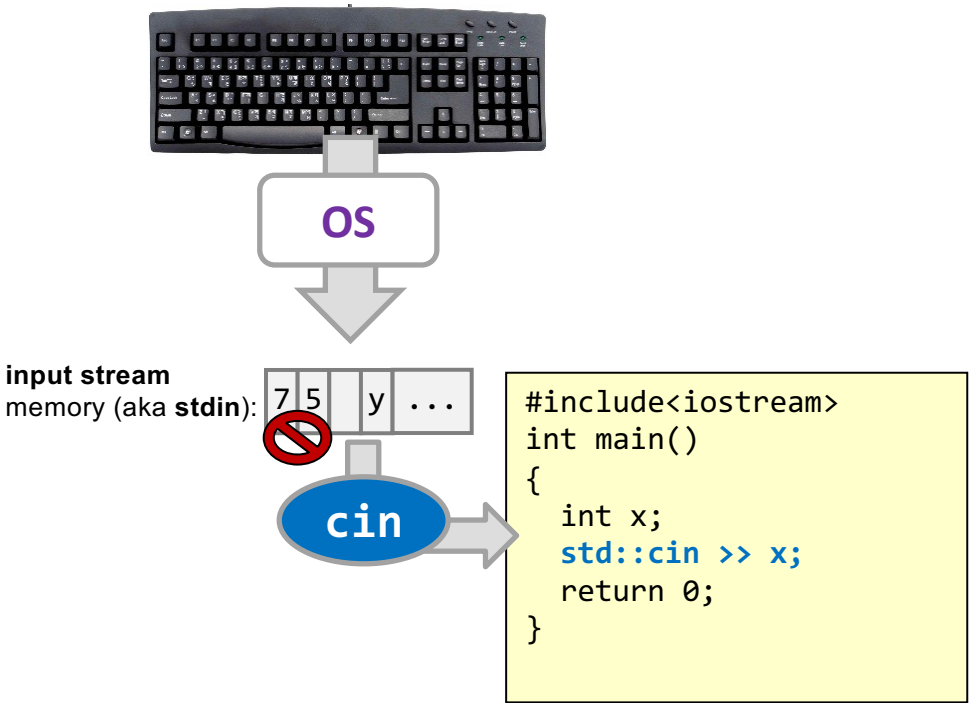
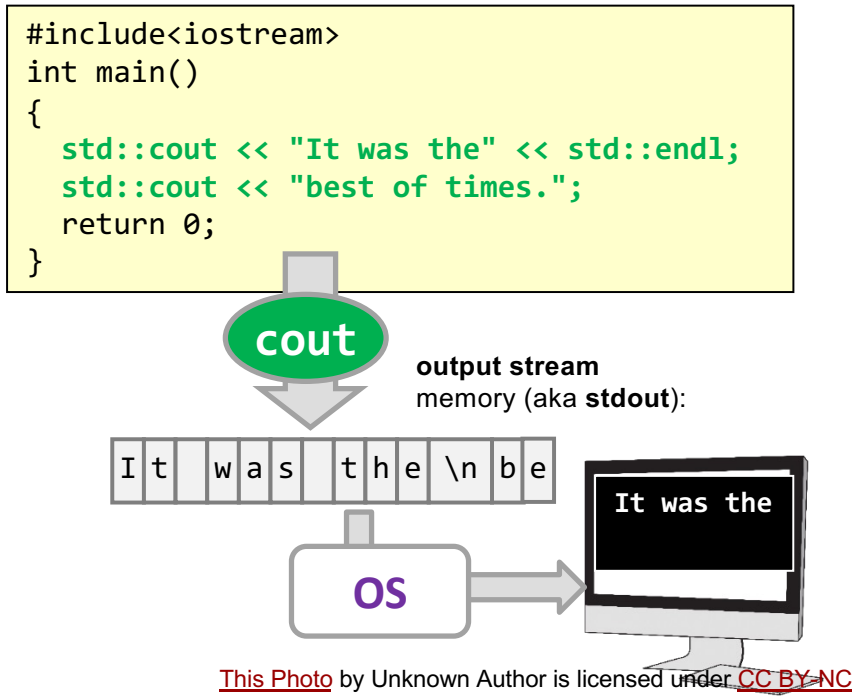
CS 103 Unit 5f – Error Checking, Parsing and Stringstreams

CSCI 103L Teaching Team

Recall: I/O Streams

- C++ and the OS use the notion of **streams** to temporarily store (aka buffer) data to be input or output and then uses the `cin` and `cout` **objects** (from the `<iostream>` library) to access those streams.
- The OS name for these streams are: **stdin** (keyboard) and **stdout** (the terminal)
- `cin` pulls data from **stdin** and `cout` places data in the **stdout** stream

