation](#)
  5. [Exploit Word Parallelism](#)



# Trade-offs

There are a few key trade-offs we can make on data structures:

- Space-for-time
- Time for Space
- Space and Time

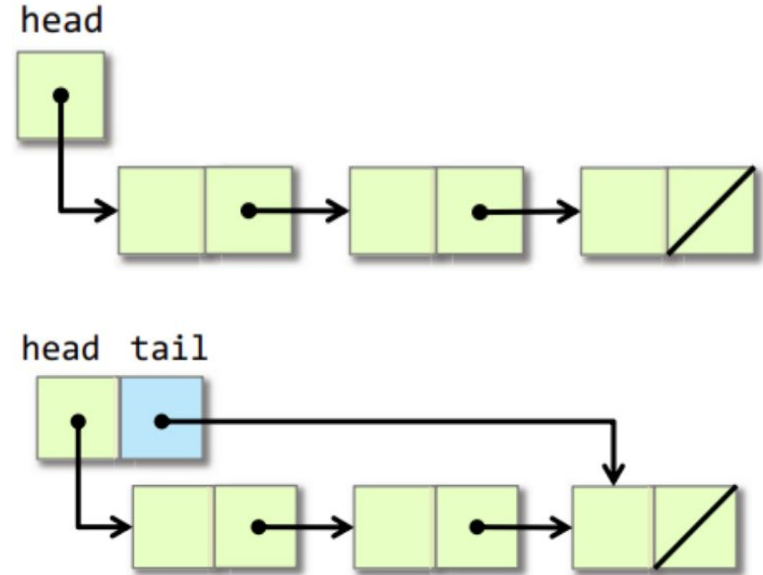
As I sometimes say, “Computer Science is all about understanding trade-offs” -  
Mike 

- (And sometimes--you are lucky enough to get both space and time benefits!)



# Modifying Data - Space-for-time | Data Structure Augmentation (1/2)

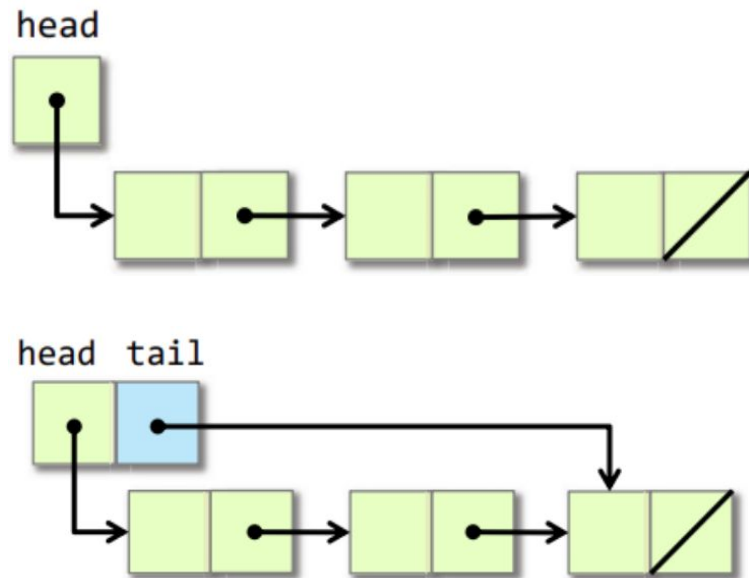
- You can add information to a data structure to make common operations faster
  - e.g. Singly Linked List 'append'
  - Normally appending requires walking the entire linked list and appending at the end of the linked list a new node
  - Can be sped up by adding a 'tail' pointer to directly access the tail





# Modifying Data - Space-for-time | Data Structure Augmentation (2/2)

- You can add information to a data structure to make common operations faster
  - e.g. Singly Linked List 'append'
  - Normally appending requires walking the entire linked list and appending at the end of the linked list a new node
  - Can be sped up by adding a 'tail' pointer to directly access the tail
    - Small memory cost overall of maintaining one additional pointer

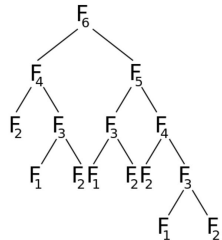


```
8 class LinkedList
9 {
10 private:
11     node* m_head;
12     node* m_tail;
```



# Modifying Data - Space-for-time | Store Pre-computed Result (1/3)

- The example on the right shows computing the 'nth' fibonacci number
  - And we compute the result multiple times throughout our program
  - This operation costs  $O(2^N)$  time
    - (or about 10 seconds on my machine running this program)
    - The reason is we are recomputing the same results frequently.



Recursion tree for Fibonacci of 6

```
1 // Compile: gcc fib.c -o fib
2 // Run with: time ./fib
3 #include <stdio.h>
4
5 long fib(long n){
6     if(n<=1){
7         return 1;
8     }
9     // Recursive call
10    return fib(n-1) + fib(n-2);
11 }
12
13
14 int main(){
15
16     const long NthNumberToCompute = 45;
17
18     long fibComputation1 = fib(NthNumberToCompute);
19     long fibComputation2 = fib(NthNumberToCompute);
20
21     printf("fibComputation1: %ld\n", fibComputation1);
22     printf("fibComputation2: %ld\n", fibComputation2);
23
24     return 0;
25 }
```

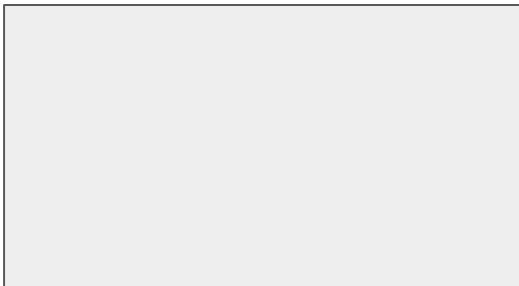
```
mike:code$ time ./fib
fibComputation1: 1836311903
fibComputation2: 1836311903

real    0m9.204s
user    0m9.203s
sys     0m0.000s
```



# Modifying Data - Space-for-time | Store Pre-computed Result (2/3)

- We can speed up Fibonacci by caching the result (Memoization)
  - This optimization works because we:
    - Have a generally expensive function
    - The argument space is relatively small (1 argument of integer type)
    - Function has no side effects
    - Function is deterministic
- Drumroll for the result....



```
1 // Compile: gcc fib_table.c -o fib_table
2 // Run with: time ./fib_table
3 #include <stdio.h>
4
5 #define PRECOMPUTED_VALUES 100
6 long FIB_TABLE[PRECOMPUTED_VALUES];
7
8 long initialize_table(){
9     for(long i= 0; i < PRECOMPUTED_VALUES; i++){
10         // Store as a sentinel value so we know
11         // it has not been precomputed
12         FIB_TABLE[i] = -1;
13     }
14 }
15
16
17 long fib_memo(long n){
18     // lookup value first and return result
19     // if it has not been previously computed
20     if(FIB_TABLE[n] != -1){
21         return FIB_TABLE[n];
22     }
23     // base case
24     if(n<=1){
25         FIB_TABLE[n] = 1;
26         return 1;
27     }
28     // Recursive call
29     FIB_TABLE[n] = fib_memo(n-1) + fib_memo(n-2);
30     return FIB_TABLE[n];
31 }
32
33
34 int main(){
35
36     const long NthNumberToCompute = 45;
37     // Setupt our table
38     initialize_table();
39
40     long fibComputation1 = fib_memo(NthNumberToCompute);
41     long fibComputation2 = fib_memo(NthNumberToCompute);
42
43     printf("fibComputation1: %ld\n", fibComputation1);
44     printf("fibComputation2: %ld\n", fibComputation2);
45
46     return 0;
47 }
```



