Checking input: iostream and cstdlib

- we usually need to perform error checking on user input, notifying the user and taking corrective actions as needed
- user might have entered the wrong type of data (e.g. entered a text string when a number is desired)
- cin and scanf each provide support for checking this and for clearing the invalid entry from the input stream
- user might have entered the write type of data but an “incorrect” value (e.g. entering a negative number when a positive is needed)
- this does not require any adjustment of the input stream (since the input operation itself was successful)

cin: checking for failed reads

- if cin fails in its attempt to read (e.g. we try to cin an int to x but the user enters “blah”) we can detect it as follows:

```
cin >> x;
if (cin.fail()) {
   cout << “you did not enter an integer!” << endl;
} else {
   cout << “you entered integer “ << x << endl;
   // here we could use x normally
}
```

cin: clearing input stream

- if cin fails (as on the previous slide) then the “garbage” input is still sitting in the input stream, we need to clear it inside our “fail” case

```
if (cin.fail()) {
   cin.clear();  // turns off cin's error flag
   // now let's throw away the line of input, up to 80 chars
   cin.ignore(80,'\n');
}
```

example: int in specific range

```
// ask user for an int in the range 0 to 100, do error checking
cout << “Enter an integer from 0 to 100” << endl;
cin >> userVal;
if (cin.fail()) {
   cout << “That was not an integer, clearing input” << endl;
   cin.clear();
   cin.ignore(80, '\n');
} else if ((userVal < 0) || (userVal > 100)) {
   cout << “That was not in the range 0..100” << endl;
   // note that we do NOT clear the input here,
   //      since cin did successfully read in the value
} else
   cout << x << “ is indeed in the range 0..100” << endl;
   // now do whatever with x
}
```

scanf: error check

- scanf does not have a “fail” check like cin, but it does return a value telling us how many values it successfully read into variables
- example 1:

```
count = scanf(“%d”, &x);
```

  - count will be 1 if scanf read an int to x, 0 otherwise
- example 2:

```
count = scanf(“%d %d”, &x, &y);
```

  - scanf will be 2 if both values were read successfully, 1 if just x was read successfully, 0 otherwise

Example: checking failed read

```
// ask user to enter an int, check if the read worked
printf(“Please enter an integer”);
int x, count;
count = scanf(“%d”, &x);
if (count == 0) {
   printf(“Error: that was not an integer\n”);
} else {
   printf(“You entered integer %d\n”, x);
}
```

scanf: clearing input

- if the user enters an invalid data type that prevents scanf from reading then we need to clear that from the input
- we do this by reading/discarding one “word” of text
- a special format string allows us to do this, “%*s”, e.g.

```
scanf(“%*s”); // reads and discards one word of input
```

example: input in specific range

```
printf(“Enter an integer from 0 to 100\n”);
int userVal, inputCount;
inputCount = scanf(“%d”, &userVal);

if (inputCount == 0) {
   printf(“That was not an integer, discarding input\n”);
   scanf(“%*s);
}

else if ((userVal < 0) || (userVal > 100)) {
   printf(“Your value, %d, was outside range 0-100\n”, userVal);
   // note we do NOT do a scanf *s here, scanf did read in the int
}

else {
   printf(“Correct, %d is in the range 0-100\n”, userVal);
   // and we could then use the value normally
}
```

Read strings to avoid input failure

- an alternative approach is to read the user input as text (rather than as an int, float, etc) so that the input operation always succeeds
- the program can then check if the entered text has the desired format, and if so it can attempt to convert to the desired data type
- (we'll visit this in detail when we get to character arrays and when we look at the string class)