slido.com
jhuintprog

601.220 Intermediate Programming

Summer 2022, Meeting 15 (July 13th)

Today's agenda

- Review exercises 25 and 26
- Day 27 recap questions
- Exercise 27
- Day 28 recap questions
- Exercise 28

Reminders/Announcements

- HW5 is due **Thursday, July 14th** (tomorrow!)
- HW7 is due **Thursday, July 21st**
  - We will have covered everything you need to know for the CTrie class by Friday
  - The TTrie class is a template class: we will cover template classes and functions on Monday
- Final project team formation: soon

Exercise 25 review

abbreviate function loop:

```
bool last_was_vowel = false;
for (size_t i = 0; i < word.size(); i++) {
  bool cur_is_vowel = is_vowel(word[i]);
  if (cur_is_vowel) {
    if (!last_was_vowel) { result += "'"; }
  } else {
    result += word[i];
  }
  last_was_vowel = cur_is_vowel;
}
```

Exercise 25 review

main function, opening input and output files:

```
ifstream in(argv[1]);
if (!in.is_open()) {
  cerr << "Couldn't open input file " << argv[1] << "\n";
  return 1;
}

ofstream out(argv[2]);
if (!out.is_open()) {
  cerr << "Couldn't open output file " << argv[2] << "\n";
  return 1;
}
```

main function main loop:

```
string line;
while (getline(in, line)) {
  stringstream line_in(line);
  string word;
  while (line_in >> word) {
    out << abbreviate(word) << " ";
  }
  out << "\n";
}
```

std::getline is useful for programs which process input one line at a time.

string s;

in >> s; // read token

classify program body of loop, variables:

```
double fpval;
int ival;
string extra;
bool is_ival = false, is_fpval = false;
```

Goal of loop body is tok classify one token.

"42.3"

Check whether token is an integer value:

```
stringstream as_i(token);
if (as_i >> ival) {
  if (!(as_i >> extra)) {
    sum_i += ival;
    is_ival = true;
  }
}
```

Idea is that when extracting an integer, there should not be any input "left over".

Exercise 25 review

Determine whether token is a floating point value:

```
if (!is_ival) {
  stringstream as_fp(token);
  if (as_fp >> fpval) {
    sum_fp += fpval;
    is_fpval = true;
  }
}
```

Exercise 25 review

Handle other tokens:

```
if (!is_ival && !is_fpval) {
  ntok++;
  ntok_c += token.size();
}
```

```
struct Bucket {
    char  letter;
    int  count;
};
```
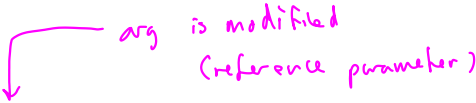
letter_freq program, initialize vector of Buckets:

```
vector<Bucket> buckets;
for (int i = 0; i < 26; i++) {
  Bucket b;
  b.letter = 'a' + i;
  b.count = 0;
  buckets.push_back(b);
}
```

Exercise 25 review

letter_freq program, open input file, read characters, classify them:

```
ifstream in(argv[1]);
if (!in.is_open()) {
  cerr << "Couldn't open input file " << argv[1] << "\n";
  return 1;
}
                            arg   is modified
                                 (reference parameter)
char c;
while (in.get(c)) {
  c = tolower(c);
  if (isalpha(c)) {
    buckets[c - 'a'].count++;
  }
}
```

Exercise 25 review

letter_freq program, bucket comparision function:

```cpp
// we want Buckets with a higher count to compare as "less"
// (so that the overall ordering is from most frequent
// to least frequent)
bool compare_buckets(const Bucket &left, const Bucket &right) {
  if (left.count > right.count) { return true; }
  if (left.count < right.count) { return false; }
  return left.letter < right.letter;
}
```

std::sort

Sorting the vector of Buckets:

comparator

```cpp
sort(buckets.begin(), buckets.end(), compare_buckets);
```

Exercise 25 review

`letter_freq` program, printing letter frequencies:

```cpp
for (vector<Bucket>::const_iterator i = buckets.cbegin();
     i != buckets.cend();
     ++i) {
  if (i->count > 0) {
    cout << i->letter << ": " << i->count << "\n";
  }
}
```

Exercise 26 review

```cpp
void make_cumulative(std::vector<double> &dist) {
  for (std::vector<double>::iterator i = dist.begin() + 1;
       i != dist.end();
       ++i) {
    *i += *(i - 1);
  }
}
```

"pointer arithmetic"

Note: `vectors` have random-access iterators, so "pointer arithmetic" on iterator values is possible.

Fun fact: in most STL implementations, `vector` iterators _are_ pointers.

Exercise 26 review

naive_find_last_iterator function:

```
std::vector<double>::const_iterator best = begin, i = begin;
while (i != end && *i <= v) {
  best = i;
  ++i;
}
return best;
```

Exercise 26 review

fast_find_last_iterator function:

```cpp
size_t n = end - begin;
if (n == 1) { return begin; }

std::vector<double>::const_iterator mid = begin + n/2;
if (*mid > v) {
  return fast_find_last_iterator(begin, mid, v);
} else {
  return fast_find_last_iterator(mid, end, v);
}
```

Note: this is *slightly* different than the standard recursive binary search, because we do not eliminate the middle element from consideration in the recursive case.

Day 27 recap questions

❶ What is object-oriented programming?
❷ What is the difference between a public and a private field/member function?
❸ Do class fields and member functions default to public or private?
❹ Can we define member functions in a struct in C? How does C++ handle structs? Can we do that in C++?
❺ What is a default constructor?
❻ Why is using an initializer list in a class constructor a better choice than not using one?

