Programming style

programming style

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.*

Motivation

- Good code should read like a book
  - straight-forward
  - concise
  - easy to look at
- Much easier to debug and maintain

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

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 ) {
   ...
}
```

Keep comments in synch


```
if( country==SINGAPORE || country==BRUNEI ||
    country==POLAND || country==ITALY ) {
   /*
    * If the country is Singapore, Brunei, or Poland, the current
    * time is the 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 (self-documenting) code, skip silly comments

Names

- Use descriptive names for globals, short names for locals
- The smaller the scope, the shorter the name
- 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


```
#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

These are simply examples you *might* follow:

- Use camelcase, or underscores
  - `leastRightDesc` vs `least_right_desc`
- 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 )) ...
```

Better:

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

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 side effects and short-circuit evaluation

Indent to show structure


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

Make it clear the body is empty:

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

Better still – idiomatic use of for loop

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

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 )
```

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" ) ;
```

Mind the side effects

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++ ;
```

consistency and idioms

Use consistent indentation and brace style


```
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

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

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 of ways to specify infinite loops:

```
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;}
```

Use `do-while` loops sparingly

Only use a `do-while` loop when the loop must be executed at least once.

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

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

Use `else-if` for multi-way decisions


```
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 */
```

- Saves duplicating one line of code

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

- Longer, but much clearer

Switch statements

- 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 ;
   ...
}
```

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

- 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 } ;
```

  - Since C99 (at least), C has `const`
- 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 */
```

```
   // Initialize total to number_received
node->total = node->number_received ;
```

Page header comments

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'd*)


```
/**-*-C-*-*****
 * myHeader.h -- example interface file
 *
 * Spiros Mancoridis
 * 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:
 *  - Watch that sine function
 */
```

Comment global data


```
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? */
```

Comment function header

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

```
  /* 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 ) ;
}
```

Clarify, don't confuse


```
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 ) ;
}
```

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

Summary

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

credits

Thanks

The contents of these slides were created by Kurt Schmidt and modified by other faculty of the Drexel University CS Department including Geoffrey Mainland, Vera Zaychick, Jeremy Johnson, Spiros Mancoridis, and others. Examples are taken from Kernighan & Pike, *The Practice of Programming*, Addison-Wesley, 1999