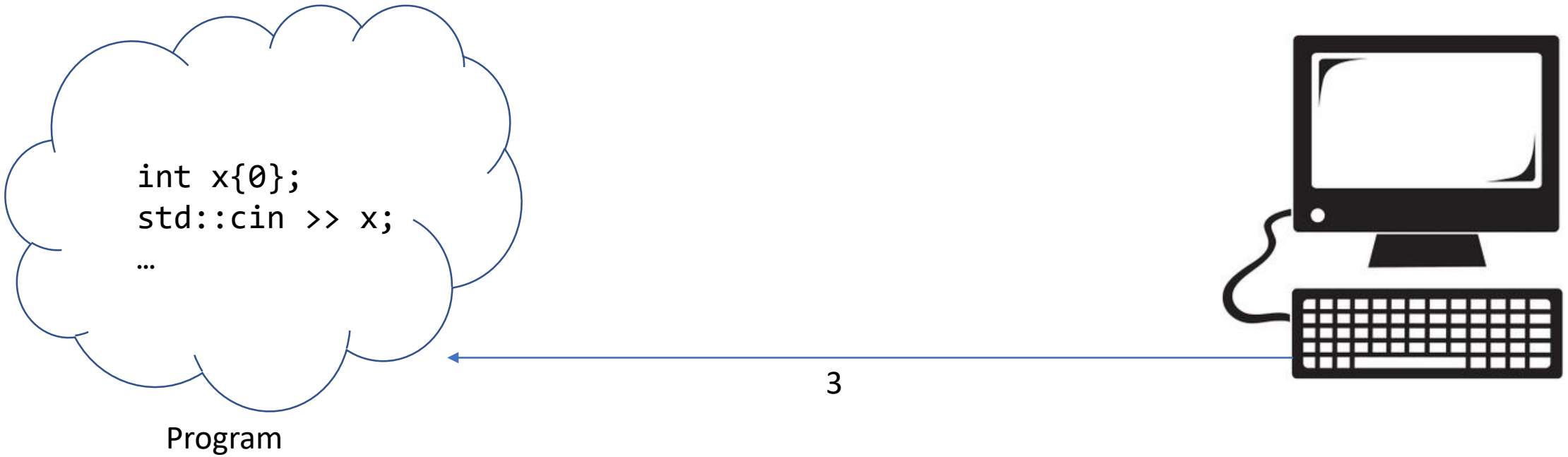
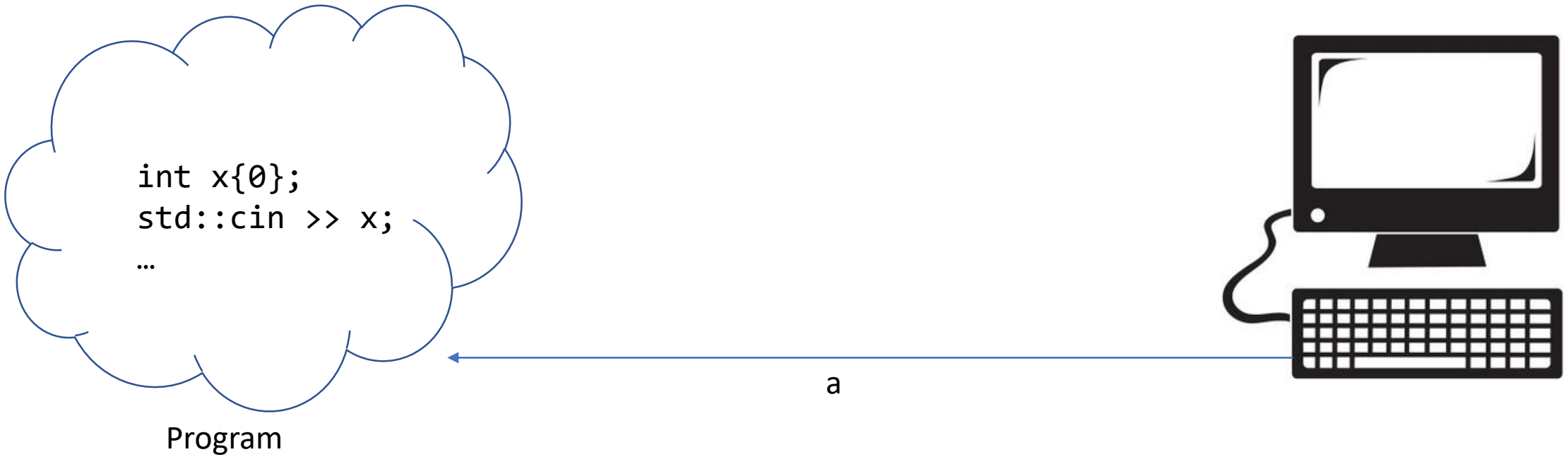
TDDE18 & 726G77

