The University Of Michigan

EECS402 Lecture 03

Andrew M. Morgan

Some Thoughts And Requirements On Style

Disclaimer

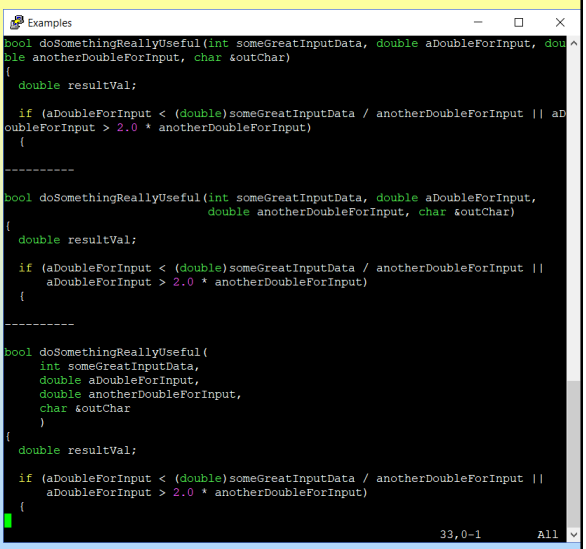
- These slides will attempt to talk about <u>some</u> of the style considerations that may cause deductions
- The examples and explanations are not all-inclusive
- We may deduct for other style issues not included in these slides, or for reasons not specifically stated in these slides

Overarching Importance of Style

- The following are some overall reasons style is so important
  - Understandability
    - Someone (either someone else, or even YOU at a later date) who is looking at your code needs to be able to understand it
    - Any programmer could analyze poorly styled code and "figure it out" eventually, but:
    - Goal is to allow fast and easy understanding without much "analysis"
  - Readability
    - While reading code, names and organization matter
    - Anything odd, or out of place, will cause a reader to stop for a second and ask "why is that like that?"
    - Even if the answer is just "oh, it's just poorly styled, but I understand now", it was a distraction and shifted focus from reading and understanding the code
  - Searchability
    - When coding, you often want to jump to some code where you did something specific
    - Knowing what you likely named the variables involved and how you use whitespace, etc., you can typically do a search to find what you're looking for without having to try several possibilities

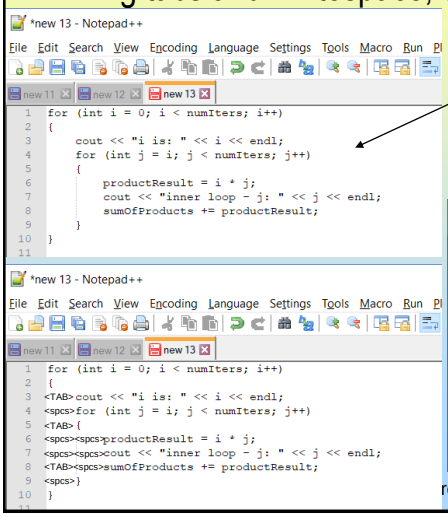
80 Character Limit

- Standard terminal dimensions are 80x24
- Using a standard text editor within a terminal doesn't included scroll-bars
- Wrapped lines look like poorly indented lines and cause confusion
- You may have 6 monitors laid out horizontally and can expand your terminal to be super wide, or you may use a graphical editor which has scrollbars – but not everyone does!



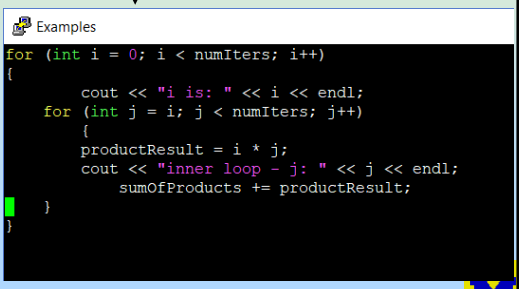
Tabs Disallowed

- Tabs are NOT standardized! Different editors will interpret them as different number of characters.
- Mixing tabs and whitespace, especially, is problematic:



How it looks in Notepad++

How it looks in vim



More On Tabs

- Many text editors will put tabs in your code even if you don't type them!
- Many text editors can DISPLAY tabs as spaces, but when writing to a file, still STORE them as tabs!

