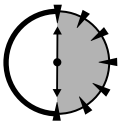
SESSION

Intro to Classes

Session Checklist

- ✔ Using the class structure to group different types of variables into one object
- ✔ Writing programs using the class structure



30 Min. To Go

Arrays are great at handling sequences of objects of the same type such as `int`s or `double`s. Arrays do not work well, however, when grouping different types of data such as when we try to combine a social security number with the name of a person into a single record. C++ provides a structure called a class to handle this problem.

Grouping Data

Many of the programs in earlier sessions read a series of numbers, sometimes into arrays, before processing. A simple array is great for stand-alone values. However, many times (if not most of the time) data comes in groups of information. For example, a program may ask the user for her first name, last name, and social security number. Alone any one of these values is not sufficient—only in the aggregate

do the values make any sense. For reasons that become clear shortly, I call such a grouping of data an *object*.

One way to describe an object is by what I call *parallel arrays*. In this approach, the programmer defines one array of strings for the first names, another for the second, and a third for the social security numbers. The three different values are coordinated through the array index.

An example

The following program uses the parallel array approach to input and display a series of names and social security numbers. `szFirstName[i]`, `szLastName[i]`, and `nSocialSecurity[i]` to combine to form a single object.

```
// ParallelData - store associated data in
//                parallel arrays
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// "parallel arrays" store associated data
// (make arrays global to give all functions
// access)
char szFirstName[25][128];
char szLastName [25][128];
int  nSocialSecurity[25];

// getData - read a name and social security
//           number; return 0 if no more to
//           read
int getData(int index)
{
    cout << "\nEnter first name:";
    cin  > szFirstName[index];

    // if the first name is 'exit' or 'EXIT'...
    if ((strcmp(szFirstName[index], "exit") == 0)
        ||
        (strcmp(szFirstName[index], "EXIT") == 0))
    {
        // ...return with a "let's quit" indicator
```

```
        return 0;
    }

    // load the remainder of the object
    cout << "Enter last name:";
    cin  > szLastName[index];

    cout << "Enter social security number:";
    cin  > nSocialSecurity[index];

    return 1;
}

// displayData - output the index'th data set
void displayData(int index)
{
    cout << szFirstName[index]
         << " "
         << szLastName[index]
         << "/"
         << nSocialSecurity[index]
         << "\n";
}

int main(int nArg, char* pszArgs[])
{
    // load first names, last names, and social
    // security numbers
    cout << "Read name/social security information\n"
         << "Enter 'exit' for first name to exit\n\n";
    int index = 0;
    while (getData(index))
    {
        index++;
}

    cout << "\nEntries:\n";
    for (int i = 0; i < index; i++)
    {
        displayData(i);
```

```
    }
    return 0;
}
```

The three coordinated arrays are declared as follows:

```
char szFirstName[25][128];
char szLastName [25][128];
int  nSocialSecurity[25];
```

The three have sufficient room to handle 25 entries. The first and last names of each entry are limited to 128 characters.

No checks are made to insure that the 128-character limits are exceeded. In a real-world application, failing to make the check is unacceptable.

The `main()` function first reads in the objects in the loop beginning with `while (getData(index))` in the function `main()`. The call to `getData()` reads the next entry. The loop exits when `getData()` returns a zero indicating that entry is complete.

The program then calls `displayData()` to display the objects entered.

The `getData()` function reads data from `cin` to the three arrays. The function returns a 0 if the user enters a first name of `exit` or `EXIT`. If the first name is not `exit` the function reads the remaining data and returns a 1 to indicate that there are more data objects to read.

The following output is from a sample run of the ParallelData program.

```
Read name/social security information
Enter 'exit' for first name to exit

Enter first name:Stephen
Enter last name:Davis
Enter social security number:1234

Enter first name:Scooter
Enter last name:Dog
Enter social security number:3456

Enter first name:Valentine
Enter last name:Puppy
```

```
Enter social security number:5678

Enter first name:exit

Entries:
Stephen Davis/1234
Scooter Dog/3456
```

The problem

The parallel-array approach is one solution to the data-grouping problem. In many older programming languages, there were no other options. For large amounts of data, keeping the potentially large number of arrays in synchronization becomes quite a problem.

The simple ParallelData program has only three arrays to keep track of. Consider the amount of data that a credit card might keep on each entry. Potentially dozens of arrays would be needed.

A secondary problem is that it isn't obvious to the maintenance programmer that the numerous arrays belong to each other. If the programmer updates any combination of the arrays without updating them all, the data becomes corrupted.

The Class

What is needed is a structure that can hold all of the data needed to describe a single object. A single object would hold both the first name and last name along with the social security number. C++ uses a structure known as a class.

The format of a class

A class used to describe a name and social security number grouping might appear as follows:

```
// the dataset class
class NameDataSet
{
    public:
        char szFirstName[128];
        char szLastName [128];
```

```
int  nSocialSecurity;
};

// a single instance of a dataset
NameDataSet nds;
```

A class definition starts with the keyword `class` followed by the name of the class and an open-closed brace pair.