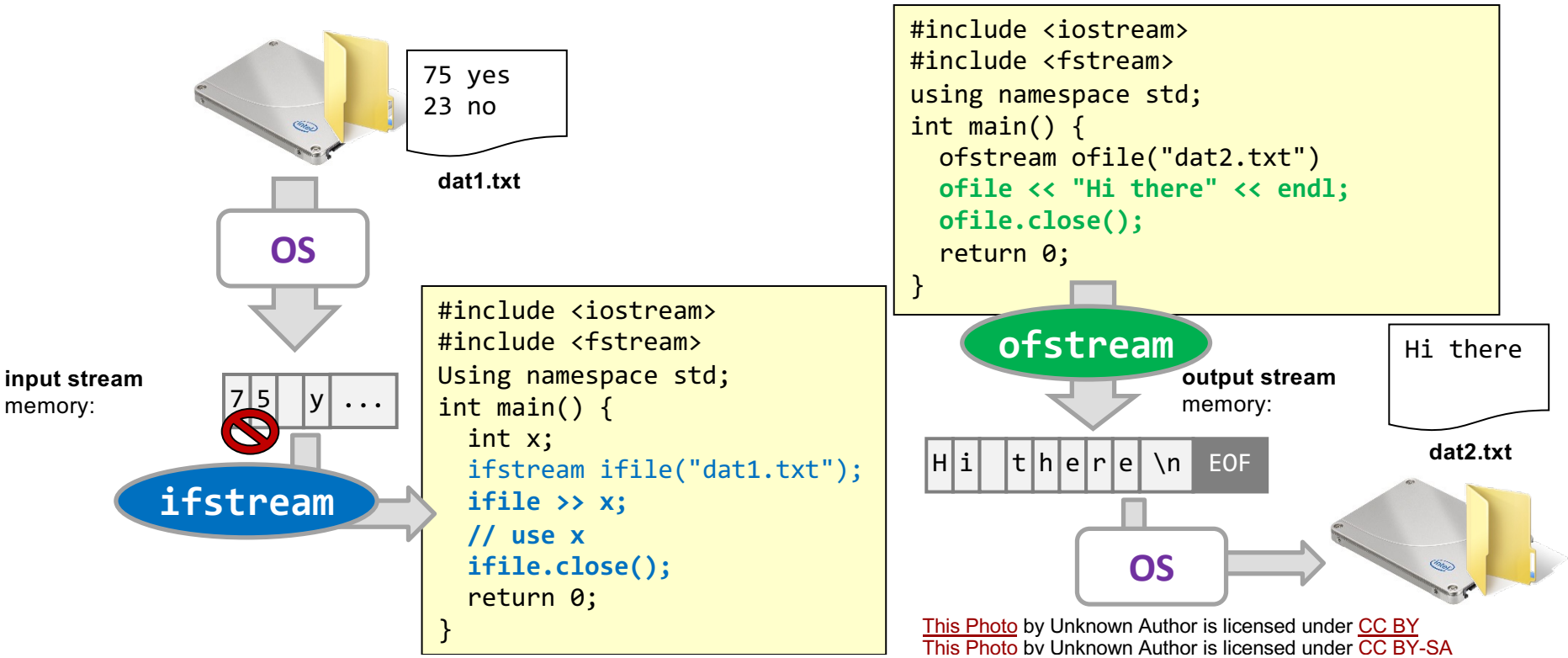




This Photo by Unknown Author is licensed under CC BY-NC

File Streams

- C++ leverages the SAME interface that cin and cout provide to:
  - Read data IN from a file (like `cin`, but data comes from a **file** not the keyboard) and
  - Write data OUT to a file (like `cout`, but data goes to a **file** not the terminal).
- The counterpart to `cin` is an `ifstream` object
- The counterpart to `cout` is an `ofstream` object

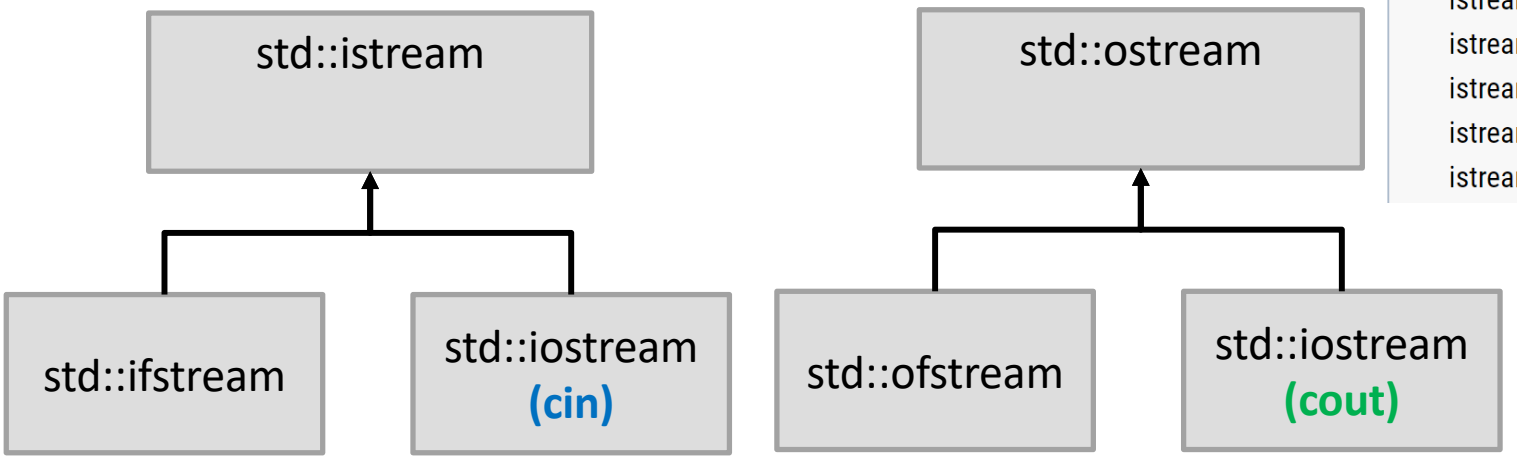


This Photo by Unknown Author is licensed under CC BY
This Photo by Unknown Author is licensed under CC BY-SA

Relationship of Input & Output Streams

- Recall, file streams behaved the same as cin and cout (which are iostreams)!
- Why is that?
- Because they are related through inheritance.
- Where are getline() and operator>> defined?
  - In the base class, std::istream

istream
- istream::~istream
- istream::istream
- **member classes**
  - istream::sentry
- **member functions**
  - istream::gcount
  - istream::get
  - istream::getline
  - istream::ignore
  - istream::operator>>
  - istream::peek
  - istream::putback
  - istream::read
  - istream::readsome
  - istream::seekg
  - istream::sync
  - istream::tellg
  - istream::unget



CHECKING FOR INPUT ERRORS AND THE END OF INPUT

Input Stream Error Checking

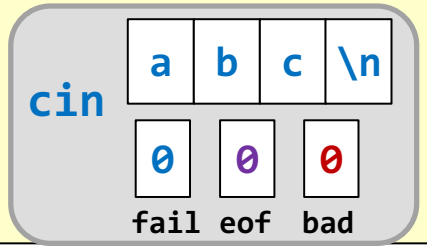
- We can check errors when **cin** receives unexpected data that can't be converted to the given type
- Use the function fail() member function (i.e. cin.fail()) which returns true if anything went wrong opening or reading data in from the file
- Internally, the istream class maintains 3 status bits:
  - Fail
  - EOF (end-of-file / end-of-stream)
  - Bad (ignore this one, it's rarely used)

```
#include <iostream>
using namespace std;

int main ()
{
  int x;
  cout << "Enter an int: " << endl;

  cin >> x;  // What if the user enters:
             //    "abc"

  // Check if we successfully read an int
  if(  cin.fail() ) {
    cout << "Error:  I said enter an int!";
    cout << " Now I must exit!" << endl;
    return 1;
  }
  cout << "You did it! You entered an int";
  cout << " with value: " << x << endl;
  return 0;
 }
```



Understanding Input Streams

User enters value "512 123" at the prompt

```
int x=0;

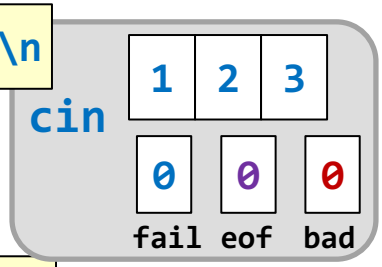
cout << "Enter x: ";

cin >> x;
```

