

In this case the `c_str()` operation provided by the C++ string returns a C-style string, which can then be assigned to a variable of type `char*`.

Which String Representation to Use

Given that we have access to two different string representations, which string abstraction is best to use? In most cases, the programmer should choose the C++ string over the C-style string because it provides a safer abstraction. Although the C-style string enables representation of character data, it is still just an array of characters that can be accessed and changed in arbitrary ways. C++ provides a real-string data type with associated operations that work only for that data type. It captures the length of the string as part of its abstraction and alleviates the burden on the programmer to worry about how the string is implemented. When you must use functions that expect C-style strings, it is important to handle C-style strings appropriately or convert them into the C++ string data type.

QUICK CHECK



- 11.1.1 Why do we say that an array is a homogeneous data structure? (p. 518)
- 11.1.2 You are solving a problem that requires you to store 24 temperature readings. Which structure would be most appropriate for this problem: a record, a union, a class, or an array? (p. 518)
- 11.1.3 Write a declaration of an array variable called `temps` that holds 24 values of type `float`. (pp. 518–519)
- 11.1.4 Write a For loop that fills every element of the `temps` array declared in Exercise 11.1.3 with the value 32.0. (pp. 524–526)
- 11.1.5 How is a component of an array accessed? (p. 519)
- 11.1.6 A string in C++ can be treated like an array using the `[]` operator. Why is it preferred to use the `at` operation to access a character in a string? (p. 521)

1110912 Cen Li

11.2 Arrays of Records

Although arrays with atomic components are very common, many applications require a collection of records. For example, a business needs a list of parts records, and a teacher needs a list of students in a class. Arrays are ideal for these applications. We simply define an array whose components are records.

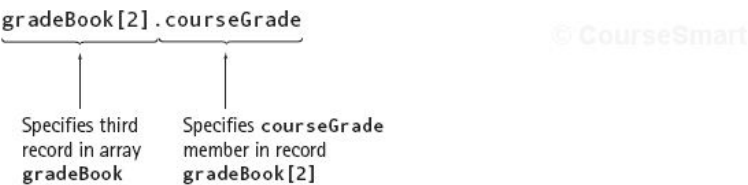
Arrays of Records

In Chapter 10, we defined a student record using a struct. We now need a collection of these student records. How do we implement a “collection”? By a file or an array. Because this chapter focuses on arrays, let's declare an array of student records and look at how each field would be accessed. To make things a little more interesting, let's add a user-defined type `GradeType` and an array of exam scores. As this is the beginning of a program, we show the structure as code.

```
//*****
// This program manipulates an array of grade records.
//*****
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
```

```
const int MAX_STUDENTS = 150;
enum GradeType {A, B, C, D, F};
struct StudentRec
{
    string stuName;
    float gpa;
    int examScore[4];
    GradeType courseGrade;
};
StudentRec gradeBook[MAX_STUDENTS];
int main()
{}
```

This data structure can be visualized as shown in **FIGURE 11.8**.
An element of `gradeBook` is selected by an index. For example, `gradeBook[2]` is the third component in the array `gradeBook`. Each component of `gradeBook` is a record of type `StudentRec`. To access the course grade of the third student, for example, we use the following expression:



The record component `gradeBook[2].examScore` is an array. We can access the individual elements in this component just as we would access the elements of any other array: We give the name of the array followed by the index, which is enclosed in brackets.

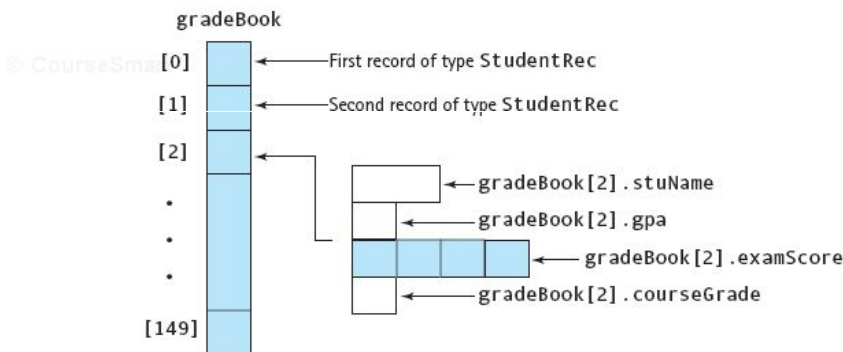
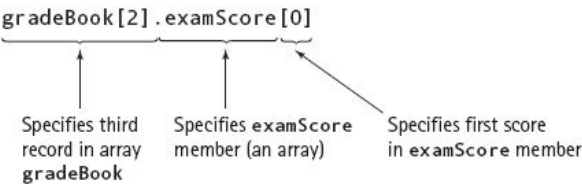


