C++ Learning Note

11-08-2015

1. Functions

calling functions

using functions: functions either return a value, or they return data type void-nothing.

2. What is RAM?

RAM is Random Access Memory. it is the elcetronic memory your computer uses while executing programs. Any information in RAM is lost when your computer is turned off. When you run your program, it is loaded into RAM from the disk file (hard drive storage). All variables are created in RAM as well. When programmers talk of memory, it is usually RAM to which they are referring.

3. Variables and constants

The size of an int is: 4bytes.
The size of a short int is: 2bytes.
The size of a long int is: 4bytes.
The size of a char is: 1bytes.
The size of a bool is: 1bytes.
The size of a float is: 4bytes.
The size of a double is: 8bytes.

-C++ is case sensitive.

unsigned int myage,myweithet; // two unisgned int variables.

long area,width,length; // three longs.

-endl stands for end line and is end-L rather than end-one. it is commonly pronounced "end-ell".

-typedef, which stands for type definition.

typedef unsigned short int USHORT

creates the new name USHORT that you can use anywhere you might have written unsigned short int.

-Defining Constants with const

const unsigned short int studentsPerClass = 15;

-Enumerated Constants

create a set of constants. the syntax for enumerated constants is to write the keyword enum.

enum COLOR { RED, BLUE, GREEN, WHITE, BLACK};

This statement performs two tasks:

1). It makes COLOR the name of an enumeration, that is, a new type.

2). It makes RED a symbolic constant with the value 0, BLUE a symbolic constant with the value 1,and so on.

Any one of the constants can be initialized with a particular value.

enum Color {RED=100, BLUE, GREEN=500, WHITE, BLACK=700};

then RED will have the value 100; BLUE, the value 101; GREEN, the value 500; WHITE, the value 501; and BLACK, the value 700.

4. Expressions and Statements

-All C++ statements end with a semicolon.

-Arithmetic operators (+,-,*,/,%)

+ addition
- subtraction
* multiplication
/ division
% modulo

-compound assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=):
expression: value += increase; is equivalent to: value + increase
expression: **a -=5**; is equivalent to: **a=a-5**;
expression: **a /=b**; is equivalent to: **a=a/b**;
expression: **price *= units+1**; is equivalent to: **price =price * (units +1)**;

-Increase and decrease (++, --)
c++;
++c;
c+=1;
c=c+1;
The prefix variety (++c), increment the value and then fetch it.
the postfix variety (c++), fetch the value and then increment the original.
Example: B=3; A=++B; // A contains 4 , B contains 4
Example: B=3; A=B++; // A contains 3, B contains 4

-Relational and equality operatiors (==, !=, >, <, >=, <=):
== Equal to
!= Not equal to
> Greater than
< Less than
>= Greater than or equal to
<= Less than or equal to

-Logical operators (!, &&, ||):
&&: if the left-hand side expression is false, the conmbined result is false (right-hand side expression not evaluated).
||: if the left-hand side expression is true, the combined result is true (right-hand side expression not evaluated).

-Conditional operator (?):
The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:
condition ? result1 : result2.
If condition is true the expression will return result1, if it is not it will return result2.

5. Functions

-Declaring and Defining Functions

parameter type
unsigned short int FindArea (*int* length, *int* width);
return type name parameter name

Note that all functions have a return type.

Examples of function prototypes:
long FindArea(long length, long width); // returns long, has two parameters
void PrintMessage(int messageNumber); // returns void, has one parameter
int GetChoice(); // returns int, has no parameters

-Using Variables with Functions
Local Variables, exist only locally within the function itself. when the function returns, the local variables are no longer available.
Global Variables,In C++, are avoided because they can create very confusing code that is hard to maintain.

6. Creating Basic Classes

- **Implementing Member Functions**

Every class member function that you declare also must be defined.
A member fuction definition begins with the name of class followed by the scope resolution operator (::) and the name of the function.
Here's an example:

```
void Tricycle :: pedal ()
{
   std :: cout << "Pedaling trike \n";
}
```

Class functions have the same capabilities as functions, they can have parameters and return a value.

