

Code Layout (2)

September 21, 2017

Byung-Gon Chun

(Slide credits: George Candea, EPFL)

Code Layout (Statements)

Braces

```
for (i=1 ; i<N ; ++i) {  
    Table[i] = Table[i-1] + 1;  
}
```

Same line as previous statement

Always on a line by itself

Use braces even for
single-statement blocks

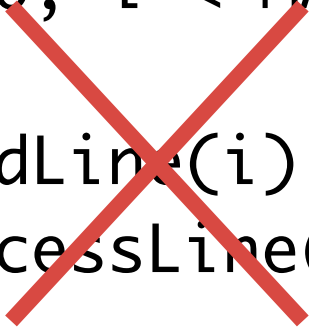
```
for (i=1 ; i<N ; ++i)  
    Table[i] = Table[i-1] + 1;
```

Statement Indentation

Use 4 spaces

```
for (int i=0; i<MAX_LINES; ++i) {  
    ReadLine(i);  
    ProcessLine(i);  
}
```

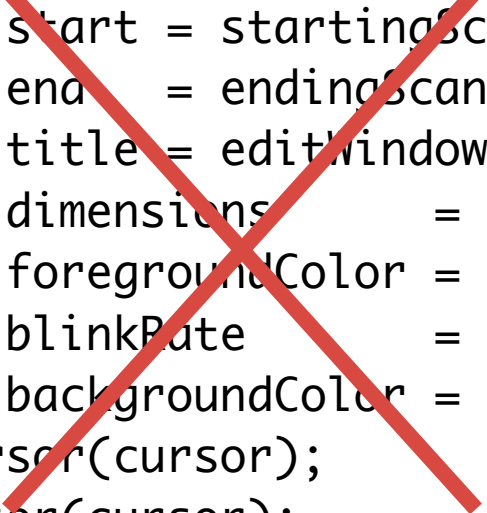
```
for (int i=0; i < MAX_LINES; ++i)  
{  
    ReadLine(i);  
    ProcessLine(i);  
}
```



Demarcate Blocks with Whitespace

- Some blocks are not "naturally" demarcated
 - use blank lines to "ungroup" unrelated statements

Demarcate Blocks with Whitespace



```
cursor.start = startingScanLine;  
cursor.end   = endingScanLine;  
window.title = editWindow.title;  
window.dimensions = editWindow.dimensions;  
window.foregroundColor = userPreferences.foregroundColor;  
cursor.blinkRate = editMode.blinkRate;  
window.backgroundColor = userPreferences.backgroundColor;  
SaveCursor(cursor);  
SetCursor(cursor);
```

```
window.dimensions = editWindow.dimensions;  
window.title = editWindow.title;  
window.backgroundColor = userPreferences.backgroundColor;  
window.foregroundColor = userPreferences.foregroundColor;
```

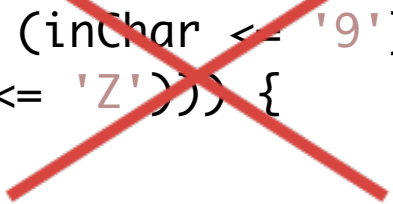
```
cursor.start = startingScanLine;  
cursor.end = endingScanLine;  
cursor.blinkRate = editMode.blinkRate;  
SaveCursor(cursor);  
SetCursor(cursor);
```

Alignment

```
static final List<String> favorites = Arrays.asList ("Comedy", "Action", "Musical", "Drama");  
static final List<Integer> monthIdx = Arrays.asList (2, 5, 1, 11 );  
  
static final List<String> favorites = Arrays.asList ("Comedy", "Action", "Musical", "Drama");  
static final List<Integer> monthIdx = Arrays.asList (2, 5, 1, 11 );
```

Split Complex Predicates Cleanly

```
if (((('0' <= inChar) && (inChar <= '9')) || (('a' <= inChar) && (inChar <= 'z')) || (('A' <= inChar) && (inChar <= 'Z')) {  
    // ...  
}
```



```
if ( ( ( ('0' <= inChar) && (inChar <= '9')) ||  
      ( ('a' <= inChar) && (inChar <= 'z')) ||  
      ( ('A' <= inChar) && (inChar <= 'Z')) )  
    ) {  
    // ...  
}
```


