Types and Streams II Summary

Key Takeaways

- >> extracts the next variable of a certain type, up to the next whitespace.
- The >> and << operators return a reference to the stream itself, so in each instance the stream is the left-hand operand.
- Yes, it’s called a stringstream. Reading and writing simultaneously can often lead to subtle bugs, be careful!

set stream

stringstream positioning functions

	input stream	output stream
get position	`oss.tellp();`	`iss.tellg();`
set position	`oss.seekp(pos);`	`iss.seekg(pos);`
create offset	`streamoff(n)`	

These methods let you manually set the position.
Most useful is the offset which can be added to positions.

Note: the types are a little funky. Read the documentation!

state bits

Four bits indicate the state of the stream.

- G Good bit: ready for read/write.
- F Fail bit: previous operation failed, all future operations frozen.
- E EOF bit: previous operation reached the end of buffer content.
- B Bad bit: external error, likely irrecoverable.

Common reasons why that bit is on.

G Nothing unusual, on when other bits are off.

F Type mismatch, file can’t be opened, seekg failed.

E Reached the end of the buffer.

B Could not move characters to buffer from external source.
(e.g. the file you are reading from suddenly is deleted)

Important things about state bits.

G and B are not opposites! (e.g. type mismatch)

G and F are not opposites! (e.g. end of file)

F and E are normally the ones you will be checking.

Conclusion: You should rarely be using G.

cout and cin

cin的行为：
1. 跳whitespace 2. 有无eof：有 -> wait user to type+enter -> try to read stuffs in the buffer -> stop at whitespace/type mismatch(failed!)

Key Takeways

- When does the program prompt the user for input?
- Why does the cout operation not immediately print the output onto the console? When is the output printed?
- Does the position pointer skip whitespace before the token or after the token with each >> operation? (this is important!)
- Does the position pointer always read up to a whitespace? If not, come up with a counterexample.

Key Takeaways

- The program hangs and waits for user input when the position reaches EOF, past the last token in the buffer.
- All input operations will flush cout.
- The position pointer does the following:
    - consume all whitespaces (spaces, newlines, etc.)
    - reads as many characters until:
        - a whitespace is reached, or…
        - for primitives, the maximum number of bytes necessary to form a valid variable.
        - example: if we extract an int from “86.2”, we’ll get 86, with pos at the decimal point.

What is getline?

区分读入的position和返回值内容的区别：
前者consume了\n，后者没包括\n

- Covered in CS 106B (probably)?
- Reads up to the next delimiter (by default '\n') and consumes it the delimiter. Position is now *past* the delimiter.
- Returns the "line" without the whitespace.

type of line is string

- Always use getline with cin instead of >> (why?)
- Always check the return value of getline (why?)

Be careful about mixing >> with getline!

- To solve the issue of getline retrieving a whitespace, use the ignore function.
- Generally try to avoid this problem by…
    - Using getline for cin, not >>.
    - Using >> when you are trying to parse space by space.
    - Using more advanced regex libraries if doing more advanced parsing.
- Do not use >> with the Stanford libraries which use getline.

Summary of Types and Streams II

- Use modern C++ constructs! (auto, uniform initialization, etc.)
- If you need error checking for user input, best practice is to:
    - use getline to retrieve a line from cin,
    - create a istringstream with the line,
    - parse the line using a stringstream, usually with >>.
- Use state bits to control streams and perform error-checking.
    - fail bit can check type mismatches
    - eof bit can check if you consumed all input

seekg（setting）
tellg

modern C++ types

note: the slides in this section is meant mostly as a reference, and doesn’t have a logical flow. During lecture we mostly focused on an example.

When to use type aliases?

- When a type name is too long and a simpler alias makes the code more readable.
- In libraries there is a common name for a type within each class. Example:
  - vector::iterator, map::iterator, string::iterator
  - vector::reference, map::reference, string::reference

When to use auto?