X = 0 stdin =

X = 0 cin = 5 1 2 ␣ 1 2 3 \n

X = 512 cin = ␣ 1 2 3 \n



`cin.fail()` is `false`

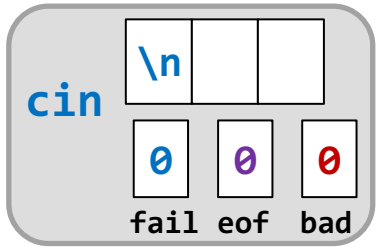
```
int y = 0;

cout << "Enter y: ";

cin >> y;
```

Y = 0 cin = ␣ 1 2 3 \n

Y = 123 cin = \n



`cin.fail()` is `false`

Take Care Mixing >> and Getline

```
int x=0;

cout << "Enter x: ";

cin >> x;

char y[80];

cout << "Enter y: ";

cin.getline(y, 80);
```

User enters the following:

```
23
Hi :)
```

X = 0 cin = [ ]

X = 0 cin = [2][3][\n][H][i][ ][:][)][\n]

X = 23 cin = [\n][H][i][ ][:][)][\n]

`cin.fail()` is `false`

Y = [ ][…][ ] cin = [\n][H][i][ ][:][)][\n]

Y = [\0][…][ ] cin = [H][i][ ][:][)][\n]

cin.fail() is false

Warning: >> stopped before the '\n' and so getline will read that in and stop.

Solution (Use 2 Getlines?)

User enters the following:

```
23
Hi :)
```

```
int x=0;
```

X = 0 cin = ( )

```
cout << "Enter x: ";
```

X = 0 cin = 2 3 \n H i ␣ :) \n

```
cin >> x;
```

X = 23 cin = \n H i ␣ :) \n

```
char y[80];
```

cin.fail() is false

```
cout << "Enter y: ";
```

Y = ␣ … ␣ cin = \n H i ␣ :) \n

```
cin.getline(y, 80);
```

Y = \0 … ␣ cin = H i ␣ :) \n

```
cin.getline(y, 80);
```

Y = H i ␣ :) \0 cin = ( )

cin.fail() is false

1st getline() gets the rest of what's on the line with 23, second gets the next line.

Alternate Solution (Use std::ws)

Use `std::ws` to advance to the next NON-WHITESPACE character in the stream.

```
int x=0;

cout << "Enter x: ";

cin >> x;

char y[80];

cout << "Enter y: ";

cin >> ws;

cin.getline(y, 80);
```

User enters the following:

```
23
Hi :)
```

X = 0 cin = ( )

X = 0 cin = 2 3 \n ␣ H i ␣ :) \n

X = 23 cin = \n ␣ H i ␣ :) \n

Before `cin >> ws`

Y = ␣ … ␣ cin = \n ␣ H i ␣ :) \n

After `cin >> ws`

Y = \0 … ␣ cin = H i ␣ :) \n

Y = H i ␣ :) \0 cin = ( )

cin.fail() is false

When Does It Fail

- **User enters value "23abc" at 1st prompt, 2nd prompt fails**

```
int x=0;

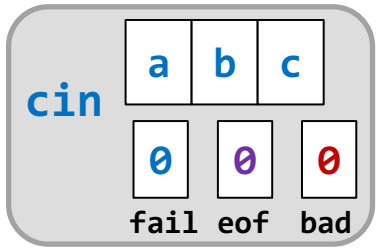
cout << "Enter x: ";

cin >> x;
```

X = 0 cin =

X = 0 cin = 2 3 a b c \n

X = 23 cin = a b c \n



`cin.fail()` is `false`

```
int y = 0;

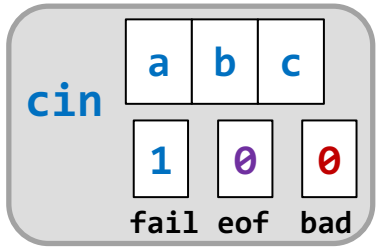
cout << "Enter y: ";

cin >> y;
```

