Programming Style

Kurt Schmidt

Dept. of Computer Science, Drexel University

March 31, 2016

Examples are taken from Kernighan & Pike, *The Practice of Programming*, Addison-Wesley, 1999

Intro

Objective: For students to appreciate the importance of good programming style and to develop good programming style themselves

Well-written programs are better than badly-written ones – they have fewer errors and are easier to debug and to modify – so it is important to think about style from the beginning.

- Good code should read like a book
    - Straight-forward
    - Concise
    - Easy to look at
- Much easier to debug and maintain
- Don't irritate Jobu

Themes

- **Consistency!**
- Code should be clear and simple:
  - Straightforward logic
  - Natural expression
  - Conventional (idiomatic) language use
  - Meaningful names
  - Neat formatting
  - Helpful comments
  - Avoid clever tricks and unusual constructs

Oh, yeah...

Did I mention consistency?

Names

Choose Good names

```
if( country==SG || country==BN || country==PL )
{
...
}
```

So, maybe ISO country codes aren't all that clear to everybody.

```
if( country==SINGAPORE || country==BRUNEI || country==POLAND )
{
    ...
}
```

```
if( country==SINGAPORE || country==BRUNEI
    || country==POLAND || country==ITALY )
{
  /*
   * If the country is Singapore, Brunei, or Poland, the current
   * time is te answer time, rather than the off-hook time.
   * Reset answer time and set day of week.
   */
  ...
}
```

- Update comments when code gets updated
- Better still, write legible code, skip silly comments

Names

Programming
Style

Kurt Schmidt

Intro

Names

Accuracy

Expressions
and
Statements

Consistency
and Idioms

Function
Macros

Magic
Numbers

Comments

- Use descriptive name for globals, short names for locals
    - The smaller the scope, the shorter the name
- Namespaces
    - Use them to avoid clashes, and contrived-sounding names
- Follow consistent conventions
    - You'll develop your own style, over time
    - Larger projects should have their own style guides

- Use active names for functions
    - Make it clear what the function does
    - Make the meaning of the return value easy to infer
- Be accurate
- Comment units

Use Meaningful Names

Programming
Style

Kurt Schmidt

Intro

Names

Accuracy

Expressions
and
Statements

Consistency
and Idioms

Function
Macros

Magic
Numbers

Comments

```
#define ONE 1
#define TEN 10
#define EIGHT 16
```

Much more helpful:

```
#define INPUT_MODE 1
#define INPUT_BUFSIZE 10
#define WORD_BITS 16
```

Descriptive Names for Globals, Shorter for Local

```
int nPending = 0 /* current length of input queue */
```

```
for( theElementIndex = 0 ;
     theElementIndex < numberOfElements ;
     ++theElementIndex )
```

```
for( i=0; i<nelemens; ++i )
   elem[i] = i
```

Conventions

Programming
Style

Kurt Schmidt

Intro

Names

Accuracy

Expressions
and
Statements

Consistency
and Idioms

Function
Macros

Magic
Numbers

Comments

These are simply examples you *might* follow:

- Use camelcase, or underscores
    - `leastRightDesc` vs. `least_right_desc`
- Decorate pointers, globals
    - `p_head, gName`
- Initial capital letter for types, or for globals
- All caps for constants
- Be Consistent!

Use Namespaces

Don't be silly

```
class UserQueue {
  public:
    int noOfItemsInQ, frontOfTheQueue, queueCapacity ;
    int noOfUsersInQueue() {...}
}

queue.queueCapacity ;
```

```
class UserQueue {
  public:
    int nItems, front, capacity ;
    int nUsers() {...}
}
```

