Coding with Style

This guide describes basic style expectations for code, and why we ask you to meet them.

1 Making your Code Readable

We write programs so that computers can execute them, right?

Here's a program that the C0 compiler will happily compile to an executable that your computer can execute:

```
int g(int x,int y){if(y==0)return 1;return g(x,y-1)*x;}int f(int x,int y){int z=x;int w=y;in
```

While the computer can make sense of this code, it is unreadable for us.

The fact is that **we write programs for people**, not just for computers. These people include:

- *you!* As you develop and debug your program, you will be reading parts over and over. You want to write your code so that it is as easy as possible to understand it.
- *the course staff.* When you go to office hours to get help debugging your code, your TAs will often want to read what you did. If it is written in a way that is hard to make sense of, they will not be able to give you the best advice. Well-written code is a lot easier to debug!
- *your future self.* You may want to extend or fix bugs in apps you developed yourself months or years after you first wrote them. By then, you will have forgotten the details and will need to familiarize yourself with your old code anew by reading it.
- *your team.* Chances are that you will be working on software that is a lot larger than what we see in 15-122, software built by a team. You will need to write your parts of the code in a way that makes them as easy as possible for your teammates to work with.

> *"Programs must be written for people to read, and only incidentally for machines to execute"*
>
> — *from H. Abelson and G. Sussman, The Structure and Interpretation of Computer Programs*

So, how do we write readable code?

1.1 Include Line Breaks and White Spaces!

Let's start with asking what is wrong with the above snippet. First, there are no line breaks and very few spaces. Let's add some.

```
int g(int x, int y) {
if (y == 0) return 1;
return g(x, y-1) * x;
}

int f(int x, int y) {
int z = x;
int w = y;
```

```
int v = 1;
while (w > 0) {
if (w % 2 == 1) {
v = z * v;
}
z = z * z;
w = w / 2;
}
return v;
}

int main() {
int x = f(3, 2);
assert(x == g(3, 2));
return x;
}
```

In general, you want have each statement (e.g., an assignment) on a separate line, blank lines between functions and other significant chunks of code, and spaces between operators.

1.2 Indent your Code!

Much better, but not that great. C0, like the vast majority of high-level programming languages, organizes code into *blocks* — for example everything between `{` and the matching `}` belongs to the same block. **Indentation** is a good way to visualize the block structure of code.

Let's properly indent this example:

```
int g(int x, int y) {
  if (y == 0) return 1;
  return g(x, y-1) * x;
}

int f(int x, int y) {
  int z = x;
  int w = y;
  int v = 1;
  while (w > 0) {
    if (w % 2 == 1) {
      v = z * v;
    }
    z = z * z;
    w = w / 2;
  }
  return v;
}

int main() {
  int x = f(3, 2);
  assert(x == g(3, 2));
```

```
  return x;
}
```

By how much you indent the statements in each block doesn't matter as long as you are consistent — it's hard to read inconsistently indented code.

1.3 Don't use Tabs!

Modern editors automatically indent code in a sensible way (most of the time). Beware of tabulations though! Your editor may use tabulations its own way to indent your code. As a result, code that looks beautifully indented to you may look horrible in your friends' editors (which interprets tabulations differently from yours!) or on Autolab. Here's what they may see for the above code:

```
int g(int x, int y) {
  if (y == 0) return 1;
    return g(x, y-1) * x;
}

int f(int x, int y) {
  int z = x;
    int w = y;
  int v = 1;
  while (w > 0) {
    if (w % 2 == 1) {
    v = z * v;
    }
        z = z * z;
     w = w / 2;
      }
  return v;
}

int main() {
    int x = f(3, 2);
  assert(x == g(3, 2));
  return x;
}
```

Yuk! You can configure all modern editors so that they use only spaces for indentation, not tabs.

We are making progress! Now we have code that looks good from a distance — we can easily understand its structure. But what does it do? This code does not make it easy to understand its meaning.

1.4 Write Comments!

There are multiple ways to make the meaning of code understandable, and we should use all of them when writing programs.