Multi-Line Statements

- Make incompleteness obvious

```
while ( (pathNames[startPath + position] != ';') &&  
        ((startPath + position) <= pathName.length()) )
```

```
// ...
```

```
totalBill = totalBill + customerPurchases[customerID] +  
    SalesTax(customerPurchases[customerID]);
```

```
// ...
```

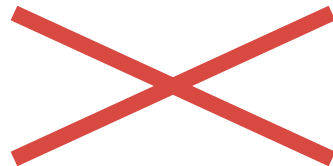
```
DrawLine (window.north, window.south, window.east, window.west,  
    currentWidth, currentAttribute);
```

Multi-Line Statements

- Can also put continuation at start of line

```
while (pathName[startPath + position] != ';')  
    && ( (startPath + position) <= pathName.length() )  
  
    // ...
```

```
totalBill = totalBill + customerPurchases[customerID]  
    + SalesTax(customerPurchases[customerID]);  
  
    // ...
```



One Statement per Line

~~i = 0; j = 0; k = 0; DestroyBadLoopNames (i, j, k);~~

~~if (map.isEmpty()) ++i;~~

Each Line Should Do One Thing

~~PrintMessage (++n, n+2);~~

Naming Variables

Choosing Good Names

- Think very deeply about variable names !

```
int n;  
int noErr;  
int nCompConns;
```

```
int connectionIndex;  
int numErrors;  
int numCompletedConnections;
```

Choosing Good Names

Purpose of Variable	Good Names, Good Descriptors	Bad Names, Poor Descriptors
Running total of checks written to date	<i>runningTotal, checkTotal</i>	<i>written, ct, checks, CHKTTL, x, x1, x2</i>
Velocity of a bullet train	<i>velocity, trainVelocity, velocityInMph</i>	<i>velt, v, tv, x, x1, x2, train</i>
Current date	<i>currentDate, todaysDate</i>	<i>cd, current, c, x, x1, x2, date</i>
Lines per page	<i>linesPerPage</i>	<i>lpp, lines, l, x, x1, x2</i>

Name Length

- “Aurea mediocritas”
– debug effort minimized when variable names have 10-16 chars [Gorla & Benander 1990]
- Just make sure short names are clear enough
– keep in mind to **read-optimize**, not write-optimize !
- Use longer names for rarely used variables [Shneiderman 1980]

Loop Indexes

- Customary: i, j, k, ... *BUT*
- Use meaningful name when outside the loop

```
recordCount = 0;
while (moreScores()) {
    score[recordCount] = getNextScore();
    ++recordCount;
}

neededCapacity = recordCount * bytesPerRecord;
...
```

Loop Indexes

- Use full index names in long loops
- Avoid index cross talk in nested loops
 - often i and j get mixed up

```
for (teamIndex=0; teamIndex<teamCount ; ++teamIndex) {  
    for (eventIndex=0; eventIndex<eventCount[teamIndex] ; ++eventIndex) {  
        score[teamIndex][eventIndex] = 0;  
  
        // ...  
    }  
}
```

- Only use i, j, k for loop indexes, no other variables
 - otherwise developers get confused

Computed-Value Qualifiers

- Frequently used
 - Total, Sum, Average, Max, Min, etc.
- Place at end of variable name
 - usersTotal, consumptionAverage, etc.
- Exception: Num
 - use as prefix: numUsers
 - avoid using it (choose Count or Total instead)

Use Common Opposites

- begin/end
- first/last
- locked/unlocked
- min/max
- next/previous
- old/new
- opened/closed
- visible/invisible
- source/target
- source/destination
- up/down

Purpose-Driven Naming

- Use one variable for one purpose only
- Example: solve $ax^2 + bx + c = 0$

```
tmp = Sqrt(b*b - 4*a*c);  
solution[0] = (-b + tmp) / (2*a);  
solution[1] = (-b - tmp) / (2*a);  
// swap the solutions  
tmp = solution[0];  
solution[0] = solution[1];  
solution[1] = tmp;
```

Purpose-Driven Naming

- Explicit names leads to clearer code

```
discriminant = Sqrt( b*b - 4*a*c );  
solution[0] = ( -b - discriminant ) / ( 2*a );  
solution[1] = ( -b + discriminant ) / ( 2*a );
```

```
oldSolution = solution[0];  
solution[0] = solution[1];  
solution[1] = oldSolution;
```

