Object-Oriented Programming

Lecture 3: Organizing Programs and Data

01286131 Object-Oriented Programming

Software Engineering Programme
International College, KMITL

Contents

- Organizing Programs
- Programming Style
- Organizing Data
- C++ Compilation Process

Organizing Programs

Organizing Programs (1)

- As we follow more examples, programs start to get larger
- They would have been larger without **`vector`**, **`string`**, and **`sort`**
- These library facilities share several qualities
  - They solve a paricular kind of problem
  - They are independent of most of the others
  - They have a name

Organizing Programs (2)

- C++ offers two fundamental ways of organizing programs
  - Functions (subroutines)
  - Data structures
- We will explore a **class** which is a way to combine functions and data structures into a single unit later on

Writing C++ Functions (1)

A function must be declared in every source file that uses it, and defined only once.

```
ret-type function-name(parm-decls);            // function declaration

[inline] ret-type function-name(parm-decls)    // function definition
{
    // function body goes here
}
```

Example:

```cpp
// compute a student's overall grade
// from midterm and final exam grades and homework grade
double grade(double midterm, double final, double homework)
{
    return 0.2 * midterm + 0.4 * final + 0.4 * homework;
}
```

Writing C++ Functions (2)

Previously, we computed a grade by writing:

```
cout << "Your final grade is " << setprecision(3)
     << 0.2 * midterm + 0.4 * final + 0.4 * sum / count
     << setprecision(prec) << endl;
```

With `grade` function, we could have written:

```
cout << "Your final grade is " << setprecision(3)
     << grade(midterm, final, sum / count)
     << setprecision(prec) << endl;
```

Example: Finding Medians

```
// compute the median of a `vector<double>'
// note that calling this function copies the entire argument `vector'
double median(vector<double> vec)
{
    typedef vector<double>::size_type vec_sz;

    vec_sz size = vec.size();
    if (size == 0)
        throw domain_error("median of an empty vector");

    sort(vec.begin(), vec.end());

    vec_sz mid = size / 2;
    return size % 2 == 0? (vec[mid] + vec[mid - 1]) / 2: vec[mid];
}
```

- Notice the use of **exception** for error handling (**`throw domain_error`**)

Example: Finding Medians (with `auto`)

```
// compute the median of a `vector<double>'
// note that calling this function copies the entire argument `vector'
double median(vector<double> vec)
{
    auto size = vec.size();
    if (size == 0)
        throw domain_error("median of an empty vector");

    sort(vec.begin(), vec.end());

    auto mid = size / 2;
    return size % 2 == 0? (vec[mid] + vec[mid - 1]) / 2: vec[mid];
}
```

Example: Reimplementing Grading Policy

```cpp
// compute a student's overall grade from midterm and final exam grades
// and vector of homework grades.
// this function does not copy its argument, because `median' does so for us.
double grade(double midterm, double final, const vector<double>& hw)
{
    if (hw.size() == 0)
        throw domain_error("student has done no homework");
    return grade(midterm, final, median(hw));
}
```

- Notice the use of **`const vector<double>&`**
- The use of several functions with the same name is called function **overloading**

Example: Reading Homework Grades

```
// read homework grades from an input stream into a `vector<double>'
istream& read_hw(istream& in, vector<double>& hw)
{
    if (in) {
        // get rid of previous contents
        hw.clear();

        // read homework grades
        double x;
        while (in >> x)
            hw.push_back(x);

        // clear the stream so that input will work for the next student
        in.clear();
    }
    return in;
}
```

Function Input/Output Choices (C++98)

With function parameters and return value, reasonable default choices for choosing the function input/output is as shown in the table below:

	Cheap to Copy (e.g., `int`)	**Moderate cost to copy** (e.g., `string`, `BigPOD`) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to copy** (e.g., `vector`, `BigPOD[]`)
Out	`X f()`		`f(X&)`*
In/Out	`f(X&)`		
In	`f(X)`	`f(const X&)`	
In & retain copy			

"Cheap" ≈ a handful of hot int copies
"Moderate cost" ≈ memcpy hot/contiguous ~1KB and no allocation

* or return **X*** at the cost of a dynamic allocation

Reference: Essentials of Modern C++ Style, Herb Sutter, CppCon 2014

Function Input/Output Examples (1)

```cpp
double median(vector<double> vec); // by value input (`vec')

median(hw); // call by value, `hw' is copied
```

```cpp
// by const reference input (`hw')
double grade(double midterm, double final, const vector<double>& hw);

grade(midterm, final, homework); // `homework' is not copied
```

```cpp
// by reference output (`hw')
istream& read_hw(istream& in, vector<double>& hw);

read_hw(cin, homework); // `homework' value is possibly changed
```

