Programming Studio CSCE 315
Communicating in Code: Naming, Style, and Layout

18 January 2010
Outline

1. Purpose of programming
2. Names
3. Layout and Style
What is the purpose of programming?

- Instructing a computer?
What is the purpose of programming?

- Instructing a computer?

- Certainly, but communicating to the computer is just one part of programming.

- Programming is also about communicating
  - to yourself (later on) and
  - to others

- Purpose of coding is to effectively express ideas of what you want the computer to do.
Outline

1. Purpose of programming
2. Names
3. Layout and Style
Variable names

- A name of a variable conveys information of its purpose
- Should be
  - informative
  - concise
  - memorable
  - pronounceable (if possible)
- The broader the variable’s scope, the more information should go into the name
  - include a brief comment for each global

```c
int npending = 0 // current length of input queue
```
Descriptive names for globals, short names for locals

- Compare

```c
for (theElementIndex = 0; theElementIndex < numberOfElements;
    ++theElementIndex)
{
    elementArray[theElementIndex] = theElementIndex;
}
```
Descriptive names for globals, short names for locals

- **Compare**

```plaintext
for (theElementIndex = 0; theElementIndex < numberOfElements;
    ++theElementIndex)
{
    elementArray[theElementIndex] = theElementIndex;
}
```

- **with:**

```plaintext
for (i = 0; i < nelems; ++i)
{
    elem[i] = i;
}
```
Naming conventions

- Naming conventions vary
  - Use what your project requires
  - Use what the programming/language/library community expects
  - Be consistent

- Examples
  - use names that begin or end with a `p` for pointers
  - initial capital letters for `Globals`
  - all capitals for `CONSTANTS` and `MACROS`
  - start with `n` a variable describing a count or size
  - `i, j` for loop indices
  - `c, ch` for characters
  - `s, ch` for a string
Example: Hungarian notation

- Base types
  - wn Window
  - scr Screen Region
  - fon Font
  - ch Character
  - pa Paragraph

- Examples: wnMain, scrUserWorkspace
Example: Hungarian notation

- Prefixes
  - `a` array
  - `c` count
  - `d` difference between two variables
  - `e` element of array
  - `g` global variable
  - `h` handle
  - `i` index into array

- Example: `iwnUserView = index into array of windows giving user views`
Spelling of names with multiple parts

- Two main conventions
  1. camelCaseNotation
  2. underscore_notation

- Often a matter of preference

- Sometimes language / library / project context determines
  - C++ standard library uses underscores
  - Java typically uses CamelCase

- CamelCase sometimes confusing, compare `numXMLElements` and `num_XML_elements`
Choosing names

- Choose a name that describes all what the function does
  - `Print()` vs. `PrintAndCloseFile()`

- Avoid meaningless names
  - `PerformAction()`, `HandleData()`

- Use active names for procedures (verb followed by a noun)
  - `anotherStudent()` vs. `addStudent()`

- For pure functions, sometimes no need for a verb
  - `size()`, `parent()`

- For predicates, choose the name that indicates what is returned unambiguously
  - `if (check_size(v)) ...
  - `if (is_full(v)) ...`
Choosing names: Be consistent

- What is wrong with this snippet?

```java
class UserQueue {
    int noOfItemsInQ, frontOfTheQueue, queue_capacity;
    public int noOfUsersInQueue() { ... }
}
```
Choosing names: Be consistent

- What is wrong with this snippet?

```java
class UserQueue {
    int noOfItemsInQ, frontOfTheQueue, queue_capacity;
    public int noOfUsersInQueue() { ... } }```

- many variations of spelling for `queue`
- `queue` in member names is redundant
  - `queue.queue_capacity`
Choosing names: Be consistent

- Better version of the previous example

```java
class UserQueue {
    int nitems, front, capacity;
    public int nusers () { ... }
}
```

- Now usage much clearer:

```java
queue.capacity++;  
n = queue.nusers();
```
Rely on language’s modularity features to keep names short

- Namespaces, packages, classes, modules etc. constrain a name’s visibility
- Explicit requests can bring names into scope

```cpp
namespace boost {
namespace lambda {
  template <class T1, class T2>
  lambda<...> bind(T1 t1, T2 t2) { ... };
}

// qualify name explicitly
boost::lambda::bind(f, x);

// bring a specific name into scope
using boost::lambda::bind;
bind(f, x);
bind(f, y)

// bring all names into scope
using namespace boost::lambda;
```
Outline

1. Purpose of programming

2. Names

3. Layout and Style
About layout

- Many conventions, source of endless debates
  - Consistency is key, so adhere to commonly accepted conventions, instead of inventing new “better” ones
- Layout of code can help or hinder communication
- Layout and style should be chosen to increase clarity
Fundamental "Theorem of Formatting"

- Good visual layout shows the logical structure of the program
Fundamental "Theorem of Formatting"

- Good visual layout shows the logical structure of the program
- Organization of code is as important to understanding as details
Whitespace

- Used to indicate logical grouping
  - Spacing between characters
  - Indentation
  - Blank lines

- Compare