Y = 0 cin = a b c \n

Y = 0 cin = a b c \n

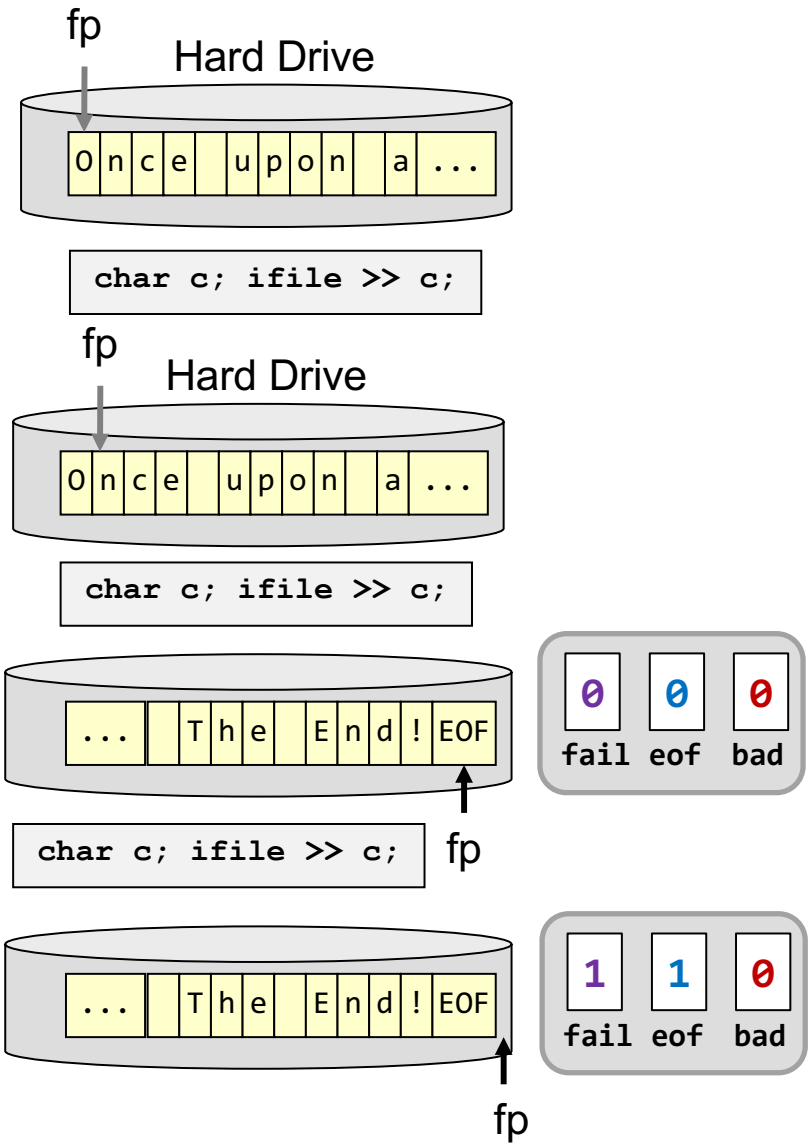
Y = ?? cin = a b c \n



`cin.fail()` is `true`

File Streams and EOF

- Your `ifstream` object implicitly keeps track of where you are in the file using a "file pointer" (fp)
- EOF (end-of-file) or other error means no more data can be read. Use the `fail()` function to ensure the file is okay for reading/writing
- Input streams also allow you to check if you've read the EOF character by calling an `eof()` function, but fail will be set when eof is and so it's easier to just use `fail()`



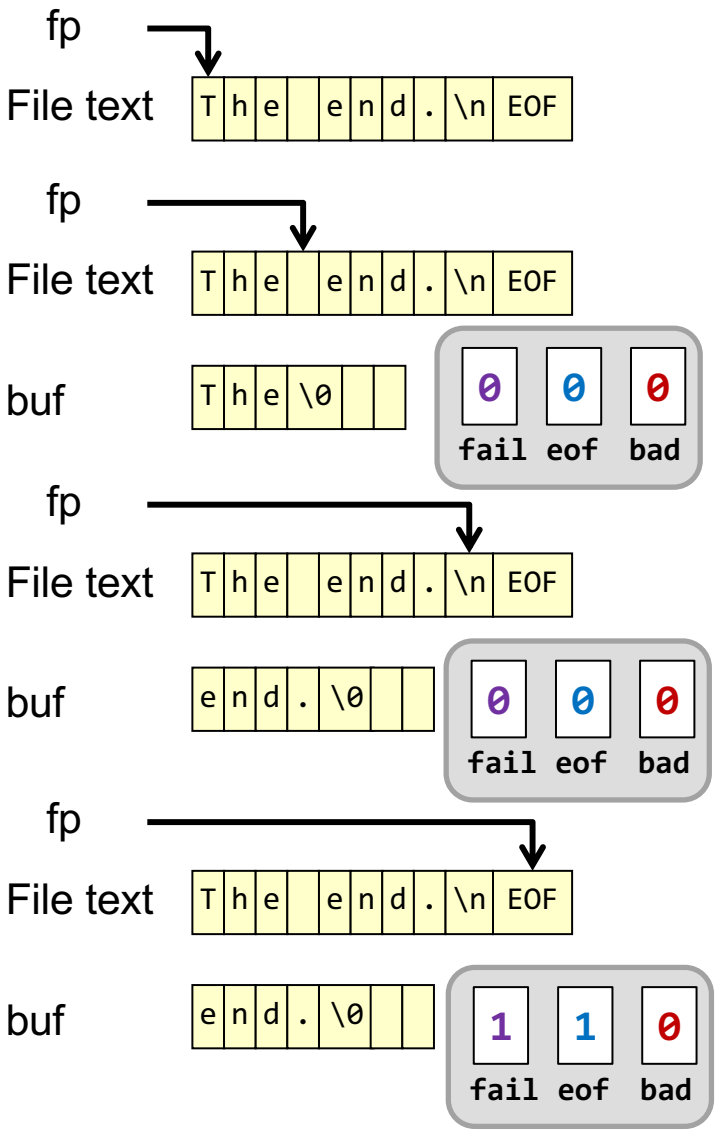
When Does It Fail

```
char buf[40];
ifstream inf(argv[1]);
```

```
inf >> buf;
```

```
inf >> buf;
```

```
inf >> buf;
```



For filestreams & soon stringstreams the stream does NOT fail until you read ***PAST*** the EOF.

Reading something that stops ON or AT the EOF will NOT cause fail() or eof() to return true

Pattern for File I/O or Streams

- Step 1: **Try** to read data (>> or getline)
- Step 2: **Check** if you succeeded or failed
- Step 3: Only **use** the data read from step 1 if you succeeded

- If you read from a stream and then blindly use the data you read without checking for failure, you will likely get one **BOGUS** data value at the end!

Getting the order right

- Be sure you CHECK whether the input failed **before** you USE the result!
  - See top example
- If you don't CHECK and the input fails, you will use a garbage value
  - See bottom example

3 Step Process
1. Try
2. Check
3. Use

Correct Approach

```
int main ()
{
  int x, sum = 0;
  cout << "Enter an int: " << endl;
  cin >> x;

  // Check if we successfully read an int
  while( ! cin.fail() ) {
     sum += x;
     cin >> x; // What if the user enters abc
  }
  cout << "sum = " << sum << endl;
  return 0;
}
```

```
int main ()
{
  int x, sum = 0;
  while( ! cin.fail() ) {
    cout << "Enter an int: " << endl;
    cin >> x;  // What if the user enters: abc
    sum += x;  // May use BAD value
  }
  cout << "sum = " << sum << endl;
  return 0;
}
```

Incorrect Approach

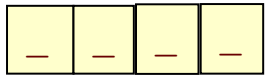
Which Option Works?

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
 vector<int> nums;
 ifstream ifile("data.txt");
 int x;
 while( !ifile.fail() ){
   ifile >> x;
   nums.push_back(x);
 }
 ...
}
```

data.txt

```
7 8 EOF
```

nums



```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
 vector<int> nums;
 ifstream ifile("data.txt");
 int x;
 while( true ){
   ifile >> x;
   if(ifile.fail()) break;
   nums.push_back(x);
 }
 ...
}
```

Remember:
3 Step Process
1. Try
2. Check
3. Use

Goal is to read all integers from the file into a vector. Which of these works?

A Tangent: Implicit Type Conversion

- Would the following if condition make sense?
- No! If statements want Boolean variables, not objects
- But let's go back to "operator overloading". C++ provides **type-conversion operators**.
  - **operator <type>()**
- Student::operator bool()
  - Code to specify how to convert a Student to a bool
- Student::operator int()
  - Code to specify how to convert a Student to an int

```
class Student {
  private:  int id; double gpa;
};
int main()
{
  Student s1;
  if(s1){  cout << "Hi" << endl; }
  return 0;
}
```

```
class Student {
  private:
   int id; double gpa;
  public:
   operator bool() { return gpa >= 2.0;}
   operator int() { return id; }
};

