4 Application: Grading a class

In this lecture, we provide a first introduction to *class design* in C++. The program developed in this chapter is similar, but not identical, to the grading program described in Chapter 4 of (Koenig and Moo 2000).

4.1 Problem Statement

Here is a statement of the problem that the grading program solves:

> The program reads a file of marks (e.g., Figure 31 on the next page) and writes a report of grades (e.g., Figure 32 on the following page). Each line of the marks file consists of a name, a mark for the midterm, a mark for the final, and marks for assignments. The number of assignments is not fixed; students do as many as they want to. The name is a single name with no embedded blanks.
>
> A line of the output file is similar to a line of the input file, but the first number is the total mark, computed as 30% of the midterm mark plus 60% of the final mark plus the median of the assignments. The output is written twice, once sorted by name, and once sorted by total mark.

4.2 First version

Figure 33 on page 65 shows the main program. The program is designed with two objectives in mind:

- Use an object to store a student record
- Put as much problem-specific information as possible into the corresponding class

The first paragraph of the program asks the user for a file name and tries to open the file. The second paragraph declares the principal data object of the program, a vector of `Students`. From this paragraph, we can tell that the `Student` class must provide a reading capability (`>>`) and a method `process` to compute the final mark.

It is important that the argument to `push_back` is ***passed by value***. If it was passed by reference, each entry in the vector `classData` would refer to the same local variable, `stud`!

The last part of the program opens an output file and writes the data to it twice, first sorted by name and then sorted by total marks. From this section, we deduce that the `Student` class must

provide two sorting functions, `ltNames` and `ltMarks`. We also need a free function `showClass` to write the class list.

102 The function `showClass` is straightforward; it is shown in Figure 34 on the facing page. The output stream `os` is ***passed by reference*** because the function will change it when writing. The student data is ***passed by constant reference*** because it will not be changed by the function. It uses an iterator to traverse the vector of marks data. The statement

```
os << *it << endl;
```

requires class `Student` to provide an ***inserter*** – a function that overloads the operator (`<<`).

```
Thomas  47 83 9 8 4 7 6 9 8
Georges 36 88 8 6 7 4 9 7 8 7
Tien    49 91 9 6 7 8 5 6 7
Lei     41 82 8 8 8 8
Oanh    45 76 9 9 8 9 9 8 8 9
Lazybones  31 45
Mohamad 39 99 8 6 7 9 5 6 9
Jane    36 64 7 5 8
```

Figure 31: Input for grading program

```
Sorted by name:
Georges    67.5 36 88  8  6  7  4  9  7  8  7
Jane       52.6 36 64  7  5  8
Lazybones  33.2 31 45
Lei        65.4 41 82  8  8  8  8
Mohamad    74.2 39 99  8  6  7  9  5  6  9
Oanh       63.6 45 76  9  9  8  9  9  8  8  9
Thomas     67.2 47 83  9  8  4  7  6  9  8
Tien       71.4 49 91  9  6  7  8  5  6  7

Sorted by marks:
Lazybones  33.2 31 45
Jane       52.6 36 64  7  5  8
Oanh       63.6 45 76  9  9  8  9  9  8  8  9
Lei        65.4 41 82  8  8  8  8
Thomas     67.2 47 83  9  8  4  7  6  9  8
Georges    67.5 36 88  8  6  7  4  9  7  8  7
Tien       71.4 49 91  9  6  7  8  5  6  7
Mohamad    74.2 39 99  8  6  7  9  5  6  9
```

Figure 32: Output from grading program

103

Figure 35 on page 66 shows the declaration for class `Student`. There are several points to note:

```
int main()
{
    string classFileName;
    cout << "Please enter class file name: ";
    cin >> classFileName;
    ifstream ifs(classFileName.c_str());
    if (!ifs)
    {
        cerr << "Failed to open " << classFileName << endl;
        return 1;
    }

    vector<Student> classData;
    Student stud;
    while (ifs >> stud)
    {
        stud.process();
        classData.push_back(stud);
    }

    ofstream ofs("grades.txt");
    sort(classData.begin(), classData.end(), ltNames);
    ofs << "Sorted by name:\n";
    showClass(ofs, classData);

    sort(classData.begin(), classData.end(), ltMarks);
    ofs << "\nSorted by marks:\n";
    showClass(ofs, classData);
}
```