- **Creating and Deleting Objects**

Classes have a special member function called a constuctor that is called when an object of the class is instantiated. The job of the constructor is to create a valid object of the class, which often includes initializing its member data. The constructor is a function with the same name as the class but no return value. Constructors may or may not have parameters, just like any other function of the class.
Here's a constructor for the Tricycle class:

```
Tricycle :: Tricycle (int initialSpeed)
{
   setSpeed (initialSpeed);
}
```

This constructor sets the initial value of the speed member variable using a parameter.

When you declare a constructor, you also should declare a destructor. A destructor always has the name of the class preceded by tilde (~). Destructors take no parameters and have no return value.
Here's a Tricycle destructor:

```
Tricycle :: ~Tricycle()
{
   // do nothing
```

}
The destructor for the class requires no special actions to free up memory, so it just includes a comment.

- **Constructors Provided by the Compiler**

If you declare no constructors, the compiler creates a default constructor for you. the default constructor the compiler provides takes no action no action.
There are two important points to note:

- The default constructor is any constructor that takes no parameters. You can define it yourself or get it as a default from the compiler.
- If you define any constructor (with or without parameters), the compiler does not provide a default constructor for you. In that case, if you want a default constructor, you must define it yourself.

If you fail to define a destructor, the compiler also provides one of those, which also has an empty body and does nothing.
If you define a constructor, be sure to define a destructor even if your destructor does nothing.

In OOP (object-oriented programming), a program consists of one or more objects, each of which has its own data in the form of member data and functions in the form of functions. The objects are separate from each other and specialize in a specific and narrow purpose. By designing objects to be independent of each other, you create code that's more easily reused elsewhere.

7. Moving into Advanced Classes

- **const Member Functions**

To declare a function as constant, put the keyword const after the parentheses, as in this example:
void displayPage () const;
It is good programming practice to declare as many function to be const as possible.

- **Interface Versus Implementation**

The parts of a program that create and use objects are the clients of the class. The class declaration serves as a contract with these clients. The contract tells client what data the class has available and what the class can do.

- **Organizing Class Declarations and Function Definitions**

Class definitions often are kept separate from their implementations in the source code of C++ programs. Each function that you declare for your class must have a definition.
The definition must be in a file that the compiler can find. Most C++ compilers want that file to end with .cpp.
A convention that most programmers adopt is putting the declaration in a header file with the same name but ending in .hpp (or less commonly .h or .hp).
So if you've put the declaration of the Tricycle class in a file named Tricycle.hpp, the definition of the class functions would be in Tricycle.cpp. The header file can be incorporated into the .cpp file with a preprocessor directive:

```
# include “Tricycle.hpp”
```

- **Inline Implementation**

Here’s an example:

```
class Tricycle
{
public:
   int getSpeed() const { return speed; }
   void setSpeed(int newSpeed);
}
```

Instead of a semicolon after the keyword const, there’s a short block of code within braces. there’s no semicolon after the parentheses. Whitespace doesn’t matter, so the declaration could be formatted like this:

```
class Tricycle
{
publie:
   int getSpeed () const
   {
      return speed;
   }
   void setSpeed(int newSpeed);
}
```

It is not uncommon to build a complex class by declaring simpler classes and including them in the declaration of more complicated class.