Use Active Names for Functions

```
now = date.getTime() ;
putchar( '\n' ) ;
```

Name should make sense of the return value:

```
if( checkoctal( c )) ...
```

Okay. Sure. Yes. I checked it.
Better:

```
if( isoctal( c )) ...
```

Accuracy

Use Active Names for Functions

Programming
Style

Kurt Schmidt

Intro

Names

Accuracy

Expressions
and
Statements

Consistency
and Idioms

Function
Macros

Magic
Numbers

Comments

This may stray outside of "programming style" a bit, but, worth mentioning

```
#define isoctal( c ) ((c) >= '0' && (#c) <= '8')
```

```
#define isoctal( c ) ((c) >= '0' && (#c) <= '7')
```

```
public boolean inTable( Object obj )
    int j = this.getIndex( obj ) ;
    return( j == nTable ) ;
}
```

```
public boolean inTable( Object obj )
    int j = this.getIndex( obj ) ;
    return( j < nTable ) ;
}
```

Expressions and Statements

Expression and Statements

- Use indentation to show structure
- Use the natural form of an expression
- Parenthesize to resolve ambiguity
- Break up complex expressions
- Mind those side effects!

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

Make it clear the body is empty:

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

Better still – idiomatic use of for loop

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

Use Natural Form for Expressions

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

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

Remember DeMorgan's Laws

```
if( !( r=='n' || r=='N' ))
```

```
if( r!='n' && r!='N' )
```

Use Parentheses to Resolve Ambiguity

Even if parentheses aren't strictly necessary.

```
if( x & ( MASK==BITS )) /* Incorrect */
if( x & MASK == BITS ) /* Correct (maybe) */
```

```
if( (x&MASK) == BITS )
```

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

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

Break up Complex Expressions

```
*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 ;
```

Be Clear

```
subkey = subkey >> (bitoff - ((bitoff >> 3) << 3)) ;
```

We can clean the logic up, make it easier to read:

```
subkey = subkey >> (bitoff & 0x7) ;
subkey >>= bitoff & 0x7 ;
```

Here are some acceptable uses of the ternary operator:

```
max = a>b ? a : b ;
```

```
printf( "The list has %d item%s\n"
        n, (n==1)?"":"s" ) ;
```

Save "clever" for your design.

Don't Abuse Coercion

Don't treat pointers as booleans

```
child=( !LC&&!RC )? 0 : (!LC?RC:LC) ;
```

Expanded out:

```
if( LC==0 && RC==0 )
   child = 0 ;
else if( LC==0 )
   child = RC ;
else
   child = LC ;
```

Better, simplify the logic:

```
if( LC==0 )
   child = RC ;
else
   child = LC ;
```

Assignment associates right-to-left; however, the order in
which the operands are evaluated is **not** defined.

```
str[i++] = str[i++] = ' ' ;
```

```
str[i++] = ' ' ;
str[i++] = ' ' ;
```

Actually, no harm in the above. Consider this one:

```
array[i++] = i ;
```

```
array[i] = i ;
i++ ;
```

Mind Evaluation

Programming
Style

Kurt Schmidt

Intro

Names

Accuracy

Expressions
and
Statements

Consistency
and Idioms

Function
Macros

Magic
Numbers

Comments

- Order in which arguments to a function are evaluated is not defined
- **All** arguments to a function are evaluated before the function is called

Here, `profit[yr]` is evaluated before `yr` is read:

```
scanf( "%d %d", &yr, &profit[yr] );
```

Must read `yr` first:

```
scanf( "%d", &yr );
scanf( "%d", &profit[yr] );
```

Consistency and Idioms

Use Consistent Indentation and Brace Style

Programming
Style

Kurt Schmidt

Intro

Names

Accuracy

Expressions
and
Statements

**Consistency
and Idioms**

Function
Macros

Magic
Numbers

Comments

```
if( month==FEB ) {
   if( isLeap( yr ))
      if( day>29 )
         legal = FALSE ;
      else
   if( day > 28 )
   {
      legal = FALSE ;
   }
}
```

- Generally, braces are recommended, even if not needed
- If omitted for small scopes, be careful

```
if( month==FEB ) {
   if( isLeap( yr )) {
      if( day>29 )
         legal = FALSE ;
      else if( day > 28 )
         legal = FALSE ;
   }
}
```

