**Style Guide**

Above all, remember: clever code is unreadable code. Save your cleverness for algorithms and document them well.

Key:  
- **YOU MUST** follow this rule
- We recommend you follow this rule

### Syntax Specific Guidelines

**IF**

- IFs with ELSE IF clauses should always include an ELSE block even if it's only an error message.

<table>
<thead>
<tr>
<th>INCORRECT</th>
<th>CORRECT</th>
</tr>
</thead>
</table>
| if (a>5) {  
    printf("a is large\n");  
} /* end if large case */  
else if ( (a<=5) && (a>3) ) {  
    printf("a is small\n");  
} /* end else if small case */  
else {  
    printf("error!\n");  
} /* end else error case */ | if (a>5) {  
    printf("a is large\n");  
} /* end if large case */  
else if ( (a<=5) && (a>3) ) {  
    printf("a is small\n");  
} /* end else if small case */  
else {  
    printf("error!\n");  
} /* end else error case */ |

- IFs with AND conditions should only use fully elaborated Boolean expressions, and not rely on ordering for implicit conditions.

<table>
<thead>
<tr>
<th>INCORRECT</th>
<th>CORRECT</th>
</tr>
</thead>
</table>
| if (x>10)  
    if (y>6)  
        z = a + b; | if ((x>10) && (y>6)) {  
        z = a + b;  
} /* end if to check x and y */ |

- IFs with ELSE IF clauses should form an exhaustive set of alternatives.

<table>
<thead>
<tr>
<th>INCORRECT</th>
<th>CORRECT</th>
</tr>
</thead>
</table>
| if (a>5) {  
    printf("PosNum is large\n");  
} /* end if large case */  
else if ( (a<=5) && (a>3) ) {  
    printf("a is small\n");  
} /* end else if small case */  
else {  
    printf("error!\n");  
} /* end else error case */ | if (a>5) {  
    printf("a is large\n");  
} /* end if large case */  
else if ( (a<=5) && (a>3) ) {  
    printf("a is medium\n");  
} /* end else if small case */  
else if (a<=3) {  
    printf("a is small\n");  
} /* end else if small case */  
else {  
    printf("error!\n");  
} /* end else error case */ |
• IFs that serve only to guard a block of statements (i.e. test only one condition) and have no alternatives (ELSE IF blocks) do not necessarily require an ELSE block

Unnecessary

```c
if (a==0) {
    printf("invalid entry for x
\n");
} /* end x is 0 */
else {
    printf("x is ok\n");
} /* end else x is not 0*/
```

CORRECT

```c
if (a==0) {
    printf("invalid entry for x
\n");
} /* end x is 0 */
else if (a>5 && (b>5) ) {
    printf("ok
");
} else if (a>5 && (b<=5) ) {
    printf("not ok
");
} else if (a<=5) {
    /* do nothing */
} else {
    printf("error
");
} /* end else error case */
```

• Nested IFs contain implicit AND conditions, making them hard to maintain. Be very careful when you use them.

INCORRECT

```c
if (a>5) {
    if (b>5) {
        printf("ok\n");
    } /* end if b>5 */
    else {
        printf("not ok\n");
    } /* end else not ok */
} /* end if a>5 */
```

CORRECT

```c
if (a>5) {
    if ( (a>5) && (b>5) ) {
        printf("ok\n");
    } /* end if x is ok */
    else if ( (a>5) && (b<=5) ) {
        printf("not ok\n");
    } /* end else if x is not ok */
    else if (a<=5) {
        /* do nothing */
    } /* end if a<=5 */
    else {
        printf("error\n");
    } /* end else error case */
```

• ELSE statements should be followed by a comment that explains the conditional to which it is attached.

INCORRECT

```c
if (x==0) {
    printf("invalid x value\n");
} else {
    z = q/x;
}
```

CORRECT

```c
if (x==0) {
    printf("invalid x value\n");
} /* x is not valid */
else {
    z = q/x;
} /* x was not 0 */
```

• Combine multiple IF/ELSE IF alternatives that execute the same code. Duplicated code requires more effort to maintain.

INCORRECT

```c
if (x>10)
    z = a + b;
else if (y>6)
    z = a + b;
```

CORRECT

```c
if ( (x>10) || (y>6) ) {
    z = a + b;
} /* end if x and y are in range */
```

• It is preferred that all if statements, even those with only one line, use braces.