Student s1;
if(s1){    // calls operator bool() and
  int x = s1; // calls operator int()
}
...
```

Getting All The Inputs

- The `istream` class defines an operator bool() which returns true if the stream didn't fail
  - `istream::operator bool() { return !fail(); }`
- So we can combine the TRY and CHECK into a single if/while statement

```
if/while(cin >> val)
  { /* process val */  }
```

- In this approach cin does two things
  - It does try to extract input into the variable 'val'
  - It returns 'true' if it successfully got input, 'false' otherwise
- Keeps grabbing values one at a time until the user types Ctrl-D

```
#include <iostream>
using namespace std;
int main()
{
  int val, sum = 0;
  // reads until user hits Ctrl-D
  // which is known as End-of-File(EOF)
  cout << "Enter a sequence of integers";
  cout << " or Ctrl-D (EOF) to quit: ";
  cout << endl;

  while(cin >> val){
    sum += val;  // we know val is good

  }
  cout << "Sum is " << sum << endl;
  return 0;
}
```

A More Compact Way

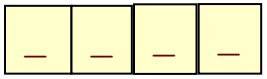
```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
 vector<int> nums;
 ifstream ifile("data.txt");
 int x;
 while( !ifile.fail() ){
   ifile >> x;
   nums.push_back(x);
 }
 ...
}
```



data.txt

```
7 8 EOF
```

nums



```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
 vector<int> nums;
 ifstream ifile("data.txt");
 int x;
 while( true ){
   ifile >> x;
   if(ifile.fail()) break;
   nums.push_back(x);
 }
 ...
}
```



```
int x;
while( ifile >> x ){
  nums.push_back(x);
}
...
```



Calling >> on an input stream will essentially return a Boolean:

- true = success
- false = failure

Getline() for char* (C-Strings)

- We can get a whole line of text (including spaces) with getline()
  - **istream& istream::getline(char *buf, int bufsize);**
  - Reads through a newline OR a max of bufsize-1 characters and then adds the null character to the end of the given character array
- But **getline()** uses **char* (C-Strings)**… what if we want to use C++ strings???

```
#include <iostream>
#include <fstream>
using namespace std;

int main ()

{

  char myline[100]; int i = 1;

  ifstream ifile ("input.txt");

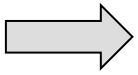
  if(  ifile.fail() ){  // can't open?
    return 1;
  }


  while ( ifile.getline(myline, 100) ) {
   cout << i++ << ": " << myline << endl;
  }

  ifile.close();
  return 0;
}
```

input.txt

```
The fox jumped over the log.
The bear ate some honey.
The CS student solved a hard problem.
```



```
1: The fox jumped over the log.
2: The bear ate some honey.
3: The CS student solved a hard problem.
```

Getline() for std::string (C++ strings)

- C++ string library (#include <string> defines a global function (not a member of istream) that can read a line of text into a C++ string
- Prototype: **istream& getline(istream &is, string &str, char delim='\n');**
  - `is` = any input stream (ifstream, cin), etc.)
  - `str` = A C++ string that it will fill in with text
  - `delim` = A char to stop on (by default it is '\n') which is why it's called getline
  - Returns the updated istream (the 'is' object you passed in as the 1st arg)
- The text from the input stream will be read up through the first occurrence of 'delim' (defaults to '\n') and placed into str. The delimiter will be stripped from the end of str and the input stream will be pointing at the first character after 'delim'.

```
ifstream myfile(argv[1]);
string myline;
// Not a member function
myfile.getline( myline );  // doesn't work

// global scope function...correct
getline(myfile, myline);
```

```
int line_no = 0;
ifstream myfile(argv[1]);
string myline;

while ( getline( myfile, myline ) ) {
  cout << "Line: " << myline << endl;
}
```

STRINGSTREAMS

Introducing…Stringstreams

- I/O streams
  - Keyboard (`cin`) and terminal (`cout`)
- File streams – Contents of file are the stream of data
  - #include <fstream> and #include <iostream>
  - `ifstream` and `ofstream` objects
- Stringstreams – Contents of a string are the stream of data
  - #include <sstream> and #include <iostream>
  - `stringstream` objects

C++ String Stream

- If streams are just sequences of characters, aren't strings themselves like a stream?
  - The <sstream> library lets you treat C++ string objects like they were streams
- Why would you want to treat a string as a stream?
  - Parse out (break up) the pieces of a string
  - Buffer up output for later display
  - Data type conversions
- Very useful in conjunction with string's `getline(...)`

C++ Stringstream: Application 1

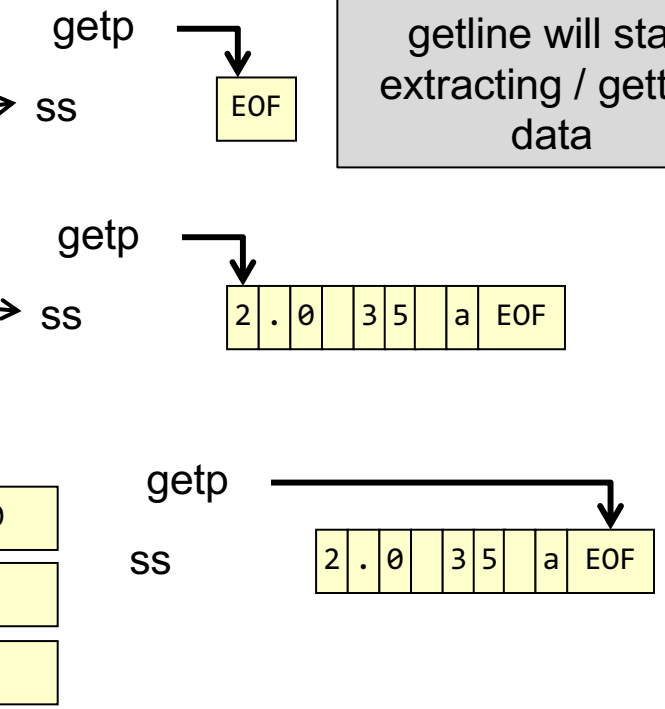
- Can parse (split) a string of many values into separate variables