Streams & Classes

Input – correct type



Input – incorrect type



Error sources

- End of file - Reached the end of the input
- Logic failure - Conversion from characters to number fail
- Serious error - Like hardware failure

- After ANY failure the source will SILENTLY REFUSE further operations!
  - Remember what happened in lab0 if you enter something weird when it asked for an int?

Error checking

- Must be performed just after each operation
- Provide possibility to detect error
- Almost never used explicitly
- Four operations available:

```
bool eof();          // end-of-file
bool fail();         // logical error
bool bad();          // serious error
bool good();         // no error
```

Peculiarities (that make sense)

- bad() and good() are not opposites
  - A stream may be !good() even if it’s !bad()

- fail() and good() are not opposites
  - A stream may be ! good() even if it’s !fail()

- eof() can be set when fail() is not set
  - Input can succeed despite slamming into eof!

Error checking – table form

eofbit	failbit	badbit	good()	**fail()**	bad()	eof()	operator bool	operator!
false	false	false	true	false	false	false	true	false
false	false	true	false	true	true	false	false	true
false	true	false	false	true	false	false	false	true
false	true	true	false	true	true	false	false	true
true	false	false	false	false	false	true	true	false
true	false	true	false	true	true	true	false	true
true	true	false	false	true	false	true	false	true
true	true	true	false	true	true	true	false	true

Error checking

```
int a{};
cin >> a;
if (cin.eof()) {
   ...
} else if (cin.fail()) {
   ...
} else if (cin.bad()) {
   ...
} else {
   // Here we know that everything went well
}
```

Error checking - better

```
int a{};
cin >> a;
if (cin) {
   // Here we know that everything went well
} else {
   // Bad stuff happened
}

if (cin)
```

is the same thing as

```
if (!cin.eof() and !cin.fail() and !cin.bad())
```

Error checking – even better

```
int a{};
if (cin >> a) {
   // Everything went well
} else {
   // Something went wrong
}
```

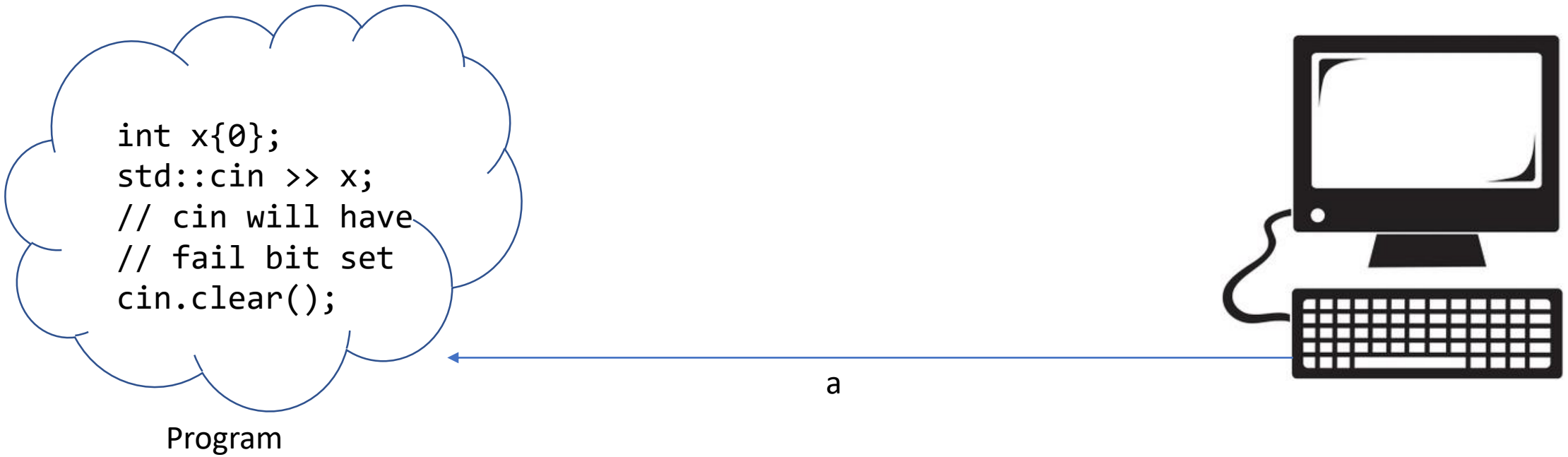
`if (cin >> a)` is the same thing as `if (cin.good())` with another benefit. It also read the value into the variable `a`.