Function Input/Output Examples (2)

```
vector<double> empty_vec()
{
    vector<double> v; // no elements
    return v;
}
```

```
  // C++11 and later
vector<double> empty_vec()
{
    return {};
}
```

```
grade(midterm, final, empty_vec()); // throws an exception!

read_hw(cin, empty_vec()); // error: `empty_vec()' is not an `lvalue'

    // this example doesn't work
    try {
        streamsize prec = cout.precision();
        cout << "Your final grade is " << setprecision(3)
             << grade(midterm, final, homework) << setprecision(prec);
    } ...
```

Reimplementing the Grading Program

```
int main()    // grading1.cpp
{
    // ask for and read the student's name
    cout << "Please enter your first name: ";
    string name;
    cin >> name;
    cout << "Hello, " << name << "!" << endl;

    // ask for and read the midterm and final grades
    cout << "Please enter your midterm and final exam grades: ";
    double midterm, final;
    cin >> midterm >> final;

    // ask for the homework grades
    cout << "Enter all your homework grades, "
            "followed by end-of-file: ";

    vector<double> homework;

    // read the homework grades
    read_hw(cin, homework);

    // compute and generate the final grade, if possible
    try {
        double final_grade = grade(midterm, final, homework);
        streamsize prec = cout.precision();
        cout << "Your final grade is " << setprecision(3)
             << final_grade << setprecision(prec) << endl;
    }
    catch (const domain_error&) {
        cout << endl << "You must enter your grades."
                        "  Please try again." << endl;
        return 1;
    }
    return 0;
}
```

Organizing the Grading Program

```cpp
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>

//// using std::cin; using std::cout; ...

double median(vector<double> vec)
{
    // ...
}

double grade(
    double midterm, double final, double homework)
{
    // ...
}

double grade(
    double midterm, double final,
    const vector<double>& hw)
{
    // ...
}

istream& read_hw(istream& in, vector<double>& hw)
{
    // ...
}

int main()
{
    // ...
}
```

Or alternatively putting function declarations first:

```cpp
//// #include <> ...

double median(std::vector<double> vec);
double grade(
    double midterm, double final, double homework);
double grade(
    double midterm, double final,
    const std::vector<double>& hw);
std::istream& read_hw(
    std::istream& in, std::vector<double>& hw);

//// using std::cin; using std::cout; ...

int main()
{
    // ...
}

//// Definitions for `median', `grade', and `read_hw'
//// may goes here or in other source files!
```

Programming Style

Background

- Like writing, programming is a form of **communication**
- Code is read much more often than written, so the code must be **understandable**
- Though subjective, **guidelines** or **conventions** are often useful
- No one true style: one size doesn’t fit all
- Choose one style and **be consistent**

Coding Conventions

Common coding conventions often cover the following areas:

- Naming Convention
- Indentation
- Comments
- Line Length

Coding Conventions (2)

The following areas has broader scope beyond introductory programming:

- Best Practices
- Programming Principles
  - Defensive Programming
  - Separation of Concerns
  - etc.
- Code Refactoring

Coding Conventions: Examples

- Programming: Principles and Practice using C++ (PPP) Style Guide
- The C++ Core Guidelines
- JSF-AV: https://www.stroustrup.com/JSF-AV-rules.pdf
- Misra C++: https://www.cppdepend.com/misra-cpp

Naming Convention

- Use meaningful names
  - Noun for variables, verb for functions
  - Simple names (i, x, y, p, etc.) are OK in small scopes
- Don’t use acronyms
- Don’t use excessively long names
- Beware of confusing letters and digits: **0Oo1lL**

Multiple-word Identifiers

- **`isupper`**: flat case
- **`ISUPPER`**: upper flat case
- **`isUpper`**: camel case
- **`IsUpper`**: pascal case, upper camel case
- **`is_upper`**: snake case
- **`IS_UPPER`**: macro case, constant case

Other variants: `is_Upper`, `Is_Upper`, `is-upper`, `IS-UPPER`, `Is-Upper`

Naming Convention: Example

C and C++

- Variables: **`some_var`**
- Functions: **`do_something(…)`**
- Types: **`Student_info`**
- Constants and macros: **`NUM_ITEMS`**

Naming Convention: Example (2)

Java, C#, Javascript, etc.

