CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 8
September 24, 2018

Etudes in Coding

Overview

Software construction is much like other activities that combine design with skills.

Piano students practice scales and études as well as learning to play Beethoven piano sonatas.

Ballet dancers do barre exercises to acquire the skills needed to dance Nutcracker.

Authors learn good writing style by having others criticize their own work.

Today I present some examples of programs and try to point out the design decisions that impact the cleanliness and robustness of the result.

Problem Set 1 Design Issues

```
// Solution by Michael J. Fischer
// Calculate a user's age
void run() {
    string first;
    string last;
    int birthYear;
    int age;

    // Get current year
    const time_t now = time( nullptr );          // get current time
    struct tm* today = localtime( &now );        // break into parts yr-mon-day
    const int thisYear = 1900 + today->tm_year; // tm_year counts years from 1900

    cout << "Please enter your first name: ";
    cin >> first;
    if (!cin.good()) fatal("Error reading first name");

    cout << "Please enter your last name: ";
    cin >> last;
    if (!cin.good()) fatal("Error reading last name");

    cout << "Please enter the year of your birth: ";
    cin >> birthYear;
    if (!cin.good()) fatal("Error reading age");

    age = thisYear - birthYear;
    cout << first << " " << last << " becomes " << age << " years old in "
         << thisYear << "." << endl;
}
```

Comments on my code

Good points:

- Logical progression towards solution: get year, get first name, get last name, get birth year, compute age, print results.
- Most obscure part of getting current year is commented.
- Identifiers are compromise between length and clarity.
- All I/O errors are detected, reported, and handled as required.

Drawbacks:

- Code is monolithic.
- User-interaction is intermixed with computation.
- Variables related to user (`first, last, birthYear, age`) are not separated from intermediate variables (`now, today, thisYear`).
- General computation is not isolated from input-specific code.

A student solution, function isgood()

```
// ----------------------------------------------------------------------------
// Function to check for input errors and then concatenate first and last name
// string inputs.
void isgood(string *name, string *temp)
{
    cin >> *temp;
    if (cin.good()) {
        *name = *name + *temp;
    }
    else {
        fatal("Invalid input.");
    }
}
```

Comments on `isgood()`

Good points:

- Clear separation from surrounding code.
- Clear statement of purpose, but incomplete.
- Uses `cin.good()` for error checking as required.

Drawbacks:

- Statement of purpose omits mention of string read.
- Function name suggests only the checking part.
- A check-only founction should be `const` and return a `bool`.
- The actions to take with a successful or unsuccessful read should not be the concern of the checking function.
- `name` should not be a parameter.
- Output parameter `temp` should be of reference type `string&`.

A student solution, function calctime()

```
// --------------------------------------------------------------------------
// Function to check for input errors and then calculate both the current year
// and the age of the user using time() and localtime().
void calctime(int *age, int *year)
{
    int birth;
    cin >> birth;
    if (cin.good()) {
        time_t current;
        struct tm * localhold;

        time(&current);
        localhold = localtime(&current);

        *year = 1900 + localhold->tm_year;
        *age = *year - birth;
    }
    else {
        fatal("Invalid input.");
    }
}
```

Comments on `calctime()`

Similar coments to `isgood()`.

Main drawback is that user interaction, data reading, error checking, and time calculations are carried out by the same function.

When we get to classes,`age` and `year` would be data members of the class containing `calctime()`, and `calctime()` would need no parameters.

Minor formatting problem: Left bracket { should be at end of `isgood` line, not on a line by itself. Applies to `isgood()` as well.

A student solution, function run()

```
// ----------------------------------------------------------------------------
// Run function that prints out user prompts and calls subsidiary functions for
// processing submitted inputs.
void run() {
    string name;
    string temp;

    cout << "Please enter your first name: ";
    isgood(&name, &temp);
    name = name + " ";              // adds a space between first and last name
    cout << "Please enter your last name: ";
    isgood(&name, &temp);

    int age;
    int year;

    cout << "Please enter the year of your birth: ";
    calctime(&age, &year);

    cout << name << " becomes " << age << " years old in " << year << ".\n";
}
```

Comments on `run()`

Good points:

- Correctly formatted function definition.
- Checks both first name and last name for read errors.
- Checking code is not replicated.
- Consistent top-level structure for handling names and birth year.