Status Variables

- Avoid using *flag* in the name
 - it always means something, so name it accordingly

```
if (flag) ...  
if (statusFlag & 0x0F) ...  
if (16 == printFlag) ...  
if (0 == computeFlag) ...
```

```
flag = 0x1;  
statusFlag = 0x80;  
printFlag = 16;  
computeFlag = 0;
```

Status Variables

- Use explicit variable names instead

```
if (dataReady) ...  
if (characterType & PRINTABLE_CHAR) ...  
if (ReportType_Annual == reportType) ...  
if (True == recalcNeeded) ...
```

```
dataReady = true;  
characterType = CONTROL_CHARACTER;  
reportType = ReportType_Annual;  
recalcNeeded = false;
```


Avoid Hybrid Coupling

- Avoid variables with hidden meanings – have different values for the variable mean different things
- E.g., bytesRead
 - indicates # of bytes received
 - except if < 0 , in which case indicates error value
- E.g., customerId
 - indicates a customer number
 - except if $> 500,000$, in which case you subtract 500,000 to get the number of a delinquent account

Separate Namespaces

- Java packages
 - can distinguish Database::Table from GUI::Table
- C++ *namespace* keyword
- Languages without namespace support
 - use naming conventions (e.g., in C)

Use Enums Liberally

```
public class Card {  
    public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX,  
        SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }  
  
    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }  
  
    ...  
}
```

Boolean Variables

- Use typical boolean names
 - done, error, found, success, ok, ...
- Can only take on true/false values
 - bad names: status, sourceFile, ...
 - using the "is" prefix protects you from bad names (isStatus... ?)
- Favor using *positive* boolean names
 - negatives are cumbersome to negate

Shortening Names

- Some abbreviation tricks
 - remove non-leading vowels (screen -> scrn, pages -> pgs)
 - remove articles (and, the, ...)
 - remove useless endings (-ing, -ed, ...)
- Abbreviate consistently
 - can produce a "Standard Abbreviations" document
- Should pass the telephone test [Kernighan & Plaugher]

Data Declaration

- Put in comments what cannot go in name
- Put unambiguous units in name of variable
 - if not possible, then at least put them in comments

```
Color previousColor; // color prior to window update
Color currentColor;
Color nextColor;      // color for next update
Point previousTop;    // previous location in pixels
Point previousBottom;
...
Temperature temperatureInKelvin;
```

Names to Avoid

- Words that sound similar
 - wrap vs. rap -> remember telephone test !
- Names should differ in at least two characters
- Avoid numerals (as in file1, file2)
- Avoid commonly mis-spelled words
 - acummmulate, calender, concieve, independant, reciept
- Use a single natural language for project
 - and dialect/spelling (e.g., US vs. UK English)

Using Variables

Declarations & Initialization: The Dangers

- Improper initialization → fertile source of bugs
 - result: variable has unexpected value
- Know your programming language's semantics!
- Sources of bugs
 - not initialized at all (default in Java, random in C/C++)
 - inconsistent value (e.g., constructor initializes only part of the class)

Declarations & Initialization: What You can Do

- Move initialization close to declaration

```
double area = 3.14 * radius * radius;  
Board myBoard = new Board(8,8);
```

- Move declaration/initialization close to point of first use

```
int accountIndex = 0;  
// code using accountIndex  
// ...  
boolean done = false;  
while (!done) {  
    // ...  
}
```

Exception to the rule...

```
double dist;  
try {  
    // lots of code  
    dist = distance();  
} catch (Exception e) {  
    // handle ...  
}  
retVal = 2*dist;
```

Declarations & Initialization: What You can Do

- Be disciplined with initialization
 - *always make values explicit, even if language provides defaults*
 - *initialize all class members in the **constructor***
 - *method parameters: instead of initializing, these should be checked for validity*

```
double hypotenuse(double c1, double c2) throws BadParametersException {  
    if (c1<=0 || c2<=0) {  
        throw new BadParametersException();  
    }  
    return Math.sqrt(c1*c1 + c2*c2);  
}
```