FIGURE 11.8 `gradeBook` Array with Records as Elements

The next step is to write a function that reads values from a file into the fields of the records. The file name and the record are parameters of the function. Both must be reference parameters: Files are always passed by reference, and we are sending back values to the calling code through the record.

```
void ReadValues(istream& inFile, StudentRec& record)
{
    char letter;
    char throwAway;
    getline(inFile, record.stuName);
    inFile >> record.gpa >> record.examScore[0]
        >> record.examScore[1] >> record.examScore[2]
        >> record.examScore[3] >> letter;
    inFile.get(throwAway);
    switch (letter) // Convert from char to GradeType
    {
        case 'A' : record.courseGrade = A;
                    break;
        case 'B' : record.courseGrade = B;
                    break;
        case 'C' : record.courseGrade = C;
                    break;
        case 'D' : record.courseGrade = D;
                    break;
        case 'F' : record.courseGrade = F;
                    break;
    }
}
```

The only tricky part is getting past the `coln` at the end of each student's entry. This `coln` follows the letter grade entered as a character. We can't use a regular character read operation because it returns the first letter of the next student's name. Instead, we must use `inFile.get`.

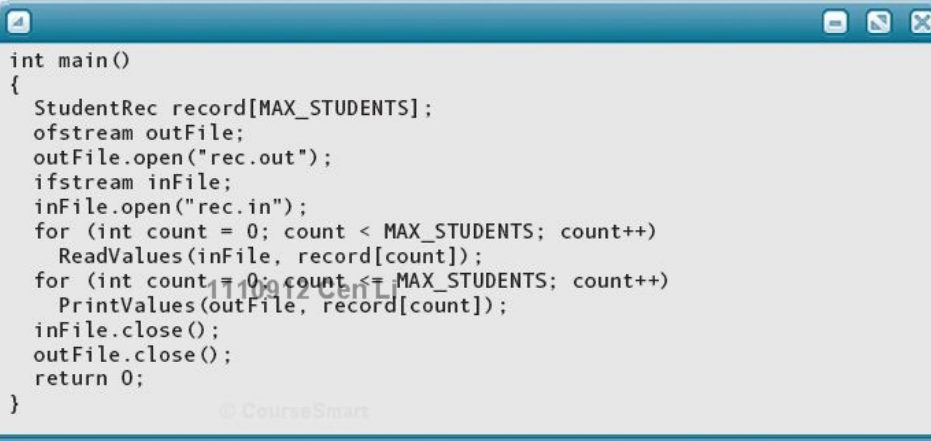
The function to write each student's record is a mirror image of the input function.

```
void PrintValues(ofstream& outFile, StudentRec& record)
{
    outFile << record.stuName << endl;
    outFile << record.gpa << ' ' << record.examScore[0] << ' '
        << record.examScore[1] << ' '
        << record.examScore[2] << ' '
        << record.examScore[3] << ' ';
    switch (record.courseGrade)
    {
        case A : outFile << 'A';
                    break;
        case B : outFile << 'B';
                    break;
        case C : outFile << 'C';
                    break;
        case D : outFile << 'D';
                    break;
    }
}
```

© CourseSmart

```
    case F : outFile << 'F';  
            break;  
    }  
    outFile << endl;  
}
```

Now we must create a `main` program that uses these two functions to input the file of records, store them into an array, and print them on an output file.



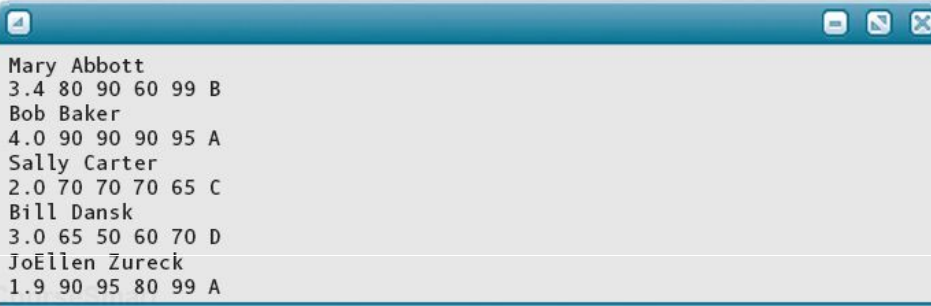
```
int main()  
{  
    StudentRec record[MAX_STUDENTS];  
    ofstream outFile;  
    outFile.open("rec.out");  
    ifstream inFile;  
    inFile.open("rec.in");  
    for (int count = 0; count < MAX_STUDENTS; count++)  
        ReadValues(inFile, record[count]);  
    for (int count = 0; count < MAX_STUDENTS; count++)  
        PrintValues(outFile, record[count]);  
    inFile.close();  
    outFile.close();  
    return 0;  
}
```

© CourseSmart

© CourseSmart

Here is an input file and the resulting output file. The constant for `MAX_STUDENTS` was changed to 5 for this run (for obvious reasons).

rec.in



```
Mary Abbott  
3.4 80 90 60 99 B  
Bob Baker  
4.0 90 90 90 95 A  
Sally Carter  
2.0 70 70 70 65 C  
Bill Dansk  
3.0 65 50 60 70 D  
JoEllen Zureck  
1.9 90 95 80 99 A
```


