MAE 573 Fall 2001 - Lecture 4: C++
Programming

1 Introduction

The C++ programming language was created to address shortcomings that its creators saw in plain old C. That, of course, has been a subject of debate of some die hard C programmers. C is designed to let the programmer design global control and behavior of the program. C programs have one master control loop that walks through and manipulates various sets of data. C++ programs, on the other hand, function by letting a bunch of functional units each do their own work on their own data. In C++, the programmer designs classes, which incorporate data and functions (called methods). The programmer than creates (or instantiates) instances of these classes (called objects) and then lets them do the work of the program.

2 Classes

A class is similar to a struct in C, except in addition to data members, as mentioned above, classes can have functions (methods). Additionally, classes specify private and public methods and data members, determining what parts of the program can access what. Public data members and methods can be accessed at any time from anywhere in the program where the object has scope\(^1\). Private data members and methods, on the other hand, can only be accessed by the class methods. This can be confusing and a bit overwhelming, so let’s look at an example.

\(^1\)See any beginning Computer Science text for a definition of scope
/first declare a class TalkingDoll

class TalkingDoll
{
public:
    void Talk();
    void SetSpeech(char *);
private:
    char Speech[255];
};

//next, declare the functions you've prototyped,
//use the '::' specifier

void TalkingDoll::Talk()
{
    printf("%s\n",Speech);
}

void TalkingDoll::SetSpeech(char *s)
{
    //should check for overflow error
    strcpy(Speech,s);
}

//now, let's create an instance, set its speech, and make it talk

main()
{
    TalkingDoll a;
    a.SetSpeech("Hello There");
    a.Talk();
    //this is illegal- Speech is private
    a.Speech[0]='a';
}

There are a couple of things to note here. First, notice how when designing a class, you first declare a prototype of the class, specifying what data and methods are and then you declare the actual methods themselves, using the specifier ::.

Another thing to note is that in C++, unlike in C, all methods must specify a return type, even if it's void. If you declare a method without a return type, the compiler will complain.

Next, notice how you declare an instance of a class- you simply give the class name, then the name of the object. This is exactly what you do when declaring
any other kind of type- you enter the line:

```cpp
int a;
float b;
TalkingDoll d;
```

This is called static declaration, and "." operator is used to address member functions and data as seen below:

```cpp
a.Talk();
```

Here, the method Talk, which is a member of the object a. You can imagine having multiple objects and calling their own individual Talk methods.

The other way to declare instances of classes is to use dynamic declarations. Here, we use the new operator. And instead of using direct referencing, pointers are used:

```cpp
TalkingDoll *doll;
doll=new TalkingDoll();
doll->Talk();
delete doll;
```

Note, too, the use of the "->" operator to indicate methods (and data members), instead of the "." operator.

### 3 Constructors and Destructors

When an object is created (when a C++ class is instantiated), one particular method is automatically called. Usually, this method functions as an initialization routine, setting default values to data members. This is called a constructor. Analogously, when the object is deleted or if it goes out of scope, the destructor is called. In the destructor, cleanup is performed- so if memory has been allocated, for example, in the constructor, it is deallocated in the destructor. The constructor method has the same name as the class name and the destructor method has the same name as the class name with the ~ prepended to it.

There can be multiple constructors, and they can be passed parameters. So, you can design a simple constructor that takes no arguments and sets default values, and you can design a more complicated one where the user passes values which are then set. The constructor is the only method that doesn’t have a return type (all other methods need at least to be declared with a void return type). Here’s an example:
//first define the class
class Pet {
    public:
        Pet();
        Pet(char *n);
        void Introduce();

    private:
        char Name[255];
};

//now implement the constructors
Pet::Pet() {
    strcpy(Name, "Fido");
}

Pet::Pet(char *name) {
    strcpy(Name, name);
}

//now implement the Introduce method
void Pet::Introduce() {
    printf("Hello, my name is %s\n", Name);
}

//and here’s the main loop
main()
{
    Pet a;
    Pet b("Ralph");
    Pet *c = new Pet("Jimbo");
    a.Introduce();
    b.Introduce();
    c->Introduce();
    delete(c);
}

Note that when objects are created dynamically with the new operator, they
must be explicitly destroyed with the delete operator.

4 A sample C versus C++ program

To give you an idea of the differences between C and C++, here’s an example of a simple program written in C and a functionally identical program in C++. The program simply keeps track of a list of people and their phone numbers.