8. Creating Pointers

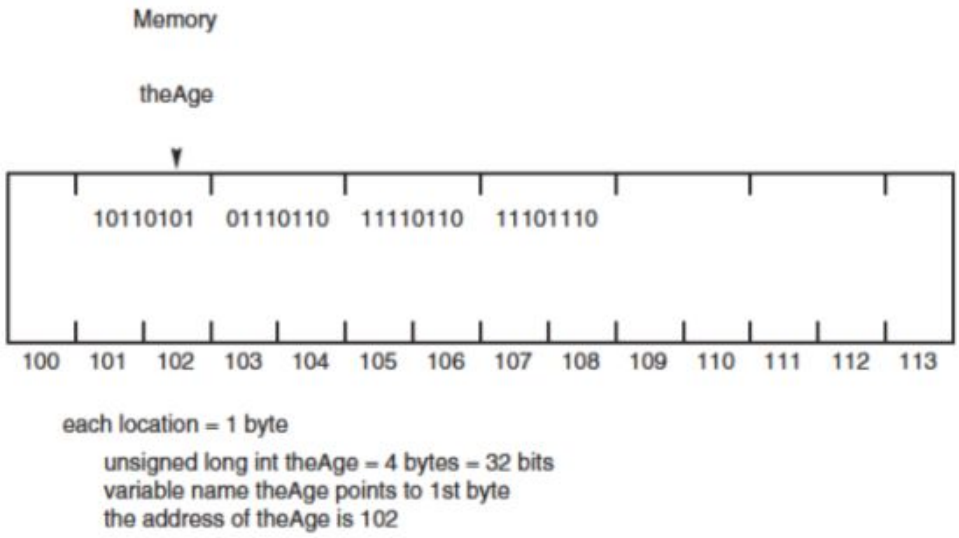
- **Understanding Pointers and Their Usage**

Pointers provide the capability to manipulate computer memory directly.

A pointer is a variable that holds a memory address.

Computer memory is where variable values are stored. By convention, computer memory is divided into sequentially numbered memory locations. Each of these locations is a memory address.

Every variable of every type is located at a unique address in memory. Figure 10.1 shows a schematic representation of the storage of an unsigned long integer variable, theAge.



The special character \t in causes a tab character to be inserted in the output. This is a simple way of creating columns. There are other useful characters like this in addition to the newline character \n.
The \\ character is used to display a backslash.
The \" character is used to display a double quote.
The \' character is used to display a single quote.
Address of operator &.
The compiler takes care of assigning the actual address.

When a pointer is allocated, the compiler assigns enough memory to hold an address in your hardware and operating system environment. The size of a pointer might or might not be the same size as an integer, so be sure you make no assumptions.

- **Storing the Address in a Pointer**

Every variable has an address. Even without knowing the specific address of a given variable, you can store that address in a pointer.
To declare a pointer called pAge to hold its address, the following statemet:
int *pAge = NULL;
The convention of naming all pointers with an initial p and a second letter capitalized, as in pAge.
A pointer is just a special type of variable that holds the address of an object in memory.
A pointer whose value is NULL is called a null pointer. All pointers, when they are created, should be initialized to something. If you don't know what you want to assign to the pointer, assign NULL.

The next version of C, C++0x, has a new nullptr constant that represents a null pointer. When your C++ compiler supports this new version, use nullptr instead of a or NULL.

If you initialize the pointer to 0 or NULL, you must specifically assign the address of howOld to pAge.

```
int howOld = 50; // make a variable
int *pAge = 0; // make a pointers
pAge = &howOld; // put howOld's address in pAge
```

Assigning a nonpointer to a pointer variable is a common error. Fortunately, the compiler will detect this and fail with an “invalid conversion” error.
You could have accomplished this with fewer steps:

```
unsigned short int howOld =50; // make a variable
unsigned short int *pAge=&howOld; // make pointer to howOld
```

- **The Indirection Operator, or Using Pointers Indirectly**

The indirection operator * also is called the dereference operator. When a pointer is dereferenced, the value at the address stored by the pointer is retrieved.
A pointer provides in direct access to the value of the variable whose address it stores.

```
unsigned short int howOld =50; // create the variable howOld
unsigned short int *pAge = &howOld; // pAge points to the address of howOld
unsigned short int yourAge; // create another variable
yourAge = *pAge; // assign value at pAge (50) to yourAge
```

The indirection operator * in front of the variable pAge means “the value stored at.”

