Why and How You Should Comment Your Code
David Kieras, EECS Department
September, 2003

Why comment?

The purpose of comments is to help yourself to understand your own code, and especially to help other people read and understand your code.

Experience shows that grading comments for quality in detail is difficult, and computer grading of comments is technically not really possible. But we can grade whether your comments are generally adequate, and whether you have enough comments to be reasonable.

Here is how we will enforce commenting requirements: If you ask for help with your code, and it i poorly commented and hard to read, we reserve the right to tell you to go away until you have commented it and cleaned it up, especially if there are people waiting for our help.

Comments are there to help people read the code, so if you leave them out, it will be *your* problem, not ours! And if you don't put comments in your code from the beginning, chances are it will be harder for you to write the code. So put comments in while you are writing the code, when they will help you, and help us to help you; don't leave them until the end!

Then if we are assessing the project for quality, if you have commented your code in a way that helps develop the code and debug it, you should be in good shape when we assess the quality of your code.

How comment?

What should your comments be like, and how long should they be? Many textbooks seem to go overboard on this. But the working assumption in industry is that the code is going to be read by a programmer familiar with the language and with the specifications or purpose of the program. So comments should be limited to saying things that are not obvious from the code or from external documents. Also, by using good function and variable names, and the expressiveness allowed by the syntax of the language, one can say a lot about the code in the code itself.

Each function should have a short comment before it that says what the purpose of the function is, or summarizes its operation, or key design decisions or assumptions involved in it. Note that the result or effect of a function is usually less obvious than what its inputs are, which the parameter names and types should make obvious.

Then inside the function, use brief comments for anything that would slow the reader down - e.g. i it took you a while to figure out how to write the code, it would take a reader a long time to figure out what the code is doing, so you tell them up-front with a few selected comments. The best comments are essentially an outline for what the code is supposed to do. An old programmer's tric is to write the comments first, and then fill in the actual code! Writing the comments first ensures that they actually describe important ideas about the code, and so help both you and the reader.

But never write comments that are totally synonymous with the code, like

```
// add one to i
i++;
```

```
      // call foo with i
      foo(i);
      // go back to caller
      return;
```

Such comments are a waste of time - unless you are really at the earliest stage of learning. If so, be sure to eliminate them when you no longer need them.

Some real examples

Below is a function from some of my own research code, that took a non-trivial amount of time to design. My research colleagues who use my code have often given me positive feedback about the comments, so I must be doing something right; if anything, I comment somewhat too heavily.

Anyway, in this example, I am assuming that the reader knows the Standard Library container classes, which are used everywhere in the project, and the #includes should tell the reader this. It would be crazy to document in every function how these Standard Library things work.

But I wanted to make sure that when *I* or someone else has to look at this code, we don't have to figure the design decisions out all over again. We can read the comments and see critical design issues and the basic logic of the function without puzzling over any of the details in the code. As ar example of the code explaining itself, the "const Binding_set& bs" in the parameter list tells the reader that the input is a "binding set" that is neither copied nor modified. While the name "bs" for the parameter isn't very informative, it is the only binding set in the picture, so it's pretty clear that is the binding set that is being removed.

```
/*
Remove a binding set by finding it and removing it.
add_binding_set ensured each binding set is unique, so
removal here does not have to check for duplication.
Return true if a change was made.
*/
bool Binding_set_list::remove_binding_set(const Binding_set& bs)
{
      // if list is empty, no removal, no change
      if (binding_set_list.size() == 0)
            return false;
      // search nonempty list for binding set
      Binding_set_list_t::iterator pos =
            find(binding_set_list.begin(), binding_set_list.end(), bs);
      if (pos == binding_set_list.end())
            return false;     // not present, no change

      binding_set_list.erase(pos);
      return true;      // changed
}
```

Understanding the purpose of even a very simple function can be helped by a comment, to spare th
reader from having to parse the code. Another example from my own code:

```
// output a pair in parentheses, with a colon after the varname
ostream& operator<< (ostream& os, const Binding_pair& bp)
{
      os << '(' << bp.get_var_name() << ':' << bp.get_var_value() << ')';
      return os;
}
```

Here is another short example, showing a couple of comments that explain how the code deals wit
obscure facts about the math library that might otherwise stump the reader:

```
// construct a Polar_vector from a Cartesian_vector
Polar_vector::Polar_vector(const Cartesian_vector& cv)
{
      r = sqrt((cv.delta_x * cv.delta_x) + (cv.delta_y * cv.delta_y));
      // atan2 will return negative angle for Quadrant III, IV.
      theta = atan2 (cv.delta_y, cv.delta_x);
      if (theta < 0.)
            theta = 2. * GU_pi + theta; // normalize theta positive
}
```

Note: I haven't spruced this code up for other people yet! So if I find it helpful to comment my ow
code in this way for myself, chances are you will too!