Figure 33: The main program

```
void showClass(ostream & os, const vector<Student> & classData)
{
    for (  vector<Student>::const_iterator it = classData.begin();
           it != classData.end();
           ++it )
        os << *it << endl;
}
```

Figure 34: Function `showClass`

```
class Student
{
    friend ostream & operator<<(ostream & os, const Student & stud);
    friend istream & operator>>(istream & is, Student & stud);
    friend bool ltNames(const Student & left, const Student & right);
    friend bool ltMarks(const Student & left, const Student & right);
public:
    void process();
private:
    static string::size_type maxNameLen;
    string name;
    int midterm;
    int final;
    vector<int> assignments;
    double total;
};
```

Figure 35: Class `Student`

- In C++, the declaration of a class and the definitions of its functions are separate. The function definitions may be – and often are – in a different file (as described later in this chapter).

`friend` functions

- A class declaration may introduce functions as `friend`s. Although these functions are not member functions, they have access to the classes' private data.

`public` and `private` visibility modifiers

- The declarations `public` and `private` introduce a ***group*** of declarations with the given accessibility (unlike in Java, where each member has its own visibility modifier).

  Some authors put the `private` declarations before the `public` declarations. This seems backwards: the users of a class need to know about only the `public` attributes and these should therefore appear ***first***.[17]

- Functions declared within a class are called ***member functions***. Functions declared outside a class are called ***free functions***.[18] Member functions can be called only with an object, as in `obj.fun()`. Free functions are called without an object, as in `fun()`.

default constructor performs no initializations

- There is no constructor. We rely on the default constructor provided by the compiler; this constructor allocates space for the object but performs no other initialization. When we define a new instance of `Student`, we must ensure that all fields are correctly initialized.

- There is only one public method, `process`, which performs any necessary computation on the data read from the marks file.

- The private data includes the information that is read from the marks file (`name`, `midterm`, `final`, and `assignments`) and computed information, `total`.

[17] It would be even better if the `private` attributes could be hidden from users altogether. Although C++ does not allow that, documentation tools such as *Doxygen* (see Section B.3) can provide the required effect.

[18] Note that Java does not have free functions. However, classes such as `Math` provide static functions that are effectively the same as free functions. In Java, you write `Math.sqrt(x)`, in C++, you just write `sqrt(x)`.

- For formatting the output, we need to know the length of the longest name. This is an attribute of the class, not the object, and so it is declared as a `static` data member. Memory for static variables is automatically allocated at program start and cleaned up when it terminates.

  static data members

- We need methods for input (`>>`) and output (`<<`); these are declared as friends.
- We need comparison functions that will be used for sorting: `ltNames` orders by student's names, and `ltMarks` orders by student's total marks.

  There is an important design choice here. The four `friend` functions cannot be member functions, because of the way they are called. The alternatives are either to provide access functions to private data in the class or to declare these functions as `friend`s. Access functions should be avoided if possible, especially functions with write access, as would be required for `>>`. Although `friend` functions should be used only where necessary, they sometimes provide better encapsulation, as in this case.[19]

  `friend` functions vs. member access functions

- A class declaration – just like any other declaration – must end with a '`;`'

  don't forget the '`;`' at the end of a class declaration!

The next step is to complete the implementation of class `Student` by providing definitions for functions and initial values for static variables. The static data member must be initialized like this:

initializing `static` data members

```
string::size_type Student::maxNameLen = 0;
```

This is the ***only*** way to initialize a static data member. It has the form of a declaration rather than an assignment (the type is included) and it appears at global scope, that is, outside any function.

The public function `process` has two tasks: it maintains the length of the longest name seen so far and it calculates the total mark. Calculating the total mark requires finding the median of the assignments. The median is meaningless for an empty vector, and the median function requires a non-empty vector as its argument (see Figure 37 on the following page). Thus `process`, shown in Figure 36, calls `median` only if the student has done at least one assignment.

104

```
void Student::process()
{
    if (maxNameLen < name.size())
        maxNameLen = name.size();
    total = 0.3 * midterm + 0.6 * final;
    if (assignments.size() > 0)
        total += median(assignments);
}
```

Figure 36: Function `process` for class `Student`

In general a function should not perform two unrelated tasks, as `process` does. The rationale in this case is that `process` performs ***all*** of the processing that is needed for one student record. There might be more tasks to perform than just these two. An alternative would be to define

[19] We will discuss the undesirability of access functions later in the course, when we address class design issues.

two functions, one to update `maxNameLen` and the other to calculate `total`. These two functions would always be called together, so it makes sense to combine them into a single function.