- Variables: **`someVar`**
- Functions: **`doSomething(…)`**
- Types: **`StudentInfo`**
- Constants and macros: **`NUM_ITEMS`**

Language-specific Name

In C and C++:

- Names are case sensitive
- Keywords are lowercase
- Names from standard library are mostly lowercase
- Reserved names
  - **`_Reserved`** (begin with an underscore and a capital letter)
  - **`__reserved`** (containing double underscore)

Indentation: PPP Style

```
// if statement
if (a == b) {
    // ...
}
else {
    // ...
}

// loop
for (int i = 0; i < 10; ++i) {
    // ...
}
```

```
// switch statement
switch (a) {
case A:
    // ...
    break;
case B:
    // ...
    break;
default:
    // ...
}
```

Indentation: PPP Style (2)

```
/// function
double sqrt(double d)
{
    // ...
}
```

```
/// class or struct:
class Temperature_reading {
public:
    // ...
private:
    // ...
};
```

Whitespace

- Vertical whitespaces (empty lines)
  - Between functions, structs, etc.
  - Separate different sections of code
- Tabs vs spaces
  - Be consistent with indentation
  - Pick one style and stick with it throughout the project

Indentation: Opinions

- Spaces are easier to manage across various people and softwares
- When you try to maintain consistent code layout and line limit, tabs are quite difficult to maintain
- Tabs can be problematic when changing editors and managing source code repository
- My choice is often **4 spaces** indentation, often the default for many text editors

Comments

Comments are good for:

1. Stating **intent** (what the code is supposed to do)
2. Explaining **ideas** related to the code
3. Stating **invariants**, pre- and post-conditions

Things to consider:

- Comments are **not for translating** program statements
- If the code is hard to read, **consider rewriting** it

Line Length

Characters Per Line

- Plain text (RFC 678): 72
- Python: 79
- GNU: 80
- Google: 80
- My (biased) opinion: **78**

Documentation

- Requirements
- Developer’s Manual
  - Program Design/Model
  - Implementation Details
  - Programming Interface
- User Manual

Organizing Data

Data Input/Output

Instead of computing just on student’s grade, what if we want to compute grades from a file that list students' names and grades like:

```
Smith 93 91 47 90 92 73 100 87
Carpenter 75 90 87 92 93 60 0 98
...
```

From the above input, the output would be:

```
Carpenter        86.8
Smith            90.4
...
```

Student’s Data

- Use **`struct`** to define a data structure that group related data together.
- We can define a data structure for student’s data as follows:

```
struct Student_info {
    string name;
    double midterm, final;
    vector<double> homework;
}; // note the semicolon
   // -- it's required
```

Alternatively:

```
struct Student_info {
    string name;
    double midterm;
    double final;
    vector<double> homework;
};
```

Managing the Records (1)

Reading the student’s name and exam grades:

```cpp
istream& read(istream& is, Student_info& s)
{
    // read and store the student's name and midterm and final exam grades
    is >> s.name >> s.midterm >> s.final;

    // read and store all the student's homework grades
    read_hw(is, s.homework);
    return is;
}
```

Computing the student’s grade:

```cpp
double grade(const Student_info& s)
{
    return grade(s.midterm, s.final, s.homework);
}
```

Managing the Records (2)

For sorting records, we need to write the comparison function for **`Student_info`**:

```
bool compare(const Student_info& x, const Student_info& y)
{
    return x.name < y.name;
}

    // ...
    vector<Student_info> students;

    // ...
    // alphabetize the records
    sort(students.begin(), students.end(), compare);
```

Iterating Over the Records

C++98

```
for (vector<Student_info>::size_type i = 0;
     i != students.size(); ++i) {

    // write the name, padded on the right to `maxlen + 1' characters
    cout << std::left << setw(maxlen + 1) << students[i].name;

    // ...
}
```

With **range-based for loop** (C++11 and later)

```
// `const auto&' will match `const Student_info&'
for (const auto& s: students) {
    cout << std::left << setw(maxlen + 1) << s.name;

    // ...
}
```

Generating the Report

```cpp
int main()    // grading2.cpp
{
    vector<Student_info> students;
    Student_info record;
    string::size_type maxlen = 0;

    // read and store all the records,
    // and find the length of the longest name
    while (read(cin, record)) {
        maxlen = max(maxlen, record.name.size());
        students.push_back(record);
    }

    // alphabetize the records
    sort(students.begin(), students.end(), compare);

    for (const auto& s: students) {
        // write the name, padded on the right to `maxlen + 1' characters
        cout << std::left << setw(maxlen + 1) << s.name;

        // compute and write the grade
        try {
            double final_grade = grade(s);
            streamsize prec = cout.precision();
            cout << setprecision(3) << final_grade
                 << setprecision(prec);
        }
        catch (const domain_error& e) {
            cout << e.what();
        }
        cout << endl;
    }
    return 0;
}
```

