DESIGN BY CONTRACT

Why Comment?

For some very *practical* reasons:

- To help you **write** the program.
  It is hard to write correct code until you know what you want to do! Your comments should serve as a guide.
- To help other programmers (clients of your code) **use** the classes and functions you write. They need to know only what your code does, not how.
- To help you and your team **maintain** the code you own. You and your team need to know in detail how your code works.
- To **convince** yourself and others that your code is correct. Certain comments specify what "correct" means, and others, called "assertions," help establish that the code is correct.

Helping Client Programmers: External Comments

Client programmer: someone who writes code that uses yours.

Clients *don't need to see* how your classes are implemented. This is the only way programmers can deal with the complexity of huge programs. But it won't work unless you've written complete, clear, and precise external comments.

In fact, clients typically *can't see* your function bodies, local variables, and private variables, because they don't have the code and are working from external documentation.

Clients need to know:

- **The purpose of the class:**
  What does your class represent?
  (e.g., student, network connection, queue.)
  A fairly brief summary comment at the top of the class is enough.
- **What the functions do (not how):**
  See "function specifications" below.
- **Important facts about performance:**
  How fast is your code, and how much memory does it use? (More on this later.)

Rule: Do not mention private members, algorithms, or implementation details in external class and function comments.
Describe everything in terms of the abstract contents of the class, and the parameters to the functions.

External: Function Specifications

Function specs state precisely what the function does under all circumstances. Always include:

Function Summary: A brief summary of the function's purpose.

Precondition: A boolean statement that must be true when the function is called in order for it to work correctly, otherwise the behaviour is unspecified.

Preconditions may, for example:

- restrict a parameter's value
  Example: `p must not be null.`
- restrict how parameters relate to each other
  Example: `0 <= size < a.length.`
- require that a condition be checked before calling the function
  Example: `isFull() must be false.`

Postcondition: A boolean statement that is guaranteed to be true when the function returns, as long as the precondition was met.

Often written informally, as a part of the function summary.

The postcondition must specify:

- the purpose of every parameter
  Example: `size is the number of array elements in use.`
- what the function returns (unless it is `void`)
  Example: `Returns the position of the maximum value in a[0..size-1], or -1 if size is zero.`
- how exactly the function affects the object
  Example: `Removes the front element of the queue.`

Ideally, a function has weak preconditions (some even have none) and strong postconditions.

Exercises

Identify flaws in the following function specification comments:

```
From a template for a queue class:
    // Return the first object in array 'contents'.
    Type head();


From a template for a queue class:
    // Append o to me, wrapping tail around to the
    // front of the array if necessary.
    void enqueue(Type o);


From a stringBuffer's replace function:
    // Replace part of me with a new string.
    void replace(int start, int end, string str);
```


Design by Contract

External comments are like a contract between you and the client: provided that the client meets the precondition, your code will behave as stated.

Design by Contract is a very important concept in software engineering.

Rule: When you design a class, write a contract for it *before* you write all the code.

Example: A Queue contract

Here are two function specifications from a template for a `Queue` class:

```
// Append o to me.
void enqueue(Type o);

// Remove and return my front object.
// Precondition: I am not empty.
Type dequeue();
```

Note the precondition on `dequeue`. Here is how to read the specification as a contract:
"Provided that the queue is not empty, a call to `dequeue()` will remove and return the front object in the queue."

There is no precondition on `enqueue()`. This means the programmer is claiming there are no conditions that cause it not fulfill its contract.

Helping Your Team: Internal Comments

Your programming team will include other people working on the same product as you. They (and you) must quickly figure out your code, in order to track down bugs and add features.

Internal comments should explain *how* the code works and *why* it was designed that way. They should:

- explain design decisions, algorithm choices, tricky bits of code, and the purpose of member and local variables.
- state assertions and representation invariants (see below).
- summarize major steps within each fcn.

Internal comments should appear on member variables and *inside* function bodies.

Internal Comments: Assertions

Assertions are boolean statements describing what had better be true at a point in a program's execution (otherwise the code must be incorrect).

Assertions can be expressed simply as comments. Some programming languages, including C and C++ have assertions that can be executed to check whether code fulfills its contract.

You can use assertions to help you *write* code. Example (pseudo-code):

```
a print function for CircularQueue:
    loop i from head to tail (wrapping around) {
        Assertion: 0 <= i < A.length
        print contents[i]
    }
```

Writing down the assertion reminds me to check that my funky loop arithmetic with wraparound doesn't cause `contents[i]` to go out of bounds.

Internal Comments: Representation Invariants

Invariant: Something that never changes. Here we mean something that is always true.

A representation invariant describes the properties that instance variables of a class must meet at all times (except while one of its functions is executing).

A representation invariant should express:

- How the instance variables represent the abstract thing.
- Constraints on the values of the data members.
- Relationships among the data members.

Why bother?

A representation invariant is very helpful because it tells the person writing the functions:

- exactly what they must maintain in order for the instance variables to make sense, and subsequent function calls to work. This is called maintaining *internal consistency*.
- to watch out for particular borderline cases and so avoid errors.

Rule: Whenever you write a class, provide a representation invariant.

Summary of Comment Types

External comments:

- class summary
- function specifications
- performance facts

Internal comments:

- representation invariants
- assertions
- other comments explaining how the code works and why it was designed that way.

3 Ways to do a Circular Queue

Tail is where to enqueue next

Need a size counter to distinguish an empty queue from a full queue.

```
// Let c be the capacity of array 'contents'.
// 0 <= size <= c, is the number of elements in me.

// 0 <= head < c.
// 0 <= tail < c.

// If size is 0, I am empty and head=tail.
// Otherwise:
//    contents[head] is the head.
//    contents[tail-1] is the tail.
//    if head < tail,
//        contents[head .. tail-1] contains my elements in
//        the order they were inserted, and
//        size=tail-head.
//    if head >= tail,
//        contents[head .. c-1, 0 .. tail-1] contains
//        my elements in the order they were inserted, and
//        size=tail-head+c.
```


Tail is where we last inserted

What is tail when the queue is empty?

When queue size is 2, tail is 1 past head. When queue size is 1, tail is at head. So when queue size is 0, it makes sense to have tail be one less than head (with wraparound).

Then we need a size counter to distinguish an empty queue from a full queue.

The representation invariant is slightly different.

Note: "slightly" here doesn't mean insignificant; it means subtle. And that means watch out for bugs.

Keep an extra unused slot

Don't need a size counter.

Again, the RI is different.

```
// Let c be the capacity of array 'contents'.
// We can store  at most c-1 elements in the queue.

// 0 <= head < c.
// 0 <= tail < c.

// If (tail+1)%c = head, I am empty.
// Otherwise:
//    contents[head] is the head.
//    contents[tail-1] is the tail.
//    if head < tail,
//        contents[head .. tail-1] contains my elements in
//        the order they were inserted, and
//        size=tail-head.
//    if head >= tail,
//        contents[head .. c-1, 0 .. tail-1] contains
//        my elements in the order they were inserted, and
//        size=tail-head+c.
```