As a general principle, it should always be possible to explain the purpose of a function with a one-line description. If you need three sentences to say what a function does, there's probably something wrong with it. We could describe the purpose of function `process` as "perform all calculations needed to generate the marks file".

> ***Every function should have a complete, one-line description.***

105 The median calculation is performed by the function in Figure 37. The main design issue for this function is how to pass the vector of scores. Since we have to sort the vector in order to find the median, we cannot pass it by constant reference. If we pass it by reference, the caller will get back a sorted vector. Although this does not matter much for this program, a function should not in general change the data it is given unless the caller needs the changed value. Consequently, we choose to pass the vector by value, incurring the cost of copying it.

Finally, note that `median` has a precondition: it does not accept an empty vector. The only use of `median` in this program is in the context

```
if (assignments.size() > 0)
    total += median(assignments);
```

It follows that, if the assertion fails, there is a logical error in the program.

Perhaps, after this program has been used for a few years, another programmer decides to extend it. The function `median` is called from another location, without the check for an empty vector. During testing, the assertion fails, and the programmer immediately sees the problem with the extension.

```
// Requires: scores.size() > 0.
double median(vector<int> scores)
{
    typedef vector<int>::size_type szt;
    szt size = scores.size();
    assert(size > 0);
    sort(scores.begin(), scores.end());
    szt mid = size / 2;
    return size % 2 == 0 ?
        0.5 * (scores[mid - 1] + scores[mid]) :
        scores[mid];
}
```

Figure 37: Finding the median

106

Figure 38 on the next page shows the comparison functions that we need for sorting. The parameter lists of these functions are determined by the requirements of the `sort` algorithm:

there must be two parameters of the same type, both passed by constant reference. Since we have declared these functions as `friend`s of `Student`, they have access to `Student`'s private data members. The type of `name` is `string` and the type of `total` is `double`; both of these types provide the comparison operator `<`.

After sorting, the records will be arranged in increasing order for the keys. Names will be alphabetical: `Anne`, `Bo`, `Colleen`, `Dingbat`, etc. Records sorted by marks will go from lowest mark to highest mark. To reverse this order, putting the students with highest marks at the "top" of the class, all we have to do is change "`<`" to "`>`" in `ltMarks`. As a courtesy to readers, it would also be a good idea to change the name to `gtMarks`.

```
bool ltNames(const Student & left, const Student & right)
{
    return left.name < right.name;
}

bool ltMarks(const Student & left, const Student & right)
{
    return left.total < right.total;
}
```

Figure 38: Comparison functions

The compiler has to perform a number of steps to determine that these functions are called by the statements

```
sort(classData.begin(), classData.end(), ltNames);
sort(classData.begin(), classData.end(), ltMarks);
```

The reasoning goes something like this:

1. The type of the argument `classData.begin()` is `vector<Student>::const_iterator`
2. The compiler infers from this that the elements to be sorted are of type `Student`
3. The comparison functions must therefore have parameters of type `const & Student`
4. There are functions `ltNames` and `ltMarks` with parameters of the correct type 107

Figure 39 on the following page shows the extractor (`>>`) for class `Student`. It is a bit tricky, because we rely on the failure management of input streams. The key problem is this: since students complete different numbers of assignments, how do we know when we have read all the assignments? The method we use depends on what follows the last assignment: it is either the name of the next student or the end of the file. If we attempt to read assignments as numbers, either of these will cause reading to fail. Consequently, we can use the following code to read the assignments:

```
while (ifs >> mark)
    stud.assignments.push_back(mark);
```

```
istream & operator>>(istream & ifs, Student & stud)
{
    if (ifs >> stud.name)
    {
        ifs >> stud.midterm >> stud.final;
        int mark;
        stud.assignments.clear();
        while (ifs >> mark)
            stud.assignments.push_back(mark);
        ifs.clear();
    }
    return ifs;
}
```

Figure 39: Extractor for class `Student`

However, we must not leave the stream in a bad state, because this would prevent anything else being read. Therefore, when the loop terminates, we call

```
ifs.clear();
```

to reset the state of the input stream.