C++ Compilation Process

Compiling and Linking

```
  // success.cpp
int main()
{
    return 0;
}
```

```
  // failed.cpp
int main()
{
    return 1;
}
```

Example Session

```
$ g++ -c -Wall -Wextra success.cpp
$ g++ -o success success.o
$ ./success
$ echo $?
0
$ g++ -c -Wall -Wextra failed.cpp -o failed.o
$ g++ -o failed failed.o
$ ./failed
$ echo $?
1
```

```
# To compile each source file
g++ -c <flags> <source>.cpp -o <object>.o

# To link object files into an executable
g++ -o <exe> <obj1>.o <obj2>.o ...
```

Managing Projects with CMake (1)

```
cmake_minimum_required(VERSION 3.10)

project(example1)

add_executable(success success.cpp)
add_executable(failed failed.cpp)

# set C++ standard
set_target_properties(
    success failed
    PROPERTIES
    CXX_STANDARD 17
    CXX_STANDARD_REQUIRED YES
    CXX_EXTENSIONS NO)

# add more warning to the compiler options
if (MSVC)
    target_compile_options(success PRIVATE /Wall /WX)
    target_compile_options(failed PRIVATE /Wall /WX)
else()
    target_compile_options(success PRIVATE -Wall -Wextra)
    target_compile_options(failed PRIVATE -Wall -Wextra)
endif()
```

Managing Projects with CMake (2)

Set a CMake variable to avoid repeating commands:

```
set(TARGETS success failed)

set_target_properties(
    ${TARGETS}
    PROPERTIES
    CXX_STANDARD 17
    CXX_STANDARD_REQUIRED YES
    CXX_EXTENSIONS NO)

foreach(T ${TARGETS})
    if (MSVC)
        target_compile_options(${T} PRIVATE /Wall /WX)
    else()
        target_compile_options(${T} PRIVATE -Wall -Wextra)
    endif()
endforeach()
```

Configure and Build with CMake

To **configure and generate** build files for a project:

```
$ mkdir build
$ cd build
$ cmake -GNinja ../example1
```

To **build** a project:

```
$ cmake --build .
```

In the above example session, the directory layout created after the session is as follows:

```
- <root-dir>
  - build
  - example1
```

Source and Header Files

```
#ifndef ACPP_MEDIAN_HPP
#define ACPP_MEDIAN_HPP

// `median.hpp' -- final version
#include <vector>

double median(std::vector<double>);

#endif /* ACPP_MEDIAN_HPP */
```

```
#include "median.hpp"

// `median.cpp': source file for the `median' function
#include <algorithm>    // to get the declaration of `sort'
#include <stdexcept>    // to get the declaration of `domain_error'
#include <vector>       // to get the declaration of `vector'

using std::domain_error;    using std::sort;    using std::vector;

// compute the median of a `vector<double>'
// note that calling this function copies the entire argument `vector'
double median(vector<double> vec)
{
    // function body as shown previously
}
```

Requesting Access to the Interface

```
  // use `median' function
#include "median.hpp"
#include <vector>

int main()
{
    // ...

    median(hw);

    // ...
}
```

This is seen by the compiler:

```
  // declarations from
  // #include <vector>

double median(std::vector<double>);

  // declarations from
  // #include <vector>
  // (normally guarded by #ifndef)

int main()
{
    // ...

    median(hw);

    // ...
}
```

`#ifndef` Guard Pattern

In every header file, we usually use **`#ifndef`** pattern to guard against multiple inclusions of the header contents into the same source code:

```
#ifndef SOME_UNIQUE_NAME
#define SOME_UNIQUE_NAME

// ...

#endif /* SOME_UNIQUE_NAME */
```

We must ensure that **SOME_UNIQUE_NAME** is **really unique** throughout the entire application project.