Error cleaning

- Every stream will REFUSE further operation after any failure!
- Error must be cleared before stream can be operated again!
- Clearing an error will NOT EVER fix the problem!

```
void clear();          // clear all error flags
```

Error cleaning



Fixing the problem

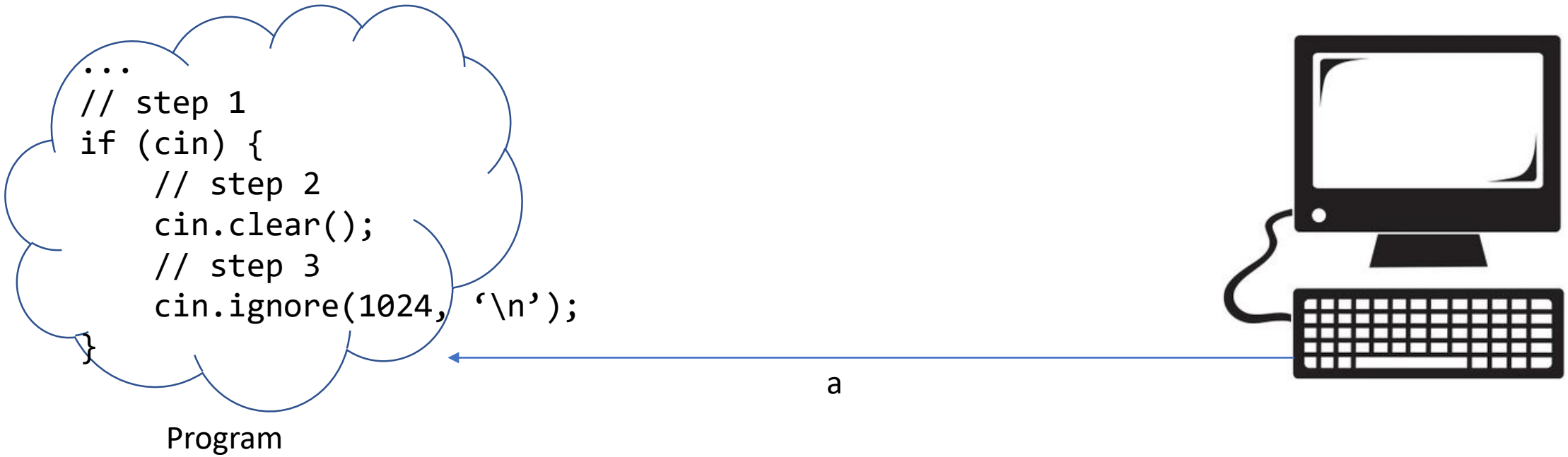
- Step one
  - detect that you have a problem

  if (! (cin >> i))
- Step two
  - clear the stream so it will cooperate

  cin.clear();
- Step three
  - remove the data that caused the error!

  cin.ignore(1024, ‘\n’);

