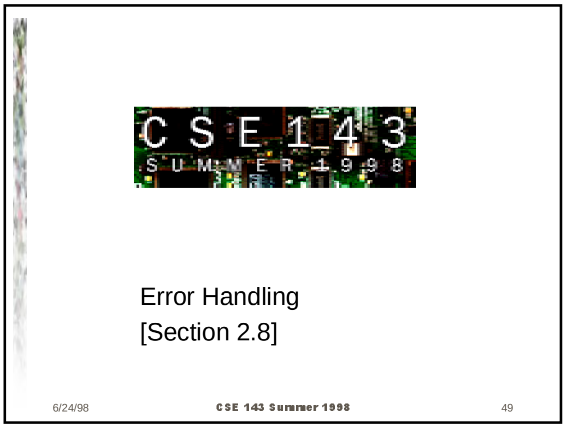
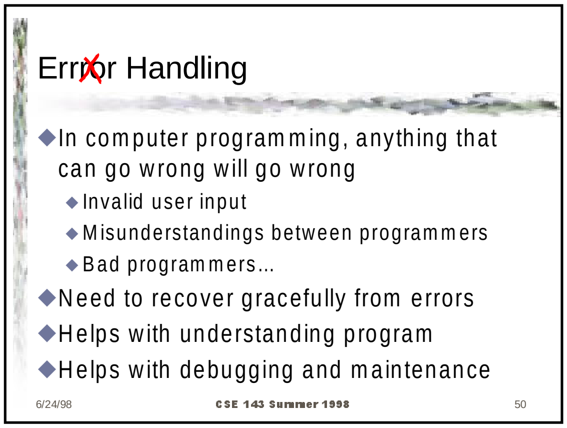


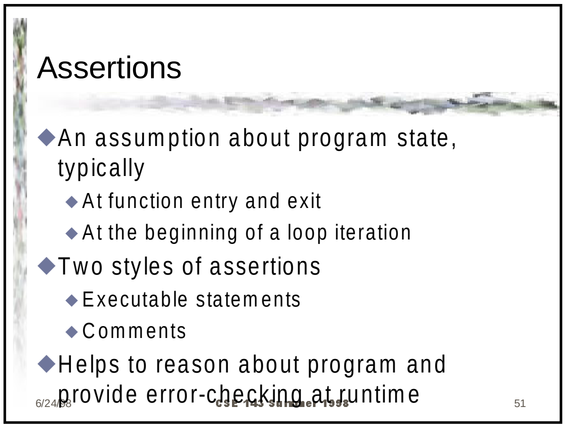
CSE 143
SUMMER 1998

Error Handling
[Section 2.8]



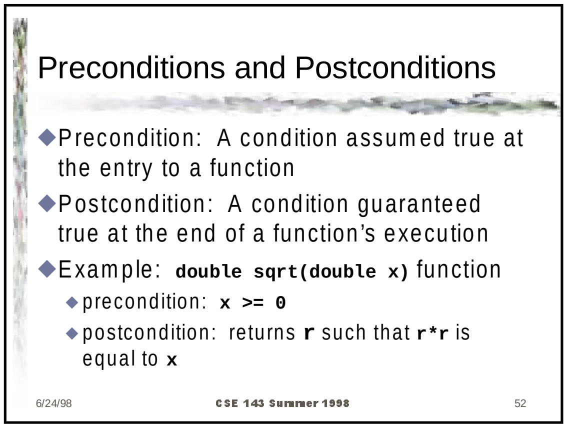
Err~~r~~or Handling

- In computer programming, anything that can go wrong will go wrong
  - Invalid user input
  - Misunderstandings between programmers
  - Bad programmers…
- Need to recover gracefully from errors
- Helps with understanding program
- Helps with debugging and maintenance