Consistent Indentation and Brace Style

Programming
Style

Kurt Schmidt

Intro

Names

Accuracy

Expressions
and
Statements

Consistency
and Idioms

Function
Macros

Magic
Numbers

Comments

Rearrange the logic the improve the legibility of the previous example:

```
if( month==FEB ) {
   int nday = 28 ;

   if( isLeap( yr ))
      nday = 29 ;
   if( day > nday )
      legal = FALSE ;
}
```

Use Idioms for Consistency

Programming
Style

Kurt Schmidt

Intro

Names

Accuracy

Expressions
and
Statements

Consistency
and Idioms

Function
Macros

Magic
Numbers

Comments

Each of these loops does the same:

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

```
for( i=n; i>=0; --i )
   array[i] = 1.0 ;
```

```
for( i=0; i<n; )
   array[i++] = 1.0 ;
```

```
for( i=0; i<n; ++i )
   array[i] = 1.0 ;
```

- A non-standard construct will catch the eye
- If the loop is doing something non-standard (going right-to-left through the array) it **should** catch the eye
- Otherwise, it shouldn't

Use Idioms for Consistency

Standard for walking a linked list:

```
for( p=list; p!=NULL; p=p->next )
    ...
```

A couple infinite loops (I prefer the latter, though industry seems to favor the former).

```
for( ;; )
    ...
```

```
while( 1 )
```

- Unless the loop *actually* is meant to run forever, this is lazy design
- It is handy to be able to look at the first line, have an idea of the loop's purpose

Use Idioms – Avoid Sprawl

- Sprawling layouts also force code onto multiple screens
- General rule of thumb: A function, loop body, etc., should fit on a screen

```
for (
    ap = arr ;
    ap < arr + 128 ;
    ++ap
    )
{
    *ap = 0 ;
}
```

Don't sacrifice legibility for compactness, either.

```
i=0;while(i<12){if(i%2==0)printf("%d\n",i*i);++i;}
```

Only use a `do-while` loop when the loop must be exectued
at leat once.

```
do {
   c = getchar() ;
   putchar( c ) ;
} while( c != EOF ) ;
```

```
while( (c=getchar()) != EOF )
   putchar( c ) ;
```

Consistent use of idioms draws attention to non-idiomatic
constructs, a frequent source of trouble.

```
iArray = (int*) malloc( nmemb * sizeof( int )) ;
for( i=0; i<=nmemb; ++i )
   iArray[i] = i ;
```

```c
if( argc==3 )
   if( (fin=fopen( argv[1], "r" )) != NULL )
      if( (fout=fopen( argv[2], "w" )) != NULL ) {
         while( (c=getc( fin )) != EOF )
            putc( c, fout ) ;
         fclose( fin ) ;
         fclose( fout ) ;
      } else
         printf( "Can't open output file %s\n", argv[2] ) ;
   else
      printf( "Can't open input file %s\n", argv[1] ) ;
else
   printf( "Usage: cp inputfile outputfile\n" ) ;
```

- Marches across the screen
- Point of the mess is buried in the middle of the mess
- The alternative is not near the consequent

Use `else-if` for Multi-Way Decisions

- Flip the tests in the antecedent
- Leave the `else-if` at the same indent

```
if( argc!=3 )
  printf( "Usage: cp inputfile outputfile\n" ) ;
else if( (fin=fopen( argv[1], "r" )) == NULL )
  printf( "Can't open input file %s\n", argv[1] ) ;
else if( (fout=fopen( argv[2], "w" )) == NULL ) {
  printf( "Can't open output file %s\n", argv[2] ) ;
  fclose( fin ) ;
} else {
  while( (c=getc( fin )) != EOF )
    putc( c, fout ) ;
  fclose( fin ) ;
  fclose( fout ) ;
}
```