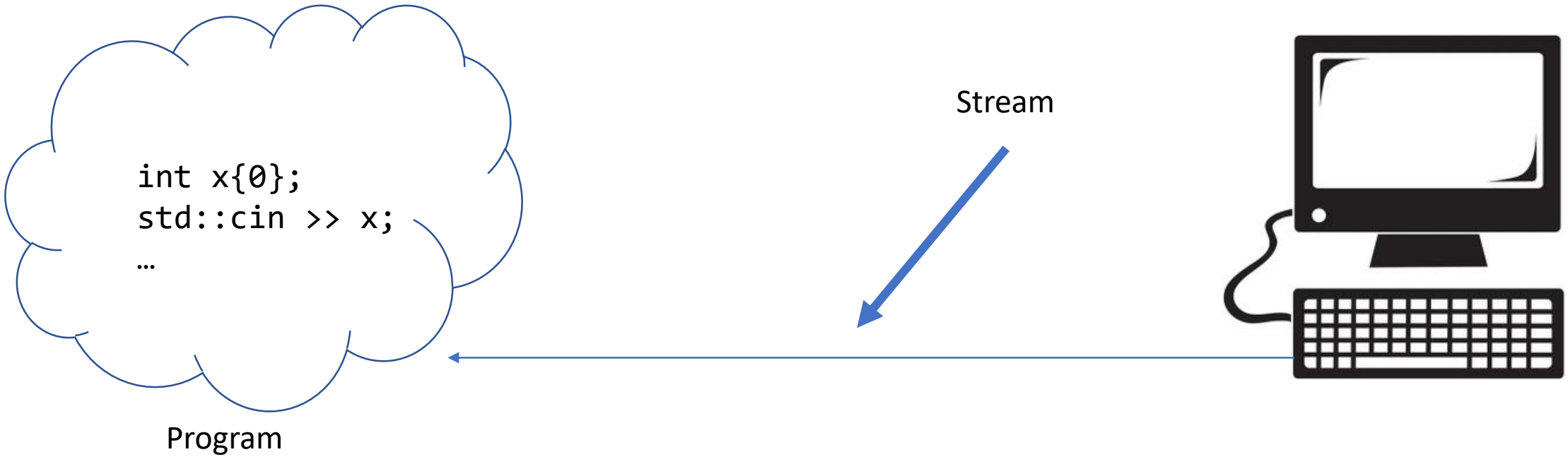
Error cleaning



Stream concept

- An ordered stream of bytes
- One source generates bytes
- One destination consumes bytes
- Not possible to break the given order
- Destination can not receive again
- You cant look ahead, only the first in line

