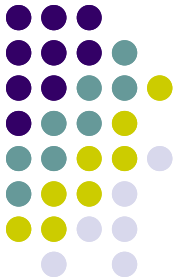
C++ Basics



Optimized C++

Ed Keenan

31 January 2019

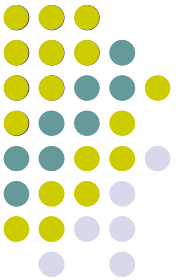
13.0.6.3.13 Mayan Long Count

DePaul University

Goals

- C++ Basics
  - Stuff you should know

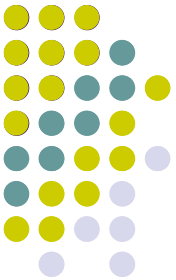
C++ Basics



- Every class basics
  - Big Four
    - Make sure you define
- Default constructor
  - dog()
- Copy constructor
  - dog(const dog &in)
- Assignment operator
  - dog & operator= (const dog &in)
- Destructor
  - ~dog()

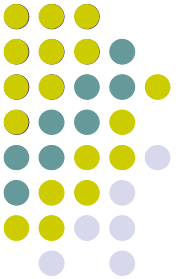
- Example convention:
  - class dog is in almost every example
  - Traditionally you'll see class foo
    - but I use dog

Const – your friend



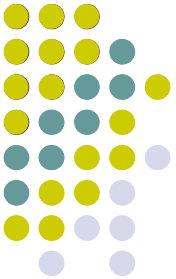
- Place const everywhere you can
  - Yes you!
- More const you use,
  - Gives hints to the compiler to optimize
  - Communicates to the user that data is Read-Only
- Several ways it's used
  - See how good you understand const
  - Quiz on next slide

Const – Quiz



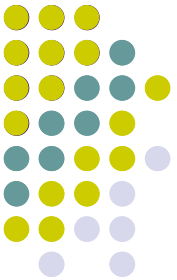
- Explain
  a) Dog::foo(***const*** Dog & r);
  b) Dog::foo(Dog ***const*** & r);
  c) Dog::foo(***const*** dog * p);
  d) Dog::foo(dog ***const*** * p);
  e) Dog::foo(dog * ***const*** p);
  f) Dog::foo(dog * p ***const***);
  g) Dog::foo(dog * p) ***const***;
  h) ***const*** int Dog::foo(dog * p);
- Big daddy:
  - ***const*** int Dog::foo(***const*** dog * ***const*** p) ***const***;

Encapsulated data



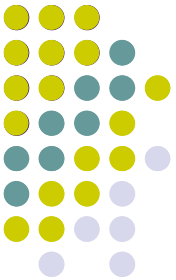
- What is it?
  - Hide all data from direct access.
    - Why do we like this?
- How do you do accomplish this property?
  - Make you data private
  - Give accessors to update data
- Global data, public data
  - Is evil
  - Causes support and maintenance issues

Good interfaces



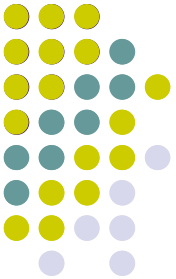
- What is it?
  - Do know, but I know what's not.
- Best practices
  - Needs to cleanly without manual
- User should understand
  - Expected parameters by naming convention
  - Error conditions
  - Return parameters
  - Dangerous parameters

Example:
int strcmp (const char * str1, const char * str2);



- What's wrong (yes related to Basics4)
  - Missing *consts*
    - Read only pointer
  - What happens if *str1* or *str2* is null (0)?
    - Crashes
  - What is the magic int number?
    - Make it obvious, not intuitive
      - 0 is success
      - 1 str1 > str2 (strict weak ordering)
      - -1 str1 < str2 (strict weak ordering)

Example:
int strcmp (const char * str1, const char * str2);



- How do I fix?
  - Wrap functions that suck!

```
returnCode myStrcmp( const char * const str1, const char * const str2);

enum returnCode {
        Success_StringEqual,
        Fail_String1Null,
        Fail_String2Null,
        Fail_BothStringNull,
        Fail_String1Greater,
        Fail_String2Greater
};
```

Char *

- What is the difference between initializations:

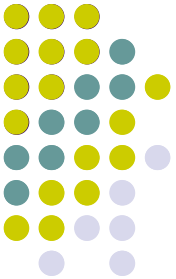
```
void dog ( )
{
    a)  char *p = “MonkeyBrains”;
    b)  char  s[10] = “JellyBeans”;
    c)  char  t[5] =
        “GummyWorms”;
    d)  char m[5] = “HiChew”;
    e)  char  *r = new char[15];
}
```