Don't Be Clever With Switch Statements

- Avoid fall-throughs in switch statements
- Comment, if you must

```
switch( c ) {
   case '-':  sign = -1 ;
   case '+':  c = getchar() ;
   case '.':  break ;
   default:
      if( !isdigit( c ))
         return 0 ;
} /* switch c */
```

```
switch( c ) {
   case '-':
      sign = -1 ;
      /* fall through */
   case '+':
      c = getchar() ;
      break ;
   case '.':
      break ;
   default:
      if( !isdigit( c ))
         return 0 ;
} /* switch c */
```

- Saves duplicating one line of code

- Longer, but much clearer

Switch Statements

Programming
Style

Kurt Schmidt

Intro

Names

Accuracy

Expressions
and
Statements

Consistency
and Idioms

Function
Macros

Magic
Numbers

Comments

- Might be better to express using `else-if`

```
if( c == '-' ) {
    sign = -1 ;
    c = getchar() ;
} else if( c == '+' ) {
    c = getchar() ;
} else if( c != '.' && !isdigit(c))
    return 0 ;
```

- Example of acceptable fall-throughs
- No comment needed

```
switch( c ) {
    case 'h':
    case 'H':
    case '?':
        usage() ;
        break ;
    ...
}
```

Function Macros

Avoid Function Macros

- Macros have been used to avoid the overhead of function calls
    - No longer necessary
    - In C99, C++, we have inline functions
- Note, actual arguments might be evaluated more than once
    - Again, side effects become a problem

Read 2 characters:

```
#define isUpper(c) ( 'A'<=(c) && (c)<='Z' )
...
while( isUpper( c=getchar()))
   ...
```

Correct, but, inefficient:

```
#define round_to_int(x) ( (int)( (x)+( (x)>0 ? 0.5 : -0.5 )))
size = rounded_to_int( sqrt( dx*dx + dy*dy ))
```

Each occurrence of a macro argument should be put in parentheses:

```
1  #define square(x) x * x
2  i=3 ; j=4 ;
3  k = square( i+j ) ;
```

- Line 3 expands to 3+4*3+4
- k has value 19, rather than 49

Entire macro definition should be enclosed in parentheses:

```
1  #define square(x) x * x
2  f = 3 ;
3  g = 1.0/square(i)
```

- Line 3 expands to 1.0/3*3
- k has value 1.0, rather than 0.111

Magic Numbers

Avoid *Magic Numbers*

- Unnamed, meaningful, numerical constant
- Obscures developer's intent in choosing that number
- Increases opportunities for subtle errors
    - Is 3.14159265358979 correct?
    - Is it equal to 3.14159265359?
- Easier to alter the number's value

```
x = 12 * d ;
  /* mo/yr? eggs/dozen? */
f = 6.672e-11 * 5 * 8 / (7*7)
  /* force due to gravity? G might change */
```

Define Numbers as Constants, not Macros

Programming
Style

Kurt Schmidt

Intro

Names

Accuracy

Expressions
and
Statements

Consistency
and Idioms

Function
Macros

Magic
Numbers

Comments

- C preprocessor changes the lexical structure of the program
  - We lose type info
  - Symbols don't appear in debugger
- Use the C `enum` for integer constants

```
enum { MAXROW=24, MAXCOL=80 } ;
```

- C++ provides the `const` keyword

```
const int MAXROW=24, MAXCOL=80 ;
```

- Java has `final`

```
static final int MAXROW=24, MAXCOL=80 ;
```

Use Character Constants, not Ordinals

```
if( 65<=c && c<=90 )
   ...
```

This is more legible:

```
if( 'A'<=c && c<='Z' )
   ...
```

But, still dependent upon a representation.
These always work:

```
if( isupper( c ))
   ...
```

```
if( Character.isUpperCase( c ))
   ...
```

Use the Language to Calculate Size of an Object

- Use `sizeof` operator in C/C++:

```
char buf[1024] ;
fgets( buf, sizeof(buf), stdin ) ;
```