Finally, here is a long example, in which a complex algorithm is being used that I learned for the first time when I wrote this function some years ago, so the comments helped me understand what was going on; recently I had to revisit this code when I moved it into a new environment. My original comments helped me make sense of it, greatly simplifying the modifications I had to make Howver, because it presupposes a reader willing to work at it, not all the complexities are commented on.

```
/*
Calculate the line that intersects a rectangle through its center, using a specializat
of the Cohen-Sutherland clipping algorithm (see Foley, van Damm, Feiner, & Hughes).

The start-to-center line goes from the start point P0 to the end point P1,
which is the center of the rectangular target region, whose boundaries are
min_edges.x, max_edges.x, min_edges.y, max_edges.y.

Since P1 is always inside the rectangle, we check only for whether P0 is also inside,
and then which edge of the rectangle the line crosses, and compute the point of
intersection.

clipped_line is the line ending in the center that intersects the edge of rectangle.
return true if starting point is outside and d and s values are valid; false otherwise
*/

bool compute_center_intersecting_line(const Line_segment& start_to_center, const Size
rect_size, Line_segment& clipped_line)
{
      // the start of the line is p0, the starting point, and will get moved to
// points of intersection during the computation
Point p0(start_to_center.get_p1());

// the end of the line is p1, the center of the target, and stays fixed.
const Point p1(start_to_center.get_p2());

// max and min_edges are the boundaries of the target rectangle
// max edges are the top right coordinates
Point max_edges(p1.x + rect_size.h/2, p1.y + rect_size.v/2);
// min edges are the lower left coordinates
Point min_edges(p1.x - rect_size.h/2, p1.y - rect_size.v/2);

// Where does the line p0-p1 intersect the rectangle?
// Calculate intersection accordingly.
// if the point is not outside, then start point must be inside the rectangle,
// and no movement should be made

Point p; // point of intersection
bool top, bottom, right, left;
bool first = true;

while(true) {
top = bottom = left = right = false;
// calculate where p0 is outside
if (p0.y > max_edges.y) {
      top = true;
      }
else if (p0.y < min_edges.y) {
      bottom = true;
      }
if (p0.x > max_edges.x) {
      right = true;
      }
else if (p0.x < min_edges.x) {
      left = true;
      }
// if p0 is not outside, we are done
// If it was the first value of p0, then starting point was inside,
// and result is invalid
// otherwise stop and compute the final result
// if p0 is still outside, continue the computation with the new value
if (!(top || bottom || right || left)) {
      if(first)
            return false;
      else
            break;
      }
else
      first = false;
// Where does the line intersect the rectangle?
// On the top
if (top) {
      p.x = p1.x + (p0.x - p1.x) * (max_edges.y - p1.y)/(p0.y - p1.y);
      p.y = max_edges.y;
      }

// On the bottom
```

```
        else if (bottom) {
                p.x = p1.x + (p0.x - p1.x) * (min_edges.y - p1.y)/(p0.y - p1.y);
                p.y = min_edges.y;
                }

        // On the right
        else if (right) {
                p.x = max_edges.x;
                p.y = p1.y + (p0.y - p1.y) * (max_edges.x - p1.x)/ (p0.x - p1.x);
                }

        // On the left
        else if (left) {
                p.x = min_edges.x;
                p.y = p1.y + (p0.y - p1.y) * (min_edges.x - p1.x)/ (p0.x - p1.x);
                }

        // move p0 to the point of intersection and repeat
        p0 = p;
        }       // main loop
        // return the line segment between the intersection and the center
        clipped_line = Line_segment(p0, p1);
        return true;
}
```