1. What is object-oriented programming?

An object is an instance of a `class` (or `struct`) type. The class or struct type can have *member variables* and *member functions*.

Member variables (a.k.a. *fields*): define the data contained in an object.

Member functions (a.k.a. *methods*): define the *behavior* of an object.

In an object-oriented program, computations are done by calling member functions on objects.

2. What is the difference between a public and a private field/member function?

Public member: can be directly accessed from code outside the member functions of the class.

Private member: can only be directly accessed from member functions of the class.

General rules for member visibility

General rules:

- member variables (fields) should be private
- member functinos to be used by the program as a whole should be public (these are sometimes called "API functions")
- member functions that are only needed internally (helper functions for the class) should be private

The idea that internal implementatino details (fields, helper functions) are private is known as *encapsulation*.

3. Do class fields and member functions default to public or private?

For `class` types: default is `private`

For `struct` types: default is `public`

Otherwise, there is no difference between `class` types and `struct` tyypes.

4. Can we define member functions in a struct in C? How does C++ handle structs? Can we do that in C++?

In C, a struct type cannot have member functions.

In C++, a struct type can have member functions.

The only difference between C++ struct types and class types is that the members of struct types are public by default, and members of class types are private by default.

5. What is a default constructor?

A default constructor is a constructor which requires no arguments.
In is invoked to initialize an object if no other constructor is invoked.
Like all constructors, its main job is to initialize the fields of the
object being created.

Assume that Foo is a class:

```
Foo f; // default constructor is called to initialize
       // fields of f

Foo arr[10]; // default constructor is called on each
             // element of arr

Foo *p = new Foo[5]; // default constructor is called on
                     // each element of the dynamically
                     // allocated array
```

6. Why is using an initializer list in a class constructor a better choice than not using one?

It avoids an initialization of a field by its default constructor, followed by an *assignment* to change the value of the field.

```
struct Foo {
  std::string s;
  Foo();
};

Foo::Foo()  // best approach
  : s("initial value for s")
  { }

Foo::Foo()  // not as good
  { s = "initial value for s"; }
```

① (for the `s("initial value for s")` line)

② → `def ctor for s is invoked`

constructor ⟱ *initialize member vars*

11:20

- Practice working with a class (`grade_list`) and member variables and functions
- Breakout rooms 1–10 are "social"
- Use Slack to let us know if you have questions!

Day 28 recap questions

1. What is a non-default (or "alternative") constructor?
2. If we define a non-default constructor, will C++ generate an implicitly defined default constructor?
3. When do we use the this keyword?
4. What is a destructor?
5. A destructor will automatically release memories that are allocated in the constructor- true or false?

1. What is a non-default (or "alternative") constructor?

A non-default constructor has one or more parameters. Usually, these are used to initialize the field(s) of the object being initialized.

Example:

(handwritten) Point p (3.0, 4.5);

```cpp
class Point {
private:
  double x, y;
public:
  Point() : x(0.0), y(0.0) { }   // default constructor
  Point(double x, double y)      // non-default constructor
    : x(x), y(y) { }
  // ... other member functions ...
};
```

2. If we define a non-default constructor, will C++
generate an implicitly defined default constructor?

No. For example:

```
class Point {
private:
  double x, y;
public:
  Point(double x, double y)
    : x(x), y(y) { }
  // ... other member functions ...
};

// ...elsewhere...

Point p; // will not compile
```

struct Foo {
    string s;
};

implicit
default
constructor

Foo f;

3. When do we use the `this` keyword?

The `this` keyword us useful for explicitly referring to the object a member function is called on, sometimes called the "receiver" object. It is a *pointer* to the receiver object.

Among other uses, `this` can be useful for disambiguating a member variable that has the same name as a parameter. Example:

```
class Point {
private:
  double x, y;
public:
  // ...
  void set_x(double x) { this->x = x; }
  // ...
};
```
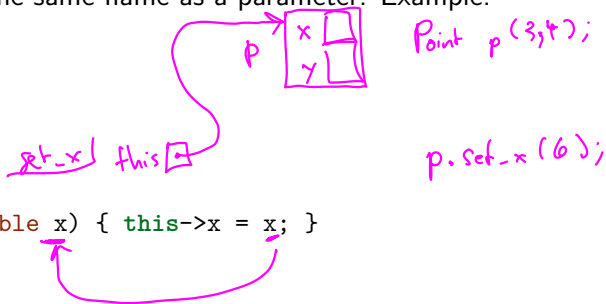
4. What is a destructor?

A `class` (or `struct`) type's destructor member function is called automatically when an object's lifetime ends. It's purpose is to deallocate any dynamic resources associated with the object.

Examples of dynamic resources:

- dynamically allocated memory
- file resources not automatcially closed by a destructor

Destructor example

```
class CBuf {
private:
  char *buf;
  size_t size;

public:
  CBuf(size_t sz) : buf(new char[sz]), size(sz) { }
  ~CBuf() { delete[] buf; }

  // ...other member functions...
};
```

5. A destructor will automatically release memories that are allocated in the constructor- true or false?

False.

The destructor must *explicitly* deallocate dynamically-allocated memory using either delete or delete[] (depending on whether or not the memory being deallocated is an array.)

Exercise 28

- `grade_list` again, but this time storing grades in a dynamically allocated array
  - This is very much like how `std::vector` works!
- Breakout rooms 1–10 are "social"
- Use Slack to ask us if you have questions!

Notes

Notes

Notes

Notes

Notes