ECS 40 **Program Standards**

A computer program has three audiences: 1) the computer that compiles and runs it; 2) the programmer[s] who create it; and 3) other programmers that must read, evaluate, and/or maintain it. Some programming style standards are required simply for efficient code production by the second audience. When a program will be long lived, and hence the third audience will often read a program, the need arises for extensive comments. You should have had a taste of this thorough commenting style in ECS 30. Because our programs are short lived, and your limited time to spend on programming, we will forego addressing the third audience. The following standards are the minimum needed to efficiently write correct code for small programs. Remember that 20% of your program grade is determined by your adherence to these standards. You will find **.vimrc** in my home directory provides *most* of the proper spacing and indents when you use vi.

The body of a function should not be more than 35 lines long including comments, except in the case of switch statements. Long switch statements should be placed in functions by themselves. For long switch statements, if a case has extensive code, then place the code in a short function and have the case call the function.

Capitalization and Names

Keywords, standard functions, data types, and single word user-defined identifiers are in lowercase. Multi-word identifiers, e.g., getInfo, should contain uppercase letters at the start of each word after the first word. The exception to this rule is for common abbreviations, such as OK and AM. Class names and typedefs are similar to other identifiers, except that they start with an uppercase letter. Macros, and enumerated constants are entirely in UPPERCASE, with underscores used to separate words.

You should choose function and variable names that makes your code self-documenting. To allow for more self-documenting variable and function names, Standard C style uses an underscore to join two word in a name, e.g. get_name. We will use getName. Though it quite common, and often acceptable to use single letter variable names, e.g. i or j, for counter variables, you should think about using counter variable names that relate to the physical object represented by the data structure, e.g. row, line, or col.

Indentation and Spacing

You should use an indent of two spaces. You should indent one more level whenever you start a new block of code with a '{' (which be on a line by itself), and when there is single statement following a flow control expression. Since the word wrapping of printers will not automatically indent lines properly, you must not have any line longer than 80 characters. If you have a multi-line statement, you must break the statement up so that expressions are kept together on one line. By starting successive lines with Boolean operators, you will find it easier to recognize that they are an extension of the previous line. Every line after the first line is indented. Since literal strings may not cross line boundaries, you must break long strings into multiple parts as in case 2 of the switch statement below. For switch statements, if there is only one short statement associated with a case, then place it adjacent to the case, otherwise start a new indented block.

Use blank lines to differentiate related sections of your code. For example, if four adjacent lines of code deal with a file's name, and the following six lines deal only with the file's contents, then you should place a blank line between the two sections. You should also place blank lines between the variable declarations of a function and the rest of the body. There should be two or three blank lines between the end of a function and the start-of-function comments of the next function. When in doubt add an extra blank line--they do not add errors, and often help you to quickly locate the code you are looking for.

Colons, and binary operators should have one space on either side of them. Commas and semi-colons should have one space after them. For example, `z = x + foo(y, w); break;` and not `z=x+foo(y,w);break;`

Open curly brackets, '{', should appear directly below the first letter of the line above, not at the end of that line. Closed curly brackets should be aligned with the associated open curly bracket. This is called Allman style brackets.

```
if (!strcmp(programs[i].getName(), programs[i + 1].getName())
  && programs[i + 1].getStartTime() % 30 != 0)
      // same name and later one is a continuing program
  programs[i].setEndTime(programs[i + 1].getEndTime());

switch (num)
{
  case 0 : cout << "zero\n"; break;
  case 2 :
    cout << "This is number is even and prime. We are looking for odd numbers or";
      << " zero. Please enter a different number\n";
    break;
  case 3 : cout << "Good choice!\n"; break;
} // switch
```

Comments

You should use comments to make the purpose of each part of a program clear. All comments should help you to better understand your own code when you return to it later in the quarter. Write your comments as if you were writing to yourself two months from now. Do not make the mistake that because a particular block of code or function is obvious to you now, that it will be clear in a month. In all likelihood, if you spent a while figuring out how to write a particular part of your code, then it's actions will not be clear in a month. I promise, the time you spend on commenting now will be well rewarded later in the quarter when you must modify or re-use a function you wrote a month earlier. For this course, we will have three kinds of comments: 1) End-of-block comments; 2) If-else comments; and 3) Line comments. I have tried to keep the formatting requirements to a minimum for each of these. Try to concentrate on what is important to know about your code.

End of Block Comments:

Each end of block '}' must be labeled with the corresponding keyword that started the block unless it is a do-while closing bracket. The end-of-function brackets should be labeled with the name of the function. The end-of-class brackets should be labeled with the name of the class. When I am writing code, and I type a '{', I immediately type below it a '}' and its appropriate end-of-block comment. I then insert the appropriate code between these two brackets. Besides ensuring that I've labeled my '}' correctly it also ensures that my brackets are always paired and indented properly.

If-Else Comments:

If statements and else statements should have short descriptions of what the tests mean. This may describe what is excluded or what is included. The comment placement may be before, adjacent, or after the if or else. You may omit such comments from if statements when the test is truly trivial, e.g., `if(x == 1)`.

```
  if (strcmp(ptr->program.getName(), ptr->next->program.getName())
    || ptr->next->program.getStartTime() % 30 == 0)      // different shows
    ptr->program.setEndTime(ptr->next->program.getStartTime() - 1);
  else // same name
    ptr->program.setEndTime(ptr->next->program.getEndTime());
```

Line Comments:

Your line comments will not be graded. They are completely up to you. Use comments to explain lines or blocks of code that are not immediately obvious. You can place them above, below, or at the end of the same line. Common places for line comments are near flow control statements.

```
int Schedule::findChannel(int channelNum)
{
  int index;

  for (index = 0; index < channelCount; index++)
  {
    if (channels[index].getChannelNum() == channelNum)  // found the channel
      return index;
    else  // channel not found
      if (channels[index].getChannelNum() > channelNum)  // beyond the channel
        break;
  }  // for each channel

  // In any case, index will now be where the channel belongs.

  if (channelCount == size)  // Array is full
    resize();

  for (int i = channelCount++; i > index; i--)  // move channels to right in array
    channels[i] = channels[i - 1];

  channels[index].clear();

  return index;
} // findChannel()
```