We assume that, if a student name can be read successfully, the rest of the record is also readable. If the name is ***not*** read successfully, the function immediately returns the input stream in a bad state, telling the user that we have encountered end of file.

What happens if there is a format error in the input? Some markers, although they are asked to provide integer marks only, include fractions. Suppose that the input file contains this line:

```
Joe    45 76 9 9 8.5 9 9 8 8 9
```

The corresponding output file contains these lines:

```
Joe       63.6 45 76  9  9  8
.5         15.2  9  9  8  8  9
```

We see that Joe has lost all his assignment marks after 8.5 and we have a new student named "`.5`". It is clear that, if this was a production program, we would have to do more input validation.

108 The inserter (`<<`) in Figure 40 on the next page does not have these complications. The main points to note are:

- The manipulators:
  - `left` aligns text to the left
  - `right` (the default) aligns text to the right
  - `fixed` chooses fixed-point (as opposed to scientific) format for floating-point numbers

```
ostream & operator<<(ostream & os, const Student & stud)
{
    os <<
        left << setw(static_cast<streamsize>(Student::maxNameLen)) <<
        stud.name << right <<
        fixed << setprecision(1) << setw(6) << stud.total <<
        setw(3) << stud.midterm <<
        setw(3) << stud.final;
    for (  vector<int>::const_iterator it = stud.assignments.begin();
           it != stud.assignments.end();
           ++it )
       os << setw(3) << *it;
    return os;
}
```

Figure 40: Inserter for class `Student`

- `setprecision(1)` requests one decimal digit after the decimal point
- `setw(`n`)` requests a field width of N characters

- We use the longest name to align columns. The type of the variable `Student::maxNameLen` is `string::size_type` but the type expected by `setw` is `std::streamsize`. To avoid warnings from the compiler, we cast the type. Since the cast can be performed at compile time, we use a ***static cast***: static_cast

  ```
  static_cast<streamsize>(Student::maxNameLen)
  ```

- We use an iterator to output the assignment marks.

Extractors and Inserters. All extractors and inserters follow the same pattern: 109

```
istream & operator>>(istream & is, T & val)
{
    // perform read operations for fields of T
    return is;
}

ostream & operator<<(ostream & os, const T & val)
{
    // perform write operations for fields of T
    return os;
}
```

defining operator>> and operator<<

In both cases, the function is passed a reference to a stream and returns a reference to a stream. In fact, of course, both references are to the same stream, but the state of the stream changes during the operation. The second argument for the extractor is passed by reference, because its value will be updated when the stream is read. The second argument for the inserter is passed

by constant reference, because the inserter should not change it. When writing inserters and extractors, remember to return the updated stream.

110 Compiling this program requires the inclusions shown in Figure 41. The comments are not necessary, because experienced C++ programmers are familiar with these names.

```
#include <algorithm>  // sort
#include <cassert>    // assertions
#include <fstream>    // input and output file streams
#include <iomanip>    // stream manipulators
#include <iostream>   // input and output streams
#include <string>     // STL string class
#include <vector>     // STL vector class
```

Figure 41: Directives required for the grading program

4.3 Program Structure

translation unit

A C++ program consists of ***header files*** and ***implementation files***. The program is compiled as a collection of translation units. A ***translation unit*** normally consists of a single implementation file that may `#include` several header files. This is known as *separation of interface and implementation* (Weiss 2004, Section 4.11).

Building a program is a process that consists of compiling each translation unit and linking the resulting object files. A ***build*** is the result of building. Some companies have a policy such as "daily build" to ensure that an application under development can always be compiled and passes basic tests.

4.3.1 Header Files

Header files contain declarations but not definitions. This implies that the compiler:

1. does not generate any code while processing a header file
2. may read a header more than once during a build

If a header file that contains a definition such as

```
int k;
```

never put definitions into header files!

is read more than once during a build, the linker will complain that `k` is redefined and will not link the program. This is why it is important not to put definitions into header files.