- Java arrays have a `length` attribute:

```
char [] buf = new char[1024];
for( int i=0; i< buf.length; ++i )
    ...
```

- Idiom for finding length of array in C/C++ (in scope):

```
#define NELEMS(array) ( sizeof(array) / sizeof(array[0]) )
double dbuf[100] ;
for( i=0; i<NELEMS(dbuf); ++i )
    ...
```

Comments

Comments

- Don't belabor the obvious
- Comment functions and global data
- Don't comment bad code – rewrite it
- Don't contradict the code
- Clarify, don't confuse

Don't Belabor the Obvious

```
/*
 * default
 */
default :
    break ;
```

```
/* return SUCCESS */
return SUCCESS ;
```

```
zerocount++ ; /* Increment zero entry counter */
```

```
// Inialise totoal to number_received
node->total = node->number_received ;
```

Minimally, comments should include

- Filename
- Purpose
- Your name
- Date
- Platform information
- Usage notes (if it's a client-facing file)
- Change log

Page Header Comments (*cont*)

Programming
Style

Kurt Schmidt

Intro

Names

Accuracy

Expressions
and
Statements

Consistency
and Idioms

Function
Macros

Magic
Numbers

Comments

```
/**-*-C-*-*****
 * myHeader.h -- example interface file
 *
 * Kurt Schmidt
 * MAR 2016
 *
 * gcc (Ubuntu 4.8.4-2ubuntu1~14.04.1) 4.8.4 on
 * Linux 3.16.0-67-generic
 *
 * EDITOR: tabstop=3, cols=80
 *
 * NOTES:
 *  - Have fun
 *  - Watch that sine function
 */
```

Comment Global Data

Programming
Style

Kurt Schmidt

Intro

Names

Accuracy

Expressions
and
Statements

Consistency
and Idioms

Function
Macros

Magic
Numbers

Comments

```
struct State {          /* prefix & suffix list */
   char *pref[NPREF] ; /* prefix words */
   Suffix *suf ;        /* list of suffices */
   State *next ;        /* next State in list */
} ;
```

Supply units, where appropriate!

```
double weight ; /* Pounds? Newtons? */
double radius ; /* Inches? Furlongs? Light years? */
```

- These should serve as a user guide.
- Describe *inputs*, *outputs*, and *side-effects*
  - Alternatively, *preconditions* and *postconditions*
- Warn client of side-effects
- Units!

```c
/* mySine - computes sine of an angle
 * Requires: global PI, x in radians
 * Ensures: sine(x) returned; all your chocolate is gone
 */
double mySine( x ) {
   rv = magic( x, PI ) ;
   stealChocolate() ;
   return( rv ) ;
}
```

Don't Comment Bad Code...

```
/* If 'result' is 0 a match was found so return
true (non-zero). Otherwise, 'result' is non-zero
so return false (zero). */
#ifdef DEBUG
printf( "*** isword returns !result=%d\n", !result ) ;
fflush( stdout ) ;
#endif
return( !result ) ;
```

. . . rewrite it

```
#ifdef DEBUG
printf( "*** isword returns matchFound=%d\n", matchFound ) ;
fflush( stdout ) ;
#endif
return( matchFound ) ;
```

```c
int strcmp( char *s1, char *s2 )
 /* string comparison routine returns -1 if s1 is above s2 */
 /* in ascending order list, 0 if equal, 1 if s1 below s2 */
{
   while( *s1==*s2 ) {
      if( *s1=='\0' )
         return( 0 ) ;
      ++s1 ; ++s2 ;
   if( *s2 > *s1 ) return( 1 ) ;
   return( -1 ) ;
}
```

```c
/* strcmp: return <0 if s1<s2, >0 if s1>s2, 0 if equal */
/*   ANSI C, section 411.4.2 */
```

- Your code should be legible
- "Good style should be a matter of habit."