# Modifying Data - Space-for-time | Store Pre-computed Result (3/3)

- We can speed up Fibonacci by caching the result (Memoization)
  - This optimization works because we:
    - Have a generally expensive function
    - The argument space is relatively small (1 argument of integer type)
    - Function has no side effects
    - Function is deterministic
- Drumroll for the result....

```
mike:code$ time ./fib_table
fibComputation1: 1836311903
fibComputation2: 1836311903

real    0m0.002s
user    0m0.002s
sys     0m0.000s
```

```
1 // Compile: gcc fib_table.c -o fib_table
2 // RUN with: time ./fib_table
3 #include <stdio.h>
4
5 #define PRECOMPUTED_VALUES 100
6 long FIB_TABLE[PRECOMPUTED_VALUES];
7
8 long initialize_table(){
9     for(long i= 0; i < PRECOMPUTED_VALUES; i++){
10         // Store as a sentinel value so we know
11         // it has not been precomputed
12         FIB_TABLE[i] = -1;
13     }
14 }
15
16
17 long fib_memo(long n){
18     // lookup value first and return result
19     // if it has not been previously computed
20     if(FIB_TABLE[n] != -1){
21         return FIB_TABLE[n];
22     }
23     // base case
24     if(n<=1){
25         FIB_TABLE[n] = 1;
26         return 1;
27     }
28     // Recursive call
29     FIB_TABLE[n] = fib_memo(n-1) + fib_memo(n-2);
30     return FIB_TABLE[n];
31 }
32
33
34 int main(){
35
36     const long NthNumberToCompute = 45;
37     // Setupt our table
38     initialize_table();
39
40     long fibComputation1 = fib_memo(NthNumberToCompute);
41     long fibComputation2 = fib_memo(NthNumberToCompute);
42
43     printf("fibComputation1: %ld\n", fibComputation1);
44     printf("fibComputation2: %ld\n", fibComputation2);
45
46     return 0;
47 }
```



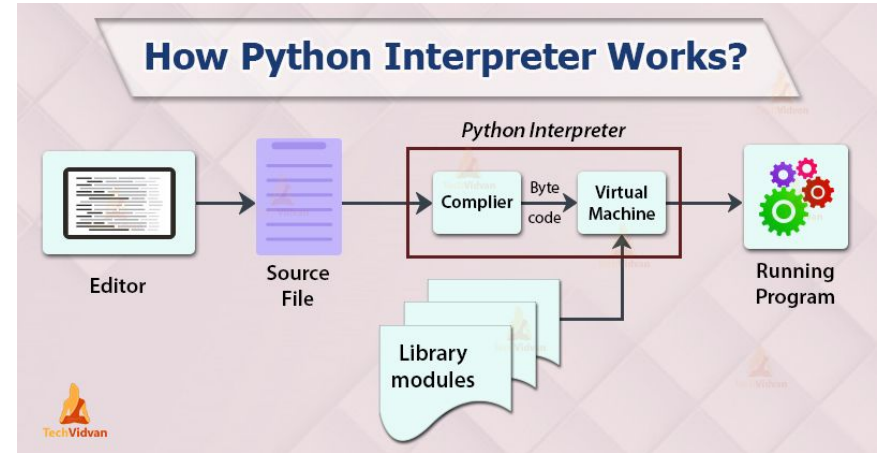
# Modifying Data - Time-for-Space | Packing/Compression

- Reduce space of data by storing processed results
  - e.g. Data compression(e.g. .zip, .rar) by eliminating repetitions ([LZ77](#))
  - Just a cool example: <https://www.youtube.com/watch?v=2NBG-sKFaB0>
- Useful on embedded devices for example
- Or if you are trying to limit bandwidth usage on networked applications
- Other practical tips
  - Use smaller data sizes
    - i.e. If your range is only 0-255, use a char not an 'int' --
      - (Aside: very common use case for storing RGB color values for instance, and frequently I see folks use 'int')



# Modifying Data - Time-for-Space | Interpreters

- e.g. Python
  - It's an interpreted language (reads byte code)
  - No need to generate binaries (.o, .exe, etc.) files, just need the source code!
  - The language thus describes the computation, no need to store opcodes
- Does not have to be a full language either
  - Could be reading in data from a file during run-time for example as opposed to storing in the binary.





# Modifying Data - Space-and-Time | Packing (1/2)

- We try to store (or encode) more data into a machine word
  - Why does it make things faster?
    - This results in less 'fetches' to memory for data.
    - (This is also more space efficient!)
- Here's an example using 'bit fields' in C.

```
1 // 96-bit representation of date
2 // 1 int is 32 bits (4 bytes)
3 // 3 ints thus is 3*32 = 96 bits.
4 typedef struct{
5     int year;
6     int month;
7     int day;
8 } date_t;
```

```
1 // 22-bit representation of date
2 typedef struct{
3     int year: 13; // 2^13 or 8192 years [-4096-4095]
4     int month: 4; // 4 bits is 2^4=16 for months
5     int day: 5;   // 5 bits--months of 32 days max
6 } date_t;
```

(Slight caveat, that compiler may 'pad' struct to align it better to say 32-bits)



# Modifying Data - Space-and-Time | Packing (2/2)

- We try to store (or encode) more data into a machine word
  - Why does it make things faster?
    - This results in less 'fetches' to memory for data.
    - (This is also more space efficient!)
- Here's an example using 'bit fields' in C.

```
1 // 96-bit representation of date
2 // 1 int is 32 bits (4 bytes)
3 // 3 ints thus is 3*32 = 96 bits.
4 typedef struct{
5     int year;
6     int month;
7     int day;
8 } date_t;
```

```
1 // 22-bit representation of date
2 typedef struct{
3     int year: 13; // 2^13 or 8192 years [-4096-4095]
4     int month: 4; // 4 bits is 2^4=16 for months
5     int day: 5;   // 5 bits--months of 32 days max
6 } date_t;
```

(Slight caveat, that compiler may 'pad' struct to align it better to say 32-bits)

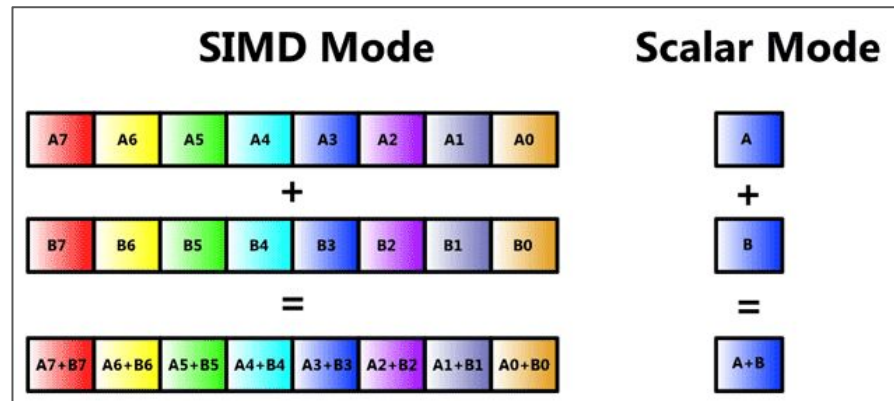
Second caveat--decoding (unpacking) may take more time--in which case the optimization may involve more work if you have to decode before using this data.

More: [https://compileroptimizations.com/category/bitfield\\_optimization.htm](https://compileroptimizations.com/category/bitfield_optimization.htm)



# Modifying Data - Space-and-Time | SIMD

- Single Instruction Multiple Data
  - Execute a single operation on multiple data items
  - Both faster and less storage
- Can be used
  - If same operation is used on all data items.
  - (We'll explore this a bit more later in the course!)





# Modifying the Code Structure



# Modifying Code

There are a few key trade-offs we can make on how we structure our code:

- Loops
- Logic
- Functions (Procedures)
- Expressions
- Parallelism
  - (We'll discuss in future lecture as they are more architecture specific)

Some of these are common enough, our compilers can actually assist us as well!