The Revised Grading Program (1)

```cpp
#include "student_info.hpp"
#include "grade.hpp"

#include <algorithm>
#include <iomanip>
#include <ios>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>

using std::cin;     using std::setprecision;
using std::cout;    using std::sort;
using std::domain_error;
using std::streamsize;
using std::endl;    using std::string;
using std::max;     using std::vector;
using std::setw;

int main()    // grading3.cpp
{
    vector<Student_info> students;
    Student_info record;

    // the length of the longest name
    string::size_type maxlen = 0;

    // read and store all the students' data.
    // Invariant: `students' contains all
    //                       the student records
    //                       read so far
    //            `maxlen' contains the length
    //                     of the longest name
    //                     in `students'
    // ...
```

```cpp
    // ...
    while (read(cin, record)) {
        // find the length of the longest name
        maxlen = max(maxlen, record.name.size());
        students.push_back(record);
    }

    // alphabetize the student records
    sort(students.begin(), students.end(), compare);

    // write the names and grades
    for (const auto& s: students) {
        // write the name, padded on the right
        // to `maxlen + 1' characters
        cout << std::left << setw(maxlen + 1) << s.name;

        // compute and write the grade
        try {
            double final_grade = grade(s);
            streamsize prec = cout.precision();
            cout << setprecision(3) << final_grade
                 << setprecision(prec);
        }
        catch (const domain_error& e) {
            cout << e.what();
        }
        cout << endl;
    }
    return 0;
}
```

The Revised Grading Program (2)

`Student_info` structure and related functions

```cpp
#ifndef ACPP_STUDENT_INFO_HPP
#define ACPP_STUDENT_INFO_HPP

  // `student_info.hpp' header file
#include <iostream>
#include <string>
#include <vector>

struct Student_info {
    std::string name;
    double midterm, final;
    std::vector<double> homework;
};

bool compare(
    const Student_info&,
    const Student_info&);
std::istream& read(
    std::istream&, Student_info&);
std::istream& read_hw(
    std::istream&, std::vector<double>&);

#endif /* ACPP_STUDENT_INFO_HPP */
```

```cpp
  // source file for `Student_info'-related functions
#include "student_info.hpp"

using std::istream;  using std::vector;

bool compare(
    const Student_info& x, const Student_info& y)
{
    return x.name < y.name;
}

istream& read(istream& is, Student_info& s)
{
    // ...
}

  // read homework grades from an input stream
  // into a `vector<double>'
istream& read_hw(istream& in, vector<double>& hw)
{
    // ...
}
```

The Revised Grading Program (3)

`grade` functions

```cpp
#ifndef ACPP_GRADE_HPP
#define ACPP_GRADE_HPP

  // `grade.hpp'
#include "student_info.hpp"
#include <vector>

double grade(double, double, double);
double grade(
    double, double,
    const std::vector<double>&);
double grade(const Student_info&);

#endif /* ACPP_GRADE_HPP */
```

```cpp
#include "grade.hpp"
#include "student_info.hpp"

  // median.hpp and median.cpp is as shown previously
#include "median.hpp"

#include <vector>
#include <stdexcept>

using std::domain_error;  using std::vector;

  // ...
double grade(
    double midterm, double final, double homework)
{
    return 0.2 * midterm + 0.4 * final + 0.4 * homework;
}

  // ...
double grade(
    double midterm, double final,
    const vector<double>& hw)
{
    // ...
}

double grade(const Student_info& s)
{
    return grade(s.midterm, s.final, s.homework);
}
```

The Revised Grading Program (4)

CMake project file:

```
cmake_minimum_required(VERSION 3.10)

project(acpp_ch04)

add_executable(grading1 grading1.cpp)
add_executable(grading2 grading2.cpp)
add_executable(
    grading3
    grading3.cpp grade.cpp median.cpp student_info.cpp)

set(TARGETS grading1 grading2 grading3)

set_target_properties(
    ${TARGETS}
    PROPERTIES
    CXX_STANDARD 17
    CXX_STANDARD_REQUIRED YES
    CXX_EXTENSIONS NO)

# ...
```

The Revised Grading Program (5)

CMake project file (cont'):

```
# ...

foreach(T ${TARGETS})
    if (MSVC)
        target_compile_options(${T} PRIVATE /Wall /WX)
    else()
        target_compile_options(${T} PRIVATE -Wall -Wextra)
    endif()
endforeach()

# ...
```

The Revised Grading Program (6)

CMake project file (cont') (for copying the data files):

```
# ...

set(DATA_FILES single_grade grades)

foreach(FN ${DATA_FILES})
add_custom_command(
    OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/${FN}
    COMMAND ${CMAKE_COMMAND} -E copy
        ${CMAKE_CURRENT_SOURCE_DIR}/${FN}
        ${CMAKE_CURRENT_BINARY_DIR}/${FN}
    DEPENDS ${FN})

add_custom_target(
    data-${FN} ALL DEPENDS ${CMAKE_CURRENT_BINARY_DIR}/${FN})
endforeach()
```

Building and Running the Program

To **configure and generate** build files for a project:

```
$ mkdir build
$ cd build
$ cmake -GNinja ../acpp-ch04
$ cmake --build .
```

Using I/O redirection to avoid typing the same input over multiple runs:

```
$ ./grading3 < grades
```