First the C program:

```c
char Name[255][255];
char Phone[255];
int NumPeople;

// prototype for AddPerson Function
AddPerson(char *,char*);

main()
{
    int NumPeople=0;
    int counter;

    AddPerson("Joe Smith","444-3332");
    AddPerson("Fred Hayes","123-1111");
    AddPerson("Rita Williams","222-1111");
    for(counter=0;counter<NumPeople;counter++)
    {
        printf("Name: %s Phone: %s\n",&Name[counter][0],Phone[counter]);
    }
}

AddPerson(char *name,char *phone)
{
    strcpy(&Name[NumPeople][0],name);
    strcpy(&Phone[NumPeople]);
    NumPeople++;
}
```

This is a typical procedural C program. Note how the main loop has access to the data, and it directly accesses it when it’s time to print out the database.

Now the C++ version:
//globals
int NumPeople;
Person *[255]; // an array of pointers to Person objects

// first define the Person Class
class Person
{
public:
Person(char *n, char *p);
void Dump();
private:
char Name[255];
char Phone[255];
}

// and prototype the AddPerson function
void AddPerson(char *, char *);

// and here's main

main()
{
NumPeople=0;
AddPerson("Joe Smith", "444-3332");
AddPerson("Fred Hayes", "123-1111");
AddPerson("Rita Williams", "222-1111");
for(counter=0; counter<NumPeople; counter++)
{
Person[counter]->Dump();
}
}

// now define AddPerson

// now define AddPerson

void AddPerson(char *n, char *p)
{
Person[NumPeople] = new Person(n, p);
NumPeople++;
}

Notice how in the C++ program, the data is encapsulated into the class itself. So, to get a complete listing of people and their phone numbers, you call each class's Dump method. This is the fundamental idea behind object oriented programming— that rather than have all the data stored in some central location and then accessing it all at once, the data is stored in individual units (objects)
and the objects themselves perform the functions on their own data.

5 Inheritance

Once you’ve designed a class, you may want to create a class that has similar functionality but that is more specialized than the original. For example, you may have a class called *Conveyance* which describes a generic transportation device. It might contain data members like number of passengers, cost, dimensions. If you wanted to then define a class for an automobile, you would want all the data members of the *conveyance* plus some others. Rather than duplicate all of the functionality of the *conveyance* class, you can declare a class that inherits all of the *conveyance* functionality and adds additional functionality.
//define conveyance class
class Conveyance
{
public:
    Conveyance();
    void SetCost(float c);
    float GetCost();
    void Identify();
private:
    float Cost;
};

class Automobile: public Conveyance
{
public:
    Automobile();
    void SetNumTires(int t);
    int GetNumTires(void);
    void SetColor(char *c);
    char * GetColor(void);
private:
    int NumTires;
    char Color[255];
};

main()
{
    Automobile a;
    //we can call an Automobile method
    a.SetNumTires(4);
    //and we can call all the methods from Conveyance
    a.Identify();
}

Notice that in this code we are declaring an instance of the class Automobile and then calling a Conveyance method. This is one of the benefits of inheritance— you can create complex hierarchies of classes and not have to duplicate functionality among classes of similar behavior. Figure 5 shows what a sample class hierarchy might look like continuing the automobile theme.

6 Operators

You can define operators that take your classes and perform operations on them. The difference between an operator and a function is that an operator takes the form of object operator object, while a function takes the form of function(object,
object). For example:

```cpp
// operator
class A; // defined elsewhere
A first, second;
A third = first * second; // implement the '*' operator on A
// now, a function
A third = Mult(first, second);
```

Operators are defined similarly to methods, and use the `operator` keyword. The vector example illustrates the definition of an operator.

## 7 Vector Example

This example creates a simple nx1 vector class. The addition (+) operator is then defined on this class.