The indirection operator * is used in two distinct ways with pointers: declaration and dereference. When a pointer is declared, the star indicates that it is a pointer, not a normal variable. For example:

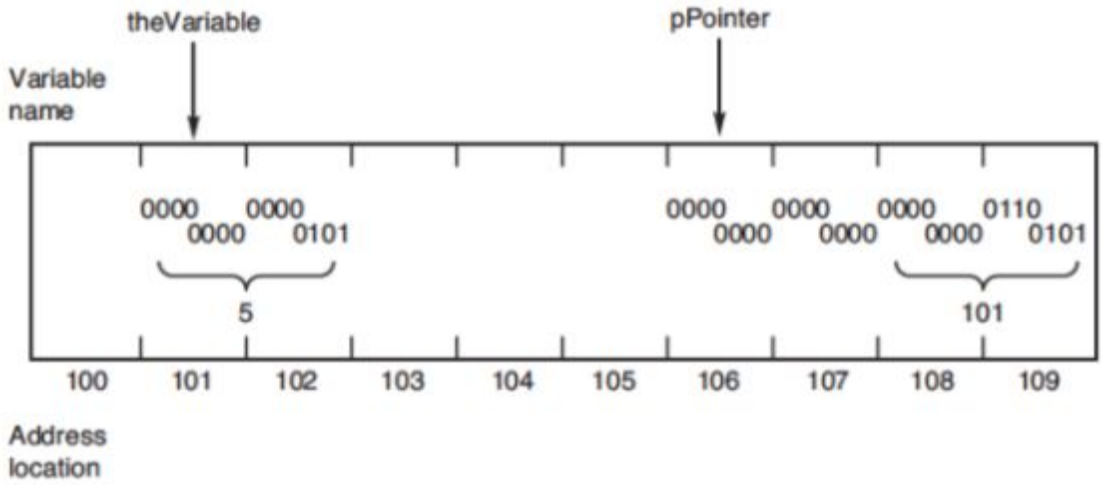
```
unsigned short *pAge =NULL; // make a pointer to an unsigned short
```

When the pointer is dereferenced, the indirection operator indicates that the value at the memory location stored in the pointer is to be accessed, rather than the address itself:

```
*pAge =5; // assign 5 to the value at pAge
```

- Pointera, Addresses, and Variables

Consider the following code fragment:
```
int theVariable =5;
int *pPointer = &theVariable;
```
The address that pPointer holds is the address of theVariable. The value at the address that pPointer holds is 5.



- **Manipulating Data by Using Pointers**

After a pointer is assigned the address of a variable, you can use that pointer to access the data in that variable.
Pointers enable you to manipulate addresses without ever knowing their real value.

unsigned short int myAge =5;
unsigned short int *pAge =&myAge;
cout << “*pAge: \t” << *pAge << “\n\n”;

It displays the result of dereferencing pAge, which displays the value at pAge - the value in myAge, or 5. it shows how to get the value stored in myAge by dereferencing the pointer pAge.

- **Why to Use Pointers**

Pointers are employed most often for three tasks:
- Managing data on the heap
- Accessing class member data and functions
- Passing variables by reference to functions

- **The Stack and the Heap**

Programmers generally deal with five areas of memory:
- Global name space
- The heap
- Registers
- Code space
- The stack

Local variables are on the stack, along with function parameters. Code is in code space, and global variables are in global name space. The registers are used for internal housekeeping functions, such as keeping track of the top of the stack and the instruction pointer. Just about all remaining memory is given over to the heap, which is sometimes referred to as the free store.

When the funtion returns, the local variables are thrown away. Global variables being accessible without restriction throughout the program, leads to the creation of bug-prone code. Putting data in the heap solves both these problems.

The stack is cleaned automatically when a function returns. All the local variables go out of scope, and they removed from the stack. The heap is not cleaned until your program ends.

The advantage to the heap is that the memory you reserve remains available until you explicitly free it.

- **Using the new Keyword**

You allocate memory on the heap in C++ by using the new keyword.
The return value from new is a memory address. It must be assigned to a pointer. To create an unsigned short on the heap, write:

```
unsigned short int *pPointer;
pPointer = new unsigned short int;
```

Or

```
unsigned short int *pPointer = new unsigned short int;
```

pPointer points to an unsigned short int on the heap.

```
*pPointer =72;
```

This means “put 72 at the value in pPointer” or “assign the value 72 to the area on the heap to which pPointer points.”

- **Using the delete Keyword**

When you have finished with your area of memory, you must call delete on the pointer, which returns the memory to the heap. The pointer itself is a local variable. When the function in which it is declared returns, that pointer goes out of scope and is lost. The memory becomes unavailable - a situation called a memory leak.