# Modifying Code | Loops

- Loops are especially important to optimize?
- Why--because we spend so much of our time executing in loops
- Let's look at a few optimizations within loops
  - Code Motion
  - Sentinel Loop Exit Test
  - Loop Unrolling
  - Partial Loop Unrolling
  - Loop Fusion



# Modifying Code | Loops -- Code Motion (1/3)

- Move code outside of loop that does not need to be recomputed.
  - More on lazy code motion: <https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/lazy-code-motion/>

```
1 // gcc code_motion_off -o code_motion_off
2 // time ./code_motion_off
3 #define ITERATIONS 1000000
4
5
6 double approx_pi(){
7     return 22.0/7.0;
8 }
9
10 int main(){
11
12     double circumferences[ITERATIONS];
13     for(int i=0; i < ITERATIONS; i++){
14         // 2 * PI * r = circumference
15         circumferences[i] = 2*approx_pi()*i;
16     }
17
18     return 0;
19 }
```



```
1 // gcc code_motion_on -o code_motion_on
2 // time ./code_motion_on
3 #define ITERATIONS 1000000
4
5
6 double approx_pi(){
7     return 22.0/7.0;
8 }
9
10 int main(){
11
12     double circumferences[ITERATIONS];
13     double PI_times_2 = 2*approx_pi();
14     for(int i=0; i < ITERATIONS; i++){
15         // 2 * PI * r = circumference
16         circumferences[i] = PI_times_2*i;
17     }
18
19     return 0;
20 }
```



# Modifying Code | Loops -- Code Motion (2/3)

- Move code outside of loop that does not need to be recomputed.
  - More on lazy code motion: <https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/lazy-code-motion/>)

```
1 // gcc code_motion off
2 // time ./code_motion
3 #define ITERATIONS 10
4
5
6 double approx_pi(){
7     return 22.0/7.0;
8 }
9
10 int main(){
11
12
13
14
15
16
17
18     return 0;
19 }
```

Careful however! Experimental results show code motion made this example slower!

Why could this be?

```
mike:code$ gcc code_motion_off.c -o code_motion_off
mike:code$ time ./code_motion_off

real    0m0.009s
user    0m0.000s
sys     0m0.009s
```

```
10 int main(){
11
12
13
14
15
16
17
18
19     return 0;
20 }
```

```
mike:code$ gcc code_motion_on.c -o code_motion_on
mike:code$ time ./code_motion_on

real    0m0.019s
user    0m0.011s
sys     0m0.008s
mike:code$ time ./code_motion_on
```



# Modifying Code | Loops -- Code Motion (3/3)

- Move code outside of loop that does not need to be recomputed.
  - More on lazy code motion: <https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/lazy-code-motion/>

Careful however! Experimental results show code motion made this example slower!

Why could this be?

- Memory fetches (reads of variable) might be more expensive!
  - Actual computation is thus not that costly to perform each iteration (sqrt, or some other operation may be however)
- We need to see the assembly if the compiler would actually perform this optimization!

```
mike:code$ gcc code_motion_off.c -o code_motion_off
mike:code$ time ./code_motion_off

real    0m0.009s
user    0m0.000s
sys     0m0.009s
```

```
mike:code$ gcc code_motion_on.c -o code_motion_on
mike:code$ time ./code_motion_on

real    0m0.019s
user    0m0.011s
sys     0m0.008s
mike:code$ time ./code_motion_on
```



# Modifying Code | Loops -- Sentinel Loop Exit Test

- Exiting early is another way to save on performance--no need to continue iterating through the entire collection when a value is found.

```
1 // Return 'some' index of a character found
2 int indexOf(char* str, char ch, int size){
3     int index = -1;
4     for(int i =0; i < size; i++){
5         if(str[i]==ch){
6             index =i;
7         }
8     }
9     return index;
10 }
```

```
1 int indexOf(char* str, char ch, int size){
2     for(int i =0; i < size; i++){
3         if(str[i]==ch){
4             return i;
5         }
6     }
7
8     return -1;
9 }
```



# Modifying Code | Loops -- Loop Unrolling (elimination of the loop)

- Small loops can be ‘unrolled’ to avoid comparison computations.
  - Generally something the compiler will figure out for you--but you can control this by doing it yourself.:w

```
1 // gcc loop_unroll.c -o loop_unroll
2
3 int main(){
4
5     // We can unroll this loop
6     int sum =0;
7     int A[4] = {1,2,3,4};
8
9     for(int i=0; i < 4; i++){
10         sum = sum + A[i];
11     }
12     // to...
13     sum = A[0] + A[1] + A[2] + A[3];
14
15
16     return 0;
17 }
```



# Modifying Code | Loops -- Partial Loop Unrolling

- A similar idea where we can partially unroll the loop
  - Can be especially powerful when combined with SIMD

```
1 // gcc partial_loop_unroll.c -o partial_loop_unroll
2
3 int main(){
4
5     // We can partially unroll this loop
6     int sum =0;
7     int A[4] = {1,2,3,4};
8
9     for(int i=0; i < 4; i++){
10         sum = sum + A[i];
11     }
12     // to...
13     for(int i=0; i < 4; i+=2){
14         sum = sum + A[i];
15         sum = sum + A[i+1];
16     }
17
18
19     return 0;
20 }
```



# Modifying Code | Loops -- Loop Fusion

- We can merge loops together that are otherwise performing independent computations.

```
for (i = 0; i < 300; i++)  
    a[i] = a[i] + 3;  
  
for (i = 0; i < 300; i++)  
    b[i] = b[i] + 4;
```

Below is the code fragment after loop fusion.

```
for (i = 0; i < 300; i++)  
{  
    a[i] = a[i] + 3;  
    b[i] = b[i] + 4;  
}
```



# Modifying Code | Logical Expression - Strength Reduction

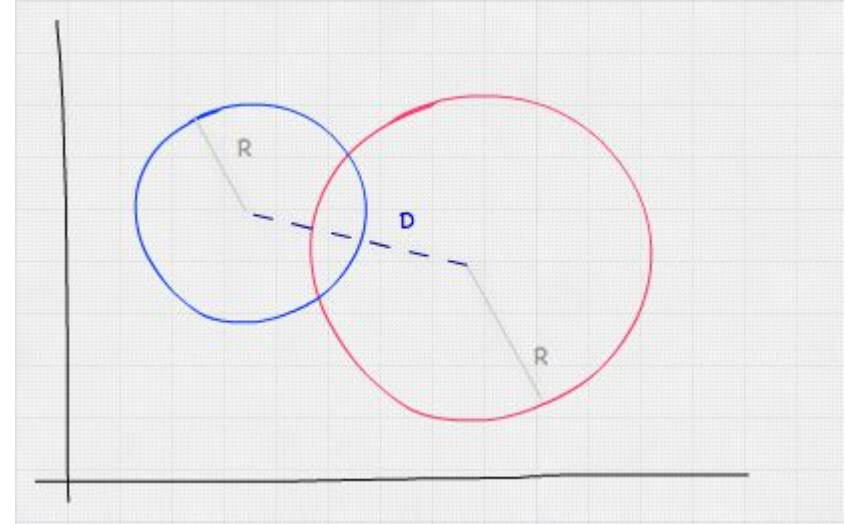
- Occasionally we can make a better substitution that logically gives us the same control flow

$\text{sqrt}(x) > 0$	$x \neq 0$
$\text{sqrt}(x*x + y*y) < \text{sqrt}(a*a + b*b)$	$x*x + y*y < a*a + b*b$
$\ln(A) + \ln(B)$	$\ln(A*B)$
$\sin(x)*\sin(x) + \cos(x)*\cos(x)$	1



# Modifying Code | Logic - Reorder Tests (1/2)

- Logical tests should be arranged so that inexpensive and often successful tests precede expensive and rarely successful tests.

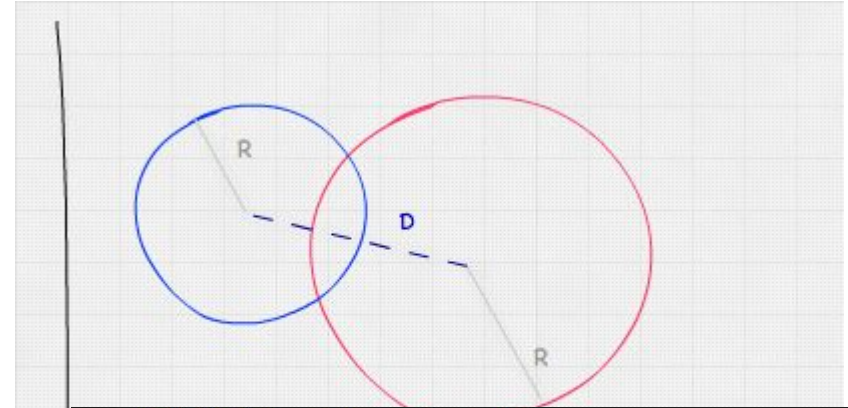


```
3 // Checks if a collision occurred
4 if( sqrt(sqr(x1-x2) + sqr(y1-y2)) < (r1 + r2)){
5     return 1;
6 }else{
7     return 0;
8 }
```



