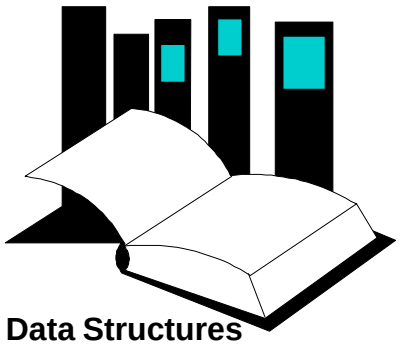
# Preconditions and Postconditions



**Data Structures and Other Objects Using C++**

- An important topic: **preconditions** and **postconditions**.
- They are a method of specifying what a function accomplishes.

This is the first of several lectures which accompany the textbook *Data Structures and Other Objects Using C++.* Each lecture chooses one topic from the book and expands on that topic - adding examples and further material to reinforce the students' understanding.

This first lecture covers the topic of Preconditions and Postconditions from Chapter 1.

## Preconditions and Postconditions

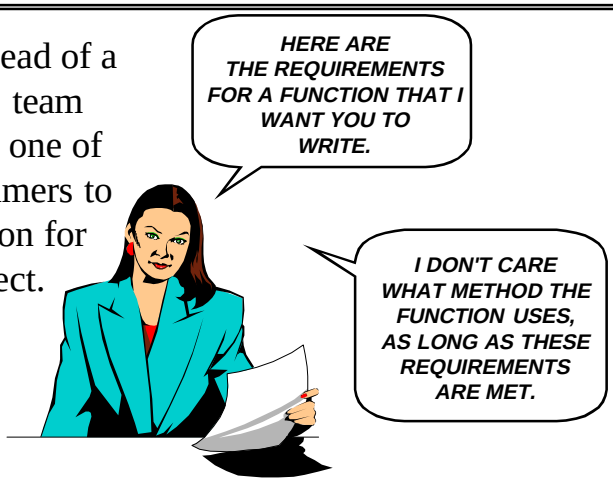
Frequently a programmer must communicate precisely **what** a function accomplishes, without any indication of **how** the function does its work.

*Can you think of a situation where this would occur ?*

Throughout the book, preconditions and postconditions are used to specify precisely what a function does. However, as we will see, a precondition/postcondition specification does not indicate anything about how a function accomplishes its work. This separation between what a function does and how the function works is extremely important - particularly for large programs which are written by a team of programmers.

## Example

- You are the head of a programming team and you want one of your programmers to write a function for part of a project.




As an example, suppose that you are the head of a programming team. Your team is writing a large piece of software, perhaps with millions of lines of code. Certainly nobody can keep all those lines of code in their head at once (not even me!). So, the large problem is broken into smaller problems. Those smaller problems might be broken into still smaller problems, and so on, until you reach manageable problems.

Each of the manageable problems can be solved by a function - but you won't be writing all these functions. The functions are written by members of your team.

As each team member is given a function to write, you will specify the requirements of the function by indicating what the function must accomplish. But most of the details about how a function works will be left up to the individual programmers.

## What are Preconditions and Postconditions?

- One way to specify such requirements is with a pair of statements about the function.
- The **precondition** statement indicates what must be true before the function is called.
- The **postcondition** statement indicates what will be true when the function finishes its work.

There are many ways to specify the requirements for a function. In this class, and in the textbook, we will use a pair of statements for each function, called the function's precondition and postcondition.

As we will see, the two statements work together: The precondition indicates what must be true before the function is called. The postcondition indicates what will be true when the function finishes its work.

An example can clarify the meanings...

## Example

```
void write_sqrt( double x)

//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.


   . . .
```

This is an example of a small function which simply writes the square root of a number. The number is given as a parameter to the function, called x. For example, if we call write_sqrt(9), then we would expect the function to print 3 (which is the square root of 9).

What needs to be true in order for this function to successfully carry out its work? Since negative numbers don't have a square root, we need to ensure that the argument, x, is not negative. This requirement is expressed in the precondition:

Precondition: x >= 0.

The postcondition is simply a statement expressing what work has been accomplished by the function. This work might involve reading or writing data, changing the values of variable parameters, or other actions.

Notice that the information shown on this slide is enough for you to use the function. You don't need to know what occurs in the function body.