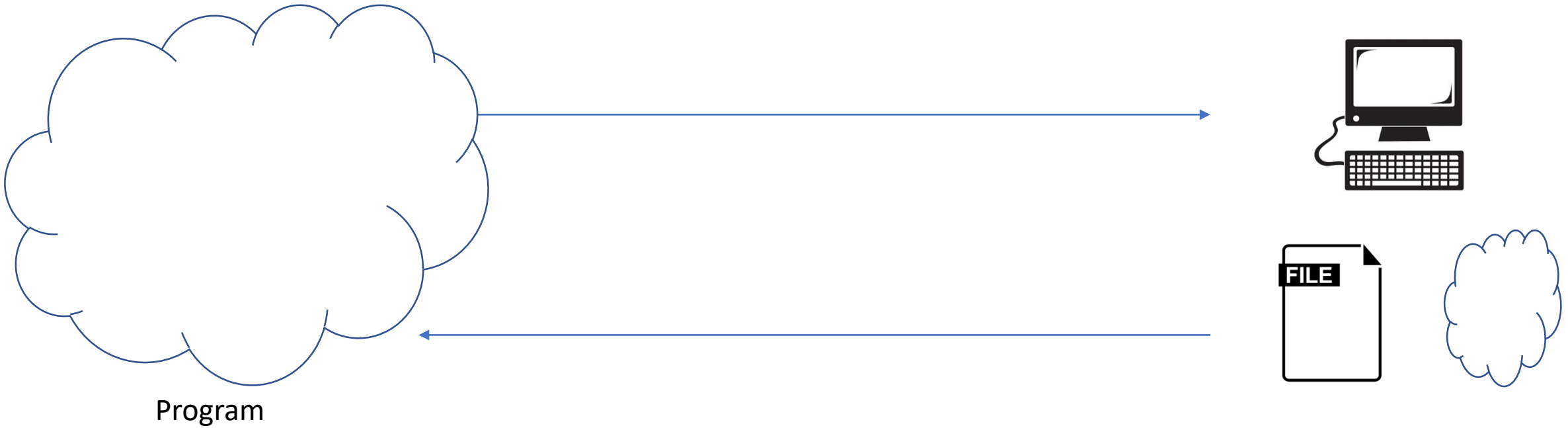
Stream



Three kinds of specific streams

- General I/O: cin, cout, cerr, clog
- File stream (file on disk act as source or destination)
- String streams (string variable in memory act as source of destination)

Three kinds of streams

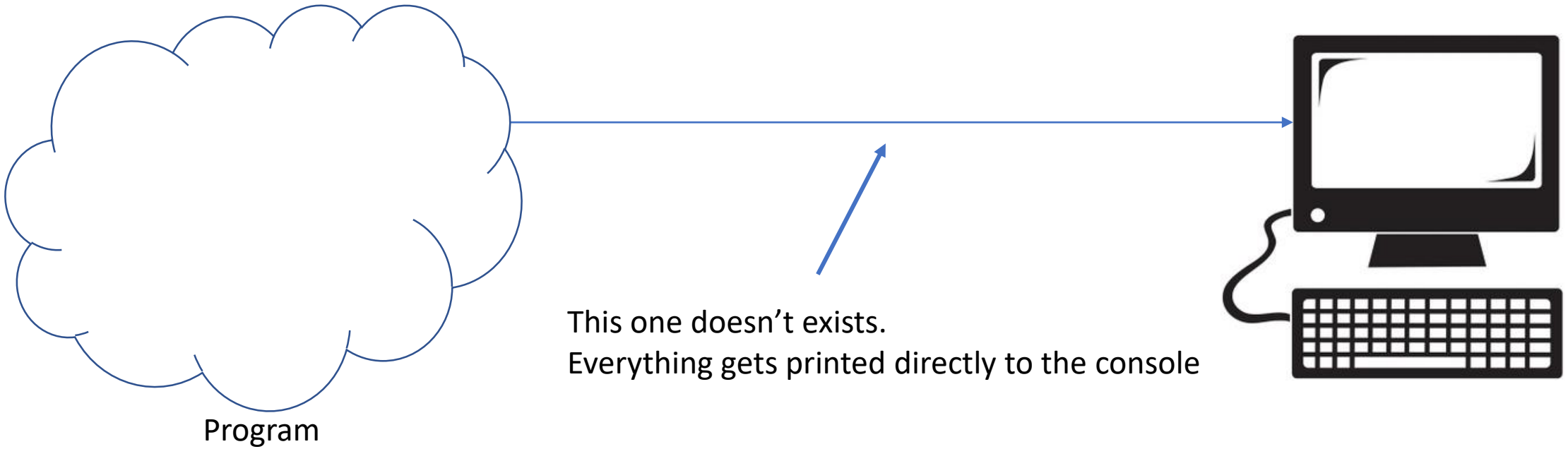


General I/O – Standard error

- Always write everything directly to the console, no need to flush first.
- No buffer

```
cerr << “this will always print to the console”;
```