# Modifying Code | Logic - Reorder Tests (2/2)

- Logical tests should be arranged so that inexpensive and often successful tests precede expensive and rarely successful tests.



```
3 // Checks if a collision occurred
4 if( sqrt(sqr(x1-x2) + sqr(y1-y2)) < (r1 + r2)){
5     return 1;
6 }else{
7     return 0;
8 }
```

```
13 if(abs(x1-x2) > r1 + r2){
14     return 0; // fast exit
15 }
16 if(abs(y1-y2) > r1 + r2){
17     return 0; // fast exit
18 }
19 if( sqrt(sqr(x1-x2) + sqr(y1-y2)) < (r1 + r2)){
20     return 1;
21 }else{
22     return 0;
23 }
```



# Modifying Code | Procedures - Inlining

- Eliminates function call overhead by moving small functions into body of code.
- Also provides further optimization opportunities for compilers to perform after the inlining takes place.
  - Generally speaking this is one of the biggest optimizations, because we often (not always) optimize on a function level.
  - [https://compileroptimizations.com/category/function\\_inlining.htm](https://compileroptimizations.com/category/function_inlining.htm)

```
int add (int x, int y)
{
    return x + y;
}
```

```
int sub (int x, int y)
{
    return add (x, -y);
}
```

Expanding add() at the call site in sub() yields:

```
int sub (int x, int y)
{
    return x + -y;
}
```

which can be further optimized to:

```
int sub (int x, int y)
{
    return x - y;
}
```



# Modifying Code | Expression Rules - Constant Propagation

- Simply propagate the result
  - This may also save us on both time and space of computing and storing intermediate values.

In the code fragment below, the value of x can be propagated to the use of x.

```
x = 3;  
y = x + 4;
```

Below is the code fragment after constant propagation and constant folding.

```
x = 3;  
y = 7;
```



# Modifying Code | Expression Rules - Compile-Time Initialization

- If a value is constant, we can make a compile-time constant
  - We'll see 'constexpr' in C++
- Saves the effort of computation
- This may allow us to perform further constant propagation
- Again enables further optimizations!

```
// constexpr function for product of two numbers.  
// By specifying constexpr, we suggest compiler to  
// to evaluate value at compile time  
constexpr int product(int x, int y)  
{  
    return (x * y);  
}  
  
int main()  
{  
    const int x = product(10, 20);  
    cout << x;  
    return 0;  
}
```

<https://www.geeksforgeeks.org/understanding-constexpr-specifier-in-c/>



# Vec3 Class

Example of more performance patterns and how to possibly iterate through optimizations



# VecN class

- So to ground us in some simple examples to learn from, let's start with a class like this
  - It's an 'n-element' vector where we have a few member functions
  - We'll use a `std::vector` to store individual elements.
  - The data structure is also templated so that we can consider storing any type.

```
1 // g++ -g -Wall -std=c++20 vecN.cpp -o prog && ./prog
2 #include <iostream>
3 #include <vector>
4
5 // Generic n-dimensional mathematical vector
6 template<typename T>
7 struct VecN{
8     // ===== Member Variables =====
9     // Store individual components
10    std::vector<T> components;
11
12    // ===== Member Functions =====
13    // Constructor
14    VecN(size_t elements);
15    // Print out the components
16    void Print() const;
17    // Add some components
18    VecN<T>& operator+=(const VecN<T>& rhs);
19 };
```



# Optimization Strategy/Pattern #1: **Caching**

A run-time space versus run-time trade-off



# Vec3N Member Functions

- I've gone ahead and implemented three member functions

```
21 // Constructor
22 template<typename T>
23 VecN<T>::VecN(size_t elements){
24     // Initialize components
25     for(size_t i=0; i < elements; ++i){
26         components.push_back(i);
27     }
28 }
```

```
29
30 // Print
31 template<typename T>
32 void VecN<T>::Print() const{
33     for(size_t i=0; i < components.size(); ++i){
34         std::cout << components.at(i) << "." << std::endl;
35     }
36 }
```

```
37
38 // generic addition overload
39 template<typename T>
40 VecN<T>& VecN<T>::operator+=(const VecN<T>& rhs){
41     for(size_t i=0; i < components.size(); ++i){
42         components[i] += rhs.components[i];
43     }
44     return *this;
45 }
```



# Recomputation

- It appears I am recomputing work very frequently however!
- Question to Audience:  
Anyone spot where?
  - (ans: next slide)

```
21 // Constructor
22 template<typename T>
23   VecN<T>::VecN(size_t elements){
24     // Initialize components
25     for(size_t i=0; i < elements; ++i){
26       components.push_back(i);
27     }
28 }
29
30 // Print
31 template<typename T>
32 void VecN<T>::Print() const{
33   for(size_t i=0; i < components.size(); ++i){
34     std::cout << components.at(i) << "." << std::endl;
35   }
36 }
37
38 // generic addition overload
39 template<typename T>
40 VecN<T>& VecN<T>::operator+=(const VecN<T>& rhs){
41   for(size_t i=0; i < components.size(); ++i){
42     components[i] += rhs.components[i];
43   }
44   return *this;
45 }
```



# Recomputation

- It appears I am recomputing work very frequently however!
- Question to Audience:  
Anyone spot where?
  - I'm constantly calling `components.size()` every iteration of every loop
  - For a 'print' function (which is likely `const`) why would I need to do this?

```
21 // Constructor
22 template<typename T>
23   VecN<T>::VecN(size_t elements){
24     // Initialize components
25     for(size_t i=0; i < elements; ++i){
26       components.push_back(i);
27     }
28 }
29
30 // Print
31 template<typename T>
32 void VecN<T>::Print() const{
33     for(size_t i=0; i < components.size(); ++i){
34         std::cout << components.at(i) << "." << std::endl;
35     }
36 }
37
38 // generic addition overload
39 template<typename T>
40 VecN<T>& VecN<T>::operator+=(const VecN<T>& rhs){
41     for(size_t i=0; i < components.size(); ++i){
42         components[i] += rhs.components[i];
43     }
44     return *this;
45 }
```



# Caching

- So here's the adjustment we can make before the loop.
  - For vectors of very large 'n' this may make some difference having 'len' directly on the stack (we'll have to measure)
- Note: pragmatically -- for a vector -- .size() is just a lookup and already optimized -- this function is probably 'inlined'.
  - Presumably for a 'graph' or some more complicated linked data structure traversal it may be worth performing this specific optimization.

```
21 // Constructor
22 template<typename T>
23 VecN<T>::VecN(size_t elements){
24     // Initialize components
25     for(size_t i=0; i < elements; ++i){
26         components.push_back(i);
27     }
28 }
29
30 // Print
31 template<typename T>
32 void VecN<T>::Print() const{
33     size_t len = components.size();
34     for(size_t i=0; i < len; ++i){
35         std::cout << components.at(i) << "." << std::endl;
36     }
37 }
38
39 // generic addition overload
40 template<typename T>
41 VecN<T>& VecN<T>::operator+=(const VecN<T>& rhs){
42     size_t len = components.size();
43     for(size_t i=0; i < len; ++i){
44         components[i] += rhs.components[i];
45     }
46     return *this;
47 }
```



# Measuring Optimizations



# Measurements (1/2)

- So in order to know if our optimization strategy (caching) worked -- we need to measure each strategy in an experiment
- Here's an example using 'time' running 1\_000\_000 iterations of add.

```
mike:2023_italian_cpp$ g++ -Wall -std=c++20 vecN.cpp -o prog && time ./prog  
0.  
1000001.  
2000002.
```

```
real    0m0.025s  
user    0m0.025s  
sys     0m0.000s
```

No optimization -- takes about 0.025 seconds

```
mike:2023_italian_cpp$ g++ -Wall -std=c++20 caching1.cpp -o prog && time ./prog  
0.  
1000001.  
2000002.
```

```
real    0m0.017s  
user    0m0.016s  
sys     0m0.000s
```

Caching a bit faster -- perhaps small enough that our computation is noise?



## Measurements (2/2)

- So in order to know if our optimization strategy (caching) is better, we need to measure each strategy in an experiment
- Here's an example using 'time' running 1\_000\_000 iterations.