INCORRECT

```c
if (x>10)
    z = a + b;
```

CORRECT

```c
if ( (x>10) || (y>6) ) {
    z = a + b;
} /* end if x and y are in range */
```
**BOOLEAN EXPRESSIONS**

- Short-circuiting should not be used, because the order in which multiple expressions in a single if are evaluated is not always defined. Use nested Ifs to avoid runtime errors. Comment these cases to warn other programmers that a potential problem exists.

```
INCORRECT
x = 0;
y = 3;
if ((x != 0) && (y/x > 2))
z = x + y;
/* if (y/x > 2) is evaluated * before the && a runtime * divide by zero error will * occur since x is zero */

CORRECT
x = 0;
y = 3;
if (x != 0) {
  if (y/x > 2) {
    z = x + y;
  } /* end if yx ratio is valid */
} /* end if x is not 0 */
/* Nesting protects y/x from evaluation in case x is zero */
```

- Conditional expressions should not have side effects.

```
INCORRECT
if (++j > 6) {
  printf("j > 6\n");
} /* end if */

CORRECT
j++;
if (j > 6) {
  printf("j > 6\n");
} /* end if */
```

- Use parenthesis wherever possible. Avoid relying on evaluation order.

```
INCORRECT
(x>10 && y>6 && q<4)

CORRECT
((x>10) && (y>6)) && (q<4)
```

**ARRAYS**

- Use constants to define your array bounds. It makes writing loop code more scalable.

```
INCORRECT
int Array[5];

CORRECT
#define SIZE 5
int Array[SIZE];
```

**FOR**

- When traversing arrays, use attributes of the array definition to define the bounds of iteration.

```
INCORRECT
#define SIZE 5
int Array[SIZE];
for (x=0; x<5; x++) {
  Array[x] = 0;
} /* end for to loop */

CORRECT
#define SIZE 5
int Array[SIZE];
for (x=0; x<SIZE; x++) {
  Array[x] = 0;
} /* end for to loop */
```

- Don't use for loops for tasks other than iterating. For loops are designed for iteration and imply the presence of iteration. If not iterating, use a while loop.

```
INCORRECT
/* this is a function that never ends */
int func(void) {
  const int ever = 1;

  for (ever;;) {
    printf("looping\n");
  } /* end infinite for */

  return 0;
} /* end func() */

CORRECT
/* this is a function that never ends */
int func(void) {
  const int ALWAYS = 1;

  while (ALWAYS){
    printf("looping\n");
  } /* end infinite while */

  return 0;
} /* end func() */
```
WHILE

- Use flags and/or sentinels to exit loops when necessary. Do not use multiple returns. A return should come at the end of a function. Do not use `break` or `continue`.

INCORRECT
/* function exits when x==5 */
int func(void) {
    int x = 0;
    while (1){
        x++;
        if (x==5) {
            return x;
        } /* end if x==5 */
    } /* end while to iterate x */
} /* end func() */

CORRECT
/* function exits when x==5 */
int func(void) {
    int x = 0;
    int Proceed = 1;
    while (Proceed){
        x++;
        if (x==5) {
            Proceed = 0;
        } /* end if x==5 */
    } /* end while to iterate x */
    return x;
} /* end func() */

- Try to express the conditional as a positive rather than simply using a logical NOT

INCORRECT
/* this is a while that exits when x is 5 */
x=0;
Stop=0;
while (!Stop){
    x++;
    if (x==5) {
        Stop = 1;
    } /* end if x==5 */
} /* end while to iterate x */

CORRECT
/* this is a while that exits when x is 5 */
x=0;
Continue=1;
while (Continue){
    x++;
    if (x==5) {
        Continue = 0;
    } /* end if x==5 */
} /* end while to iterate x */

SWITCH / CASE
- Include a default condition even if you think it will never be reached during normal operation.

INCORRECT
switch (a) {
    case 1: /* first case */
        printf("first case\n");
        break;
    case 2: /* second case */
        printf("second case\n");
        break;
} /* end switch to test a */

CORRECT
switch (a) {
    case 1: /* first case */
        printf("first case\n");
        break;
    case 2: /* second case */
        printf("second case\n");
        break;
    default: /* error case */
        printf("error case\n");
        break;
} /* end switch to test a */
General Guidelines

NAMING

- Try to avoid generic names like X, Y, I, etc. Exceptions to this rule include FOR loop variables, which often use I, J, K, etc. to clearly identify nesting.

  INCORRECT    CORRECT
  int x = 0;  int PositionIndex = 0;

- Name constants using all capitals and variables using mostly lowercase.

  INCORRECT    CORRECT
  const int upper_bound = 5;  const int UPPER_BOUND = 5;
  int counter = 0;  int Counter = 0;

- Global variables should be named in some way to indicate their special scope, e.g. having a 'g' as the first character of the variable’s name.

  INCORRECT    CORRECT
  int TotalCount;
  int gTotalCount;

  int main(void) {
    ...
  } /* end main */

  int main(void) {
    ...
  } /* end main */

- Pointer variable names should begin with `p` or with `p_`.

  INCORRECT    CORRECT
  int * QueueHead;
  int * p_QueueHead;

- Try to avoid uncommon abbreviations. Elaborate abbreviations fully in comments at declaration.