To restore the mempry to the heap, you use the keyword delete. For example:

```
delete pPointer;
```

When you call delete on a pointer, the memory it points to is freed; set it to NULL. Calling delete on a null pointer is guaranteed to be safe. For example:

```
Animal *pDog =new Animal;
delete pDog;    // frees the memory
pDog =NULL;  // sets pointer to null
// …
delete pDog;   // harmless
```

For every time in your program that you call new, there should be a call to delete. It is important to keep track of which pointer owns an area of memory and to ensure that the memory is returned to the heap when you are done with it.

- **Creating Objects on the Heap**

If you have declared an object of type Cat, you can declare a pointer to that class and instantiate a Cat object on the heap, just as you can make one on the stack. The syntax is the same as for intergers:

Cat *pCat = new Cat;

This calls the default constructor-the consructor that takes no parameters. The construcor is called whenever an object is created on the stack or on the heap.

- **Accessing Data Members Using Pointers**

Access data members and functions by using the dot operator(.) for Cat objects created locally. To access the Cat object on the heap, you must dereference the pointer and call the dot operator on the object pointed to by the pointer.

(*pRags).GetAge();

Parentheses are used to assue that pRags is dereferenced before GetAge() is accessed.

C++ provides a shorthand operator for indirect access:
the points -to operator ->,which is created by typing a dash (-) immediately followed by the greater than symbol (>). C++ treats this as a single symbol.

Every class member function has a hidden parameter: **the this pointer**. This points to the individual object.

```
const int *p1;    // the int pointed to is constant
int  * const p2;   // p2 is constant, it can't point to anything else
```

When you declare an object to be const, you are, in effect, declaring that the this pointer is a pointer to a const object. A const this pointer can be used only with const member funcions.

Pointers can be created to point to simple data types like integers and to objects as well.

Objects can be created and deleted on the heap. If you hace declared an object, you can declare a pointer to that class and instantiate the object on the heap.

Data members of a class can be pointers to objects pn the heap. Memory can be allocated in the class construcor or one of its functions and deleted in the destructor.

9. Creating References

- **What is a Reference?**

When you create a reference, you initialize it with the name of another object, the target. Although references are often implemented using pointers, that is a mateer of concern only to creators of compilers; as a programmer, you must keep these two ideas distinct.

Pointers are variables that hold the address of another object. References are aliases to an object.

- **Creating a Reference**

By writing the type of the target object, followed by the reference operator&, followed by the name of the references name of the reference.

References can use any legal variable name, but in this book all reference names are prefixed with r and the second leter is capitalized.

So, if you have an integer variable named someInt, you can make a reference to that variable by writing the following:
int &rSomeRef = someInt;

References must be initialized.

C++ gives you no way to access the address of the reference itself because it is not meaningful, as it would be if you were using a pointer or other variable. References are initialized when created and always act as a synonym for their target, even when the address of operator is applied.

- **Passing Function Arguments by Reference**

Passing by reference is accomplished in two ways: using pointers and using references. The syntax is different, but the net effect is the same: Rather than a copy being created within the scope of the function, the actual original object is passed into the function.

Passing an object by reference enables the function to change the object being referred to.

Functions can return only one value. What if you need to get two values back from a function? One way to solve this problem is to pass two objects into the function by reference. The function then can fill the objects with the correct values.

References vs. Pointers
References are easier to use and understand. The indirection is hidden, and there is no need to repeatedly dereference the variable.
References cannot be NULL, and they cannot be reassigned. Pointers offer greater flexibility but are slightly more difficult to use.

10. Developing Advanced References and Pointers

Each time you pass object into a function by value, a copy of the object is made. Each time you return an object from a function by value, another copy is made.

The size of a user-created object on the stack is the sum of each its member variables. These, in turn, can each become user-created objects. Passing such a massive structure by copying it onto the stack can be expensive in terms of performance and memory consumption.

There is another cost. With the classes you create, each of these temporary copies is created when the compiler calls a special constructor: the copy constructor.