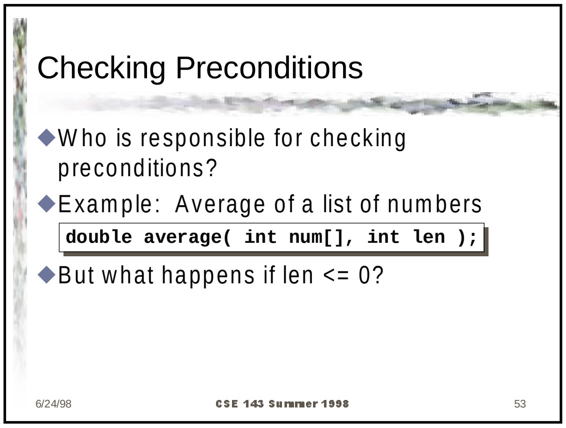
Assertions

- An assumption about program state, typically
  - At function entry and exit
  - At the beginning of a loop iteration
- Two styles of assertions
  - Executable statements
  - Comments
- Helps to reason about program and provide error-checking at runtime



Preconditions and Postconditions

- Precondition: A condition assumed true at the entry to a function
- Postcondition: A condition guaranteed true at the end of a function's execution
- Example: **`double sqrt(double x)`** function
  - precondition: **`x >= 0`**
  - postcondition: returns **`r`** such that **`r*r`** is equal to **`x`**

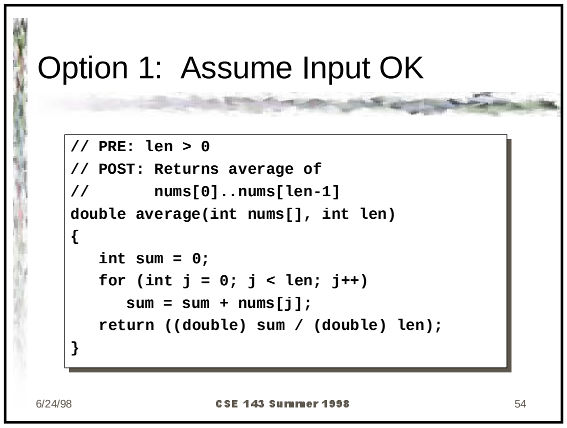


Checking Preconditions

- Who is responsible for checking preconditions?
- Example: Average of a list of numbers

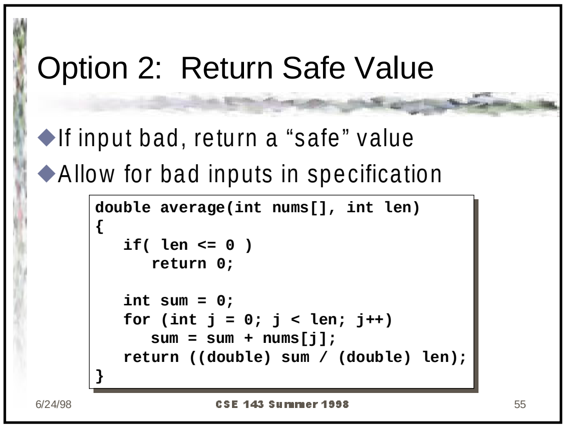
```
double average( int num[], int len );
```

- But what happens if len <= 0?



Option 1: Assume Input OK

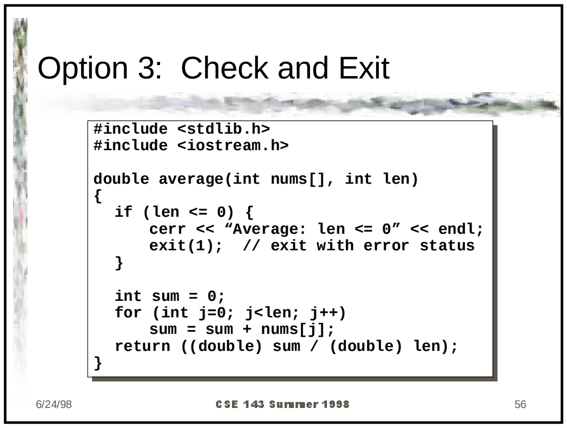
```
// PRE: len > 0
// POST: Returns average of
//       nums[0]..nums[len-1]
double average(int nums[], int len)
{
   int sum = 0;
   for (int j = 0; j < len; j++)
      sum = sum + nums[j];
   return ((double) sum / (double) len);
}
```