General I/O – Standard error

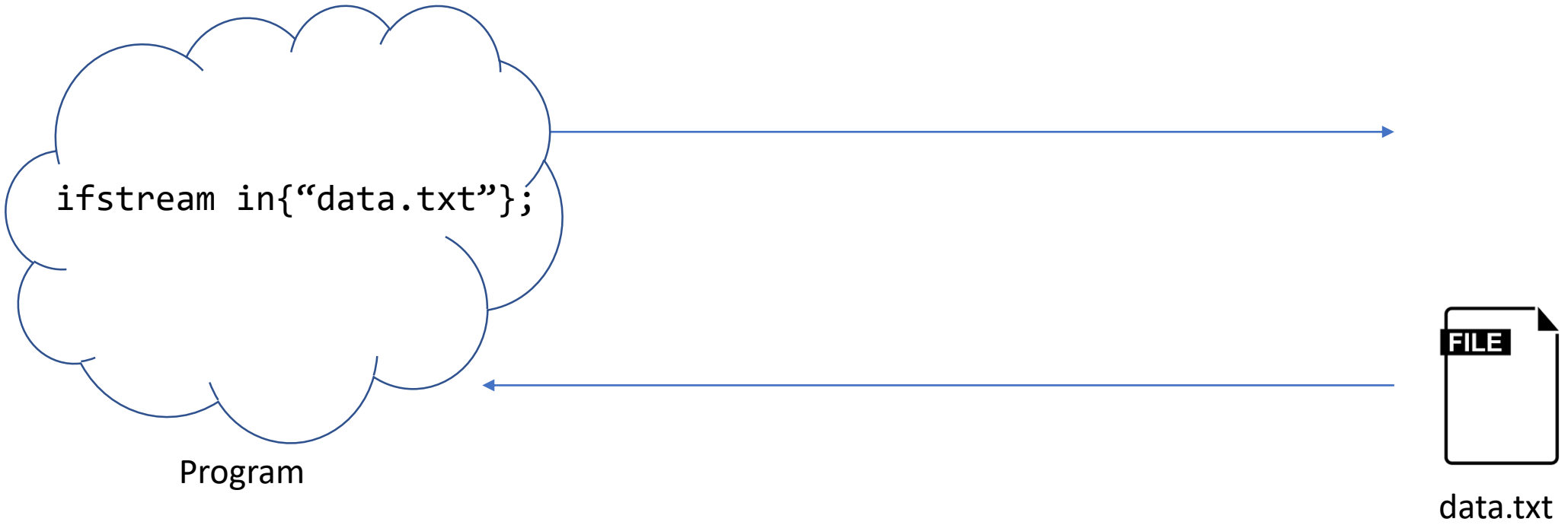


File stream input

Must be connected to a file on disk before use

```
#include <fstream>

ifstream infile{};            // Create stream object
infile.open(“data.txt”);      // Connect stream object

ifstream in{“data.txt”};      // Create and connect stream object

int a{};
int b{};
in >> a;                      // Read from ‘in’
infile >> b;                  // Read from ‘input’
in.close();                   // Disconnect
```