The first is to simply include **comments** that say what important parts do. Here's what the developer of our example had in mind:

```
// Computes x^y (in y steps)
int g(int x, int y) {
  if (y == 0) return 1;   // if y == 0, then x^y == 1
  return g(x, y-1) * x;   // otherwise, x^y == x^(y-1) * x
}

// Computes x^y in log y steps
int f(int x, int y) {
  int z = x;
  int w = y;
  int v = 1;
  while (w > 0) {
    if (w % 2 == 1) {     // if w is odd
      v = z * v;
    }
    z = z * z;
    w = w / 2;
  }
  return v;
}

int main() {
  int x = f(3, 2);        // f should compute 3^2 == 9
  assert(x == g(3, 2));   // checking that it does
  return x;
}
```

Ah! These computed the `x` to the `y`. Who would have guessed?!

1.5 Pick Good Names!

So, both `f` and `g` implement the power function, but in ways that take different time. This was not obvious, and one reason for this is that `f` and `g` are completely generic function names. For readability, we want instead to choose names that make it clear what these functions are meant to do. In this code, it's the same thing with the variables: they are all called `x`, `y`, ...

Here's a version of this same code that uses **meaningful and memorable names**:

```
// Computes base^exp (in exp steps)
int pow(int base, int exp) {
  if (exp == 0) return 1;          // if exp == 0, then base^exp == 1
  return pow(base, exp-1) * base;  // otherwise, base^exp == base^(exp-1) * base
}

// Computes x^y in log y steps
int fast_pow(int x, int y) {
  int base = x;
  int exp = y;
  int res = 1;
  while (exp > 0) {
```

```
    if (exp % 2 == 1) {     // if exp is odd
      res = base * res;
    }
    base = base * base;
    exp = exp / 2;
  }
  return res;
}

int main() {
  int r = fast_pow(3, 2);   // fast_pow should compute 3^2 == 9
  assert(r == pow(3, 2));   // checking that it does
  return r;
}
```

Besides renaming `f` and `g` to `fast_pow` and `pow` (which are much more meaningful names), we also gave variables names that match their role, like `base` and `exp` for the base and the exponent of the power functions. Note that `x` and `y` are meaningful names in some contexts (e.g., when implementing mathematical functions like here). What matters is that their purpose is easy to figure out.

1.6 Write Contracts!

Now the way the code is written makes the meaning of the variables and functions apparent. How the code works and how to use it not as obvious however. We could express that with comments, but — this being C0 — we can use **contracts** instead:

```
// Computes base^exp (in exp steps) -- SPECIFICATION FUNCTION
int pow(int base, int exp)
//@requires exp >= 0;
{
  if (exp == 0) return 1;          // if exp == 0, then base^exp == 1
  return pow(base, exp-1) * base;  // otherwise, base^exp == base^(exp-1) * base
}

// Computes x^y in log y steps
int fast_pow(int x, int y)
//@requires y >= 0;
//@ensures pow(x, y) == \result;
{
  int base = x;
  int exp = y;
  int res = 1;
  while (exp > 0)
  //@loop_invariant exp >= 0;
  //@loop_invariant pow(base, exp) * res == pow(x, y);
  {
    if (exp % 2 == 1) {     // if exp is odd
      res = base * res;
    }
```

```
    base = base * base;
    exp = exp / 2;
  }
  //@assert exp == 0;
  return res;
}

int main() {
  int r = fast_pow(3, 2);   // fast_pow should compute 3^2 == 9
  assert(r == pow(3, 2));   // checking that it does
  return r;
}
```

We are also noting in a comment that `pow` is a specification function.

You should write your contracts before you write your code — not like we just did. Doing so will significantly cut down on the time it takes to obtain correct — bug free — code.

Now, picture yourself in 6 months. You have not looked back at this code since but for some reason you need to use it. Which version would you like you had written?

