C++ Structures

In addition to naming places and processes, an identifier can name a collection of places. When a collection of places is given a name, it is called a composite data type and is characterized by how the individual places within variables of the data type can be accessed. In this section we briefly look at the **struct** and then we will spend more time on **classes**. A struct is normally used as a simple version of a class.

The struct is a very versatile data type. It is a collection of components of any data type in which the components are given names. The components of a record are called *fields* or *members*. Each field has its own field identifier and data type.

C++ has its own vocabulary in relation to the general concept of a record. Structs are called records. We will also see the terms member and instance.

To define a struct, use the keyword struct followed by the name of the structure. Then use curly braces followed by variable types and names:

```
struct StructName
{
        type1   var1;
        type2   var 2;
        …
        type3   var N;
};
```

Note the need for a semicolon at the end of the right curly brace!

The above defines a structure named "StructName". You can use StructName like it is a new data type. For example, the following will declare a variable to be of type "StructName":

```
StructName myVar;
```

Some older compilers will require you to use **struct StructName** as the type instead of just StructName.

To access the members (variables) within the structure, use the variable name followed by a dot and then the variable within the struct. This is called the member selector:

```
myVar.var1;
```

Here is an example structure:

```
struct  Recording
{
    string    title;
    string    artist;
    float     cost;
    int       quantity;
};

Recording song;
```

Recording is a pattern for a group of four variables. song is a struct variable with four members: title, artist, cost, and quantity are the member names. The accessing expression for members of a struct variable is the struct variable identifier followed by the member name with a period in between.

song.title is a string variable.
song.artist is a string variable.
song.cost is a float variable.
song.quantity is an int variable.

The only aggregate operation defined on structures is assignment, but structures may be passed as parameters and they may be the return value type of a function. For example, the following is valid:

```
Recording song1, song2;

song1.title = "Lady Java";
song1.artist = "Jenny Skavlan";
song1.cost = 3.50;
song1.quantity = 500;
song2 = song1;
cout << song2.title << endl;
```

This will print out "Lady Java" as the contents of song1 get copied to song2. C++ makes a copy of the variables, so the behavior is likely what you would expect from assignment (it is the same as if we copied an int or a string).

Rather than spending more time on structs, let's transition to discussing Classes. A class can do everything a struct can, and more, so it is pretty common in object oriented programming to skip structs and go straight to classes (in the C language there are no classes, only structs.)

C++ Class Data Type

The class data type is a C++ language feature that encourages good programming style by allowing the programmer to encapsulate both data and actions into a single object, making the class the ideal structure for representing an abstract data type. To define a class, use the keyword "class" much like you would define a structure. An example is below with the reserved words in bold:

```
class CreditCardTransaction
{
public:
    void  initialize(string name, string number, double amount);
    string getName();
    string getNumber();
    double getAmount();
private:
    string name;
    string number;
    double amount;
};
CreditCardTransaction t1;
```

CreditCardTransaction is a class data type. Like any type, it is a pattern for a structure. It is like a function prototype – it defines the structure, but not how it is actually implemented. This pattern has three data items (member variables): name, number and amount; and four actions (member functions): Initialize, getName, getNumber, and getAmount (if this were real you would want expiration date and other info). The word **public** means that the members defined in this section are accessible to anyone using the class (defining a variable of the class). The members defined following the word **private** are accessible only to the class's member functions. Member variables and functions defined between public and private form the interface between the class and its clients. A client is any software that declares variables of the class. Notice that member function prototypes, not definitions, are used in the class definition. This interface provides no information on how the member functions are implemented. This definition would normally go into the .h file of your project although you could implement them in the interface (this is sometimes done for very short one-line implementations).

t1 is an instance of the class data type CreditCardTransaction; it is called a **class instance** or a **class object**. t1 has three member variables and four member functions. To apply a class instance's member functions, you append the function name to the instance of the data type separated by a period. For example, the following code segment instructs t1 to apply its member functions initialize, and getAmout.

```
t1.initialize("Ash Williams", "4299 1299 1293 3939",99.95);
cout  << t1.getAmount() << endl;              // Outputs 99.95
```

Member Function Definitions

A member function is defined like any function with one exception: The name of the class type within which the member is declared precedes the member function name with a double colon in between (::). The double colon operator is called the **scope resolution operator**.

```
void CreditCardTransaction::initialize(string newName,
                         string newNumber, double newAmount)
{
  name = newName;
  number = newNumber;
  amount = newAmount;
}

string CreditCardTransaction::getName()
{
 return name;
}

string CreditCardTransaction::getNumber()
{
 return number;
}

double CreditCardTransaction::getAmount()
{
 return amount;
}
```