File stream input

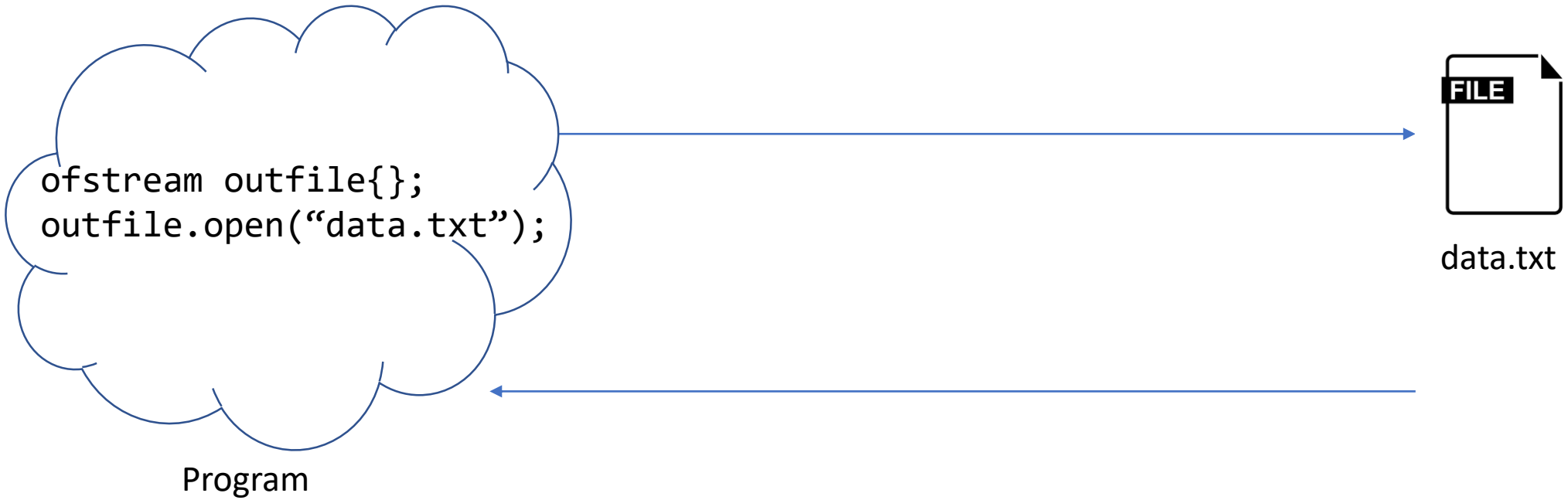


File stream output

Must be connected to a file on disk before use

```
#include <fstream>

ofstream outfile{};             // Create stream object
outfile.open(“data.txt”);       // Connect stream object

ifstream out{“data.txt”};       // Create and connect stream object

int a{123};
int b{512};
out << a;                       // Write to out
outfile << b;                   // Write to outfile
out.close();                    // Disconnect
```

File stream input



Modes for filestreams

- Files can be opened in several modes.

```
void open(string const& name, openmode mode);
```

- Possible openmodes:

```
ios::app  ios::ate      ios::trunc
ios::out  ios::in       ios::binary
```