- Use final to protect from reassignment
 - *if you want constants, make them final static \Rightarrow initialized at class load time*

Binding Variables to Values

- Binding vs. flexibility
 - *earlier binding → less flexibility → less complexity → less risk of introducing errors*
 - *later binding → more flexibility → more complexity → more risk of introducing errors*

- Can bind at

- *coding time (i.e., use magic values)*

```
titleBar.color = 0xFF;
```

- *compile time (i.e., use named constants)*

```
private static final int TITLE_BAR_COLOR = 0xFF;  
// ...  
titleBar.color = TITLE_BAR_COLOR;
```

Variable-to-Value Binding

- *at run time (e.g., use properties file or Windows registry)*

```
Properties windowProperties = new Properties();  
FileInputStream in = new FileInputStream(propertiesFileName);  
windowProperties.load(in);  
titleBar.color = windowProperties.getProperty("TITLE_BAR_COLOR");
```

- *object instantiation time: every time you create a window*
- *just-in-time: every time you draw a window*

Using Variables

- Know your language semantics
- Initialize close to declaration
- Declare close to first use
- Flexibility vs. reliability trade-off

Variable Span, Liveness, and Scope

Variable Span

Localize References to Variables

```
public boolean onTouch(View v, MotionEvent event) {
    ByteBuffer bb = null;
    int action = event.getAction();
    int actionCode = action & MotionEvent.ACTION_MASK;
    int actionPidIndex = action >> MotionEvent.ACTION_POINTER_ID_SHIFT;
    int msgType = 0;
    MultiTouchChannel h = mHandler;

    switch (actionCode) {
        case MotionEvent.ACTION_MOVE:
            if (h != null) {
                bb = ByteBuffer.allocate(event.getPointerCount() *
                    ProtocolConstants.MT_EVENT_ENTRY_SIZE);
                bb.order(h.getEndian());
                for (int n=0; n < event.getPointerCount(); n++) {
                    mImageView.constructEventMessage(bb, event, n);
                }
                msgType = ProtocolConstants.MT_MOVE;
            }
            break;
        case MotionEvent.ACTION_DOWN:
            if (h != null) {
                bb =
                    ByteBuffer.allocate(ProtocolConstants.MT_EVENT_ENTRY_SIZE);
                mImageView.constructEventMessage(bb, event, actionPidIndex);
                msgType = ProtocolConstants.MT_FIRST_DOWN;
            }
            break;
        // ...
    }
    return true;
}
```

A variable's span is
space (LOC) between successive references

Variable Liveness

Keep Variables “Live” for as Short a Time Possible

```
public boolean onTouch(View v, MotionEvent event) {
    ByteBuffer bb = null;
    int action = event.getAction();
    int actionCode = action & MotionEvent.ACTION_MASK;
    int actionPidIndex = action >> MotionEvent.ACTION_POINTER_ID_SHIFT;
    int msgType = 0;
    MultiTouchChannel h = mHandler;

    switch (actionCode) {
        case MotionEvent.ACTION_MOVE:
            if (h != null) {
                bb = ByteBuffer.allocate(event.getPointerCount() *
                                         ProtocolConstants.MT_EVENT_ENTRY_SIZE);

                bb.order(h.getEndian());
                for (int n=0; n < event.getPointerCount(); n++) {
                    mImageView.constructEventMessage(bb, event, n);
                }
                msgType = ProtocolConstants.MT_MOVE;
            }
            break;
        case MotionEvent.ACTION_DOWN:
            if (h != null) {
                bb = ByteBuffer.allocate(ProtocolConstants.MT_EVENT_ENTRY_SIZE);
                bb.order(h.getEndian());
                mImageView.constructEventMessage(bb, event,
                    actionPidIndex); msgType = ProtocolConstants.MT_FIRST_DOWN;
            }
            break;
        // ...
    }
    return true;
}
```

The diagram illustrates the liveness of two variables, `h` and `actionPidIndex`, within the `onTouch` method. A green line traces the path of `actionPidIndex` from its declaration to its last use, with a label `liveness interval=17` in a green box. An orange line traces the path of `h` from its declaration to its last use, with a label `liveness interval=14` in an orange box. The lines are curved to show the flow of execution through the `switch` statement.