Let's see if we can tease out more information from this using a different profiler

```
mike:2023_italian_cpp$ g++ -Wall -std=c++20 vecN.cpp -o prog && time ./prog
0.
1000001.
2000002.
```

```
real    0m0.025s
user    0m0.025s
sys     0m0.000s
```

No optimization -- takes about 0.025 seconds

```
mike:2023_italian_cpp$ g++ -Wall -std=c++20 caching1.cpp -o prog && time ./prog
0.
1000001.
2000002.
```

```
real    0m0.017s
user    0m0.016s
sys     0m0.000s
```

Caching a bit faster -- perhaps small enough that our computation is noise?



# perf profiler

- We need a more fine grained measurement to try to understand what our optimization strategy did -- otherwise again it may just be noise.
  - The perf profiler is a well known tool on linux, and your platform may otherwise provide other useful tools

```
PERF(1)                                perf Manual                                PERF(1)

NAME
    perf - Performance analysis tools for Linux

SYNOPSIS
    perf [--version] [--help] [OPTIONS] COMMAND [ARGS]
```



# Observing Perf

- So from this output, it appears that we do have:
  - Less instructions executed
  - fewer cpu cycles
  - fewer branches
  - (oddly more branch-misses though!)

```
mike:2023_italian_cpp$ g++ -Wall -std=c++20 vecN.cpp -o prog
mike:2023_italian_cpp$ sudo perf stat ./prog
0.
1000001.
2000002.
```

No optimization

Performance counter stats for './prog':

20.91 msec	task-clock	#	0.988 CPUs utilized
0	context-switches	#	0.000 K/sec
0	cpu-migrations	#	0.000 K/sec
117	page-faults	#	0.006 M/sec
83,507,688	cycles	#	3.994 GHz
213,323,341	instructions	#	2.55 insn per cycle
31,578,068	branches	#	1510.467 M/sec
16,847	branch-misses	#	0.05% of all branches

0.021152750 seconds time elapsed

0.021166000 seconds user

0.000000000 seconds sys

```
mike:2023_italian_cpp$ g++ -Wall -std=c++20 caching1.cpp -o prog
mike:2023_italian_cpp$ sudo perf stat ./prog
0.
1000001.
2000002.
```

Caching

Performance counter stats for './prog':

18.78 msec	task-clock	#	0.983 CPUs utilized
0	context-switches	#	0.000 K/sec
0	cpu-migrations	#	0.000 K/sec
118	page-faults	#	0.006 M/sec
76,223,347	cycles	#	4.059 GHz
165,337,456	instructions	#	2.17 insn per cycle
25,579,300	branches	#	1362.301 M/sec
249,690	branch-misses	#	0.98% of all branches

0.019094558 seconds time elapsed

0.019133000 seconds user

0.000000000 seconds sys



# Perf - was it worth it? (1/3)

- One of the first questions we should have even asked was if it was worth complicating our code
  - (i.e. remember Knuth's warning?)
- Stepping back, we can generate a 'perf report' by 'recording' execution of our program.

Samples: 216 of event 'cycles', Event count (approx.): 84105708

Overhead	Command	Shared Object	Symbol
39.14%	prog	prog	[.] VecN<int>::operator+=
30.17%	prog	prog	[.] std::vector<int, std::allocator<int> >::size
12.35%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
11.72%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
3.31%	prog	prog	[.] main
0.54%	prog	ld-2.27.so	[.] _dl_lookup_symbol_x
0.52%	prog	[kernel.kallsyms]	[k] 0xffffffffb4a2865b
0.52%	prog	[kernel.kallsyms]	[k] 0xffffffffb4a12cee
0.50%	prog	[kernel.kallsyms]	[k] 0xffffffffb4a288b6
0.48%	prog	ld-2.27.so	[.] strcmp
0.46%	prog	libstdc++.so.6.0.29	[.] std::locale::operator=
0.25%	prog	[kernel.kallsyms]	[k] 0xffffffffb4a6e62a
0.03%	perf	[kernel.kallsyms]	[k] 0xffffffffb52d1c30
0.00%	perf	[kernel.kallsyms]	[k] 0xffffffffb483ca5c
0.00%	perf	[kernel.kallsyms]	[k] 0xffffffffb48104be
0.00%	perf	[kernel.kallsyms]	[k] 0xffffffffb4878ada
0.00%	perf	[kernel.kallsyms]	[k] 0xffffffffb4878ad8

No optimization

Samples: 127 of event 'cycles', Event count (approx.): 6610875

Overhead	Command	Shared Object	Symbol
54.21%	prog	prog	[.] VecN<int>::operator+=
15.00%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
10.98%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
8.97%	prog	prog	[.] main
5.78%	prog	prog	[.] std::vector<int, std::allocator<int> >::size
1.19%	prog	[kernel.kallsyms]	[k] filemap_map_pages
1.17%	prog	[kernel.kallsyms]	[k] change_protection_range
0.99%	prog	ld-2.27.so	[.] _dl_lookup_symbol_x
0.94%	prog	ld-2.27.so	[.] do_lookup_x
0.75%	prog	[kernel.kallsyms]	[k] strlen_user
0.01%	perf	[kernel.kallsyms]	[k] intel_pmu_enable_all
0.00%	perf	[kernel.kallsyms]	[k] native_write_msr

Caching



# Perf - was it worth it? (2/3)

- The perf report tells us where we spent our time
- At first glance it looks like we made things worse!
  - (i.e. 39.14% is less than 54.21%)
  - (next slide)

Samples: 216 of event 'cycles', Event count (approx.): 84105708

Overhead	Command	Shared Object	Symbol
39.14%	prog	prog	[.] VecN<int>::operator+=
30.17%	prog	prog	[.] std::vector<int, std::allocator<int> >::size
12.35%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
11.72%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
3.31%	prog	prog	[.] main
0.54%	prog	ld-2.27.so	[.] _dl_lookup_symbol_x
0.52%	prog	[kernel.kallsyms]	[k] 0xffffffffb4a2865b
0.52%	prog	[kernel.kallsyms]	[k] 0xffffffffb4a12cee
0.50%	prog	[kernel.kallsyms]	[k] 0xffffffffb4a288b6
0.48%	prog	ld-2.27.so	[.] strcmp
0.46%	prog	libstdc++.so.6.0.29	[.] std::locale::operator=
0.25%	prog	[kernel.kallsyms]	[k] 0xffffffffb4a6e62a
0.03%	perf	[kernel.kallsyms]	[k] 0xffffffffb52d1c30
0.00%	perf	[kernel.kallsyms]	[k] 0xffffffffb483ca5c
0.00%	perf	[kernel.kallsyms]	[k] 0xffffffffb48104be
0.00%	perf	[kernel.kallsyms]	[k] 0xffffffffb4878ada
0.00%	perf	[kernel.kallsyms]	[k] 0xffffffffb4878ad8

No optimization

Samples: 127 of event 'cycles', Event count (approx.): 66108775

Overhead	Command	Shared Object	Symbol
54.21%	prog	prog	[.] VecN<int>::operator+=
15.00%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
10.98%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
8.97%	prog	prog	[.] main
5.78%	prog	prog	[.] std::vector<int, std::allocator<int> >::size
1.19%	prog	[kernel.kallsyms]	[k] filemap_map_pages
1.17%	prog	[kernel.kallsyms]	[k] change_protection_range
0.99%	prog	ld-2.27.so	[.] _dl_lookup_symbol_x
0.94%	prog	ld-2.27.so	[.] do_lookup_x
0.75%	prog	[kernel.kallsyms]	[k] strlen_user
0.01%	perf	[kernel.kallsyms]	[k] intel_pmu_enable_all
0.00%	perf	[kernel.kallsyms]	[k] native_write_msr

Caching



# Perf - was it worth it? (3/3)

- The perf report tells us where we spent our time
- At first glance it looks like we made things worse!
  - (i.e. 39.14% is less than 54.21%)
  - Consider however, there is no call to 'std::vector<...>size' on the next line however
  - Looks like we have trimmed some time!