Include Guards. Although a header file may be read more than once during a build, a header file should be read only once during the compilation of a translation unit. Suppose that translation unit A includes headers for translation units B and C, and these units both include `utilities.h`. To prevent the compiler from reading `utilities.h` twice, we write it in the following way: 111

```
#ifndef UTILITIES_H
#define UTILITIES_H

// Declarations for utilities.h

#endif
```

This is the standard pattern for ***all*** header files. The directives are called "include guards" (Weiss 2004, Section 4.12). You do not have to use the exact name `UTILITIES_H`, but it is important to choose a name that is unique and has some obvious connection to the name of the header file. For example, (Koenig and Moo 2000) uses `GUARD_utilities_h`. include guards

In most cases, the guards are not logically necessary. Since header files contain only declarations, reading them more than once should not cause errors. Some header files, however, cannot be read twice, and these can cause problems if they don't have guards. A more important reason for using guards is efficiency: header files can be very long, and they may include other header files. Without the guards, the compiler may be forced to read thousands of lines of declarations that it has seen before.

Instead of using include guards in header files, you can write just #pragma once

```
#pragma once
```

as the first line of a header file. This directive is recognized by most modern compilers and will sometimes be slightly faster than include guards. It is used in header files generated by VC++.NET. However, it is not completely portable, because compilers implement it in different ways. Include guards have been standard since the early days of C and always work correctly.

Typically, a header file will contain declarations for types, constants, functions, and classes.

A header file should `#include` anything that the compiler needs in order to process it. For example, if a class has a data member of type `string`, its header file must contain

```
#include <string>
```

It is not a good idea to include `using` declarations in header files. A header file may be included in many translation units that may not want namespaces opened for them.

> ***Do not write*** `using` ***declarations in header files.***

4.3.2 Implementation Files

Implementation files contain definitions for the objects declared in header files. An implementation file is processed only once during a build. An implementation file should `#include` its own header file and header files for anything else that it needs.

As a general rule, the first `#include` directive should name the header file corresponding to the
112 implementation file. For example, `utilities.cpp` would have the following structure:

```
#include "utilities.h"

// #includes for other components of this program

// #includes for library components

// definitions of the utilities
```

Header and implementation files create ***dependencies***, which are discussed in Section 4.5 on page 78 below. A header file may depend on other header files and an implementation file may depend on one or more header files. A file should never depend on an implementation file; in other words, you should never write

```
#include "something.cpp"
```

> ***Do not*** `#include` ***implementation files.***

When an implementation file `#include`s header files, the compiler obviously reads ***all*** of the files. Amongst other things, it checks that declarations in header files match definitions in implementation files. It is important to realize that the checking is not complete. For example,
113 the header file

```
#ifndef CONFLICT_H
#define CONFLICT_H

double mean(double values[]);

#endif
```

and the implementation file

```
#include "conflict.h"
#include <vector>

double mean(std::vector<double> values)
{
    // ....
}
```

will ***not*** produce any error messages. Since C++ allows functions to be overloaded, it assumes that the two versions of `mean` are different functions and that the vector version will be declared somewhere else. If the program calls either version, the linker will produce an error message.

4.4 Structuring the grading program

We split the grading program of Section 4.2 on page 63 into three translation units: 114

1. class `Student`
2. function `median`
3. the main program

The translation unit for function `median` is rather small, but it demonstrates the idea of splitting of generally useful functions in a larger application. For a small program such as this one, we could have put `Student` and `median` into the same implementation file.

4.4.1 Translation unit `Student`

115
116

Figure 42 on the next page shows the header file for class `Student`, `student.h`. There is an `#include` directive for each library type mentioned in the class declaration. The class declaration is unchanged from the original program.

Although function `showClass` is not a `friend` of class `Student`, it is closely related to the class; consequently, we put its declaration in `student.h`.

Figures 43 and 44 show the implementation file for class `Student`, `student.cpp`. The first `#include` directive includes `student.h`; this ensures that `student.h` does not depend on anything that it does not mention (if it did, the compiler would fail while reading it). Then we include `median.h` for this program, and finally the library types that we need. Since `student.h` includes `iostream`, `string`, and `vector`, we need only include `iomanip` for the output statements.