  INCORRECT    CORRECT
  int NmCorAns = 0;
  int NumCorrectAns = 0;
  /* variable contains the number of correct answers */

- Descriptive but concise variable names help more to make code readable than any other facet of the programming, including comments.

  INCORRECT    CORRECT
  int x = 0; /* number of boats */
  int y = 0; /* number of cars */
  int z = 0; /* number of vehicles */
  int NumBoats = 0;
  int NumCars = 0;
  int NumVehicles = 0;

  NumVehicles = NumBoats + NumCars;

- Function names should be as descriptive of their purpose as possible.

  INCORRECT    CORRECT
  int convert(double E);
  /* function converts feet to angstroms */
  int convert_feet_to_angstroms(double E);
FUNCTION DECLARATIONS

- Banner comments should include:
  o The function name and a description of its function. Be sure to include any caveats or limitations.
  o A list of its dependencies
  o All inputs and outputs
  o very long argument list should be split over several lines
  o use ANSI style argument declarations, not K & R

<table>
<thead>
<tr>
<th>INCORRECT</th>
<th>CORRECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>int func(a,b,c,d)</td>
<td>/* func returns the products of its arguments. Note: Implicit conversion to int for b and d */</td>
</tr>
<tr>
<td>int a; float b; char c; double d; return a<em>b</em>c*d;</td>
<td>int func(int a, float b, char c, double d) {</td>
</tr>
<tr>
<td>} /* end func() */</td>
<td>return (a<em>b</em>c*d);</td>
</tr>
</tbody>
</table>

- Clearly demarcate where each function begins and ends. A short comment after the ending '}' is useful.

<table>
<thead>
<tr>
<th>INCORRECT</th>
<th>CORRECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>int AddNums(int Num1, int Num2) { ... }</td>
<td>int AddNums(int Num1, int Num2) { ... } /* end AddNumbers */</td>
</tr>
<tr>
<td>int MultNums(int Num1, int Num2) { ... }</td>
<td>/* ------------------------------ <em>/ int MultNums(int Num1, int Num2) { ... } /</em> end SubtractNumbers */</td>
</tr>
<tr>
<td>} /* end AddNumbers */</td>
<td>/* ------------------------------ */</td>
</tr>
</tbody>
</table>

WHITESPACE
- Make opening braces the last character on their line or else on a line alone.

<table>
<thead>
<tr>
<th>INCORRECT</th>
<th>CORRECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (a&gt;b) { b++; } /* end if a &gt; b */</td>
<td>if (a&gt;b) { b++; } /* end if a &gt; b */ OR</td>
</tr>
<tr>
<td>} /* end if a &gt; b */</td>
<td>if (a&gt;b) { b++; } /* end if a &gt; b */</td>
</tr>
</tbody>
</table>
Put the closing braces of functions on their own lines.

**INCORRECT**
```c
if (a>b) {
    b++;  }  /* end if a > b */
```

**CORRECT**
```c
if (a>b) {
    b++;  }  /* end if a > b */
```

- Place a space between comma separated elements.

**INCORRECT**
```c
int AddNums(int Num1, int Num2) {
...
}
```

**CORRECT**
```c
int AddNums(int Num1, int Num2) {
...
} /* end AddNumbers */
```

- Insert a blank line between functionally different blocks of code, and before any control flow construct.

**INCORRECT**
```c
int count = 0;
if (x>count) {
    printf("count is too small\n");
} /* end if for count warning */
```

**CORRECT**
```c
int count = 0;
if (x>count) {
    printf("count is too small\n");
} /* end if for count warning */
```

- Don’t separate comments that describe a piece of code from the code by a blank line.

**INCORRECT**
```c
/* this block prevents divide by zero errors */
if (a==0) {
    printf("a is a bad divisor\n");
} /* end if a is 0 */
```

**CORRECT**
```c
/* this block prevents divide by zero errors */
if (a==0) {
    printf("a is a bad divisor\n");
} /* end if a is 0 */
```

- Very long lines of code should be broken over several lines

**INCORRECT**
```c
if (((x>6)&&(y>3))||(flag1&&flag2)) {
    z = x+y;
} /* end if conditions right */
```

**CORRECT**
```c
if ( ( (x>6) && (y>3) ) ||
    (flag1 && flag2) ) {
    z = x+y;
} /* end if conditions right */
```

**COMMENTS**

- When programming in C (as you will be in 16.070), **always** use c-style comments.

**INCORRECT**
```c
# not shell script
// not C++
! not html
```

**CORRECT**
```c
/* Ansi C */
```
Banner comments at the top of the file are important. They should include:
- The author's name and contact information (i.e. e-mail address)
- History of recent modification with dates
- A list of any other modules and/or external data that the file depends upon
- (Not required but often helpful) A list of any modules that depend on the file
- A description of the file's functionality.