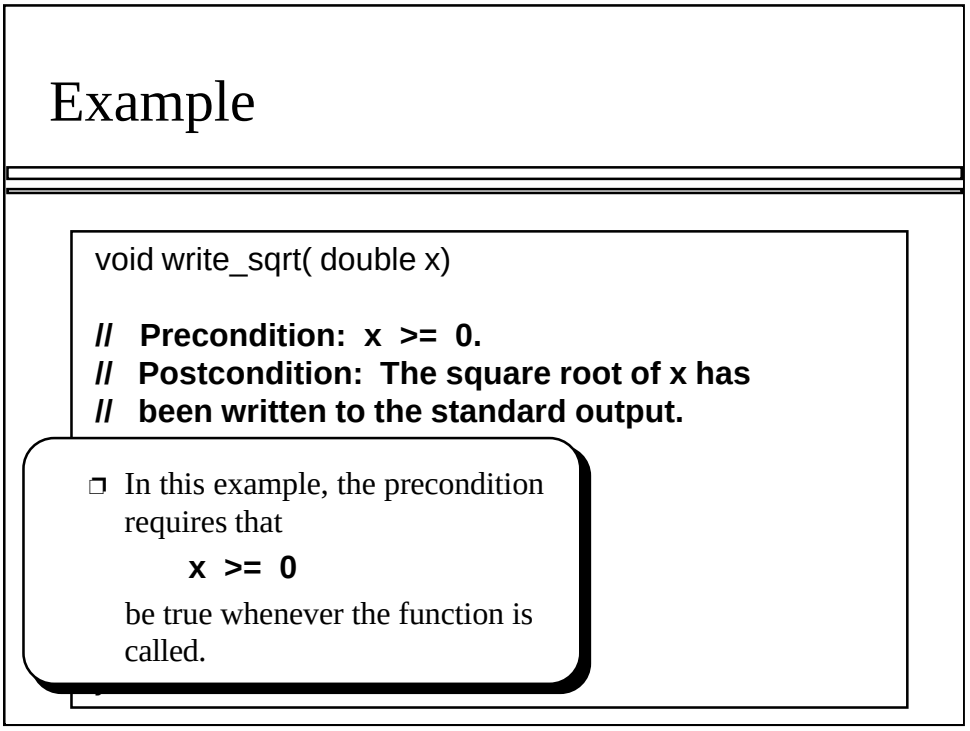
## Example

```
void write_sqrt( double x)

//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.
```

- The precondition and postcondition appear as comments in your program.
- They are usually placed after the function's parameter list.

The precondition and postcondition are not actually part of the program. It is common to place the precondition/postcondition pair in a comment immediately after the function's parameter list.




Here again you see the precondition of the example. The right way to read this is as a warning that says: "Watch Out! This function requires that x is greater than or equal to zero. If you violate this condition, then the results are totally unpredictable."

## Example

*Which of these function calls meet the precondition ?*

```
write_sqrt( -10 );
write_sqrt( 0 );
write_sqrt( 5.6 );
```

So, here are three possible function calls. Two of the calls meet the precondition and have predictable results. In one of the calls, the precondition fails, and the result of the function call is unpredictable.

Which function call is the trouble maker?

## Example

*Which of these function calls meet the precondition ?*

```
write_sqrt( -10 );
write_sqrt( 0 );
write_sqrt( 5.6 );
```

The second and third calls are fine, since the argument is greater than or equal to zero.

The second and third function calls are fine. The second call has an argument of zero, but that's acceptable since the precondition only requires that x is greater than or equal to zero.

## Example

*Which of these function calls meet the precondition ?*

```
write_sqrt( -10 );
write_sqrt( 0 );
write_sqrt( 5.6 );
```

But the first call violates the precondition, since the argument is less than zero.

But the first function call causes trouble. This function call, which violates the precondition, must never be made by a program. In a few minutes you'll see exactly how much trouble can arise from such a violation. For now, just take my word, do not violate preconditions.

## Example

```
void write_sqrt( double x)

//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.
```

- The postcondition always indicates what work the function has accomplished. In this case, when the function returns the square root of **x** has been written.

Before we continue, take one more quick look at the postcondition: As you know, it states what the function will accomplish between the time the function starts executing and the time the function finishes executing.

One more important point which isn't written on the slide: Provided that the precondition is valid, then the function is also required to finish executing. Infinite loops are not permitted, and neither is crashing the computer.

## Another Example

```
bool is_vowel( char letter )
//   Precondition:  letter is an uppercase or
//   lowercase letter (in the range 'A' ... 'Z' or 'a' ... 'z') .
//   Postcondition:  The value returned by the
//   function is true if Letter is a vowel;
//   otherwise the value returned by the function is
//   false.

   . . .
```

Here's one more example, which demonstrates how you can use ordinary English to express the precondition and postcondition.