First, we’ll define the class itself:
#include <stdio.h>

class Vector
{
public:
Vector(int n);
int GetCount();
void SetValue(int w,float v);
float GetValue(int w);
void Dump();
//addition operator
Vector operator + (Vector);
private:
float Values[255];
int Count;
};

Now, we’ll implement everything except the operator:
```cpp
int Vector::GetCount()
{
    return Count;
}

Vector::Vector(int n)
{
    Count=n;//set the number of elements
}

void Vector::SetValue(int w, float v)
{
    //really, you should check to make sure
    //that w is not > 255
    Values[w]=v;
}

float Vector::GetValue(int w)
{
    return Values[w];
}

void Vector::Dump()
{
    int counter;
    for(counter=0;counter<Count;counter++)
        printf("%.2f\n",Values[counter]);
}

Next, we’ll define the operator addition (+):
Vector Vector::operator+ (Vector a)
{
    Vector temp(Count);//make it the same size as this
    int counter;
    //first check to make sure they’re the same size
    if(Count!=a.GetCount())
    {
        printf("error- sizes must match");
    }
    else
    {
        for(counter=0;counter<Count;counter++)
        {
            temp.SetValue(counter,Values[counter]+a.GetValue(counter));
        }
    }
    return temp;//we should take care of errors way better than this
}

Notice how the operator is a public member of the class. Here’s the main() function, notice how we use the + operator just like we do with other types:

```c
main()
{
    Vector a(3),b(3),c(3);
    a.SetValue(0,1);
    a.SetValue(1,4);
    a.SetValue(2,4);
    b.SetValue(0,3);
    b.SetValue(1,2);
    b.SetValue(2,7);
    printf("a is :
");
    a.Dump();
    printf("\nb is :
");
    b.Dump();
    c=a+b;
    printf("\nc is :
");
    c.Dump();
}
```

And here’s what the output looks like:
Alternatively, you could have defined an external function `AddVectors`:

```cpp
AddVectors(Vector a, Vector b)
{
    int counter;
    Vector answer(a.GetCount());
    // First check to make sure counts are the same
    ...
    for (counter = 0; counter < a.GetCount(); counter++)
    {
        answer.SetValue(counter, a.GetValue(counter) + b.GetValue(counter));
    }
    return answer;
}
```

```cpp
Vector a(3), b(3), c(3);
// Set values
...
Vector c = AddVectors(a, b);
```

8 Conclusions

C++ represents a fundamentally different way of modeling problems. By breaking up a problem space into discrete function-data units (objects), the solution can often be more efficient, cleaner, and elegant. However, like anything else, it can sometimes be more appropriate to use a serial language like C if the problem at hand doesn’t fit naturally with the object-oriented nature of C++.

In the example code, we have created a `Vector` class. There are many problems with this class that in a real world program would need to be addressed. For example, there is no real error checking. If we attempt to add two vectors
of different sizes, there’s nothing in the return value (which is a vector) of the + operator to let the program know there is an error. We might add some sort of flag (boolean) as a private data member and set the flag on if there is an error condition, letting the program know that an error has occurred.

Another problem is that we are using an array of data that is declared statically. This can cause all sorts of problems. First of all, it takes up memory that isn’t necessarily needed. Whether your Vector has 1 or 256 members, it requires 256 float-sized chunks of memory. There is also the inevitability of overflow. If you attempt to set 257 values, your program will crash, because you’ve only declared an array of 256 floats. The solution to all this is of course, to declare the Values data member as a type float * and dynamically allocate the memory in the constructor, and deallocate it in the destructor. This prevents all of the aforementioned problems.

Finally, any function that returns a Vector type should be written with pointers, rather than statically declared objects. Here’s why. Take a look at the + operator in the example code. It declares a Vector called temp. It then sets the values of that Vector and returns it as the answer. If, instead, this function was declared as follows:

```cpp
Vector* operator + (Vector);
...
Vector* Vector::operator+ (Vector a)
{
    Vector *temp=NULL;
    int counter;
    //first check to make sure they're the same size
    if(Count!=a.GetCount())
    {
        printf("error- sizes must match");
    }
    else
    {
        temp=new Vector(Count);
        for(counter=0;counter<Count;counter++)
        {
            temp->SetValue(counter,Values[counter]+a.GetValue(counter));
        }
    }
    return temp;//we should take care of errors way better than this
}
```

Now, a few problems are fixed. First, if the sizes of the two vectors don’t match, we just return NULL- this is built in error checking. Now the program can test for the return of a NULL pointer. Secondly, if we were to implement dynamic allocation of the data member Values, we wouldn’t have to worry about orphaned pointers. This is an issue because by default, C++ handles returning
of types as \textit{pass by value}, and not \textit{pass by reference}. This is a bit beyond the scope of this document, so consult any introductory C++ book, or Computer Science text for the difference. For now, try to keep your return types as pointers when dealing with objects.