The implementation file `student.cpp`, shown in Figures 43 and 44 implements the member function of `Student`, `process`, and the `friend` functions. It also initializes the static data member `maxNameLen`. 117
118
119
120

4.4.2 Translation unit `median`

The header and implementation files for `median` are both short: see Figure 45 on page 78 and Figure 46 on page 79. In a more typical application, other useful functions might be incorporated into a single translation unit. 121
122

```
#ifndef STUDENT_H
#define STUDENT_H

#include <iostream>
#include <string>
#include <vector>

class Student
{
    friend std::ostream & operator<<(std::ostream & os,
        const Student & stud);
    friend std::istream & operator>>(std::istream & is,
        Student & stud);
    friend bool ltNames(const Student & left, const Student & right);
    friend bool ltMarks(const Student & left, const Student & right);
public:
    void process();
private:
    // Data read from file
    std::string name;
    int midterm;
    int final;
    std::vector<int> assignments;

    // Data computed by process
    double total;
    static std::string::size_type maxNameLen;
};

void showClass(std::ostream & os,
    const std::vector<Student> & classData);

bool ltNames(const Student & left, const Student & right);
bool ltMarks(const Student & left, const Student & right);

#endif
```

Figure 42: `student.h`: header file for class `Student`

4.4.3 Translation unit for the main program

123 124 125 The last step is to write an implementation file for the main program, `grader.cpp` (see Figure 48 on page 80). We do not need a header file (which would be called `grader.h`) because no other translation unit refers to anything in the main program. It is a good idea in general to avoid dependencies on the main program.

This translation unit includes only the header files for other translation units that it needs –

```
#include "student.h"
#include "median.h"

#include <iomanip>

using namespace std;

string::size_type Student::maxNameLen = 0;

void Student::process()
{
    if (maxNameLen < name.size())
        maxNameLen = name.size();
    total = 0.3 * midterm + 0.6 * final;
    if (assignments.size() > 0)
        total += median(assignments);
}

ostream & operator<<(ostream & os, const Student & stud)
{
    os <<
        left << setw(static_cast<streamsize>(Student::maxNameLen)) <<
        stud.name << right <<
        fixed << setprecision(1) << setw(6) << stud.total <<
        setw(3) << stud.midterm <<
        setw(3) << stud.final;
    for (  vector<int>::const_iterator it = stud.assignments.begin();
           it != stud.assignments.end();
           ++it )
       os << setw(3) << *it;
    return os;
}

istream & operator>>(istream & ifs, Student & stud)
{
    if (ifs >> stud.name)
    {
        ifs >> stud.midterm >> stud.final;
        int mark;
        stud.assignments.clear();
        while (ifs >> mark)
            stud.assignments.push_back(mark);
        ifs.clear();
    }
    return ifs;
}
```

Figure 43: `student.cpp`: implementation file for class `Student`: first part

```
bool ltNames(const Student & left, const Student & right)
{
    return left.name < right.name;
}

bool ltMarks(const Student & left, const Student & right)
{
    return left.total < right.total;
}

void showClass(ostream & os, const vector<Student> & classData)
{
    for (  vector<Student>::const_iterator it = classData.begin();
           it != classData.end();
           ++it )
        os << *it << endl;
}
```

Figure 44: `student.cpp`: implementation file for class `Student`: second part

```
#ifndef MEDIAN_H
#define MEDIAN_H

#include <vector>

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

#endif
```

Figure 45: `median.h`: header file for function `median`

`student.h` in this case – and any library types.

4.5 Dependencies

126

Figure 47 on the next page shows the dependencies between the files of the grading program. Dependencies on libraries are not shown. File X ***depends on*** file Y if the compiler must read Y in order to compile X. In general:

- Implementation files depend on header files
- Header files may depend on other header files
- An implementation file ***never*** depends on another implementation file
- There must be no circular dependencies
- Fewer dependencies are better (few dependencies = "low coupling")

low coupling

```
#include "median.h"

#include <algorithm>
#include <cassert>

using namespace std;

// Requires: scores.size() > 0.
double median(vector<int> scores)
{
    typedef vector<int>::size_type szt;
    szt size = scores.size();
    assert(size > 0);
    sort(scores.begin(), scores.end());
    szt mid = size / 2;
    return size % 2 == 0 ?
        0.5 * (scores[mid - 1] + scores[mid]) :
        scores[mid];
}
```