Drawbacks:

- No need to use expensive string concatenation. `name` is unnecessary. Better to have separate `first` and `last` string variables.

Brackets Example

Code demo

The 08-Brackets demo contains three interesting classes and illustrates the use of constructors, destructors, and dynamic memory management as well as a number of newer C++ features.

It is based on the example in section 4.5 of "Exploring C++", but there are several significant modifications to the code.

Many of the changes use features of c++17 and would not work under the older standard. Others reflect different design philosophies.

We briefly summarize below some of the features of the demo.

The problem

The problem is to check a file to see if the brackets match and are properly nested.

For example, `([]())` is okay, but `([)]` is not, nor is `(()))` or `[[[`.

A bracket matching algorithm

Rules for bracket matching:

1. Each left bracket is pushed onto the stack.
2. An attempt is made to match each right bracket with the top character on the stack.
3. The attempt fails if
   - The stack is empty, or
   - The top character is a different type of bracket (e.g., round instead of square).
4. If the match fails, an error comment is printed, the mismatched characters are discarded, and processing continues with the next character.
5. At end-of-file, the stack should be empty, for any remaining characters on the stack are unmatched left brackets.

Program design

The program is organized into four modules.

1. Class `Token` wraps a single character. It contains functions for determining which characters are brackets, and for each bracket, its "sense" (left or right), and its "type" (round, square, curly, or angle).
2. Class `Stack` implements a general-purpose growable stack of objects of copyable type `T`. In this case, `T` is typedef'ed to `Token`.
3. Class `Brackets` implements the matching algorithm. It reads the file and carries out the matching algorithm.
4. `main.cpp` contains the main program. It processes the command line, opens the file, and invokes the bracket checker.

Token class

Major points:

1. `enum` is used to encode the bracket type (round, square, etc.) and the sense of the bracket (left, right).
2. The two `enum` types are defined inside of class `Token` and are private.
3. `ch` is the character representing the bracket, used for printing.
4. `classify()` is a private function.
5. The definitions of `print()` and `operator<<` follow our usual paradigms.

Token class (cont.)

6. The `Token` constructor uses a ctor to initialize data member `ch`. This overrides the **default member initializer** present in the declaration of `ch`. The constructor calls `classify()` to initialize the other data members.
7. In the ctor `:ch(ch)` , the first `ch` refers to the data member and the second refers to the constructor argument.
8. In the textbook version of `Token`, the static object `brackets` is *local* to `classify()`. It is now a static *class object*, initialized in `token.cpp`.

Token design questions

1. The textbook version of `Token` uses getters to return `type` and `sense`. `getType()` was used to test if a newly-read character was a bracket, and it was also used to see if a left bracket and right bracket were the same type.
   Why were they needed?
2. The new version of `Token` replaces `getType()` with boolean functions `isBracket()` and `sameTypeAs()` functions. Similarly, `getSense()` was replaced by boolean function `isLeft()`.

   With these changes, enum `BracketType` and `TokenSense` are no longer needed outside of `Token` and hence are now private.

   What are the pros and cons of this design decision?

Token design questions (cont.)

3. Both the old and new versions of the program work whether or not `brackets` is `static`.
    - Is static a better choice here?
    - Why or why not?
    - Does your answer depend on whether the object is local (old code) or class (new code)?

Stack class

Major points:

1. `T` is the element type of the stack. This code implements a stack of `Token`. (See `typedef` declaration.)
2. Storage for stack is dynamically allocated in the constructor using `new[]` and deleted in the destructor using `delete[]`.
3. The copy constructor and assignment operator have been deleted to avoid "double delete" problens with the dynamic extension.
4. The square brackets are needed for both `new` and `delete` since the stack is an array.
5. `delete[]` calls the destructor of each `Token` on the stack. Okay here because the token destructor is null.

Stack class (cont.)

6. `push()` grows stack by creating a new stack of twice the size, copying the old stack into the new, and deleting the old stack. This results in linear time for the stack operations.
7. If `push()` only grew the stack one slot at a time, the time would grow quadratically.

Stack design questions

1. Should `pop()` return a value?
2. Why does stack have a `name` field?
3. `size()` isn't used. Should it be eliminated?
4. `Stack::print()` formerly declared `p` and `pend` at the top. Now they are declared just before the loop that uses them. Is this better, and why?
5. Could they be declared in the loop? What difference would it make?