When the statement

t1.initialize("Ash Williams", "4299 1299 1293 3939",99.95);

is executed, “Ash Williams” is stored in t1.name and “4299 1299 1293 3939” is stored in t1.number. Because initialize is an action defined within class CreditCardTransaction any invocation of initialize is applied to a specific instance of CreditCardTransaction, in this case the variable t1. The variable identifiers used in the body of the definition of function initialize refer to those of the instance to which it is applied.

The following statement prints some of the data fields of two instances of CreditCardTransaction. One is made on the stack and the other is a dynamic variable, allocated off the heap. We have to dereference the pointer before accessing the member fields, i.e. (*pt2).getName() rather than *pt2.getName(). It is kind of unwieldy to type the parenthesis all the time, so there is a shortcut notation with the arrow operator. pt2->getName() is the same as (*pt2).getName().

```
CreditCardTransaction t1;
t1.initialize("Ash Williams","1234 5678 1234 4121", 99.95);
cout << t1.getName() << " " << t1.getAmount() << endl;


CreditCardTransaction *pt2;
pt2 = new CreditCardTransaction;
(*pt2).initialize("Henrietta Knowby","3234 5678 1234 9999", 19.00);
cout << (*pt2).getName() << " " << pt2->getAmount() << endl;
delete pt2;
```

Outputs:

```
Ash Williams 99.95
Henrietta Knowby 19
```

These are called instances because they are separate instantiations of CreditCardTransaction.

Binary Operations on Another Class

When a binary operation is defined within a class type, one of the operands is passed as a parameter and the other is the class instance to which the member function is applied. For example, let's assume that a binary operation, Add, has been included as a member function in CreditCardTransaction. Here is its definition.

```
CreditCardTransaction CreditCardTransaction::transfer(
        CreditCardTransaction other)
{
    CreditCardTransaction result;
    result.name = name;
    result.number = number;
    result.amount = amount + other.amount;
    return result;
}
```

Given the following statement:

```
    CreditCardTransaction t3;
    t3 = t1.transfer(*pt2);
    cout << t3.getName() << " " << t3.getAmount() << endl;
```

Output:

```
Ash Williams 118.95
```

Name, number and amount in the code of member function “transfer” refer those members in the class object to which “transfer” has been applied; that is, to t1. The class object to which a member function is applied is often called self.

Class Constructors

Because we use classes to encapsulate abstract data types, it is essential that class objects be initialized properly. We defined a member function, Initialize, to initialize the values of the member variables. What if the client forgets to apply member function Initialize? How can we make it easy to initialize objects? This can be such a serious problem that C++ provides a mechanism to guarantee that all class instances are properly initialized, called the class **constructor**. A class constructor is a member function with the same name as the class data type. You can make as many constructors as you like as long as the parameters are different (this is called **overloading**).

```
class CreditCardTransaction
{
public:
    // Two constructors
    CreditCardTransaction();
    CreditCardTransaction(string name, string number,
                          double amount);
    string getName();
    string getNumber();
    double getAmount();
private:
    string name;
    string number;
    double amount;
};
```

These constructors are defined as follows.

```
CreditCardTransaction::CreditCardTransaction()
{
  name = "Unknown";
  number = "0000 0000 0000 0000";
  amount = 0;
}

CreditCardTransaction::CreditCardTransaction(string newName,
                       string newNumber, double newAmount)
{
  name = newName;
  number = newNumber;
  amount = newAmount;
}
```

Now in our code when we define an instance of CreditCardTransaction we can do it two ways:

```
CreditCardTransaction t1;
CreditCardTransaction t2("Ruby Knowby", "3234 5677 1234 9999", 19.00);
```

There are two constructors: one with no parameters, called the **default constructor**, and one with two parameters. They look a little strange because there is no type identifier or void before them. They are invoked differently as well. A class constructor is invoked when a class instance is declared. In this example, t1 is initialized by the default class constructor (the one with no parameters), and t2 is initialized with the name of Chuck Finley and other values.

