Communicating in Code: Layout and Style

Layout and Style
• Like naming, the goal is to communicate
• Again like naming, sometimes conventions are in place
  − Adhering to the convention in place will usually lead to more readable code than using your own “better” convention
• Goal of layout and style is to increase clarity.

Fundamental Theorem of Formatting
• Good visual layout shows the logical structure of the program.
• Studies show that organization is as important to understanding as the “details”

White Space
• Used to indicate logical grouping
  − Spacing between characters
  − Indentation
  − Blank lines
**Indentation**

- Can clarify structure, especially in odd cases.
- Studies show that 2-4 space indentation works best.
  - More indentation might “appear” better, but is not.
- Now, usually editors provide automatically.
  - But, variations for some statements:
    - switch/case
    - if/elseif
- Brace conventions differ, but be consistent.

**Parentheses**

- Parentheses can resolve ambiguity
  - Particularly important since order of operations can be problematic
- Better to use more parentheses than you think you need
- Coupled with white space, can more quickly highlight the grouping/ordering of operations
  
  ```
  leap_year = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
  ```

**Example Brace Conventions**

```java
while (something) {
  blahblahblah
}
```

```java
while (something) {
  blahblahblah
}
```

```java
while (something) {
  blahblahblah
}
```
Braces

- Like parentheses, use more braces than you need.
- One-statement operation often becomes more, later.

```cpp
if (a > b) {
    max = a;
}
```

Braces

- Like parentheses, use more braces than you need.
- One-statement operation often becomes more, later.

```cpp
if (a > b) {
    max = a;
    cout << "Set a new maximum." << endl;
}
```
Avoiding Complex Expressions

- Goal is not to write most concise and clever code.
- Break up expressions to make them clearer
- The "?" operator can be especially problematic

```c
*x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]));
```

Use “Natural Form” for Expressions

- State conditional tests positively

```c
if (!(z>=0) && !(z<a))
```

Avoiding Complex Expressions

- Goal is not to write most concise and clever code.
- Break up expressions to make them clearer
- The "?" operator can be especially problematic

```c
*x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]));
if (2*k < n-m)
    *xp = c[k+1];
else
    *xp = d[k--];
*x += *xp;
```

Use “Natural Form” for Expressions

- State conditional tests positively

```c
if (!((z>=0) && !(z<a)))
    if ((z<0) && (z>=a))
        if ((z<0) && (z>=a))
            This can vary if the way it's expressed better matches the underlying algorithm
```
Use “idomatic” forms

- There are “common” ways of expressing certain things.
  - e.g. Use a for loop appropriately – try to keep all loop control in the for statement, and keep other operations outside of the for statement

```java
for (i=0; i<n; i++)
a[i] = 0.0;
```

Use “idomatic” forms

- There are “common” ways of expressing certain things.
  - e.g. Use a for loop appropriately – try to keep all loop control in the for statement, and keep other operations outside of the for statement

```java
for (i=0; i<n;i++)
a[i] = 0.0;
for (i=0; i<n; a[i++] = 0.0);
```

Use “idomatic” forms

- There are “common” ways of expressing certain things.
  - e.g. Use a for loop appropriately – try to keep all loop control in the for statement, and keep other operations outside of the for statement

```java
for (i=0; i<n; i++)
a[i] = 0.0;
for (i=0; i<n; a[i++] = 0.0);
for (i=0; i<n;)
{
a[i] = 0.0;
i++
}
```

Idiomatic forms

- e.g. use if elseif else form

```java
if (cond1) {
dothis1();
}
else {
if (cond2) {
dothis2();
}
else {
if (cond3) {
dothis3();
}
else {
dothis4();
}
}
}
```
Idiomatic forms

- Use if elseif else form
  
  ```java
  if (cond1) {
      dothis1();
  } else if (cond2) {
      dothis2();
  } else if (cond3) {
      dothis3();
  } else {
      dothis4();
  }
  ```

If statements

- Read so that you look for the “true” case rather than a “stack” of else cases
  
  ```java
  if (a > 3) {
      if (b < 12) {
          while (!EOF(f)) {
              dothis();
          }
      } else {
          cerr << “Error 2” << endl;
      }
  } else {
      cerr << “Error 1” << endl;
  }
  ```

If statements

- Read so that you look for the “true” case rather than a “stack” of else cases
  
  ```java
  if (a <= 3) {
      cerr << “Error 1” << endl;
  } else if (b >= 12) {
      cerr << “Error 2” << endl;
  } else {
      while (!EOF(f)) {
          dothis();
      }
  }
  ```

Avoid Magic Numbers

- Rule of thumb: any number other than 0 or 1 is probably a “magic number”
- Can lead to tremendous debugging problems when these numbers are changed
- Instead, define constants to give names to those numbers.
Layout for Control Structures

- Put control in one line when possible
- Single indentation level for what it affects
- Group each part of a complicated condition on its own line

Layout of Individual Statements

- White space can improve readability
  - Spaces after commas
    ```
    EvaluateEmployee(Name.First, EmployeeID, Date.Start, Date.End);
    EvaluateEmployee(Name.First, EmployeeID, Date.Start, Date.End);
    ```
  - Spaces between parts of conditions
    ```
    if (((a<b)||(c>d))&&((a+b)<(c-d))&&((c-d)>2))
    if (((a<b) || (c>d)) && ((a+b)<(c-d)) && ((c-d)>2))
    if (((a<b) || (c>d)) && ((a+b) < (c-d)) && ((c-d) > 2))
    ```

Layout of Individual Statements

- Line up related definitions or assignments
  ```
  StudentName     = ProcessInputName();
  StudentID       = ProcessInputID();
  StudentHometown = ProcessInputName();
  ```
- Don't use more than one statement per line.
  - Likewise, define only one variable per line.
- Avoid side-effects (such as including the ++ operator when doing something else).

When a Line is Too Long

- Make it clear that the previous line is not ending (e.g. end with an operator)
- Keep related parts of the line together (don’t break single thought across line)
- Use indentation to highlight that there’s a continuation
- Make it easy to find the end of the continued line.
Layout of Routines

- Use standard indentation approach for arguments.
- Use blank lines to separate parts of routines or blocks of common actions.
- Use comments (will return to) to identify major breaks in conceptual flow.

Layout of Files

- Clearly separate (multiple line breaks) different routines in the same file.
  - Don’t want to accidentally “merge” or “break” individual routines.
  - Sequence files in a logical manner.
    - In order of header file definition.
    - In alphabetical order.
    - Constructor, accessor, destructor, other.