Option 2: Return Safe Value

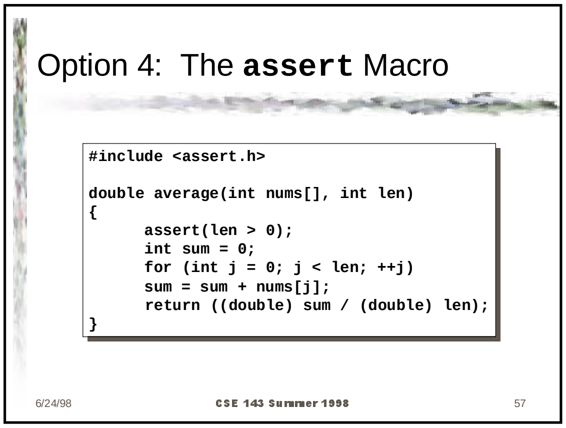
- If input bad, return a “safe” value
- Allow for bad inputs in specification

```
double average(int nums[], int len)
{
   if( len <= 0 )
      return 0;

   int sum = 0;
   for (int j = 0; j < len; j++)
      sum = sum + nums[j];
   return ((double) sum / (double) len);
}
```



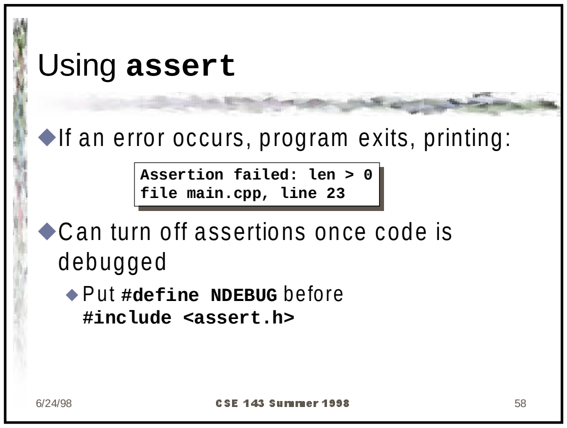
Option 3: Check and Exit

```
#include <stdlib.h>
#include <iostream.h>

double average(int nums[], int len)
{
  if (len <= 0) {
      cerr << “Average: len <= 0” << endl;
      exit(1);  // exit with error status
  }

  int sum = 0;
  for (int j=0; j<len; j++)
      sum = sum + nums[j];
  return ((double) sum / (double) len);
}
```



Option 4: The **assert** Macro

```
#include <assert.h>

double average(int nums[], int len)
{
      assert(len > 0);
      int sum = 0;
      for (int j = 0; j < len; ++j)
      sum = sum + nums[j];
      return ((double) sum / (double) len);
}
```

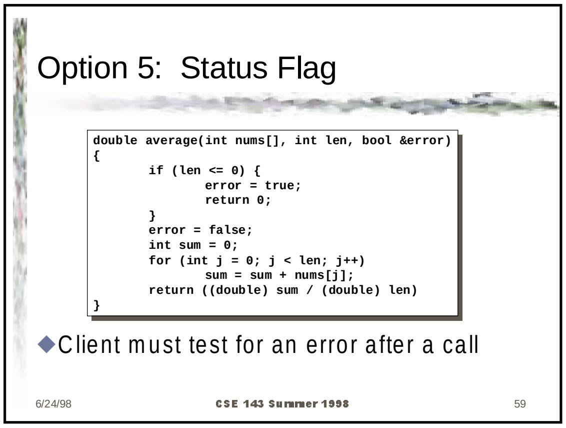


Using **assert**

- If an error occurs, program exits, printing:

```
Assertion failed: len > 0
file main.cpp, line 23
```

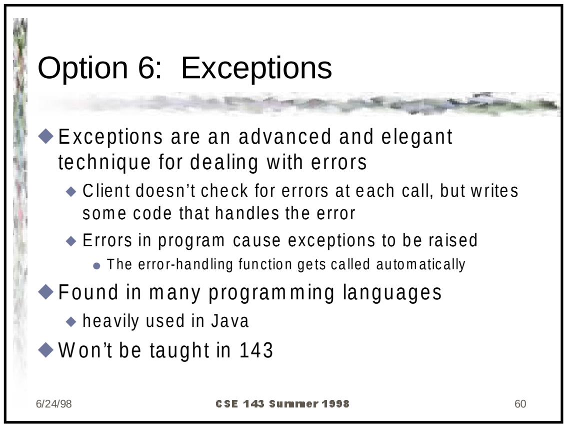
- Can turn off assertions once code is debugged
  - Put **#define NDEBUG** before **#include <assert.h>**