If we execute:

```
cout << t1.getName() << " " << t1.getAmount() << endl;
cout << t2.getName() << " " << t2.getAmount() << endl;
```

We get:

```
Unknown 0
Ruby Knowby 19
```

Notice that the class constructors do not make member function Initialize unnecessary. We could have kept it if we wanted to reinitialize the class object during run time. **Class constructors are invoked once when a class object is declared**.

There is an alternate way to initialize member variables in the constructor. The alternate formulation looks like this:

```
CreditCardTransaction::CreditCardTransaction() :
   name("Unknown"), number("0000 0000 0000 0000"), amount(0)
{
}

CreditCardTransaction::CreditCardTransaction(string newName,
                        string newNumber, double newAmount) :
    name(newName), number(newNumber), amount(newAmount)
{
}
```

Use whichever method you prefer but be prepared to see both formats.

Packaging

Because classes are used to encapsulate abstract data types and because the separation of the logical properties of an ADT from the implementation details is so important, you should package the class declaration and the implementation of the member functions in different files. The class declaration should be in a file with a ".h" extension (called the **specification or header file**), and the implementation should be in a file with the same name but with a ".cpp" extension (**called the implementation file**). The implementation file must use the #include directive to access the specification file. Any client program must use the #include directive for the specification file (.h extension) to include your files in their source code. We will say more about separate compilation later.

If you are using Visual Studio, the "Insert Class" option from the menu will automatically create a .h and a .cpp file for you.

Class Scope

Previously we said there were four kinds of scope: local scope, namespace scope, global scope, and class scope. We defined the first three, but left the definition of class scope until later. Class scope means that a member name is bound to the class or struct in which it is defined. That is, member names are not known outside of the struct or class in which they are defined.

Good Programming Practice : Private Member Variables

Note that when we declared the CreditCardTransaction class, we made the variables *private*. This means that we can't directly access them via t1.name or t1.amount. If we made them public, we would basically have the same thing as a struct with additional functions defined.

```
class CreditCardTransaction
{
public:
    // Two constructors
    CreditCardTransaction();
    CreditCardTransaction(string name, string number,
                          double amount);
    string getName();
    string getNumber();
    double getAmount();
    string name;
    string number;
    double amount;
};
```

An instance of CreditCardTransaction can now directly access dollars and cents:

```
CreditCardTransaction t1;

t1.name = "Jose";
t1.amount =100;
```

This is considered to be a poor software implementation. Why? The outside world now has direct access to the underlying data representation used in the class. It is much better to hide the underlying data representation and force the outside world to only access data through the published interface (i.e. the functions).

For example, let's say that we wrote a bunch of code that directly accesses the name. So splattered throughout the program we have:

```
n = t1.name;
t1.name = n;
if (t2.name == "Bob")
        …
```

This is in several ways more convenient than having to go through a function to access the member variable (this function is called an *accessor*. If we have a function that changes a variable it is called a *mutator*). But maybe later we decide that we need to store the first and last name separately instead of in one variable. To do this we could make two variables, one called first and the other called last, and delete the old variable called name. But this will break all the existing code that directly accesses the variables by name!

But if all code went through accessor and mutator functions, then we only need to update code in the functions to map back and forth between the old and new variables, and all the old code would still work:

```
void CreditCardTransaction::setName(string fullName)
{
  int indexOfSpace = fullName.find(" ");
  first = fullName.substr(0, indexOfSpace);
  last = fullName.substr(indexOfSpace+1);
}

string CreditCardTransaction::getName()
{
 return first + " " + last;
}
```

By making the data members of the class private, this architecture forces us to hide the implementation of a class from its clients, which reduces bugs and improves program modifiability.

Assignment of Classes

The assignment operator can be used to assign a class to another class instance of the same type. By default, these assignments are performed by memberwise copy – each member of one object is copied (assigned) individually to the same member of the other object. There is also a way to define your own assignment operator, called operator overloading (and described later).

While assignment will do what you want most of the time, note that it won't make copies of any data that is allocated using new. Consider the following class which allocates an integer:

```
class SampleClass
{
public:
    SampleClass();      // class constructor
    void SetNum(int num);
    int GetNum();
private:
    int *m_numPtr;   // Some people like to preface
                     // member vars with m_
};

// Constructor
SampleClass::SampleClass()
{
      m_numPtr = new int;                 // Allocate an int
      *m_numPtr = 0;
}

void SampleClass::SetNum(int num)
{
      *m_numPtr = num;
      return;
}

int SampleClass::GetNum()
{
      return (*m_numPtr);
}
```