Samples: 216 of event 'cycles', Event count (approx.): 84105708

Overhead	Command	Shared Object	Symbol
39.14%	prog	prog	[.] VecN<int>::operator+=
30.17%	prog	prog	[.] std::vector<int, std::allocator<int> >::size
12.35%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
11.72%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
3.31%	prog	prog	[.] main
0.54%	prog	ld-2.27.so	[.] _dl_lookup_symbol_x
0.52%	prog	[kernel.kallsyms]	[k] 0xffffffffb4a2865b
0.52%	prog	[kernel.kallsyms]	[k] 0xffffffffb4a12cee
0.50%	prog	[kernel.kallsyms]	[k] 0xffffffffb4a288b6
0.48%	prog	ld-2.27.so	[.] strcmp
0.46%	prog	libstdc++.so.6.0.29	[.] std::locale::operator=
0.25%	prog	[kernel.kallsyms]	[k] 0xffffffffb4a6e62a
0.03%	perf	[kernel.kallsyms]	[k] 0xffffffffb52d1c30
0.00%	perf	[kernel.kallsyms]	[k] 0xffffffffb483ca5c
0.00%	perf	[kernel.kallsyms]	[k] 0xffffffffb48104be
0.00%	perf	[kernel.kallsyms]	[k] 0xffffffffb4878ada
0.00%	perf	[kernel.kallsyms]	[k] 0xffffffffb4878ad8

No optimization

Samples: 127 of event 'cycles', Event count (approx.): 66108775

Overhead	Command	Shared Object	Symbol
54.21%	prog	prog	[.] VecN<int>::operator+=
15.00%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
10.98%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
8.97%	prog	prog	[.] main
5.78%	prog	prog	[.] std::vector<int, std::allocator<int> >::size
1.19%	prog	[kernel.kallsyms]	[k] filemap_map_pages
1.17%	prog	[kernel.kallsyms]	[k] change_protection_range
0.99%	prog	ld-2.27.so	[.] _dl_lookup_symbol_x
0.94%	prog	ld-2.27.so	[.] do_lookup_x
0.75%	prog	[kernel.kallsyms]	[k] strlen_user
0.01%	perf	[kernel.kallsyms]	[k] intel_pmu_enable_all
0.00%	perf	[kernel.kallsyms]	[k] native_write_msr

Caching



# Was Caching a win?

- Now, sometimes if we're not getting a huge performance boost, we might be solving the wrong problem or using the wrong technique.
  - As mentioned on my aside, caching is probably not a huge performance boost here.
  - So there's a different optimization strategy we can try

```
21 // Constructor
22 template<typename T>
23 VecN<T>::VecN(size_t elements){
24     // Initialize components
25     for(size_t i=0; i < elements; ++i){
26         components.push_back(i);
27     }
28 }
29
30 // Print
31 template<typename T>
32 void VecN<T>::Print() const{
33     size_t len = components.size();
34     for(size_t i=0; i < len; ++i){
35         std::cout << components.at(i) << " " << std::endl;
36     }
37 }
38
39 // generic addition overload
40 template<typename T>
41 VecN<T>& VecN<T>::operator+=(const VecN<T>& rhs){
42     size_t len = components.size();
43     for(size_t i=0; i < len; ++i){
44         components[i] += rhs.components[i];
45     }
46     return *this;
47 }
```



Optimization Strategy/Pattern #2:

**Compile-Time Computation**

A compile-time space versus run-time  
computation trade-off



# Compile-time (1/2)

- Ultimately we always trade time and space for performance
  - But in C++ we can choose to make that trade-off at compile-time and run-time as well!
  - Let's optimize any computation by templating our function
    - Afterall, are we going to change the 'size' of the n-dimensional vector?
      - (For this example, the answer is no)

```
6 // Generic n-dimensional mathematical vector
7 template<typename T, size_t length>
8 struct VecN{
9     // ===== Member Variables =====
10    // Store individual components
11    std::vector<T> components;
12
13    // ===== Member Functions =====
14    // Constructor
15    VecN();
16    // Print out the components
17    void Print();
18    // Add some components
19    VecN<T,length>& operator+=(const VecN<T,length>& rhs);
20 };
```

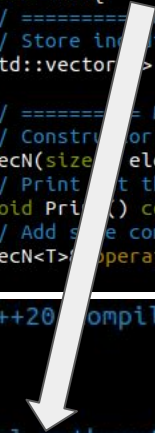


## Compile-time (2/2)

- Observe we now know the length at compile-time and no longer have to query the length at run-time for our loops

```
1 // g++ -g -Wall -std=c++20 vecN.cpp -o prog && ./prog
2 #include <iostream>
3 #include <vector>
4
5 // Generic n-dimensional mathematical vector
6 template<typename T>
7 struct VecN{
8     // ===== Member Variables =====
9     // Store individual components
10    std::vector<T> components;
11
12    // ===== Member Functions =====
13    // Constructor
14    VecN(size_t elements);
15    // Print out the components
16    void Print() const;
17    // Add some components
18    VecN<T> operator+=(const VecN<T>& rhs);
19 };
```

```
1 // g++ -g -Wall -std=c++20 compile_time.cpp -o prog && ./prog
2 #include <iostream>
3 #include <vector>
4
5 // Generic n-dimensional mathematical vector
6 template<typename T, size_t length>
7 struct VecN{
8     // ===== Member Variables =====
9     // Store individual components
10    std::vector<T> components;
11
12    // ===== Member Functions =====
13    // Constructor
14    VecN();
15    // Print out the components
16    void Print();
17    // Add some components
18    VecN<T,length> operator+=(const VecN<T,length>& rhs);
19 };
```





# Compile-Time Results

- Our fastest result yet!
- And we can try something else after our realization that length does not change
  - (an age old tradeoff...)

```
mike:2023_italian_cpp$ sudo perf stat ./prog
0.
1000001.
2000002.
```

Compile-time

Performance counter stats for './prog':

15.22 msec	task-clock	#	0.985 CPUs utilized
0	context-switches	#	0.000 K/sec
0	cpu-migrations	#	0.000 K/sec
118	page-faults	#	0.008 M/sec
60,718,858	cycles	#	3.990 GHz
145,308,561	instructions	#	2.39 insn per cycle
23,575,716	branches	#	1549.215 M/sec
16,618	branch-misses	#	0.07% of all branches

0.015454034 seconds time elapsed

0.015474000 seconds user

0.000000000 seconds sys

Samples: 41 of event 'cycles', Event count (approx.): 60031268

Overhead	Command	Shared Object	Symbol
56.94%	prog	prog	[.] VecN<int, 3ul>::operator+=
13.25%	prog	prog	[.] main
12.21%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
10.33%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
5.12%	prog	[kernel.kallsyms]	[k] kfree
1.76%	prog	[kernel.kallsyms]	[k] __mod_memcg_state
0.37%	prog	[kernel.kallsyms]	[k] security_bprm_committed_creds
0.01%	perf	[kernel.kallsyms]	[k] native_sched_clock
0.00%	perf	[kernel.kallsyms]	[k] native_write_msr



Optimization Strategy/Pattern #3:  
**Solve the right problem with the right data  
structure**

A classic space vs time data structure trade-off



# Choose the right data structure (1/2)

- Did we really need the capabilities of a vector?
  - (Note: I have to be careful here if we changing the problem)
  - Let's assume I did not however, and my domain (e.g. games) usually have vectors stay the same size (e.g. 3 components) when initialized.
- Note: This is often the best optimization strategy -- try another data structure or algorithm

```
1 // g++ -g -Wall -std=c++20 array.cpp -o prog && ./prog
2 #include <iostream>
3
4 // Generic n-dimensional mathematical vector
5 template<typename T, size_t length>
6 struct VecN{
7     // ===== Member Variables =====
8     // Store individual components
9     T components[length];
10
11     // ===== Member Functions =====
12     // Constructor
13     VecN();
14     // Print out the components
15     void Print();
16     // Add some components
17     VecN<T,length>& operator+=(const VecN<T,length>& rhs);
18 };
```



## Choose the right data structure (2/

- Faster yet again!
  - (And more important -- consistently faster!)
- But there is something bothering me
  - We are spending lots of time in +=

