Programming Principle of the Day

• Abstraction Principle
  • Related to DRY as it aims to reduce duplication
  • Each significant piece of functionality in a program should be implemented in just one place in the source code.
  • The basic mechanism of control abstraction is a function or subroutine.
  • Data abstractions include various forms of type polymorphism.

http://en.wikipedia.org/wiki/Abstraction_principle_(programming)
Abstract

• We will discuss data structures as the main means of organizing data. We extend the student grades example to work with more than one student’s grades.

• We will introduce the concept of partitioning the program into several files, separate compilation, and linking.
Language technicalities

• Are a necessary evil
  • A programming language is a foreign language
  • When learning a foreign language, you have to look at the grammar and vocabulary

• Because:
  • Programs must be precisely and completely specified
    • A computer is a very stupid (though very fast) machine
    • A computer can’t guess what you “really meant to say” (and shouldn’t try to)
  • So we must know the rules
    • Some of them (the C++ standard is 782 pages)

• However, never forget that
  • What we study is programming
  • Our output is programs/systems
  • A programming language is only a tool
Technicalities

• Don’t spend your time on minor syntax and semantic issues. There is more than one way to say everything
  • Just like in English

• Most design and programming concepts are universal, or at least very widely supported by popular programming languages
  • So what you learn using C++ you can use with many other languages

• Language technicalities are specific to a given language
  • But many of the technicalities from C++ presented here have obvious counterparts in C, Java, C#, etc.
Organizing Data

• Let’s revise student grades program for a whole course (many students)

• Read grades from a file:
  Smith 93 91 47 90 92 73 100 87
  Carpenter 75 90 87 92 93 60 0 98

• We want to produce output (overall grade)
  Carpenter 90.4
  Smith 86.8

• Alphabetical, formatting vertically lining up
Organizing Data

- Need to store all student data
  - Sorted by name
  - Line up: find longest name

- Let’s assume we can store all data about one student (student_info)
  - All students data: vector<student_info>

- Set of auxiliary functions to work with that data
  - Solve the overall problem using those
Organizing Data

• We need to hold all data items related to one student together:

```cpp
// hold all information related to a single student
struct student_info
{
    string name;  // students name
    double midterm, final;  // midterm and final exam grades
    vector<double> homework;  // all homework grades
};
```

• This is a new type holding four items (members)
  • We can use this type to define new objects of this type
  • We can store the information about all students in a

```cpp
vector<student_info> students;
```
Reading Data for one Student

• Very similar to what we already have:

```cpp
// read all information related to one student
istream& read(istream& in, student_info& s)
{
    // read the students name, midterm and final exam grades
    in >> s.name >> s.midterm >> s.final;
    // read all homework grades for this student
    return read_hw(in, s.homework);
}
```

• Any input error will cause all subsequent input to fail as well
  • Can be called repeatedly
Reading all Student Records

• Invoke read() as long as we succeed:

```cpp
vector<student_info> students; // all student records
string::size_type maxlen = 0; // length of longest name

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

• Function `std::max()` is peculiar
  - Both arguments need to have same type
Calculate Final Grade

• Again, we rely on existing function:
  
  // Calculate the final grade for one student
  double grade(student_info const& s)
  {
      return grade(s.midterm, s.final, s.homework);
  }

• Any exception thrown by underlying function grade() will be left alone
  • Nothing else to do
  • Handling left to whatever is calling this function
Sort Student Data

• We know that sorting can be done using sort():
  
  ```cpp
  vector<double> vec;
  sort(vec.begin(), vec.end());
  ```

• Let’s do the same for all students:

  ```cpp
  vector<student_info> students;
  sort(students.begin(), students.end());
  ```

• Not quite right, why?
  • What criteria to use for sorting?
  • What does it mean to sort the vector of students?
  • How to express the need to sort ‘by name’?
Sorting Student Data

- Normally, sort() uses the operator < to determine order
  - Makes no sense for student_info’s!

- We can teach sort() how to order by specifying a predicate
  - A function returning a bool taking two arguments of the type to be compared
  - Returns true if the first argument is smaller than the second (whatever that means)
Sorting Student Data

• We can teach sort() how to order by specifying a predicate:

```cpp
// compare two student_info instances, return whether 'x' // is smaller than 'y' based on comparing the stored names // of the students
bool compare(const student_info& x, const student_info& y) {
    return x.name < y.name;
}
```

• Now, we can use this function as:

```cpp
vector<student_info> students;
sort(students.begin(), students.end(), compare);
```
Sorting Student Data

• Alternatively, we could define an appropriate operator
  // compare two student_info instances, return whether 'x'
  // is smaller than 'y' based on comparing the stored names
  // of the students
  bool operator<(student_info const& x, student_info const& y)
  {
    return x.name < y.name;
  }

• Now, we can use this function as:
  vector<student_info> students;
  sort(students.begin(), students.end());
Sorting Student Data

- Alternative: lambda function (g++ 4.2, VC2010):

```
// sorting the student data using a lambda function
sort(students.begin(), students.end(),
     [](student_info const& x, student_info const& y)
     {
       return x.name < y.name;
     });
```

- Much nicer! Everything is in one place

- Note: no return type
  - Although, it could be specified (-→ bool)
  - Possible if body is ‘simple’ (one return statement)
Generating the Report

• Now we’re ready to generate the report:

```cpp
for (vector<student_info>::size_type i = 0; i != students.size(); ++i)
{
    // write the name, padded on the right side to maxlen + 1 characters
    cout << students[i].name
        << string(maxlen + 1 - students[i].name.size(), ' ');

    // compute and write the grade
    try {
        double final_grade = grade(students[i]);
        streamsize prec = cout.precision();
        cout << setprecision(3) << final_grade << setprecision(prec);
    }
    catch (domain_error e) {
        cout << e.what();
    }
    cout << endl;
}
```
Generating the Report

Now we’re ready to generate the report:

```cpp
for (student_info const& si: students)
{
    // write the name, padded on the right side to maxlen + 1 characters
    cout << si.name
        << string(maxlen + 1 - si.name.size(), ' ');

    // compute and write the grade
    try {
        double final_grade = grade(si);
        streamsize prec = cout.precision();
        cout << setprecision(3) << final_grade << setprecision(prec);
    } catch (domain_error e) {
        cout << e.what();
    }
    cout << endl;
}
```
Temporary Objects

• We can create nameless (temporary objects):

```c++
string(maxlen + 1 - students[i].name.size(), ' ')
```

• Creates a string of spaces padding the name to length maxlen + 1
• Object has no name
  • It’s a temporary object, which ‘lives’ until the semicolon
  • Equivalent to creating a named object instead
Putting it all together

• Look at that source file!
  • Even if the program is not very complex, the file is too large already

• Separate compilation
  • Example: the median function
  • It’s useful on its own, even outside the scope of this particular program

• Put part of sources into separate file, which is compiled separately
Separate Compilation

- File median.cpp (source or implementation file):

```cpp
#include <vector>    // std::vector
#include <stdexcept> // std::domain_error
#include <algorithm> // std::sort, std::max
#include "median.hpp"

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

// definition of function median
double median(vector<double> vec)
{
    // ...
}
```
Separate Compilation

• File median.hpp (header file):

```cpp
#include <vector>   // std::vector
// declaration of function median
double median(std::vector<double>);
```

• Makes function median available to other files:

```cpp
#include <vector>
#include "median.hpp"

int main() { /* use median() */ }
```
Header File Guards

• Ensure that inclusion of header more than once does no harm:

```cpp
#ifndef GUARD_MEDIAN_HPP
#define GUARD_MEDIAN_HPP

#include <vector>            // std::vector
// declaration of function median
double median(std::vector<double>);

#endif
```
Source files

- A header file (here, median.hpp) defines an interface between user code and implementation code (usually in a library)
- The same #include declarations in both .cpp files (definitions and uses) ease consistency checking
Compile Single Source File

Compile: g++ -std=c++11 -o main main.cpp

- Input is ‘main.cpp’, source code
  - Human readable text

- Output is ‘main’, executable binary (-o output_name)
  - Machine readable binary code
  - All externals resolved
  - Relocatable code (not bound to a particular address)
    - Binding to memory address is done by OS loader

- Behind the scenes:
  - Creating binary object file
  - Looking up used symbols in standard libraries known to the compiler
Compile Single Source File

**Compile:** `g++ -std=c++11 -c main.cpp`

- Input is ‘main.cpp’, source code
- Output is ‘main.o’, object binary
  - Machine readable code
  - Relocatable code
  - Externals **not** resolved

**Link:** `ld –o main main.o -lstdc++`

- Input is ‘main.o’, object binary
- Input is ‘libstdc++.a’, C++ runtime library (.a stands for ‘archive’)
  - `-lstdc++` is shortcut for `libstdc++.a`
- Output is ‘main’, executable binary

- Behind the scenes:
  - Linker knows where to look for standard libraries
Compile Multiple Source Files

Compile: `g++ -std=c++11 -o main main.cpp module.cpp`

- Input files are ‘main.cpp’ and module.cpp, source code
  - Human readable text

- Output is ‘main’, executable binary (-o output_name)
  - Machine readable binary code
  - All externals resolved
  - Relocatable code (not bound to a particular address)
    - Binding to memory address is done by OS loader

- Behind the scenes:
  - Creating binary object files
  - Looking up used symbols in standard libraries known to the compiler
  - Resolving symbols between the input files
Compile Multiple Source Files

**Compile:** `g++ -c main.cpp module.cpp`

- Input file are ‘main.cpp’ and ‘module.cpp’, source code
- Main contains references to symbols from module.cpp (or v.v.)
- Output files are ‘main.o’ and ‘module.o’, object binaries
  - Machine readable code
  - Relocatable code
  - Externals **not** resolved

**Link:** `ld -o main main.o module.o -lstdc++`

- Input files are ‘main.o’ and ‘module.o’, object binaries
- Input is ‘libstdc++.a’, C++ runtime library (.a stands for ‘archive’)
  - `-lstdc++` is shortcut for `libstdc++.a`
- Output is ‘main’, executable binary

- Behind the scenes:
  - Linker knows where to look for standard libraries