INCORRECT

/* Program by J.B. */

int main(void) {
    ...
} /* end main */

CORRECT

/* Author: Joe Be (joe@mit.edu) 
Date Begun: 10/31/01 
Date Fnl added: 11/02/01 
Dependencies: stdlib.h 
Function: This file contains a program that adds numbers. */

int main(void) {
    ...
} /* end main */

Comment closing braces with a reference to their opening braces when they are not within a few lines of each other.

INCORRECT

if (a>b) {
    ...
    ...
} /* end if a is greater than b */

CORRECT

/* this loop will continue adding numbers until the user specifies if should stop, at which point, continue will be set to 0 */

int Continue = 1;
while (Continue) {
    ...
} /* end while to add numbers */

Comment control flow structures. Especially explain exit condition for sentinel loops.
• More comments are better. Your comments should explain the general flow of your code, however you need not explain every line, especially when the function of the line is obvious.

**INCORRECT**
```c
/********************
* add one to j *
* *****************/
j++;
```

**CORRECT**
```c
/* Repeat until valid input */
printf("A number: ");
success = scanf("%d", &num);
while (success==0) {
    fflush(stdin);
    printf(" *** invalid input 
");
    printf("A number: ");
success = scanf("%d", &num);
} /* end while */
```

• When possible, comment expected ranges/states next to variable declarations.

**INCORRECT**
```c
int NumBoats=0; /*number of boats*/
int Continue=1; /*loop sentinel*/
```

**CORRECT**
```c
int NumBoats = 0; /*number of boats*/
/* expected to be between 0 and 100 
it should NEVER be negative */
int Continue = 1; /*loop sentinel*/
/* expected to always be either 
0 (false) or 1 (true) */
```

**INDENTATION**

➢ Always indent at the beginning of each new block or control structure. This applies to one-line blocks as well.

**INCORRECT**
```c
if (a>b) {
    printf("a is bigger
");
} /* end if a is greater than b */
```

**CORRECT**
```c
if (a>b) {
    printf("a is bigger 
");
} /* end if a is greater than b */
```

➢ Ensure that your indentations always line up.

**INCORRECT**
```c
if (a>b) {
    if (b>c) {
        while (q<p) {
            q++;
            printf("all set
");
        } /* end while */
    } /* end if b>c */
} /* end if a is greater than b */
```

**CORRECT**
```c
if (a>b) {
    if (b>c) {
        while (q<p) {
            q++;
            printf("all set");
        } /* end while */
    } /* end if b>c */
} /* end if a is greater than b */
```

➢ A minimum of two spaces and maximum of five should be used for your indent.

**INCORRECT**
```c
int main(void){
    int a,b;
    printf("a + b = %d",a+b);
    return 0;
} /* end main */
```

**CORRECT**
```c
int main(void) {
    int a,b;
    printf("a + b = %d",a+b);
    return 0;
} /* end main */
```
GENERAL DON'TS

- Don't use goto.

**INCORRECT**

```c
    goto anywhere;
```

**CORRECT**

```c
    /* implement same situation with a series of functions */
```

- Don't use ++ or -- unless it's on a line by itself or in a for statement.

**INCORRECT**

```c
    x = x-- + ++x;
```

**CORRECT**

```c
    x++;
    x = x + x;
    x--;  
```

- Don't use the comma operator (if you know don't know what that is the don't worry)

**INCORRECT**

```c
    for (x=0, y=0; x<5; x++) {
        printf("unnecessary operator\n");
    } /* end for to make point */
```

**CORRECT**

```c
    y=0;
    for (x=0; x<5; x++) {
        printf("correct usage\n");
    } /* end for to make point */
```

- Don't use the ternary conditional operator (<boolexp>?<exp>:<exp>)

**INCORRECT**

```c
    (x>y)?x++:y++;
```

**CORRECT**

```c
    if (x>y) {
        x++;
    } /* end if x>y */
    else {
        y++;
    } /* end else x is not greater than y */
```

- Use pointers only when necessary.

**INCORRECT**

```c
    int * p_Counter;
    counter=(int *) malloc(sizeof(int));
    for (*p_Counter=0; (*p_Counter)<5;
         (*p_Counter++)
         printf("Count: %d", *p_Counter);
    } /* end for to count */
```

**CORRECT**

```c
    int Counter = 0;
    for (Counter=0; (Counter)<5;
         Counter++) {
        printf("Count: %d", Counter);
    } /* end for to count */
```

ADDITIONAL STYLE GUIDES AND RESOURCES

- The Indian Hill Style Guide (a quasi-standard):

- The Ten Commandments for C Programmers

- C. M. Lott's index of C and C++ Style Guides

- Your problem set solutions provide many examples of well written code.