Samples: 100 of event 'cycles', Event count (approx.): 40643887

Overhead	Command	Shared Object	Symbol
79.08%	prog	prog	[.] VecN<int, 3ul>::operator+=
10.68%	prog	prog	[.] main
2.34%	prog	[kernel.kallsyms]	[k] unmap_page_range
1.43%	prog	ld-2.27.so	[.] _dl_relocate_object
1.43%	prog	ld-2.27.so	[.] _dl_lookup_symbol_x
0.78%	prog	ld-2.27.so	[.] _dl_debug_initialize
0.72%	prog	ld-2.27.so	[.] do_lookup_x
0.72%	prog	libc-2.27.so	[.] init_cacheinfo
0.71%	prog	ld-2.27.so	[.] malloc
0.71%	prog	[kernel.kallsyms]	[k] clear_page_erms
0.71%	prog	[kernel.kallsyms]	[k] get_mem_cgroup_from_mm
0.53%	prog	[kernel.kallsyms]	[k] get_page_from_freelist
0.14%	prog	[kernel.kallsyms]	[k] apparmor_bprm_committed_creds
0.02%	perf	[kernel.kallsyms]	[k] perf_event_addr_filters_exec
0.00%	perf	[kernel.kallsyms]	[k] native_write_msr

```
mike:2023_italian_cpp$ g++ -Wall -std=c++20 array.cpp -o prog
```

```
mike:2023_italian_cpp$ sudo perf stat ./prog
```

```
0.
```

```
1000001.
```

```
2000002.
```

Performance counter stats for './prog':

6.66 msec	task-clock	#	0.961 CPUs utilized
0	context-switches	#	0.000 K/sec
0	cpu-migrations	#	0.000 K/sec
119	page-faults	#	0.018 M/sec
26,987,099	cycles	#	4.050 GHz
64,342,603	instructions	#	2.38 insn per cycle
11,579,759	branches	#	1737.654 M/sec
16,397	branch-misses	#	0.14% of all branches

0.006930871 seconds time elapsed

0.006945000 seconds user

0.000000000 seconds sys



## (Aside)

- If switching to an array felt like cheating, I did go back to our very first example and just switch to a heap allocated array to see the difference.
  - results were 'noisier' do to the heap allocations (but sometimes still way faster) -- so sometimes we like more stable guarantees on time as well!

```
mike:2023_italian_cpp$ sudo perf stat ./prog
0.
1000001.
2000002.
```

Performance counter stats for './prog':

10.48 msec	task-clock	#	0.976 CPUs utilized
0	context-switches	#	0.000 K/sec
0	cpu-migrations	#	0.000 K/sec
120	page-faults	#	0.011 M/sec
43,086,088	cycles	#	4.110 GHz
99,345,603	instructions	#	2.31 insn per cycle
11,580,757	branches	#	1104.764 M/sec
16,640	branch-misses	#	0.14% of all branches

0.010739591 seconds time elapsed

0.007173000 seconds user  
0.003586000 seconds sys

```
real    0m0.038s
user    0m0.037s
sys     0m0.001s
mike:2023_italian_cpp$ time ./prog
0.
1000001.
2000002.
```

```
mike:2023_italian_cpp$
```

```
5 template<typename T>
6 struct VecN{
7     // ===== Member Variables =====
8     // Store individual components
9     T* components;
10    size_t mSize;
11
12    // ===== Member Functions =====
13    // Constructor
14    VecN(size_t elements);
15    // Print out the components
16    void Print() const;
17    // Add some components
18    VecN<T>& operator+=(const VecN<T>& rhs);
19 };
```

```
real    0m0.014s
user    0m0.010s
sys     0m0.004s
mike:2023_italian_cpp$ time ./prog
0.
1000001.
2000002.

real    0m0.015s
user    0m0.011s
sys     0m0.004s
```



# Optimization Strategy/Pattern #4: **Specialization**

A compile-time and space versus run-time  
trade-off



# Specializing functions

- So one optimization strategy we can use is to specialize functions or data structures
  - This means studying carefully a piece of code, finding the use case, and then determining that we can hand tune it to be faster.
    - And preferably do the tuning such that that our compiler cannot do better than us!
  - We're going to take advantage again of compile-time programming to specialize our code.

```
37 // generic addition overload
38 template<typename T, size_t length>
39 VecN<T,length>& VecN<T,length>::operator+=(const VecN<T,length>& rhs){
40     for(size_t i=0; i < length; ++i){
41         components[i] += rhs.components[i];
42     }
43     return *this;
44 }
```

Catch-all case generic case  
with no specialization



# Specializing functions results

- First observe that we have added a template specialization avoiding a loop (i.e. getting into compiler optimization world)
  - This appears to have reduced overall time spent in operator+= shown below.

```
37 // generic addition overload
38 template<typename T, size_t length>
39 VecN<T,length>& VecN<T,length>::operator+=(const VecN<T,length>& rhs){
40     for(size_t i=0; i < length; ++i){
41         components[i] += rhs.components[i];
42     }
43     return *this;
44 }
45
```

Catch-all case

```
46 // Add a specialization with '3' integers
47 template<>
48 VecN<int,3>& VecN<int,3>::operator+=(const VecN<int,3>& rhs){
49     components[0] += rhs.components[0];
50     components[1] += rhs.components[1];
51     components[2] += rhs.components[2];
52 }
53 return *this;
54 }
```

**\*NEW\*Specialization**

Samples: 46 of event 'cycles', Event count (approx.): 51376970			
Overhead	Command	Shared Object	Symbol
46.29%	prog	prog	[.] VecN<int, 3ul>::operator+=
22.79%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
13.78%	prog	prog	[.] std::vector<int, std::allocator<int> >::operator[]
5.94%	prog	prog	[.] main
5.68%	prog	ld-2.27.so	[.] do_lookup_x
3.27%	prog	[kernel.kallsyms]	[k] task_work_run
2.01%	prog	[kernel.kallsyms]	[k] unmap_page_range
0.22%	prog	[kernel.kallsyms]	[k] perf_output_begin
0.01%	perf	[kernel.kallsyms]	[k] native_sched_clock
0.00%	perf	[kernel.kallsyms]	[k] native_write_msr



# Specializing functions results

- From a performance standpoint, I got relatively good results
  - Perhaps our code layout has changed enough that we're not always optimized however!
  - Perhaps on a larger data structure, specialization can be more impactful -- and perhaps enable other optimizations!
    - We may have even enabled specializations like this for SIMD to get further performance.

```
mike:2023_italian_cpp$ time ./prog  
0.  
1000001.  
2000002.  
  
real    0m0.006s  
user    0m0.005s  
sys     0m0.000s
```

Specialization (with array)

```
mike:2023_italian_cpp$ time ./prog  
1000001.  
2000002.  
3000003.  
  
real    0m0.013s  
user    0m0.013s  
sys     0m0.000s
```

Specialization (with vector)



## Optimization Strategy/Pattern #5:

### **Multi-phase initialization**

A space versus time trade-off affecting  
readability/maintenance



# Multistage setup

- Consider the example to the right where we decide we want to use `std::vector` again as our underlying container
  - Often times we have data structures (including vectors) where it might be beneficial to setup the data structure in multiple stages.
    - i.e. reserve memory first, then setup components
- Note: For this particular pattern -- we probably need to increase length to something larger to be more meaningful in the results.

```
22 // Constructor
23 template<typename T, size_t length>
24 VecN<T,length>::VecN(){
25     // Initialize components
26     // in multiple steps
27     components.resize(length); // allocate and fill
28     for(size_t i =0; i < length; ++i){
29         components[i] = i;
30     }
31
32     // Initialize components growing our vector slowly
33     for(size_t i =0; i < length; ++i){
34         components.push_back(i);
35     }
36
37 }
```



# Wrapping up VecN Example



# Wrapping up VecN Example

- We've played around with a data structure thinking about 5 optimization strategies
  - Caching
  - Compile-Time Computation
  - Specialization
  - Solve the right problem with the right data structure
  - Multi-phase initialization
- We have also learned how we might investigate if our program is actually running faster
  - There exist more strategies however that I'd like to share briefly -- and may be discussed in future talks



# More Patterns/Strategies



# Hinting

- Hint on insertion
- Nice example on cppreference showing how 'hints' can be used for speeding up insertion in maps
  - [https://en.cppreference.com/w/cpp/container/map/emplace\\_hint](https://en.cppreference.com/w/cpp/container/map/emplace_hint)
- Consider another example of a list like data structure where we can 'skip' through it for faster insertion/traversals/searches [e.g. [skip list](#)]

```
int main() {  
    std::cout << std::fixed << std::setprecision(2);  
    timeit(map_emplace); // stack warmup  
    timeit(map_emplace, "plain emplace");  
    timeit(map_emplace_hint, "emplace with correct hint");  
    timeit(map_emplace_hint_wrong, "emplace with wrong hint");  
    timeit(map_emplace_hint_corrected, "corrected emplace");  
    timeit(map_emplace_hint_closest, "emplace using returned iterator");  
}
```

Possible output:

```
22.64 ms for plain emplace  
8.81 ms for emplace with correct hint  
22.27 ms for emplace with wrong hint  
7.76 ms for corrected emplace  
8.30 ms for emplace using returned iterator
```



# Precomputation

- C++ Compiler optimizations may do some of this
  - Common subexpression elimination (CSE)  
(Figure on right)
- Templates are our tool for doing work at compile-time
- C++11 and beyond has constexpr
  - You should try to constexpr as many things as possible.