```c
for(n=0;n<100;field[n++]=='\0');
```

```c
for(n = 0; n < 100; field[n++] = '\0');
```
Whitespace

- **Spaces after commas**

  ```java
  findEmployee(name.First, employeeID, date.Start, date.End);
  findEmployee(name.First, employeeID, date.Start, date.End);
  ```

- **Spaces between parts of conditions**

  ```java
  if (((a<b) || (c>d)) && ((a+b)<(c-d)) && ((c-d)>2))
  if (((a<b) || (c>d)) && ((a+b)<(c-d)) && ((c-d)>2))
  ```

  **Or more preferably**

  ```java
  if (((a<b) || (c>d)) && ((a+b)<(c-d)) && ((c-d)>2))
  ```
## 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
- Usually editors provide automatically
  - Use spaces, not tabs
- Indention and brace conventions differ, be consistent
Examples of brace conventions

```c
while (condition) {
    body
}
```

```c
while (condition)
{
    body
}
```

```c
while (condition) {
    body
}
```

```c
while (condition) {
    body
}
```
Layout of statements

- Line up related definitions or assignments

```java
studentName = validateName(n);
studentID = validateID(id);
studentHometown = validateTown(town);
```

- Usually only one statement per line
- Likewise, usually define only one variable per line
Layout of statements: 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 (do not break single thought across line)
- Use indentation to highlight that there is a continuation
- Make it easy to find the end of the continued line
Layout and Style

Layout of routines

- Use blank lines to separate parts of routines or blocks of common actions.
- Use comments (we will return to this later) to identify major breaks in conceptual flow.
Layout of files

- Clearly separate (multiple line breaks) different routines in the same file
- Sequence definitions in a logical manner within a file or within a class
  - In order of header file definition
  - Constructor, accessor, destructor, other
Parentheses

- Parentheses can resolve ambiguity
- Particularly important if order of operations not immediately clear
- Better to use more parentheses than you think you need
- Coupled with white space, can highlight the grouping/ordering of operations
  - The code lines below are semantically equivalent, but parentheses improve the clarity significantly

```python
leap_year = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;

leap_year = ((y%4 == 0) && (y%100 != 0)) || (y%400 == 0);
```
Avoid complex expressions

- Goal is clarity, not cleverness
  - conciseness is good, but not at the cost of clarity
- Break up expressions to improve clarity
- Example:

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

```c
if (2*k < n-m)
    *xp = c[k+1];
else
    *xp = d[k--];
*x += *xp;
```
Avoid negation in conditions

- Self-test: Can you read the condition aloud?
- Example

```java
if (!(block_id < actblks) || !(block_id >= unblocks))
```

```java
if ((block_id >= actblks) || (block_id < unblocks))
```
Be careful with side-effects

- Avoid side-effects (such as including the `++` operator when doing something else)
- Example:

```c
str[i++] = i;
```
Braces

- Like parentheses, use more braces than you need
- Some programmers use braces even for a one-statement body of a loop or an if-statement:

```java
if (a > b) max = a;
```

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

- Whatever your preference, be careful not to mislead

```java
if (a > b)
    max = a;
write_to_log(max);
```
Idiomatic forms

- Use established idiomatic forms

```c
for (int i = 0; i < n; ++i) array[i] = 1.0;
...
for (p = list; p != NULL; p = p->next) ...
```
Idiomatic forms

- Use established idiomatic forms

```cpp
for (int i = 0; i < n; ++i) array[i] = 1.0;
...
for (p = list; p != NULL; p = p->next) ...
```

- Do not write something like this:

```cpp
int i=0;
while (i <= n-1) array[i++] = 1.0;
```

- Don’t be inventive “in the small”
  - boring code == good code
Idiomatic forms

- Use else-ifs for multi-way decisions. Compare:

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

```java
if (cond1) {
    dothis1();
} else if (cond2) {
    dothis2();
} else if (cond3) {
    dothis3();
} else {
    dothis4();
}
```
Avoid magic numbers

- Rule of thumb: any number other than 0 or 1 is probably a “magic number”
- When they change, magic numbers can lead to tremendous debugging problems
- Define constants that name magic numbers. E.g.:

```c
draw(23, 2, ' ');

enum {
    ...
    MINCOL = 1; // left edge
    ...
    MAXROW = 24; // bottom edge
    ...

draw(MAXROW-1, MINCOL+1, ' ');
```
About coding conventions

- Decide on coding conventions upfront, hard to change later
- Accept existing conventions if working on a legacy code base
  - Otherwise source files of a long lived project will have a dozen different conventions