2 Using Abstraction

Next, let's write a function that computes (the square of) the distance between two points in 3-dimensional space. Each point is represented as a 3-element array. Here it is:

```
// returns the square of the distance between 3D points P and Q
int distance(int[] P, int[] Q)
//@requires \length(P) == 3;
//@requires \length(Q) == 3;
{
  return (P[0] - Q[0]) * (P[0] - Q[0]) + (P[1] - Q[1]) * (P[1] - Q[1]) + (P[2] - Q[2]) * (P[
}
```

Note that we did everything right based on the previous section: this code is properly indented, uses meaningful names, contains comments, and has contracts. Yet it is not all that readable.

2.1 Break Long Lines!

One issue is the very long return statement. It turns out that we, people, have a hard time reading very long lines (whether code or normal text): we get tripped up as to which one is the next line. We avoid long lines by putting line breaks in places we find pleasing. Here's one possible way to do so:

```
// returns the square of the distance between 3D points P and Q
int distance(int[] P, int[] Q)
//@requires \length(P) == 3;
//@requires \length(Q) == 3;
{
  return (P[0] - Q[0]) * (P[0] - Q[0])
       + (P[1] - Q[1]) * (P[1] - Q[1])
       + (P[2] - Q[2]) * (P[2] - Q[2]);
```

```
}
```

The lines are short, and similar expressions are aligned.

It is nowadays customary to keep lines under 80 characters long.

2.2 Factor out Common Expressions into Variables!

Notice that this code has many parts that are repeated. We can gain in readability by factoring them out. Let's start by storing the subexpressions `P[0] - Q[0]` and similar into variables:

```
// returns the square of the distance between 3D points P and Q
int distance(int[] P, int[] Q)
//@requires \length(P) == 3;
//@requires \length(Q) == 3;
{
  int dx = P[0] - Q[0];
  int dy = P[1] - Q[1];
  int dz = P[2] - Q[2];

  return dx * dx + dy * dy + dz * dz;
}
```

Factoring out repeated expressions into variable makes it easy to correct mistakes — there is only one place where to fix the issue instead of two here. Doing so also saves us a tiny bit of computation since we compute each expression once instead of twice.

Note that the three expressions we have factored out have an intrinsic meaning: they are how far apart the two points are on each of the three dimensions. By giving these quantities names, as opposed to referring to their calculation, we have achieved a simple form of **abstraction**. As we will see, abstraction often leads to clearer code because it lets us focus on *what* a quantity is as opposed to *how* it is computed.

2.3 Use Helper Functions!

In the returned value, we do the exact same thing with `dx`, `dy` and `dz`: square them. We gain in abstraction if we factor out this common computation into a helper function:

```
// returns the distance component over one dimension
int dist1(int x) {
  return x * x;
}

// returns the square of the distance between 3D points P and Q
int distance(int[] P, int[] Q)
//@requires \length(P) == 3;
//@requires \length(Q) == 3;
{
  int dx = P[0] - Q[0];
  int dy = P[1] - Q[1];
  int dz = P[2] - Q[2];

  return dist1(dx) + dist1(dy) + dist1(dz);
```

```
}
```

One advantage of doing so is that if we decide to change the way we compute the distance over one dimension, we only need to modify the helper function `dist1`. This is not as esoteric as it seems: although in the back of our mind we were returning (the square of) the Euclidean distance between two points, there are other useful notions of distance. For example, changing `dist1` to return the absolute value of its argument yields what is called the Manhattan distance.

Helper functions are a great way to make code readable. In general, if you do the same computation twice or more, you may consider factoring it out into a helper function. They are also useful in breaking long functions into smaller chunks that can be understood independently.