- Null character inserted where?
- What’s happens with these functions:

```
char *a = “12”;
strcpy( p, a);   // etc…
strcpy( s, a);
```

- What’s it’s scope?
  - String is passed into another function

Char * - shocking truth



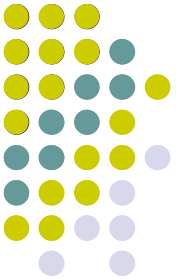
- So strcpy(p, a) crashes hard!
  - Let's look at the prototype:

    char * strcpy (char * destination, const char * source);
  - char *p = "MonkeyBrains";
    - What gives???
- In reality:

  char *p = "MonkeyBrains" ***is***

  const char *p = "MonkeyBrains";
- Prefer to be explicit and more protection:
  - const char * const p = "MonkeyBrains";

Char * - shocking truth



- Always be explicit in your declaration:
  - With char *
    - Always communicate const
  - Instead of
    - char *p = “MonkeyBrains”;
  - Write this….
    - const char *p = “MonkeyBrains”;
- Now let’s look at the String Copy
  - strcpy(p, a); ← error:
    - cannot convert parameter 1 from 'const char *' to 'char *'
  - Good! - no surprises

Thank You!

- Questions?

General Optimization

Optimized C++

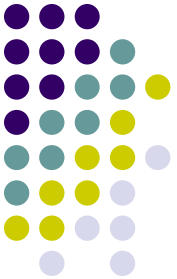
Ed Keenan

DePaul University

Goals

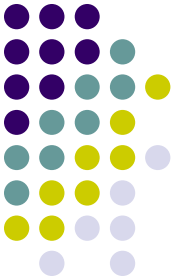
- Everyone is connected
- Easy Way
  - Compilers
  - Money
- Source of inefficiency
- Techniques
  - Loops
  - Logic
  - Strings

General observations



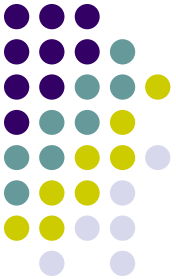
- Everything is interconnected
  - Optimizing on one section, affects others
  - You might shuffle the whole system and make is slower
- Existing architecture might be the biggest challenge.
  - It’s hard to optimized when system isn’t modular or encapsulated
  - Might need to Refactor to make the system clearer

More observations



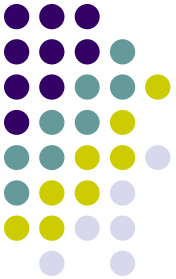
- Class design
  - Sometime the internal structure of classes and it layout dictate performance
  - Can you swap out routines easy?
  - Is their in-lining opportunities?
  - What is opaque versus observable?

Compilers



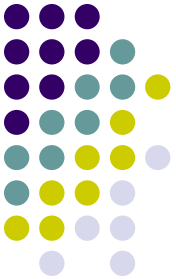
- Compilers
  - Huge difference in code performance
  - Intel creates optimal code for Intel processors.
    - How is that possible?
  - Compiler settings
    - Need to be understood and tweaked
- Compilers are smarter than you…
  - Almost, but generally better return for the dollar

Throw Money at it?



- Buy faster and better hardware
  - Is that enough?
- Why isn't my code going twice as fast?
  - I/O, Disc, Networking might be a bottleneck
  - Algorithms don't scale
  - Many processors, code only uses one.
- New systems have more but slower procs
  - Cost savings to manufacture
  - IBM servers…

Line Count?

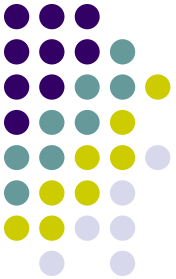


- Reducing the lines of code, improves speed
  - Many things are happening under the hood.
  - Which is faster?

```
for( i=0; i < 10; i ++)
{
    a[i]=i;
}
```

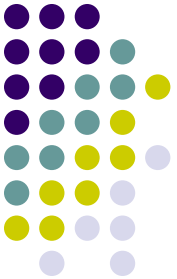
```
a[0]=0;
a[1]=1;
a[2]=2;
...
a[9]=9;
```

Understand functions



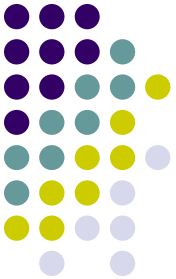
- All operations are not created equally
  - Operating system calls, like copy, read, sort take a very long time.
  - Copy constructors, passing by value are deceiving
- Optimize everything
  - You have limited time and money to work
  - Spend them where it counts
  - Resist premature optimization
  - Later we'll talk about Performance Solution Engineering (PSE) in Week 9

Premature Victory



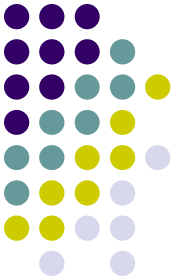
- Fast program is good enough
  - My program is:
    - 90% or almost working.
    - Practically done
  - Wrong, if it’s not working it’s not optimized.
    - Its generally the edge conditions that hurt clean streamlined code.