```
#include <sstream>
using namespace std;
int main()
{
 stringstream ss;
 ss << "2.0 35 a";

 double x, int y; char z;
 ss >> x >> y >> z;

 return 0;
}
```

sstream_test3.cpp



Getp stands for "get pointer" and is the placeholder where the next >> or getline will start extracting / getting data

C++ Stringstream: Application 2

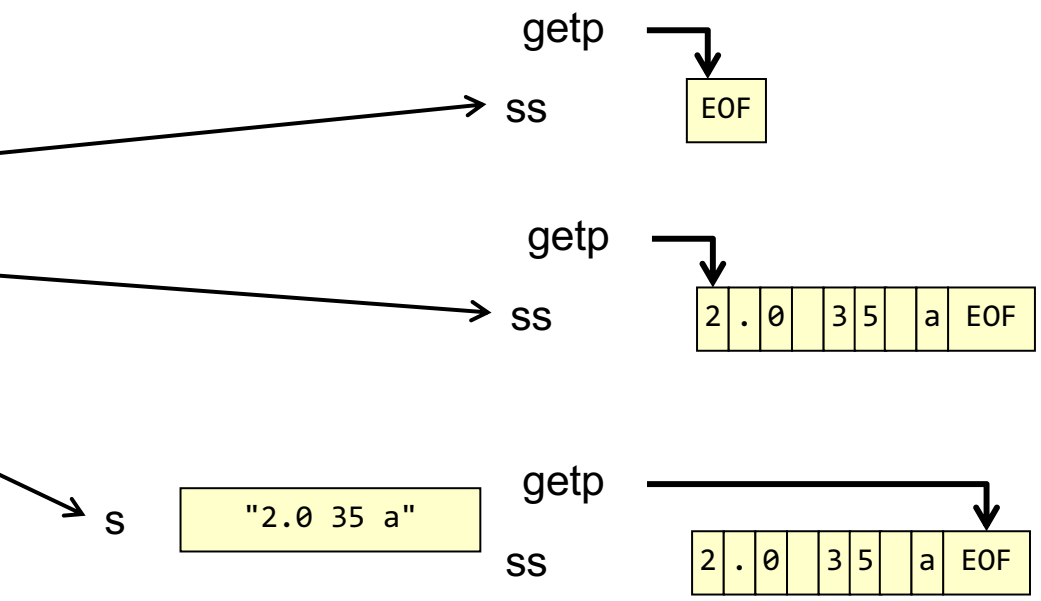
- Use the `.str()` member function to create a large string from many values (i.e. return a string with the contents of whatever is in the stream)
  - Alternative is to use to_string and the string + operator:
    `to_string(2.0) + " " + to_string(35) + " a";`

```
#include<sstream>
using namespace std;
int main()
{
 stringstream ss;
 ss << 2.0 << " " << 35;
 ss << " " << 'a';

 string s = ss.str();

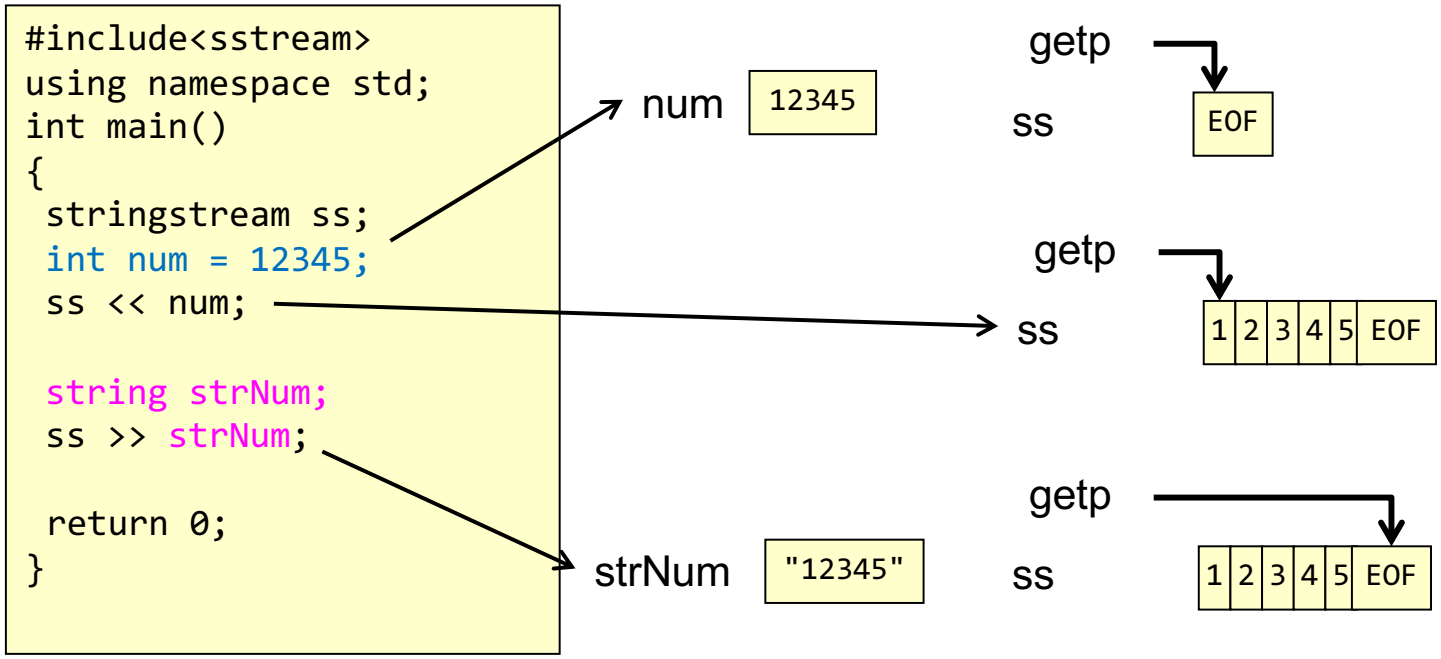
 return 0;
}
```

sstream_test4.cpp



C++ Stringstream: Application 3a

- Can be used as an alternative to `to_string()`
- Use << and >> to convert numbers into strings (i.e. 12345 => "12345")
  - Same result as `to_string(12345)`

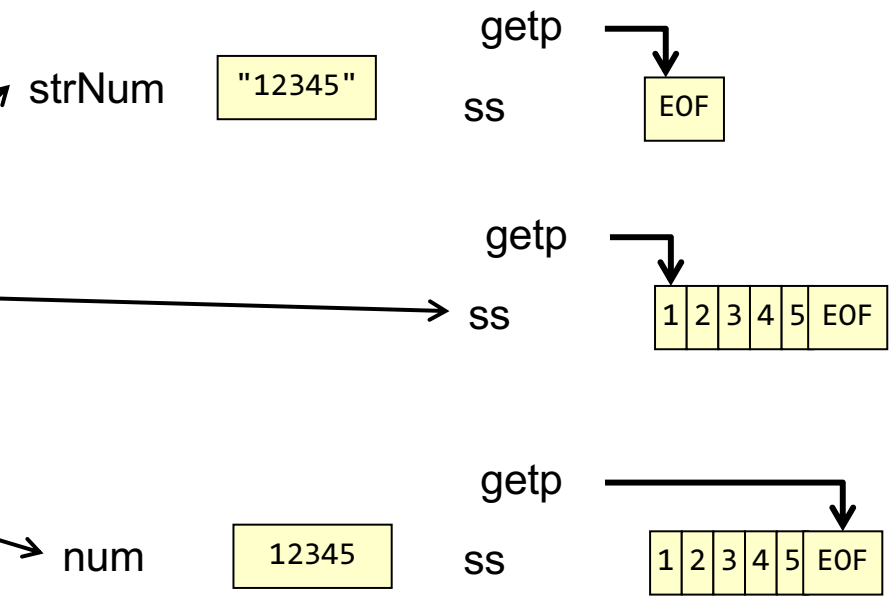


C++ Stringstream: Application 3b

- Can be used as an alternative to `stoi() or stod()`
- Use << and >> to convert strings into numbers (i.e. "12345" => 12345)
  - Same result as `stoi("12345")`