- This means you should make very very certain that the source files you submit do NOT have tabs in them
  - Every single semester, there are multiple students who insist they configured their text editor, or checked their source files, etc., but tabs are still found in there

- One way to check specifically for tabs, is with the following Linux command:
  - `grep -P "\t" *.cpp | wc -l`
  - Note that after "*.cpp" is a vertical bar (or pipe) character, and after "wc –" is a lower-case L character
  - This will report the number of lines in all cpp files that contain tabs!
    - If it reports 0, that means there are no tabs in your cpp source files
    - If it reports any number other than 0, then your source code includes that many lines with tabs and will result in a style deduction!
  - For later projects, you'll have to check ALL you source files (including, potentially *.cpp *.h and *.inl)

Booleans

- Most bool type variables should be named starting with "do" or "is" or something else that implies Boolean-ness
  - Examples:
    - "runSimulation" sounds like a function, but "doRunSimulation" sounds like a Boolean indicating whether the simulation will be run or not
    - "late" is a terrible name – not clear at all what it represents, but "isLate" allows reader to know it's a Boolean indicating whether something is late or not
- Use the C++ literal values true and false
  - isLate = 0; will work, but its confusing – the name implies a Boolean, but its being assigned to an int. The user is left wondering what "isLate" really is
  - isLate = false; is far clearer and easier to read and understand quickly
- Don't "== true" or "== false", etc..
  - Booleans are true or false by nature – use them that way
  - Example:
    - Prefer "if (isLate)" over "if (isLate == true)" and *definitely* over "if (isLate != 0)"
    - Prefer "if (!doRunSimulation)" over "if (doRunSimulation == false)"

Identifier Naming

- Naming identifiers is incredibly important!
  - Functions, variables, constants, etc.
  - Reader should essentially be able to know exactly what something is from its name
  - Examples:
    - Identifier: "sim" – what is that? Is it a variable representing the status of a simulation? Is it a function that performs a simulation? Is it a variable representing the "subscriber identity module"?
    - Identifier "performSimulation" – this is named with a verb and therefore is a function, the name clearly indicates the purpose of the function is to perform the simulation
- Note: this doesn't mean abbreviations aren't acceptable – clarity is what matters most!
  - Examples:
    - Identifier: "numberOfQuizzesGiven" is clear, but kind of wordy and long
    - Identifier: "numQuizzesGiven" is just as clear and arguably easier and faster to read

Identifier Naming, Page 2

- Very short identifier names are to be avoided
  - Example: x = (-b + sqrt(b * b – 4 * a * c)) / (2 * a);
  - This might be very clear to a reader as being the quadratic formula
  - However, think about searchability
    - If you want to go to the place in your code where you did the quadratic formula, and search for "a" because you know "a" is part of the formula, you'll find a ton of "a"s that have nothing to do with the quadratic formula
  - Improvement:
    - xResult = (-bCoeff + sqrt(bCoeff * bCoeff – 4 * aCoeff * cCoeff)) / (2 * aCoeff)
    - I'm not arguing that this is more readable or more understandable, but it is ***way more*** searchable
    - Searching for "aCoeff" will get you here quickly!

Disallowing Magic Numbers

- Literal values that show up in your code are often considered "magic numbers"
  - When the reader sees this:
    - else if (userChoice == 4)
    - they are left wondering "what is the significance of 4?" and/or "was 4 the option for doing the average or finding the max?
  - Improvement:
    - else if (userChoice == COMPUTE_AVERAGE_OPTION)
    - Now the user knows this is the part of the code where you'll be computing the average due to the user choosing that option
    - Use global constants for this type of thing, like this:
      - const int COMPUTE_AVERAGE_OPTION = 4;
  - NOT an improvement:
    - const int CHOICE_FOUR = 4;
    - else if (userChoice == CHOICE_FOUR)
    - Ok, so maybe it's a SLIGHT improvement – this tells the reader its probably a menu choice..
      - But the variable name "userChoice" tells us that too
    - This will still be considered a "magic number" and result in a style deduction