- Files should be closed as soon as possible

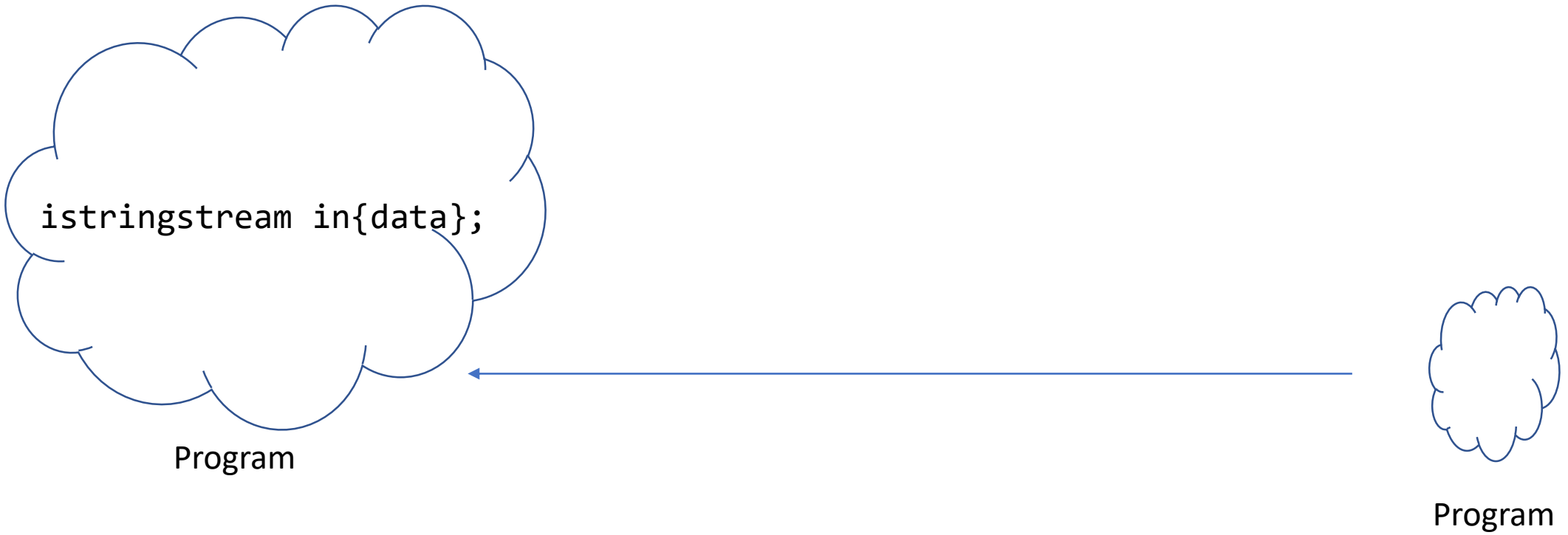
```
void close();
```

String stream input

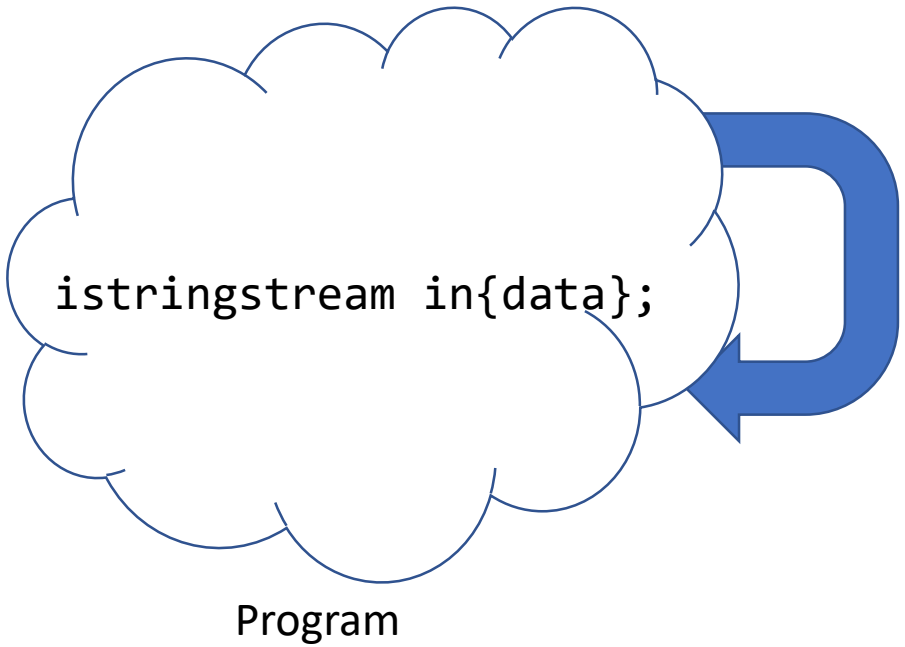
Must be connected to a string variable before use

```
#include <sstream>

string data{“4711 512”};

istringstream instr{};                  // create stream
instr.str(data);                        // connect stream

istringstream in{data};                 // create, connect

int a{};
int b{};
in >> a;                                // read from ‘in’
instr >> b;                             // read from ‘instr’
```

Three kinds of streams



Three kinds of streams



String stream output

```
#include <sstream>

ostringstream outstr{};

int a{4711};
int b{512};

outstr << a << b;

string str{outstr.str()};

// What is in ‘str’?
```

Stream references

- Stream can be sent to functions, and returned from funcations as references!
- It must be reference! You can not copy!
- Use the generic istream&, ostream& types
  - An ifstream is an istream
  - An istringstream is also an istream
  - cin is an istream

Print table to any stream

```
ostream& print_table(ostream& os)
{
    for (int i{0}; i < 10; i = i + 1) {
        os << setw(4) << i << endl;
    }
    return os;
}
```

Using the print function

```
int main() {
      ofstream file{“table.txt”};

      if ( ! file ) {
            cerr << “cant open file\n”;
      }
      else {
            print_table(file);              // Use the file
            file.close();
      }
      print_table(cout) << “.” << endl;     // Use standard output
}
```

Generate errors the C++ way

- Your goal should be to notice the program user in a clear and understandable way and then recover
- C++ ways is to throw an exception

Generate error the C++ way

```
#include <exception>
string msg{“error message”};
throw invalid_argument(msg);
throw logic_error(“bad bool”);
throw domain_error(“bad luck”);

#include <iostream>
throw ios::failure(“bad file”);
```

Error handling

```
int main() {
    ofstream file{“table.txt”};

    if ( ! file ) {
        throw ios::failure(“no file”);
    }
    print_table(file);
}
```

Command line argument

```
a.out testing 1 2 3


int main(int argc, char* argv[]) {
     for (int i{0}; i < argc; ++i) {
          cout << argv[i] << endl;
     }
}
```

Command line argument

```
sum 1 2


int main(int argc, char* argv[]) {
     cout << “The program is: “ << argv[0] << endl;
     cout << “The sum is”
          << argv[1] + argv[2] << endl;
}
```