```
#include<sstream>
using namespace std;
int main()
{
 stringstream ss;
 string strNum = "12345";
 ss << strNum;

 int num;
 ss >> num;
 return 0;
}
```



C++ Stringstream Reuse

- Beware of re-using the same stringstream object for multiple conversions. It can be weird.
  - Make sure you clear it out between uses and re-init with an empty string
- Or just make a new stringstream each time

```
stringstream ss;

//do something with ss

ss.clear();
ss.str("");
// now you can reuse ss
```

Option 1: Reuse

```
stringstream ss1;

//do something with ss1



// Just declare another stream
stringstream ss2;
// do something with ss2
```

Option 2: Use new stringstream

Exercise

- What's in each variable after execution?
  - text
  - num
  - val

```
string text;
int num;
double val;

stringstream ss("Hello 103 2.0");
ss >> text >> num >> val;
```

Exercises

- In class exercises
  - Stringstream_in
  - Stringstream_out
  - Date

Exercises

- Use the following in-class exercises to practice the prior material and illustrate the next few slides
  - wordcount-all
  - wordcount
  - wordcount_parens

getline() and stringstreams

- Imagine a file has a certain format where you know related data is on a single line of text but aren't sure how many data items will be on that line
- Can we use >>?
  - No it doesn't differentiate between different whitespace (i.e. a ' ' and a '\n' look the same to >> and it will skip over them)
- We can use getline() to get the whole line, then a stringstream with >> to parse out the pieces

```
int num_lines = 0;
int total_words = 0;

ifstream myfile(argv[1]);

string myline;
while( getline(myfile, myline) ){

   stringstream ss(myline);

   string word;
   while( ss >> word )
     {  total_words++; }
   num_lines++;
}

double avg =
   (double) total_words / num_lines;

cout << "Avg. words per line: ";
cout << avg << endl;
```

```
it was a
good day in
CS 103
```

```
The fox jumped over the log.
The bear ate some honey.
The CS student solved a hard problem.
```

Using Delimeters

- Imagine a file has a certain format where you know related data is on a single line of text but aren't sure how many data items will be on that line
- Can we use >>?
  - No it doesn't differentiate between different whitespace (i.e. a ' ' and a '\n' look the same to >> and it will skip over them)
- We can use getline() to get the whole line, then a stringstream with >> to parse out the pieces

Text file:

```
garbage stuff (words I care about) junk
```

```
vector<string> mywords;

ifstream myfile(argv[1]);

string myline;

getline(myfile, myline, '(');
// gets "garbage stuff "
// and throws away '('

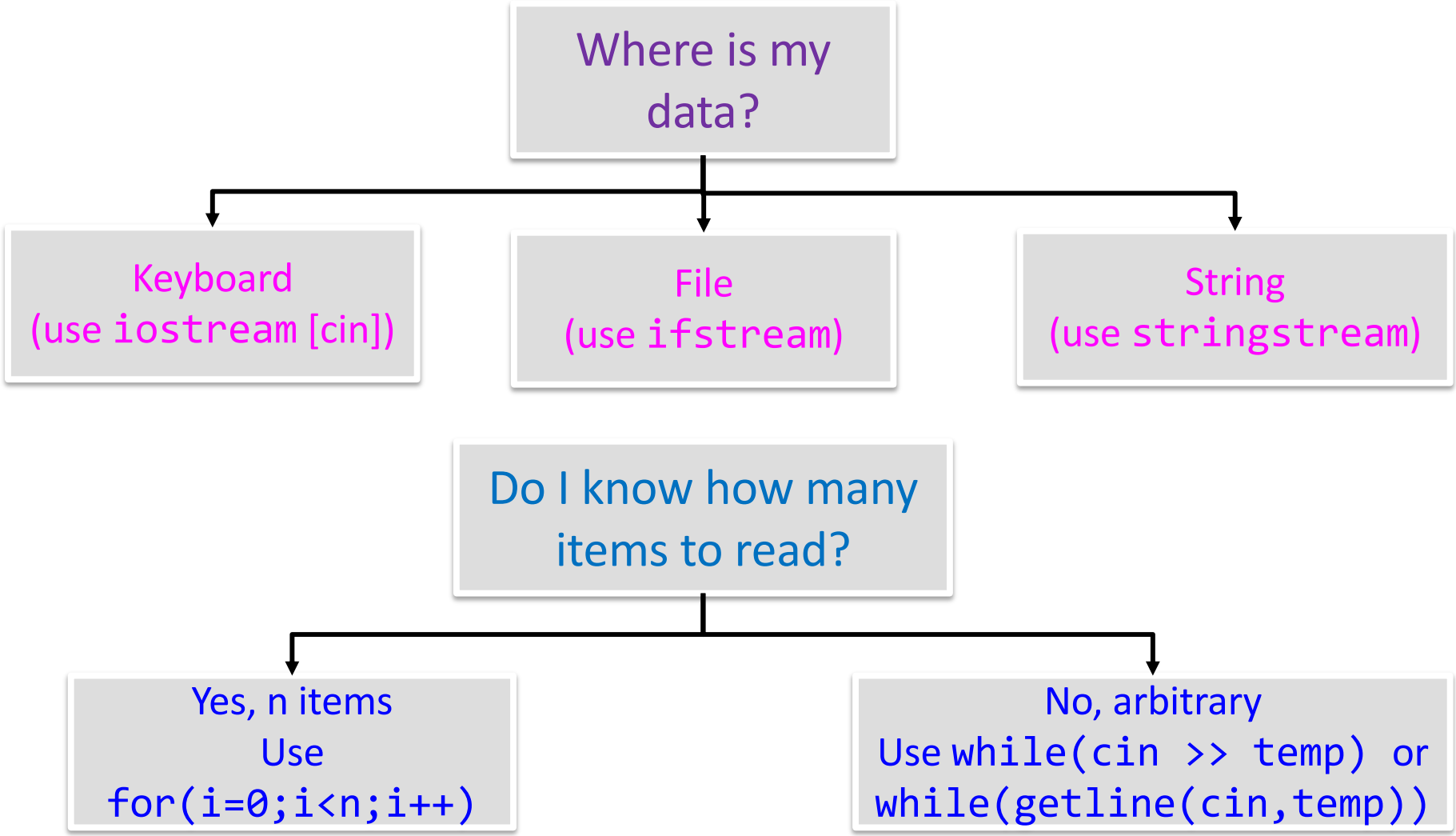
getline(myfile, myline, ')' );
// gets "words I care about"
// and throws away ')'`