© CourseSmart

rec.out

```
Mary Abbott
3.4 80 90 60 99 B
Bob Baker
4 90 90 90 95 A
Sally Carter
2 70 70 70 65 C
Bill Dansk
3 65 50 60 70 D
JoEllen Zureck
1.9 90 95 80 99 A
```

QUICK CHECK



- 11.2.1 How would you access the third letter in a string data member (called `street`) of a struct variable that is the twelfth element of an array called `mailList`? (pp. 541–542)
- 11.2.2 Imagine we are extending a web browser with functionality that requires an array of records to maintain the coordinates for HTML elements in the browser window. How might you declare the struct representing coordinates and an array of 1000 elements for holding these records? (pp. 541–542)

11.3 Special Kinds of Array Processing

Two types of array processing occur especially often: using only part of the declared array (a subarray) and using index values that have specific meaning within the problem (indexes with semantic content). We describe both of these methods briefly here and give further examples in the remainder of the chapter.

Subarray Processing

The *size* of an array—the declared number of array components—is established at compile time. We have to declare an array to be as big as it will ever need to be. Because the exact number of values to be put into the array often depends on the data itself, however, we may not fill all of the array components with values. The problem is that to avoid processing empty positions in the array, we must keep track of how many components are actually filled.

As values are put into the array, we keep a count of how many components are filled. We then use this count to process only those components that have values stored in them; any remaining places are not processed. For example, if there were 250 students in a class, a program to analyze test grades would set aside 250 locations for the grades. However, some students might be absent on the day of the test. So the number of test grades must be counted, and that number—rather than 250—is used to control the processing of the array.

If the number of data items actually stored in an array is less than its declared size, functions that receive array parameters must also receive the number of data items as a parameter. We handled this task in the `ZeroOut` function by passing the array and the number of elements as two parameters. However, there is a better way: The array and the number of actual items in the array can be bound together in a record.

Consider, for example, the collection of student records. We could change the problem a little and say that the actual number of student records is not known. The reading loop would be an eof loop rather than a For loop. Let's expand `ReadValues` and `PrintValues` to take the record containing the array and the number of items rather than just the array. Thus the loops would be found within the functions, not in `main`.

Here is the complete program with these changes. The definition of the data structure and the function prototypes are highlighted.

```
//*****
// This program manipulates an array of grade records.
//*****
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

const int MAX_STUDENTS = 150;
enum GradeType {A, B, C, D, F};
struct StudentRec
{
    string stuName;
    float gpa;
    int examScore[4];
    GradeType courseGrade;
};

struct GradeBook
{
    StudentRec grades[MAX_STUDENTS];
    int numElements;
};

void ReadValues(ifstream& inFile, GradeBook& book);
// Pre:  inFile has been opened
// Post: Values have been read from inFile and stored into the
//       proper fields of record
void PrintValues(ofstream& outFile, GradeBook& book);
// Pre:  outFile has been opened and record contains valid data
// Post: Values in record have been printed on outFile

int main ()
{
    ofstream outFile;
    outFile.open("rec.out");
    ifstream inFile;
    inFile.open("rec.in");
    GradeBook book;
    ReadValues(inFile, book);
    cout << book.numElements;
    PrintValues(outFile, book);
    return 0;
}

//*****
```

```
void ReadValues(istream& inFile, GradeBook& book)
{
    char letter;
    char throwAway;
    int count = 0;
    getline(inFile, book.grades[count].stuName);
    while (inFile)
    {
        inFile >> book.grades[count].gpa
            >> book.grades[count].examScore[0]
            >> book.grades[count].examScore[1]
            >> book.grades[count].examScore[2]
            >> book.grades[count].examScore[3] >> letter;
        inFile.get(throwAway);
        switch (letter)
        {
            case 'A' : book.grades[count].courseGrade = A;
                        break;
            case 'B' : book.grades[count].courseGrade = B;
                        break;
            case 'C' : book.grades[count].courseGrade = C;
                        break;
            case 'D' : book.grades[count].courseGrade = D;
                        break;
            case 'F' : book.grades[count].courseGrade = F;
                        break;
        }
        count++;
        getline(inFile, book.grades[count].stuName);
    }
    book.numElements = count;
}
```

```
//*****
```

```
void PrintValues(ofstream& outFile, GradeBook& book)
{
    for (int count = 0; count < book.numElements; count++)
    {
        outFile << book.grades[count].stuName << endl;
        outFile << book.grades[count].gpa << ' '
            << book.grades[count].examScore[0] << ' '
            << book.grades[count].examScore[1] << ' '
            << book.grades[count].examScore[2] << ' '
            << book.grades[count].examScore[3] << ' ';
        switch (book.grades[count].courseGrade)
        {
            case A : outFile << 'A';
                        break;
            case B : outFile << 'B';
                        break;
            case C : outFile << 'C';
                        break;
            case D : outFile << 'D';
                        break;
        }
    }
}
```