Invariants, Preconditions, and Postconditions:

Invariant is a condition that is supposed to be true all the time (except for brief, well-defined intervals). E.g., one invariant would be that if the cartesian and polar flags are both true, then the cartesian and polar member variables must represent the same complex number. Another example: in the selection sort, the part of the vector we have already scanned is sorted.
It's a good idea to document significant invariants.

Preconditions: These are conditions that must be true before a function (or code segment) executes, in order for the code to work correctly. E.g., a bisection search will work only if the vector is sorted. Another: Converting to polar form will work only if the cartesian coordinates are valid.
It's a good idea to document preconditions.

Postconditions are conditions that are true after a functions returns or at the end of a code segment. Document significant postconditions (i.e., ones that refer to the purpose of the code).

Invariants, preconditions, and postconditions represent how we expect our program to work, but we make mistakes. So it's a good idea to check conditions when it's not too expensive to do so. (What is too expensive? What are the costs of a failure?) Think of a fuse.

One simple way of doing this in C++ is with the assert() function. #Include <assert.h>, and then in program you can write:

```
assert( <condition> );
```

If the <condition> is false, your program will abort with a more or less comprehensible message.

Object-Oriented Design:

We often understand things in terms of classes and subclasses. E.g., the class of living things includes animals, plants, bacteria, fungi, etc. The class of animals includes as subclasses mammals, reptiles, fish, birds, insects, etc.
The class of mammals includes dogs, horses, cats, etc.
Dogs include collies, Dobermans, poodles, etc.
This is a class hierarchy.

Ultimately, the members (or instances) of a class such as collie are an open-ended set of individuals, such as Rover, Spot, Fido, etc.

Example:
Suppose you were designing a video game.
Working top-down from the most general class of objects, we might have:

displayObject:
Properties might include X and Y coordinates; whether it's visible; size; etc.
Behaviors might include move to some X, Y; make it appear or disappear; change size.

Subclasses of displayObject might include: notifications, menus, moving objects, scenery or background, etc.

movingObject is a subclass of displayObject:
Properties might include velocity (direction and speed)
Behaviors might include change velocity (change direction or speed), etc.
Notice that movingObjects have all the properties and behaviors of displayObjects.

We say, movingObjects inherit the properties and behaviors of displayObjects.

announcementObject might be another subclass of displayObject with
Properties such as message text, flashing, font, color, etc.
Behaviors such as show and erase.

animateObjects is a subclass of movingObjects, with
properties such as name, life meter, living/dead, gate of the legs (e.g., running or walking).
Behaviors might include walk, run, stop, turn, jump, etc.

In OOD, programming becomes more like writing a script for a play.

How do you do it in C++ (an OOPL)?

Recall that member variables and member functions can be public or private. Public members are visible to any user of the class; they are part of the interface. Private members are only visible within the class. They are part of the inner workings or "guts" of the class.

If you are defining a subclass to another class, it is essentially an extension to the parent class, and it may need access to some of those internal mechanisms. So if you are defining a class, and you want to make certain variables or functions visible to designers of subclasses of your class, then declare these protected rather than public or private.

private = visible only in the class
protected = visible only in the class and its subclasses (and potentially subsubclasses, etc.)

public = visible to everybody.

If you want to permit designers of a subclass to redefine a function defined in the parent class, then it must be declared virtual in the parent class. This allows you to make the behaviors of a subclass more specific than the parent class. For example, all animateObjects might be able to move from one place to another, but a person (subclass of animateObject) would probably move differently than a airplane (subclass of animateObject).