stringstream ss(myline);
string word;
while( ss >> word ) {
  mywords.push_back(word);
}
```

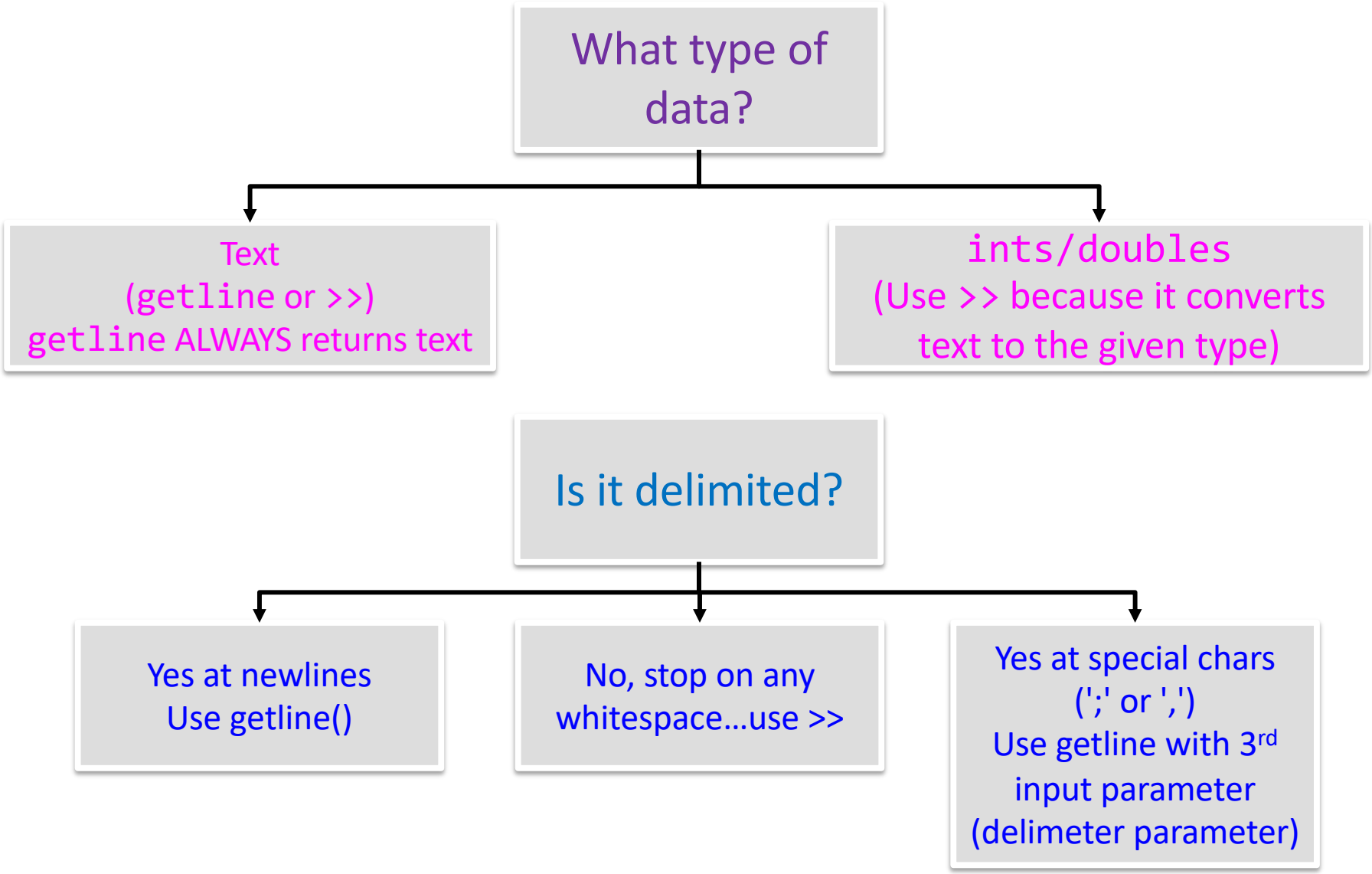
mywords

0	1	2	3
"words"	"I"	"care"	"about"

I/O Decision Tree (1)



getline or >>


What type of data?
Text (getline or >>) getline ALWAYS returns text
ints/doubles (Use >> because it converts text to the given type)
Is it delimited?
Yes at newlines Use getline()
No, stop on any whitespace…use >>
Yes at special chars (';' or ',') Use getline with 3rd input parameter (delimeter parameter)

Choosing an I/O Strategy

- Is my data delimited by particular characters?
  - Yes, stop on newlines: Use getline()
  - Yes, stop on other character: User getline() with optional 3rd character
  - No, Use >> to skip all whitespaces and convert to a different data type (int, double, etc.)
- If "yes" above, do I need to break data into smaller pieces (vs. just wanting one large string)
  - Yes, create a stringstream and extract using >> or getline()
  - No, just keep the string returned by getline()
- Is the number of items you need to read known as a constant or a variable read in earlier?
  - Yes, Use a loop and extract (>>) or getline() values placing them in array or vector
  - No, Loop while extraction doesn't fail placing them in vector

Remember: getline() always gives text/string. To convert to other types it is easiest to use >>

```
vector<string> mywords;

string myline = “garbage stuff (words I care about) junk”;
stringstream ss(myline);

getline(ss, myline, '(');
// gets "garbage stuff "
// and throws away '('

getline(ss, myline, ')' );
// gets "words I care about"
// and throws away ')'`

stringstream ss(myline);
string word;
while( ss >> word ) {
  mywords.push_back(word);
}
```