The writing of these expressions should be clear and concise. The goal is to communicate to another programmer two things:

1. What must be true in order for that programmer to use the function; and
2. What work the function will accomplish.

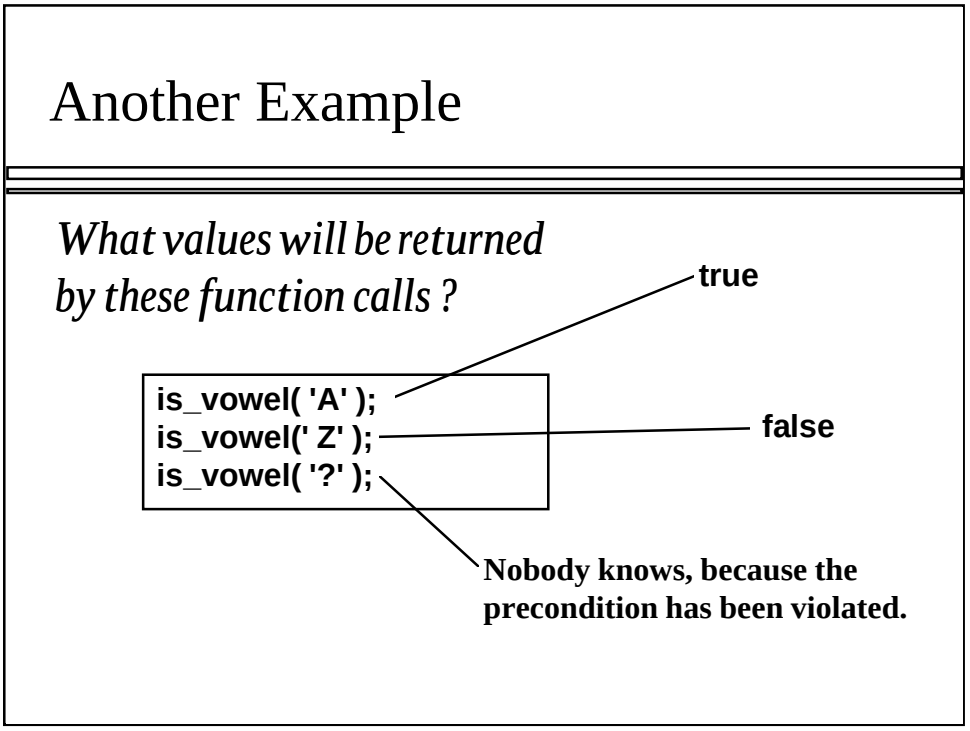
In this example, the "work accomplished" is nothing more than computing a value which the function returns. Again, notice that there is enough information for you to use the function without knowing a thing about the implementation details.

## Another Example

*What values will be returned by these function calls ?*

```
is_vowel( 'A' );
is_vowel(' Z' );
is_vowel( '?' );
```

Another quick quiz: What values will these function calls return? If you think this is a "trick question" you are right. . .




The first two function calls are fine, returning true (since 'A' is a vowel) and false since 'Z' is not a vowel.

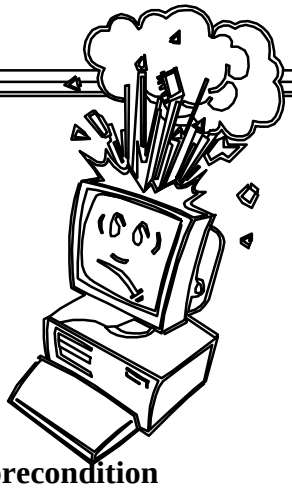
But the third function call might return true, or it might return false, nobody really knows since the precondition has been violated.

In fact, the situation is worse than that. Recall that I said to never violate a precondition. The reason is that the result of violating a precondition is totally unpredictable, including the possibility of . . .

## Another Example

*What values will be returned by these function calls ?*

```
is_vowel( '?' );
```



**Violating the precondition might even crash the computer.**

. . . crashing the computer.

Now, if I had written the is_vowel function, and the argument was a question mark, I would try to not crash the machine, I would try to not destroy files on the hard drive, I would try my best to not cause power outages across New York. But you never know for sure.

## Always make sure the precondition is valid . . .

- The programmer who calls the function is responsible for **ensuring that the precondition is valid** when the function is called.


So, let's look at the use of preconditions and postconditions in a typical situation. The programmer who calls the function is responsible for ensuring that the precondition is valid when the function is called.