## Example:

In the code fragment below, the second computation of the expression  $(x + y)$  can be eliminated.

```
i = x + y + 1;  
j = x + y;
```

After CSE Elimination, the code fragment is rewritten as follows.

```
t1 = x + y;  
i = t1 + 1;  
j = t1;
```

[https://compileroptimizations.com/category/cse\\_elimination.htm](https://compileroptimizations.com/category/cse_elimination.htm)



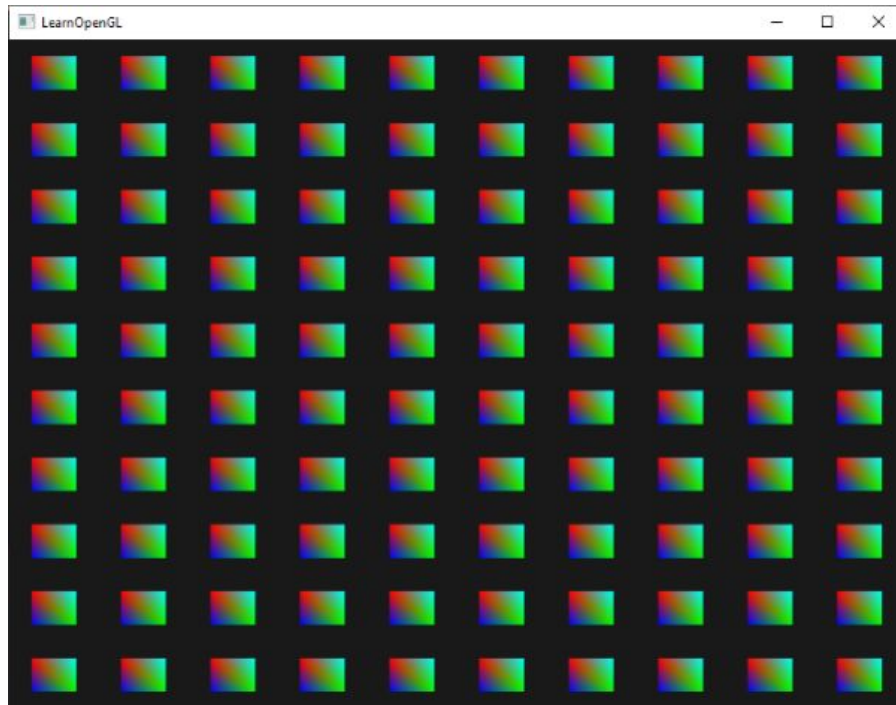
# Lazy versus Eager Evaluation

- Eager evaluation is evaluating the result immediately
- Lazy Computation is to delay our computation
  - [std::async](#) with [std::launch::deferred](#)
  - Multiple part construction of our objects as needed
- Copy-on-Write (COW) [[wiki](#)]
- Consider 'short-circuit evaluation' as another way to avoid work that does not need to be done when ordering conditionals



# Batching

- Consider in some domains like computer graphics, you want to ‘batch’ all of the draw calls together
  - (Either through instancing or some other mechanism)
- More simply -- buffered output is an example of this optimization



<https://learnopengl.com/Advanced-OpenGL/Instancing>



# Hashing

- Consider that we may want to take some long value (e.g. a large `std::string`) and compute a hash (using [`std::hash`](#))
  - We can then use this hash (i.e. an integral type) to reference the object by or otherwise compare two larger pieces of data.

<code>std::hash&lt;std::string&gt;</code>	<code>(C++11)</code>	hash support for strings (class template specialization)
<code>std::hash&lt;std::u8string&gt;</code>	<code>(C++20)</code>	
<code>std::hash&lt;std::u16string&gt;</code>	<code>(C++11)</code>	
<code>std::hash&lt;std::u32string&gt;</code>	<code>(C++11)</code>	
<code>std::hash&lt;std::wstring&gt;</code>	<code>(C++11)</code>	
<code>std::hash&lt;std::pmr::string&gt;</code>	<code>(C++17)</code>	
<code>std::hash&lt;std::pmr::u8string&gt;</code>	<code>(C++20)</code>	
<code>std::hash&lt;std::pmr::u16string&gt;</code>	<code>(C++17)</code>	
<code>std::hash&lt;std::pmr::u32string&gt;</code>	<code>(C++17)</code>	hash support for string views (class template specialization)
<code>std::hash&lt;std::pmr::wstring&gt;</code>	<code>(C++17)</code>	
<code>std::hash&lt;std::string_view&gt;</code>	<code>(C++17)</code>	
<code>std::hash&lt;std::wstring_view&gt;</code>	<code>(C++17)</code>	
<code>std::hash&lt;std::u8string_view&gt;</code>	<code>(C++20)</code>	
<code>std::hash&lt;std::u16string_view&gt;</code>	<code>(C++17)</code>	
<code>std::hash&lt;std::u32string_view&gt;</code>	<code>(C++17)</code>	



# (Silly) Anecdote

- Performance is Tricky!
- I have heard on numerous occasions adding a random 'printf' to change the address layout has improved performance by 10+% before.
  - This is in the 'lore' in optimization, I first heard about at PLDI at 2013
  - Here's a stack overflow post, and there exist possibly other notes
    - <https://stackoverflow.com/questions/42358211/adding-a-print-statement-speeds-up-code-by-an-order-of-magnitude>



# And that's all folks!

- Optimization is fun, and it comes with many trade-offs
  - It's better to say there are 'strategies' versus 'patterns' -- the reality is we have lots of strategies to choose from versus cookie cutter solutions, and optimizing is often very iterative.
  - (Slides and code will be available for this talk)
- Make sure to go read the original Knuth paper so you can tell folks that you know the full quote! (i.e. optimization is not really the root of all evil) :)

## Structured Programming with *go to* Statements

DONALD E. KNUTH

*Stanford University, Stanford, California 94305*

A consideration of several different examples sheds new light on the problem of creating reliable, well-structured programs that behave efficiently. This study focuses largely on two issues: (a) improved syntax for iterations and error exits, making it possible to write a larger class of programs clearly and efficiently without *go to* statements; (b) a methodology of program design, beginning with readable and correct, but possibly inefficient programs that are systematically transformed if necessary into efficient and correct, but possibly less readable code. The discussion brings out opposing points of view about whether or not *go to* statements should be abolished; some merit is found on both sides of this question. Finally, an attempt is made to define the true nature of structured programming, and to recommend fruitful directions for further study.

*Keywords and phrases:* structured programming, *go to* statements, language design, event indicators, recursion, Boolean variables, iteration, optimization of programs, program transformations, program manipulation systems searching, Quicksort, efficiency

*CR categories:* 4.0, 4.10, 4.20, 5.20, 5.5, 6.1 (5.23, 5.24, 5.25, 5.27)



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

Thank you  
CppIndia!

# Optimization Design Patterns

Gold Sponsors

think-cell

QUBE

intel  
software  
Bloomberg

Engineering

10:00-11:00, Fri, 4th Aug. 2023

60 minutes | Introductory Audience

95





Thank you!



# Extras and Notes