Figure 46: `median.cpp`: implementation file for function `median`

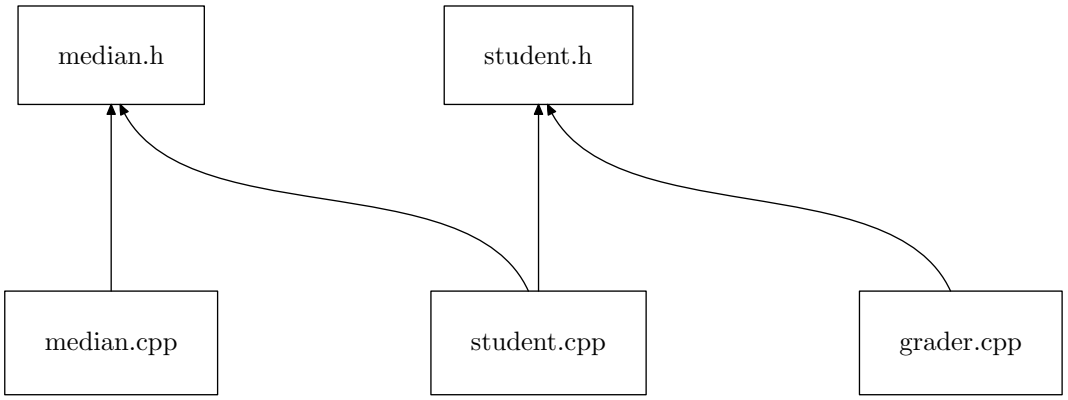

Figure 47: Dependencies in the grading program

dependencies and compilation time

Dependencies have an important effect on compilation time. A file with a high in-degree will trigger extensive recompilation when it is changed.

In Figure 47, a change to either `median.h` or `student.h` will cause two of the three implementation files to be recompiled. If `grader.cpp` depended on `median.h`, changing `median.h` would cause all three implementation files to be recompiled.

In a small program like the grader, the effect of dependencies on compilation is negligible. In large programs, the effect can be significant. Large programs require hours or even days to compile. Some header files are used by hundreds or even thousands of implementation files. A

```
#include "student.h"

#include <algorithm>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

int main()
{
    // Attempt to open file provided by user
    string classFileName;
    cout << "Please enter class file name: ";
    cin >> classFileName;
    ifstream ifs(classFileName.c_str());
    if (!ifs)
    {
        cerr << "Failed to open " << classFileName << endl;
        return 1;
    }

    // Process the file
    vector<Student> classData;
    Student stud;
    while (ifs >> stud)
    {
        stud.process();
        classData.push_back(stud);
    }

    // Print reports
    ofstream ofs("grades.txt");
    sort(classData.begin(), classData.end(), ltNames);
    ofs << "Sorted by name:\n";
    showClass(ofs, classData);

    sort(classData.begin(), classData.end(), ltMarks);
    ofs << "\nSorted by marks:\n";
    showClass(ofs, classData);
}
```

Figure 48: `grader.cpp`: implementation file for main program

change to one of the header files can trigger hours of recompilation time.

An important component of large-scale C++ design is to reduce the dependencies between source files. We will discuss ways to do this as the course progresses.

4.6 Strings and characters

Some more information on working with strings and characters.

Strings. Class `string` provides a large number of functions in addition to those that we have already seen. See http://www.cppreference.com/cppstring/all.html for a complete reference.

The following operators work for strings: 127

```
<    <=    ==    !=    >=    >    +    <<    >>    =    []
```

Operator `+` concatenates strings. There are several overloads, allowing for all combinations of `char`, C strings, and strings. Operator `[]` provides indexing for strings.

Strings function as containers for characters, working in a similar way to the type `vector<char>`. Consequently, iterators, `push_back`, and similar functions work for strings. In particular, `string` provides `insert` to insert characters into a string, `erase` to remove characters from a string, and `replace` to replace a sequence of characters in a string.

There are several functions for finding characters or substrings in strings. These functions usually return a position in the string in the form of an iterator. For example:

```
find
find_first_of
find_first_not_of
find_last_of
find_last_not_of
```

Characters. The library `cctype` provides the same functionality as the C header file `ctype.h`. Although many of these functions should now be considered obsolete (e.g., `strcpy` and friends), others are still useful. In particular: cctype library 128

`isalpha(c)` returns `true` if `c` is a letter

`isdigit(c)` returns `true` if `c` is a digit

`isspace(c)` returns `true` if `c` is a blank, tab, or linebreak

`tolower(c)` returns the lower case equivalent of an upper case character and leaves other characters unchanged

`toupper(c)` returns the upper case equivalent of an lower case character and leaves other characters unchanged

References

Koenig, A. and B. E. Moo (2000). *Accelerated C++: Practical Programming by Example*. Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.