Source of Inefficiency



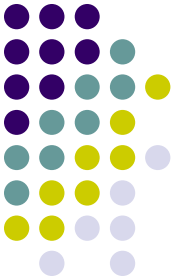
- Input / Output operations
  - Biggest and most evil
    - You have to deal with it.
  - Being clever can reduce it's effects.
    - Only use it if you have to.
  - Ways to improve it
    - Cache copies
    - Stream
    - Layout all help
    - Learn the hardware

Source of Inefficiency



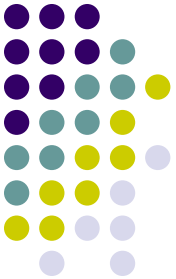
- Memory
  - Yes it comes up everywhere
    - Very slow, we can do better
  - Custom schemes
    - Writing for our use cases
  - Virtual memory manager
    - Transparent but cost time
- Thread and process switching
  - Other heavy system calls
  - They hurt ☹

Source of Inefficiency



- Language choices
  - Compiler vs interpreted
    - Interpreted is roughly 20 times slower
    - There is a place and time for them
    - Not in the most call systems or loops
    - Remember the 80/20 rule
  - Maintenance vs speed
    - Support and readability sometimes get sacrificed for optimization

Source of Inefficiency

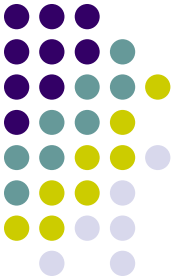


- Errors
  - Leaving debug information
  - Not freeing memory
  - Redundantly initializing memory
  - Unnecessary initializations
  - Bugs
  - Do you know what the code is really doing?
  - Many people do not step code or know how

Metrics

- Don’t be a Cowboy or Cowgirl
  - Your spidey sense isn’t that good.
  - Measure everything!
  - Assume nothing!
  - Story about Mips processor

Heisenberg Uncertainty Principle



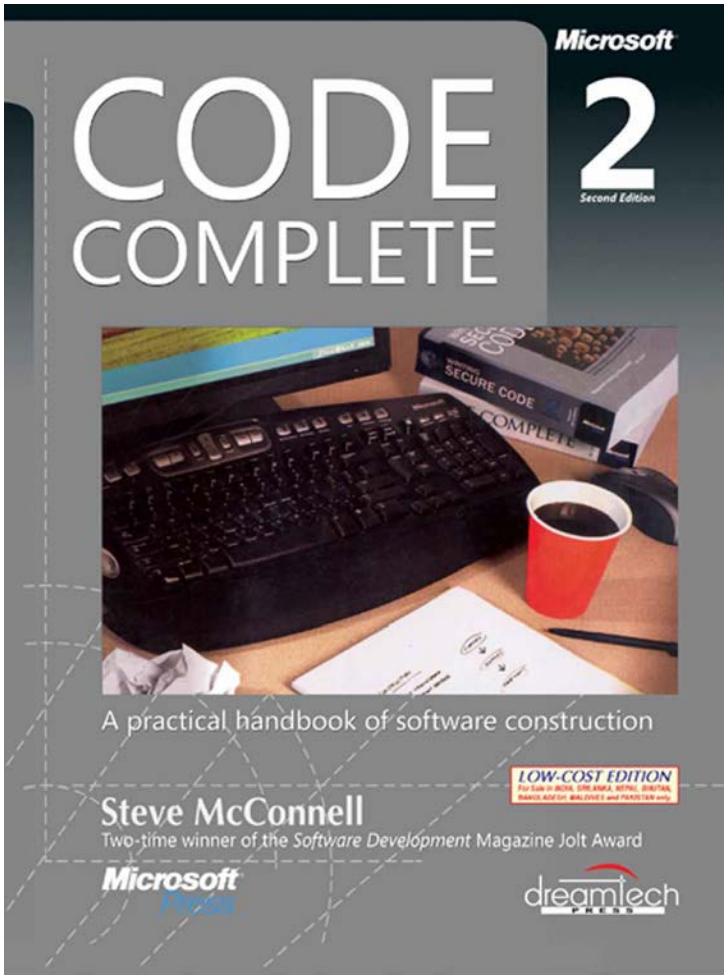
- By observing the system you affect the results of the system
  - Mexico Story
- Optimizations
  - If you alter one system,
  - The next system might be negatively altered.