2.4 Use Loops!

Now, each of `dx`, `dy` and `dz` is computed in exactly the same way on different indices of the array representation of the points. This common pattern allows us to introduce a new form of abstraction: use a loop whose body captures this shared template.

```
// returns the distance component over one dimension
int dist1(int x) {
  return x * x;
}

// returns the square of the distance between 3D points P and Q
int distance(int[] P, int[] Q)
//@requires \length(P) == 3;
//@requires \length(Q) == 3;
{
  int dx = P[0] - Q[0];
  int dy = P[1] - Q[1];
  int dz = P[2] - Q[2];

  int res = 0;
  for (int i = 0; i < 3; i++) {
    res += dist1(P[i] - Q[i]);
  }

  return res;
}
```

This last version shows another stylistic faux pas: we do not need the variables `dx`, `dy` and `dz` any more, and yet we left them lying around. They are **dead code**. It takes mental effort to understand code, so we should try to make this effort as little as possible. One way is to delete dead code once we are confident we won't need it. Here it was a couple of variables, but it could be entire functions.

2.5 Remove Dead Code!

Here's the cleaned up code that uses a loop to carry out repeated computation patterns:

```
// returns the distance component over one dimension
int dist1(int x) {
  return x * x;
```

```
}

// returns the square of the distance between 3D points P and Q
int distance(int[] P, int[] Q)
//@requires \length(P) == 3;
//@requires \length(Q) == 3;
{
  int res = 0;
  for (int i = 0; i < 3; i++) {
    res += dist1(P[i] - Q[i]);
  }

  return res;
}
```

2.6 Make it General!

One benefit of changing our dimension-by-dimension computation into a loop is that it is now trivial to generalize this code to compute the distance between two points in *any* number of dimensions!

```
// returns the distance component over one dimension
int dist1(int x) {
  return x * x;
}

// returns the square of the distance between n-D points P and Q
int distance(int[] P, int[] Q, int n)
//@requires \length(P) == n;
//@requires \length(Q) == n;
{
  int res = 0;
  for (int i = 0; i < n; i++) {
    res += dist1(P[i] - Q[i]);
  }

  return res;
}
```

Again, abstraction at work!

3 Things that Make Programmers Cringe

Good style has to do with readability. So far, we've seen that good code layout and abstraction go a long way towards making code easy to read and understand. Next, we'll look at a few ways of writing code that people consider bad style, not because they make the code particularly hard to read, but because they cry *NOVICE*! This is not the kind of code you want to write during an interview. Even when going to office hours, TAs will explain things differently based on how you wrote your code. Interestingly, most of these things have to do with the simplest of C0's types: booleans.

3.1 Consider Returning Expressions!

Consider the following function that returns whether a number is negative or not:

```
bool negative(int x) {
  if (x < 0)
    return true;
  else
    return false;
}
```

What's wrong with it? Well, nothing except that it can be written more simply this way:

```
bool odd(int x) {
  return x < 0;
}
```

3.2 Don't Compare against `true` or `false`

Building on this example, here a way to compute the absolute value of a number:

```
int abs(int x) {
  if (negative(x) == true)
    return -x;
  else
    return x;
}
```

Here, the part that will look funny to people is "`== true`". Good programmers would write:

```
int abs(int x) {
  if (negative(x))
    return -x;
  else
    return x;
}
```

or even

```
int abs(int x) {
  return negative(x) ? -x : x;
}
```

(You'll learn about this idiom later in the course — no need to worry about it now.)

Flipping the things around, this version will also look "interesting":

```
int abs(int x) {
  if (negative(x) == false)
    return x;
  else
    return -x;
}
```

while this one is fine (although the ones earlier are simpler):

```
int abs(int x) {
  if (!negative(x))
    return x;
  else
    return -x;
}
```

3.3 Use "and" and "or"!

Let's write a function that adds two numbers if both are negative and subtracts them otherwise:

```
int special_op(int x, int y) {
  if (negative(x)) {
    if (negative(y))
      return x + y;
  }
  return x - y;
}
```

Here, the nested **if**s have the purpose of returning the sum only when both `x` and `y` are negative. A simpler way to express this is by means of a conjunction (&&):

```
int special_op(int x, int y) {
  if (negative(x) && negative(y)) {
    return x + y;
  }
  return x - y;
}
```

You can think of similar patterns involving disjunction (||).