The alternative keyword `struct` may be used. The keywords `struct` and `class` are identical except that the `public` declaration is assumed in the `struct`.

The first line within the braces is the keyword `public`.

Later sessions show what other keywords C++ allows besides `public`.

Following the `public` keyword are the entries it takes to describe the object. The `NameDataSet` class contains the first and last name entries along with the social security number. Remember that a class declaration includes the data necessary to describe one object.

The last line declares the variable `nds` as a single entry of class `NameDataSet`. Programmers say that `nds` is an *instance* of the class `NameDataSet`. You *instantiate* the class `NameDataSet` to create `nds`. Finally, programmers say that `szFirstName` and the others are *members* or *properties* of the class.

The following syntax is used to access the property of a particular object:

```
NameDataSet nds;
nds.nSocialSecurity = 10;
cin > nds.szFirstName;
```

Here, `nds` is an instance of the class `NameDataSet` (that is, a particular `NameDataSet` object). The integer `nds.nSocialSecurity` is a property of the `nds` object. The type of `nds.nSocialSecurity` is `int`, while the type of `nds.szFirstName` is `char[]`.

A class object can be initialized when it is created as follows:

```
NameDataSet nds = {“FirstName”, “LastName”, 1234};
```

In addition, the programmer may declare and initialize an array of objects as follows:

```
NameDataSet ndsArray[2] = {{ "FirstFN",  "FirstLN", 1234}
                           {"SecondFN", "SecondLN", 5678}};
```

Only the assignment operator is defined for class objects by default. The assignment operator performs a binary copy of the source object to the target object. Both source and target must be of exactly the same type.

Example program

The class-based version of the ParallelData program appears below:

```
// ClassData - store associated data in
//             an array of objects
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// NameDataSet - stores name and social security
//               information
class NameDataSet
{
    public:
        char szFirstName[128];
        char szLastName [128];
        int  nSocialSecurity;
};

// getData - read a name and social security
//           number; return 0 if no more to
//           read
int getData(NameDataSet& nds)
{
    cout << "\nEnter first name:";
    cin  > nds.szFirstName;

    if ((strcmp(nds.szFirstName, "exit") == 0)
        ||
        (strcmp(nds.szFirstName, "EXIT") == 0))
```

```
    {
        return 0;
    }

    cout << "Enter last name:";
    cin  > nds.szLastName;

    cout << "Enter social security number:";
    cin  > nds.nSocialSecurity;

    return 1;
}

// displayData - output the index'th data set
void displayData(NameDataSet& nds)
{
    cout << nds.szFirstName
         << " "
         << nds.szLastName
         << "/"
         << nds.nSocialSecurity
         << "\n";
}

int main(int nArg, char* pszArgs[])
{
    // allocate 25 name data sets
    NameDataSet nds[25];

    // load first names, last names, and social
    // security numbers
    cout << "Read name/social security information\n"
         << "Enter 'exit' for first name to exit\n";
    int index = 0;
    while (getData(nds[index]))
    {
        index++;
    }
```

```
    cout << "\nEntries:\n";
    for (int i = 0; i < index; i++)
    {
        displayData(nds[i]);
    }
    return 0;
}
```

In this case, the `main()` function allocates 25 objects of class `NameDataSet`. As before, `main()` enters a loop in which entries are read from the keyboard using the function `getData()`. Rather than passing a simple index (or an index plus three arrays), `main()` passes the object that `getData(NameDataSet)` is to populate. Similarly, `main()` uses the `displayData(NameDataSet)` function to display each `NameDataSet` object.

The `getData()` function reads the object information into the `NameDataSet` object passed, which it calls `nds`.

The meaning of the ampersand added to the argument type `getData()` is fully explained in Session 13. Suffice it to say for now that the ampersand insures that changes made in `getData()` are retained in `main()`.

Advantages

The basic structure of the ClassData program is the same as that of the ParallelData program; however, the ClassData program does not track multiple arrays. In an object as simple as `NameDataSet`, it isn't obvious that this is a significant advantage. Consider what both programs would look like if `NameDataSet` included the number of entries required by a credit card company, for example. The larger the object, the greater the advantage.

As we continue developing the `NameDataSet` class in later chapters, the advantage increases.

REVIEW

Arrays can only handle sequences of the same types of objects; for example, an array of `int`s or `double`s. The class enables the programmer to group varied variable types in one object. For example, a `Student` class may contain a character string for the student name, an integer variable for the student identification, and a floating-point variable to hold the grade-point average. The combination into arrays of class objects combines the advantages of each in a single data structure.

- The elements of a class object do not have to be of the same type; however, because they are of different types, these elements must be addressed by name and not by index.
- The keyword `struct` can be used in place of the keyword `class`. `struct` is a holdover from the days of `C`.

QUIZ YOURSELF

1. What is an object? (See "Grouping Data.")
2. What is the older term for the `C++` keyword `class`? (See "The Format of a Class.")
3. What do the following italicized words mean? (See "The Format of a Class.")
   a. *Instance* of a class
   b. *Instantiate* a class
   c. *Member* of a class