Break

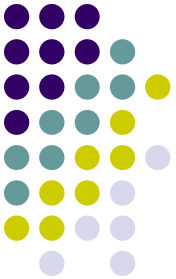
- What time is it?
  - I’m Thirsty!!!

Techniques

- Reference:
  - Chapters 25 & 26 of Code Complete 2nd Edition
- Good News – Safari free for DePaul
  - http://proquestcombo.safaribooksonline.com/book/software-engineering-and-development/0735619670

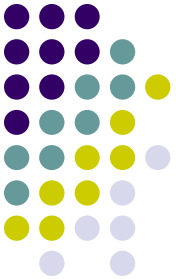


Logic



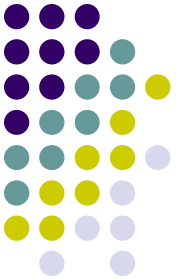
- Conditionals
  - Early out
  - Reworking conditionals to use
- Does every know their Binary Logic?
  - And, Or, Xor, Not, Xnor,
  - Associative, Commutative, Distributive
  - DeMorgan?
    - !(x+y) = !x & !y
    - Why is this important?

DeMorgan



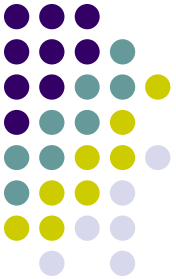
- Conditionals
  - Evaluation happens from left to right.
- If (x && y)
  - If x if false, no need to evaluate y
  - Early out.
- If (x||y) then …
  - If you can rework the logic to be negative
  - If !(x||y) then …Can use DeMorgan
- If (!x && !y) then …
  - You get the early out ☺

Switching / exiting



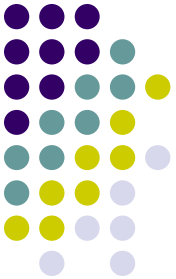
- Understand how switch() work!
  - They are implemented under the hood as
  - Many if else…
- Invariants
  - Do not have invariant states inside the loop
  - Only keep statements that change in the loop.
- Sentinals
  - What are they?
    - One time flags inside of loops
  - Evil, check is done every time
    - Move out of loop

Loops



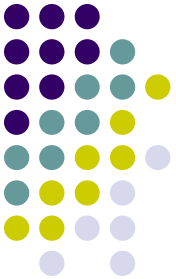
- Combining work inside with same loops
  - Sometimes this is counter intuitive
  - Need to test, caching becomes a big issue
- Unrolling
  - Sometimes helps,
  - some times confuses compilers
- Busiest loop in the inner most loop
  - Multi-nested loops
  - Less loop-context switching

Floats



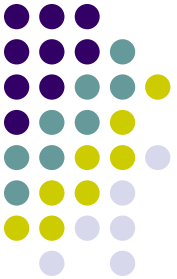
- Floats vs Integers
  - Floats are good for math
    - Not for indices or conditional testing
  - Integers are great of conditionals and indexing
    - Well kind of?
    - Should not be passed to floating point parameters
- Multi-dimensional arrays are slow
  - Indication of poor design
  - Refactor

Strings



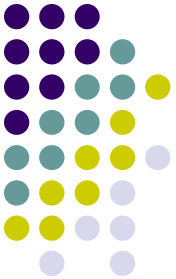
- Strings
  - Very slow to compare and use
  - Get creative, do you really need them?
  - In industry, cause of many slowdowns
  - Often, too embedded in the existing architecture to remove ☹
- MD4 or MD5 quick alternative to strings
  - Allows integer compares
  - Fixed length strings… sound weird?
    - Making all your strings the same length has advantages in processing them.

System Calls



- Understand your system calls
  - Many take doubles, when you need floats.
    - Sqrt() is big offender.
  - Why do people need 64-bit floats when we went to the moon on 16 bit fix point?
- Bit shifts and tricks don’t work anymore.
  - Look at timing and metrics
  - They are implement with slow legacy and often with slow your project down.

Assembler



- Assembler
  - Just like Copernicus
  - You can’t practically do this for the N-stage pipeline in processors, with look ahead and in order executions, cache misses, hyper-threaded context switching, and vectorize embedded coprocessors.
  - We will learn intrinsics operators
    - Like mini micro assembler functions

Thank You!

- Easy?