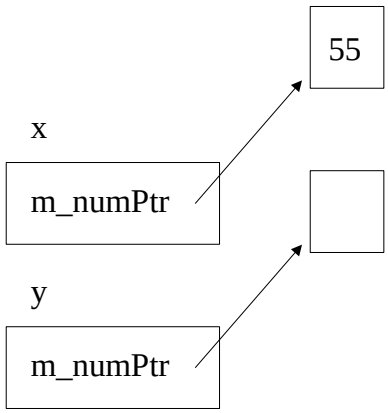
What happens with the following main?

```
int main()
{
      SampleClass x,y;
      x.SetNum(55);
      y = x;
      cout << x.GetNum() << " " << y.GetNum() << endl;
      x.SetNum(23);
      cout << x.GetNum() << " " << y.GetNum() << endl;
      return 0;
}
```

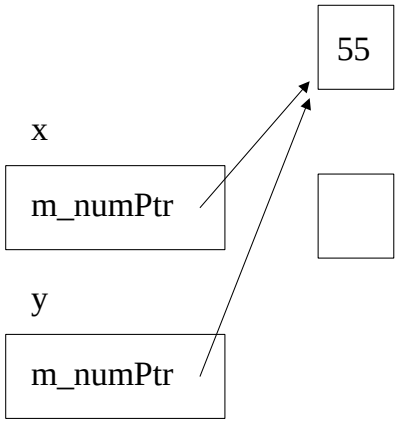
The output from this program is:

55 55
23 23

There are two problems with this program. The first problem is that when we copied x to y, we copied the x's internal pointer, but didn't copy the data. Graphically, we initially have the following after executing x.SetNum(55):

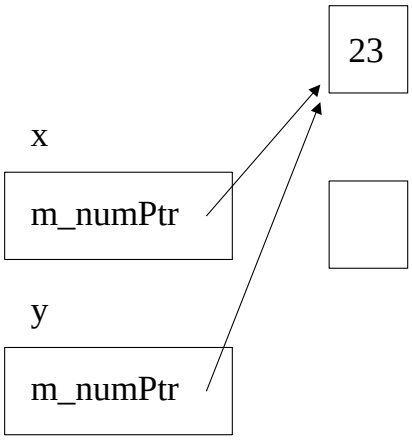


When we copy x to y, we copy the pointer but don't follow the pointer to copy the values!



Y's value is now pointing to the memory we allocated for x! What happened to the memory that y used to point to? It is still there, allocated. However we've lost the pointer to it. We have just experienced a memory leak. This memory will sit and the computer will think it is used until someone (the Operating System, hopefully) comes along and re-claims it.

Upon executing x.SetNum(23), we are now changing the picture to:



Consequently, when we print out y.GetVal(), we are getting the number 23.

This is probably not the desired result! So the bottom line is to be very careful about copying objects if there are pointers involved.

Destructors

The other problem with the above program is that the memory we allocated to each instance was never freed using the delete function call.

A logical place to free the memory is when the instance variable goes out of scope. There is a special function we can define that is automatically invoked when the variable goes out of scope. This is called the **destructor** because it is called when the object is destroyed. Any type of "cleanup" can go into the destructor. To define a destructor, use ~ in front of the class name:

```
class SampleClass
{
public:
    SampleClass();      // class constructor
    ~SampleClass();     // class DESTRUCTOR
    void SetNum(int num);
    int GetNum();
private:
    int *m_numPtr;
};

SampleClass::~SampleClass()
{
      // Cleanup code here
      delete m_numPtr;
      cout << "Deleting allocated memory!" << endl;
}
```

Here is some sample usage:

```
void TestFunc()
{
	SampleClass x;
	x.SetNum(10);
	cout << x.GetNum() << endl;
	return;
}

int main()
{
	cout << "Entering TestFunc" << endl;
	TestFunc();
	cout << "Done" << endl;
	return 0;
}
```

When this code invokes TestFunc, we create a new instance of SampleClass and call the constructor, allocating memory. Then we set the number to 10. When the function returns, the variable x goes out of scope. This means that the variable is being destroyed, and consequently the destructor is invoked.

Upon exiting TestFunc, the destructor is invoked which delete's the memory and prints out the message "Deleting allocated memory".

The output will be:

```
Entering TestFunc
10
Deleting allocated memory!
Done
```

In summary the destructor is a good place to put any cleanup/exit code you want to execute before your object goes away for good. The constructor is where you put code you want to run once, when the object is initially declared.