Option 5: Status Flag

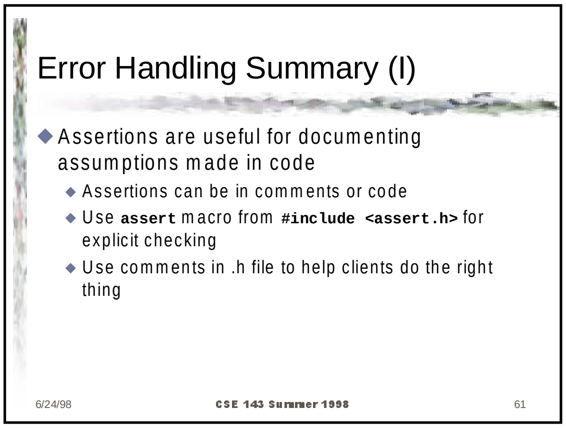
```
double average(int nums[], int len, bool &error)
{
      if (len <= 0) {
            error = true;
            return 0;
      }
      error = false;
      int sum = 0;
      for (int j = 0; j < len; j++)
            sum = sum + nums[j];
      return ((double) sum / (double) len)
}
```

- Client must test for an error after a call



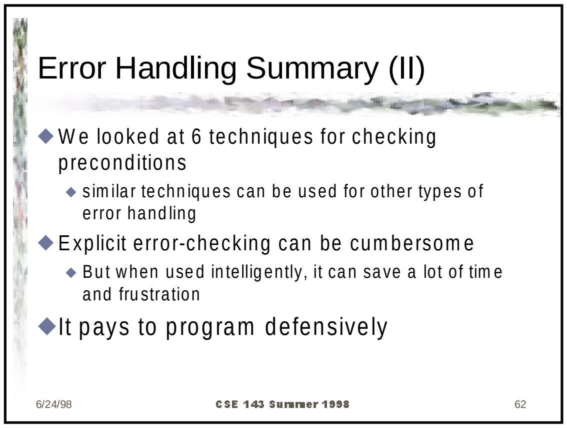
Option 6: Exceptions

- Exceptions are an advanced and elegant technique for dealing with errors
  - Client doesn’t check for errors at each call, but writes some code that handles the error
  - Errors in program cause exceptions to be raised
    - The error-handling function gets called automatically
- Found in many programming languages
  - heavily used in Java
- Won’t be taught in 143



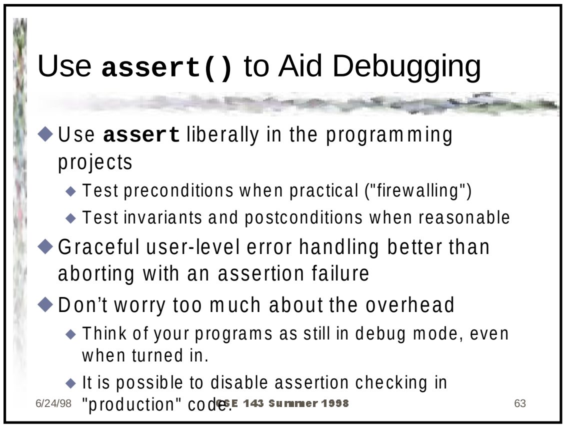
Error Handling Summary (I)

- Assertions are useful for documenting assumptions made in code
  - Assertions can be in comments or code
  - Use **`assert`** macro from **`#include <assert.h>`** for explicit checking
  - Use comments in .h file to help clients do the right thing



Error Handling Summary (II)

- We looked at 6 techniques for checking preconditions
  - similar techniques can be used for other types of error handling
- Explicit error-checking can be cumbersome
  - But when used intelligently, it can save a lot of time and frustration
- It pays to program defensively



Use **`assert()`** to Aid Debugging

- Use **`assert`** liberally in the programming projects
  - Test preconditions when practical ("firewalling")
  - Test invariants and postconditions when reasonable
- Graceful user-level error handling better than aborting with an assertion failure
- Don’t worry too much about the overhead
  - Think of your programs as still in debug mode, even when turned in.
  - It is possible to disable assertion checking in "production" code.