Disallowing Magic Numbers, Page 2

- Most literals in code are magic numbers, but not ALL are:
- Example:
  - avgResult = (firstValue + secondValue) / 2.0;
  - 2.0 is a literal value, but its not usually considered a magic number
  - What name would we give to a constant that would ***add value*** over the literal value 2.0 in this case?
    - const double AVERAGE_DENOMINATOR = 2.0;
      - This actually adds confusion.. When the reader sees:
      - avgResult = (firstValue + secondValue) / AVERAGE_DENOMINATOR;
      - they will ask "Hmm, I wonder what value AVERAGE_DENOMINATOR has? Why isn't it just 2.0? Are they doing something unexpected here???"
    - const double TWO = 2.0;
      - This may not add confusion per se, but it does not ***add value***! Providing a named constants named "TWO" to take the place of the literal value 2.0 does NOT help in any way

Vertical Whitespace

- Readability can be *greatly improved* with proper use of vertical whitespace (i.e. blank lines separating "chunks" of code)

Examples

```
numeratorVal = -bCoeff + sqrt(bCoeff * bCoeff - 4 * aCoeff * cCoeff);
denominatorVal = 2 * aCoeff;
quadFormResult1 = numeratorVal / denominatorVal;
numeratorVal = -bCoeff - sqrt(bCoeff * bCoeff - 4 * aCoeff * cCoeff);
quadFormResult2 = numeratorVal / denominatorVal;
cout << "aCoeff = " << aCoeff << " bCoeff = " << bCoeff << " cCoeff = " <<
cCoeff << " result1: " << quadFormResult1 << " result2: " <<
quadFormResult2 << endl;
doContinueLoop = promptUserForMoreInputs(numIterationsCompleted);

----------------

denominatorVal = 2 * aCoeff;

numeratorVal = -bCoeff + sqrt(bCoeff * bCoeff - 4 * aCoeff * cCoeff);
quadFormResult1 = numeratorVal / denominatorVal;

numeratorVal = -bCoeff - sqrt(bCoeff * bCoeff - 4 * aCoeff * cCoeff);
quadFormResult2 = numeratorVal / denominatorVal;

cout << "aCoeff = " << aCoeff << " bCoeff = " << bCoeff << " cCoeff = " <<
        cCoeff << " result1: " << quadFormResult1 << " result2: " <<
        quadFormResult2 << endl;

doContinueLoop = promptUserForMoreInputs(numIterationsCompleted);
```

This looks like a "mush" of code, and is difficult to read and understand

The **exact** same code, separated into chunks is far superior and easy to read.

It doesn't matter at all that its "taller"

Constant Naming

- "Global constants" or even local constants that have a set value always, are named in ALL_CAPS
- Things that we specify as "const" because we don't want to allow them to be changed in a given scope are named like regular variables (this typically means function parameters that we are indicating will not be changed in the function)

```
const double INITIAL_SPEED = 10.0;

double computeNewSpeed(
     const double accelerationVal,
     const double timeAmt
     )
{
  double newSpeed;

  newSpeed =  INITIAL_SPEED + accelerationVal * timeAmt;

  return newSpeed;
}

int main()
{
  double resultSpeed;

  //Testing with a few specific cases here
  resultSpeed = computeNewSpeed(1.5, 6.0);
  cout << "New speed after 6 seconds of 1.5 accel: " << resultSpeed << endl;

  resultSpeed = computeNewSpeed(3.0, 6.0);
  cout << "New speed after 6 seconds of 3.0 accel: " << resultSpeed << endl;

  return 1;
}
```

This is a true constant. In this program, the initial speed will ALWAYS be 10.0

These values (may) change from one call to computeNewSpeed to the next, but we want to force them to remain constant within one call to computeNewSpeed.

Use of ALL_CAPS is reserved for meaning "this identifier will always have this specific known value".

Indentation Consistency

- Pick an indentation amount, and use that same amount every time code needs to be indented (typically within { and })
  - I highly recommend using exactly two spaces for indentation
    - Enough to be clearly visible, not so much that a few indentations causes the line to start halfway across the terminal

Such simple code, but so hard to read and interpret!

```
  if (curIndex == 0)
  {
   isFirstTime = true;
    }
else
  {
   for (i = curIndex; i < NUM_VALS; i++)
   {
    for (j = i; j < NUM_VALS; j++)
   {
          productResult = i * j
   }
          productResult *= 3.0;
  }
}
```