3.4 Line up your Bounds!

One last thing: you will often need to check that a value is between two bounds — whether in a conditional, a loop or a contract. Novices often write code like this:

```
void in_bounds(int i, int n) {
  if (i >= 0 && i < n)
    printf("%d is in bounds\n", i);
  else
    printf("%d is NOT in bounds\n", i);
}
```

Again nothing wrong except that it conveys the impression that you are not a systematic, logical thinker. Much better is to make sure the three values appear from smallest to largest:

```
void in_bounds(int i, int n) {
  if (0 <= i && i < n)
    printf("%d is in bounds\n", i);
  else
    printf("%d is NOT in bounds\n", i);
}
```

3.5 Clean up Debugging Statements

Most likely, you wrote `in_bounds` to debug some other function — debugging with print statements is a useful technique. For example, the following function is meant to return how many consecutive positive numbers are at the very end of an array:

```
int count_positives_at_end(int[] A, int n)
//@requires n == \length(A);
//@ensures 0 <= \result && \result <= \length(A);
{
  int count_pos = 0;
  for (int i = n; i >= 0; i--) {
    in_bounds(i, n);  // DEBUG ONLY
    if (A[i] <= 0) return count_pos;
    count_pos++;
  }
  return count_pos;
}
```

This function has a bug — it access the array out of bounds — and you use `in_bounds` to find out what is going on.

You quickly figure out that the loop should start at `n-1` and fix your code:

```
int count_positives_at_end(int[] A, int n)
//@requires n == \length(A);
//@ensures 0 <= \result && \result <= \length(A);
{
  int count_pos = 0;
  for (int i = n-1; i >= 0; i--) {
    in_bounds(i, n);  // DEBUG ONLY
    if (A[i] <= 0) return count_pos;
    count_pos++;
  }
  return count_pos;
}
```

And that's it. All problems are solved. You move on to greater things.

The debug statement, which had the only purpose of finding your bug, is still there however. This will mildly annoy anybody reading your code: you are coming across as somebody who doesn't clean after themselves. Printing to terminal also can slow down execution considerably. This means that if you call `count_positives_at_end` on a very large array, it may take a long time to compute — minutes versus seconds!

What can you do? Commenting out debug statements solves the slow-down issue, but your code still looks messy and a bit amateurish:

```
int count_positives_at_end(int[] A, int n)
//@requires n == \length(A);
//@ensures 0 <= \result && \result <= \length(A);
{
  int count_pos = 0;
  for (int i = n-1; i >= 0; i--) {
```

```
    // in_bounds(i, n);  // DEBUG ONLY
    if (A[i] <= 0) return count_pos;
    count_pos++;
  }
  return count_pos;
}
```

What you want to do instead is to delete debug statements entirely:

```
int count_positives_at_end(int[] A, int n)
//@requires n == \length(A);
//@ensures 0 <= \result && \result <= \length(A);
{
  int count_pos = 0;
  for (int i = n-1; i >= 0; i--) {
    if (A[i] <= 0) return count_pos;
    count_pos++;
  }
  return count_pos;
}
```

In this particular example, you could have used a loop invariant to debug the function. Loop invariants are good to keep in finish code as they act as documentation. Moreover, they do not slow down execution unless it is compiled with the `-d` flag — something that is not typical of finished code.

```
int count_positives_at_end(int[] A, int n)
//@requires n == \length(A);
//@ensures 0 <= \result && \result <= \length(A);
{
  int count_pos = 0;
  for (int i = n-1; i >= 0; i--)
  //@loop_invariant 0 < i && i <= n-1;
  {
    if (A[i] <= 0) return count_pos;
    count_pos++;
    }
  return count_pos;
}
```

In general, you will want to use a combination of contracts and print statements to debug your code. Remember however to remove the print when you are done: clean after yourself!

4 Beyond the Basics

In this guide, we touched the surface of what constitutes good coding style. Most programmers tend to apply the conventions we mentioned, but there is little common agreement beyond them. Different software companies will expect their programmers to follow specific style guidelines to ensure internal consistency, but these guidelines will often vary by company.