- When you don’t care what the type is (iterators)
- When its type is clear from context (templates)
- When you don’t know what the type is (lambdas)
- Don’t use it unnecessarily for return types.

```
auto spliceString(const string& s);
```

Can you guess what this function returns? Not really.

Why use auto?

- Correctness: no implicit conversions, uninitialized variables.
- Flexibility: code easily modifiable if type changes need to be made.
- Powerful: very important when we get to templates!

- Modern IDE’s (eg. Qt Creator) can infer a type simply by hovering your cursor over any auto, so readability not an issue!

struct functions

```
struct Discount {
  double discountFactor;
  int expirationDate;
  string nameOfDiscount;
}; // don’t forget this semicolon :/

// Call to Discount’s constructor or initializer list
auto coupon1 = Discount{0.9, 30, “New Years”};
Discount coupon2 = {0.75, 7, “Valentine’s Day”};

coupon1.discountFactor = 0.8;
coupon2.expirationDate = coupon1.expirationDate;

// structured binding (C++17) – extract each component
auto [factor, date, name] = coupon1;
```

aceess by reference

auto的妙用：binding
structed binding语法：[v1, v2, ...] = sth；用 []
可以用在遍历map的时候用两个变量来bind key and value，就没必要用内置的iterator.first/.second,
for (auto [key, value] : map) {...}

dangling references

never return references to local variables!

```
char& firstCharBad(string& s) {
  string local = s;
  return local[0];
}

char& firstCharGood(string& s) {
  return s[0];
}

int main() {
  string tea = “Ito-En”;
  char& bad = firstCharBad(tea);   // undefined, ref to local out of scope
  char& good = firstCharGood(tea); // good ref to tea[0]
}
```

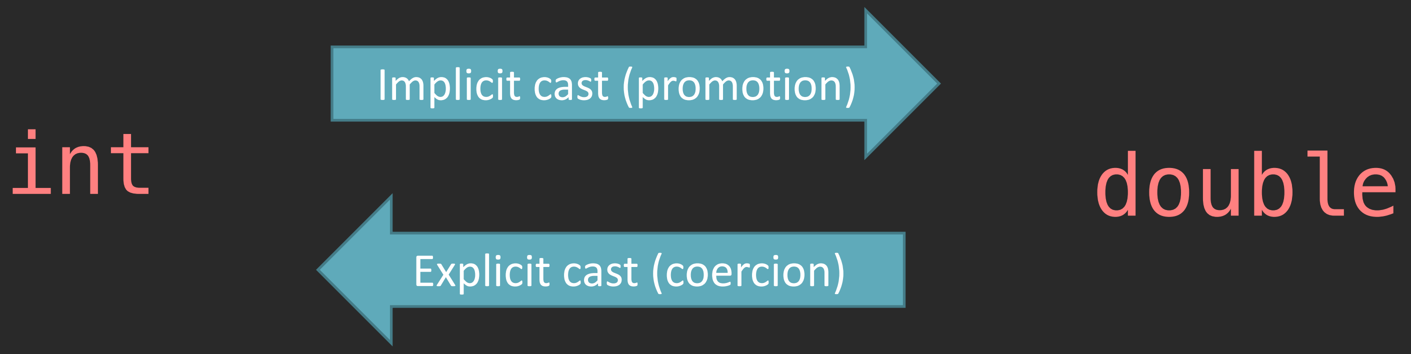
warning:Reference to stack memory associated with local variable 'b' returned

q: why char c = goodRef(a); ++c; // changing c does not change a ?

Conversions have two directions.

type cast

```
int v1 = static_cast<double>(3.4); // explicit cast
double v2 = 6;                     // promotion
```



stringstream vs. string

When should I use a stringstream?

1. Processing strings
   - Simplify "/./a/b/.." to "/a"
2. Formatting input/output
   - uppercase, hex, and other stream manipulators
3. Parsing different types
   - stringToInteger() from previous lectures

If you're just concatenating strings, str.append() is faster than using a stringstream!