Exact same code – far clearer!
Obvious now that the "*= 3.0" part is in the "i" loop, but outside the "j" loop, etc.

```
 if (curIndex == 0)
 {
   isFirstTime = true;
 }
 else
 {
   for (i = curIndex; i < NUM_VALS; i++)
   {
     for (j = i; j < NUM_VALS; j++)
     {
       productResult = i * j
     }

     productResult *= 3.0;
   }
 }
```

Curly Brace Consistency

- Pick an approach for curly brace placement, and use that same aapproach **always**
  - I highly recommend placing curlys on their own lines, aligned with what they are associated with
    - Provides some vertical whitespace automatically, and allows very simple "curly matching"

Yikes! No curly consistency, very very confusing.

```
if (curIndex == 0) {
  isFirstTime = true;
}
else
{
  for (i = curIndex; i < NUM_VALS; i++)
    {
      for (j = i; j < NUM_VALS; j++) {
        productResult = i * j }

      productResult *= 3.0;
    } }
```

So much easier to read and interpret, and determine which closing curly goes with which opening curly, etc…

```
if (curIndex == 0)
{
  isFirstTime = true;
}
else
{
  for (i = curIndex; i < NUM_VALS; i++)
  {
    for (j = i; j < NUM_VALS; j++)
    {
      productResult = i * j
    }

    productResult *= 3.0;
  }
}
```


Consistent Whitespace Around Operators

- This comes down to searchability again – if you want to jump to the code where you added aCoeff and bCoeff, you should be able to know exactly what to search for
- Inconsistency causes difficulty in guessing what to search for..
- Example options:

```
aCoeff + bCoeff
aCoeff+bCoeff
aCoeff +bCoeff
aCoeff+ bCoeff
aCoeff  +bCoeff
aCoeff +  bCoeff
<etc>
```

If I know that I **always** include one space on each side of every binary operator, I only have to search for the first option.

If I'm inconsistent, I may have to try several searched before I find what I'm looking for.

Use of Comments

- Comments should add value!

```
//Compute the factorial of the user specific value
factVal = computeFactorial(userValue);

//----------

//Use the quadratic formula to compute one of the solutions
//Reference: https://en.wikipedia.org/wiki/Quadratic_equation
numeratorVal = -bCoeff + sqrt(bCoeff * bCoeff - 4 * aCoeff * cCoeff);
denominatorVal = 2 * aCoeff;

quadFormResult1 = numeratorVal / denominatorVal;

//Now compute the other possible solution too
numeratorVal = -bCoeff - sqrt(bCoeff * bCoeff - 4 * aCoeff * cCoeff);
quadFormResult2 = numeratorVal / denominatorVal;
```

Useless comment! The reader can read code, especially code that uses good naming, etc.

Comments are NOT meant to just repeat what the code says.

This comment allows the reader to know what the purpose of the next chunk of code is. The reader need not have to recognize it as the quadratic formula, instead, they can read the comment and skip that "complex" chunk of code if its not what they're looking for

Use of Comments, Page 2

- Using comments does NOT excuse you from using otherwise good style

```
//Note: Option 4 is the one for computing the average
if (userChoice == 4)
{

//------------

//This is the quadratic formula, where "a", "b", and "c" are
//the coefficients of the function we're trying to solve
result = -b + sqrt(b * b - 4 * a * c);
```

4 is still a "magic number"! A comment describing the 4 doesn't change that.

a, b, and c are still terrible variable names! Even though the comment does a good job of describing them, they should be named better for searchability

- Humorously, in both of these examples, it probably would have been faster to just use the proper naming than it was to have to type out a comment describing why you didn't use proper naming

Use of Comments, Page 3

- Remember: you need to take credit for your source code
- Include a "header block" in every source code file you generate
  - Must include at least:
    - Your name
    - The approximate date / date range that the code was developed / updated
    - A brief description of the purpose of the source file
  - The purpose should describe the **purpose** as opposed to *why* you wrote it.
  - Example of a good purpose description:

```
//Purpose: This program will compute the median quiz score for all students
//after dropping each student's lowest score.  Results will be printed
//to the console for easy import into the records book.
```

  - Example of an insufficient purpose description:

```
//Implement project 1 for EECS402
```


Avoid Duplicated Code

- Duplicated code leads to duplicated bugs and maintenance difficulty

The same algorithm is used in two separate places in this code.