A variable's liveness interval is

interval over which changing the variable's value will affect program behavior

LOC (incl.) between first and last reference

Variable Liveness

```
int method(int x, int y) {  
    int t = x*y;  
    if ((x+1) * (x+1) == y) {  
        t = 1;  
    }  
    if (x*x + 2*x + 1 != y) {  
        t = 2;  
    }  
    return t;  
}
```

$(x+1)^2 = y$

$(x+1)^2 \neq y$

Plays role in compiler optimization

e.g., register allocation semantic

liveness is hard to use

typically resort to syntactic liveness

Variable Scope

```
public boolean onTouch(View v, MotionEvent event) {  
    ByteBuffer bb = null;  
    int action = event.getAction();  
    int actionCode = action & MotionEvent.ACTION_MASK;  
    int actionPidIndex = action >> MotionEvent.  
    ACTION_POINTER_ID_SHIFT; int msgType = 0;  
    MultiTouchChannel h = mHandler;  
  
    switch (actionCode) {  
        case MotionEvent.ACTION_MOVE:  
            if (h != null) {  
                bb = ByteBuffer.allocate(event.getPointerCount() *  
                                       ProtocolConstants.MT_EVENT_ENTRY_SIZE);  
                bb.order(h.getEndian());  
                for (int n=0; n < event.getPointerCount(); n++) {  
                    mImageView.constructEventMessage(bb, event, n);  
                }  
                msgType = ProtocolConstants.MT_MOVE;  
            }  
            break;  
        case MotionEvent.ACTION_DOWN:  
            if (h != null) {  
                bb =  
                ByteBuffer.allocate(ProtocolConstants.MT_EVENT_ENTRY_SIZE);  
                bb.order(h.getEndian());  
                mImageView.constructEventMessage(bb, event,  
                actionPidIndex); msgType = ProtocolConstants.MT_FIRST_DOWN;  
            }  
            break;  
        // ...  
    }  
    return true;  
}
```

Program space where variable is
valid and can be referenced

Typical scopes

*block within {}, method, class (and possibly
derived classes), package, whole program,*

...

Variable Scope

```
public boolean onTouch(View v, MotionEvent event) {  
    ByteBuffer bb = null;  
    int action = event.getAction();  
    int actionCode = action & MotionEvent.ACTION_MASK;  
    int actionPidIndex = action >> MotionEvent.  
    ACTION_POINTER_ID_SHIFT; int msgType = 0;  
    MultiTouchChannel h = mHandler;  
  
    switch (actionCode) {  
        case MotionEvent.ACTION_MOVE:  
            if (h != null) {  
                bb = ByteBuffer.allocate(event.getPointerCount() *  
                                       ProtocolConstants.MT_EVENT_ENTRY_SIZE);  
                bb.order(h.getEndian());  
                for (int n=0; n < event.getPointerCount(); n++) {  
                    mImageView.constructEventMessage(bb, event, n);  
                }  
                msgType = ProtocolConstants.MT_MOVE;  
            }  
            break;  
        case MotionEvent.ACTION_DOWN:  
            if (h != null) {  
                bb =  
                ByteBuffer.allocate(ProtocolConstants.MT_EVENT_ENTRY_SIZE);  
                bb.order(h.getEndian());  
                mImageView.constructEventMessage(bb, event,  
                actionPidIndex); msgType = ProtocolConstants.MT_FIRST_DOWN;  
            }  
            break;  
        // ...  
    }  
    return true;  
}
```

Program space where variable is
valid and can be referenced

Typical scopes

*block within {}, method, class (and possibly
derived classes), package, whole program, ...*

Scope

Modifier	Classes or subclasses in package	Subclasses outside package	Classes outside package
public	Yes	Yes	Yes
protected	Yes	Yes	No
no modifier	Yes	No	No
private	No	No	No

Use most restricted scope possible

less info to keep in brain → fewer errors

expand scope only if necessary

Scope scale

*block → method → private → package →
protected → public*

Variable Span, Liveness, and Scope

- Variable span
 - LOC between successive references (exclusive)
- Variable liveness interval
 - LOC between first and last reference (inclusive)
- Variable scope
 - Program context where variable is valid
- In general, try to minimize these

Code Layout