If she fails in this responsibility, then all bets are off. There is no telling what might occur.

## . . . so the postcondition becomes true at the function's end.

- The programmer who writes the function counts on the precondition being valid, and **ensures that the postcondition becomes true** at the function's end.


On the other hand, if she keeps her end of the bargain, and calls the function with a valid postcondition, then the function has a responsibility of its own.

The function must complete its execution (no infinite loops), and when the function finishes, the postcondition will be true.

In some ways, you can think of the precondition/postcondition statements as a contract between two programmers: One programmer (who uses the function) is guaranteeing that she will make sure that the precondition is valid before the function is called. The other programmer (who writes the function) is going to bank on the precondition being true. This other programmer is responsible for making sure that the postcondition becomes true when the function finishes execution.

## A Quiz

*Suppose that you call a function, and you neglect to make sure that the precondition is valid. Who is responsible if this inadvertently causes a 40-day flood or other disaster?*

① You

② The programmer who wrote that torrential function

③ Noah

Time for another quiz . . .

# A Quiz

*Suppose that you call a function, and you neglect to make sure that the precondition is valid. Who is responsible if this inadvertently causes a 40-day flood or other disaster?*

① You

The programmer who calls a function is responsible for ensuring that the precondition is valid.

Somehow I think this quiz was too easy.

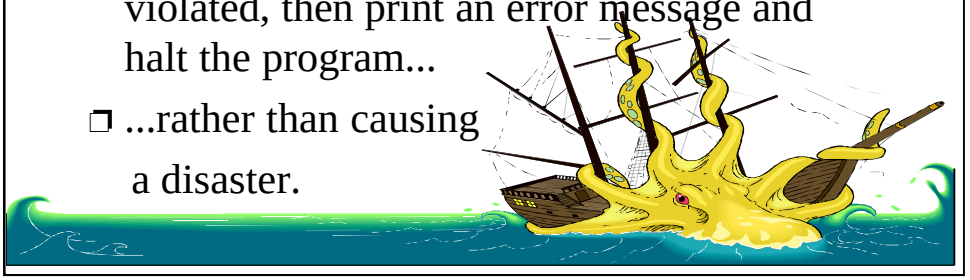
## On the other hand, careful programmers also follow these rules:

- When you write a function, you should make every effort to detect when a precondition has been violated.
- If you detect that a precondition has been violated, then print an error message and halt the program.

Well, there's no way of getting around it: The programmer who calls a function is responsible for making sure the precondition is valid. However, when you are writing a function with a precondition, you should make every attempt to try to detect when a precondition is violated. Such detections make things easier on other programmers - easier for them to debug, for example.

## On the other hand, careful programmers also follow these rules:

- When you write a function, you should make every effort to detect when a precondition has been violated.
- If you detect that a precondition has been violated, then print an error message and halt the program...
- ...rather than causing a disaster.



And such detections can also avoid disasters.

## Example

```
void write_sqrt( double x)
//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.
{
    assert(x >= 0);

  . . .
```

- The assert function (described in Section 1.1) is useful for detecting violations of a precondition.

Here's an example of how you would write a friendly function which detects when its precondition is violated. There is no need for anything fancy when the precondition fails: just print an informative error message and halt the program. In this example, I have used the C++ assert function, which has a logical expression as its argument. If the expression is true, then the assert function does nothing. But if the expression is false, the assert function prints a useful message and halts the program. You can read about the full details of the assert function in Section 1.1 of the text.

## Advantages of Using Preconditions and Postconditions

- Succinctly describes the behavior of a function...
- ... without cluttering up your thinking with details of how the function works.
- At a later point, you may reimplement the function in a new way ...
- ... but programs (which only depend on the precondition/postcondition) will still work with no changes.

Here are the primary advantages to using a method such as preconditions/postconditions to specify what a function accomplishes without giving details of how the function works.

One of the important advantages has to do with reimplementations. Often a programmer will think of a better method to accomplish some computation. If the computation is part of a function that includes a precondition/postcondition pair, then the function can be rewritten and the new, better function used instead. Any program which uses the function (and which only depends on the precondition/postcondition contract) can use the new improved function with no other changes.

# Summary

**Precondition**

- The programmer who calls a function ensures that the precondition is valid.
- The programmer who writes a function can bank on the precondition being true when the function begins execution.

**Postcondition**

- The programmer who writes a function ensures that the postcondition is true when the function finishes executing.

THE END