While debugging the main function, maybe we realize the "= 0" is a bug and should be "= 1" instead. Will we remember that the code is duplicated elsewhere and that we likely need to find it and fix it too?

Or – would we fix the one in main since that's the one we were focused on, and then have to debug the one in doSomeStatisticalStuff later when we realize its not working either?

In this case, the variable names are different too – if this were in a large code base, remembering it is there multiple times, finding all instances of it, etc., would be very challenging.

```
#include <iostream> //Need for cout.
using namespace std;

void doSomeStatisticalStuff(int inVal, int inMeanVal);

int main(void)
{
  int fact = 1;
  int val = 5;
  int i;

  for (i = 0; i <= val; i++)
  {
    fact *= i;
  }

  cout << "Fact. of " << val << " is: " << fact << endl;

  doSomeStatisticalStuff(8, 623);

  return (0);
}

void doSomeStatisticalStuff(int valToFindFactFor, int inMeanVal)
{
  int curVal;
  int resultVal;
  int finalAnswer;

  resultVal = 1;
  valToFindFactFor = 8;
  for (curVal = 0; curVal <= valToFindFactFor; curVal++)
  {
    resultVal *= curVal;
  }

  finalAnswer = resultVal + inMeanVal;

  cout << "In doSomeStatisticalStuff, computed: " << finalAnswer << endl;

}
```

Avoid Duplicated Code, Page 2

- Using functions is one of the main ways to avoid duplicated code

The factorial algorithm is now contained in "computeFactorial". Once we realize there's a bug, we debug it, fix it, and its fixed for all uses of the call to computeFactorial!

- NOTE: The amount or complexity of the duplicated code is not the way to think about it!

```
#include <iostream> //Need for cout.
using namespace std;

//Function: Returns the factorial of the number provided
int computeFactorial(int num);

//Function: Does some super useful statistics calculations
void doSomeStatisticalStuff(int inVal, int inMeanVal);

int main(void)
{
  int fact;
  int val = 5;

  fact = computeFactorial(val);

  cout << "Fact. of " << val << " is: " << fact << endl;

  doSomeStatisticalStuff(8, 623);

  return (0);
}

void doSomeStatisticalStuff(int valToFindFactFor, int inMeanVal)
{
  int curVal;
  int resultVal;
  int finalAnswer;

  valToFindFactFor = 8;
  resultVal = computeFactorial(valToFindFactFor);

  finalAnswer = resultVal + inMeanVal;

  cout << "In doSomeStatisticalStuff, computed: " << finalAnswer << endl;
}

int computeFactorial(int num) //header
{
  int result = 1; //Value to return..
  int i;          //Loop variable

  for (i = 0; i <= num; i++)
  {
    result *= i;
  }
  return (result);
}
```


An Interesting Case

- Consider this code:

```
for (i = 0; i < 10; i++)
{
```

- Is "i" a bad variable name?
  - Not in this case. Loop counting variables are very very often named "i" and "j"
  - Any reader who sees variables with that name will assume it is a loop variable
    - If it isn't, then it is definitely poorly named!
- Is 0 a magic number:
  - Probably not. Count controlled loops, like the one above, almost always start at 0. The literal value 0 tells the reader you're not doing anything strange, just starting at 0 like always
- Is 10 a magic number?
  - Yes, almost certainly.
  - Why are you stopping at 10? What significance does that value have?
  - Giving it a name allows the reader to understand what the purpose of the loop is, as opposed to knowing the exact number
- This is better:

```
for (i = 0; i < NUM_DAYS_IN_TRIAL; i++)
{
```

- The reader now knows this will iterate based on the number of days in the trial
  - They may not directly know that it will loop 10 times exactly, but knowing that the loop iterated once per day in the trial is what is most important

Reminder about Disclaimer

- These slides were a collection of some examples of good and bad style, with attempted explanations as to why they are important
- There are other style problems that come up sometimes!
  - Just because something isn't explicitly included in this document doesn't mean we won't deduct for poor style that is observed during grading
- I can't stress enough that good style should not be an afterthought
  - Use proper style through the entire coding process!
  - If you wait to "make it look good" until right before you submit, you'll run out of time or mess it up
  - Ensure that your code is ALWAYS styled well
    - Good style isn't just a requirement for this class, but it also helps others (and YOU yourself!) read, understand, maintain, and debug your code!