The copy constructor is called each time a temporary copy of the object is put on the stack. When the temporary object is destoryed, which happens when the function returns, the object's desructor is called. If an object is returned by value, a copy of that object must also be made and destoryed.

Generally, C++ programmers strongly prefer references to pointers because they are cleaner and easier to use. References cannot be reassigned, however. If you need to point first to one object and then to another, you must use a pointer. References cannot be NULL, so if there is any chance that the object in question might be, you must use a pointer rather than a reference. If you want to allocate dynamic memory from the heap, you have to use pointers as discussed in previous.

Remeber that a reference always is an alias that refers to some other object. If you pass a reference into or out of a function, be sure to ask yourself, "What is the object I'm alising, and will it still exist every time it's used?"

11. Calling Advanced Functions

- **Initializing Objects**

Constructors are created in two stages: the initializtion stage and the body of the construcotr. A member variable can be set during the intialization or by assigning it a value in the body of the constructor. The following examples shows how to initialize member variables:

```
Tricycle :: Tricycle():
speed(5),
wheelSize(12)
{
   // body of constructor
}
```

To assign values in a constructor's initialization, put a conlon after the closing parentheses of the construcor's parameter list. After the colon, list the name of a member variable followed by a pair of parentheses. Inside the parantheses, put an expression that initializes the member variable. If more than one variable is being set in this manner, separate each one with a comma.

Because references and constants cannot be assigned values, they must be initialized using this technique.

12. Using Operator Overloading

Operator overloading defines what happens when a specific operator is used with an object of a class. Almost all operators in C++ can be overloaded.

The most common way to overload an operator in a class is to use a member function. The function declaration takes this form:

returnType operatior symbol (parameter list)

```
{
   // body of overload member function
}
```

- **Limitations on Operator Overloading**

OPerators for built-in types such as int cannot be overloaded. The precedence order cannot be changed, and the arity of the operator -whether it is unary, binary, or trinary -cannot be altered, either. You also cannot make up new operators, so there's no way to do something such as ** to be the exponnentiation(power of) operator.

A shallow copy just copies the members, making both objects point to the same area on the heap. A deep copy allocates the necessary memory.

13. Extending Classes with Inheritance

- **The Syntax of Derivation**

When you create a class that inherits from another class in C++, in the class declaraion you put a colon after the class name and specify the access level of the class (public, protected, or private) and the class from which it derives.

class Dog : public Mammal

There are three access specifiers: public, protected, and private. If a function has an instance of a class, it can access all the public member data and functions of that class. The member function of a class, however, can access all the private data members and functions of any class from which they derive.

Constuctors are called in order of inheritance. Destuctors are called in reverse order of inheritance.

Overriding a function means changing the implementation of a base class function in a derived class. When you make an object of the derived class, the correct function is called.

When you override a function, it must agree in return type and in signature with the function in the base class. The signature is the function prototype other than the return type: that is, the name, the parameter list, and the keyword const, if used.

- **Overloading Versus Overriding**

The terms overloading and overriding are similar and do similar things in C++. When you overload a member function, you create more than one function with the same name but with different signatures. When you override a member function, you create a function in a derived class with the same name as a function in the base class and with the same signarure.

It is a common mistake to hide a base class method, when you intend to override it, by forgetting to include the keyword const. That keyword is part of the signature, and leaving it off changes the signature and thus hides the member function instead of overriding it.

- **Calling the Base Method**

if you have overridden the base method, it is still possible to call it by fully qualifying the name of the method. You do this by writing the base name, followed by two colons and then the method name. For example:

fido.Mammal: :move()

14. Using Polymorphism and Derived Classes

- **What virtual methods are**

A Dog object is a Mammal object. This meant that the Dog object inherited the attributes (data) and capabilities (member functions) of its base class. The relationship between a base class and derived class runs deeper than that in C++.

Polymorphism allows derived objects to be treated as if they were base objects. For example, suppose you create specialized Mammal types such as Dog, Cat, Horse, and so forth. All these derive from Mammal, and Mammal has a number of member functins factered out of the derived classes.

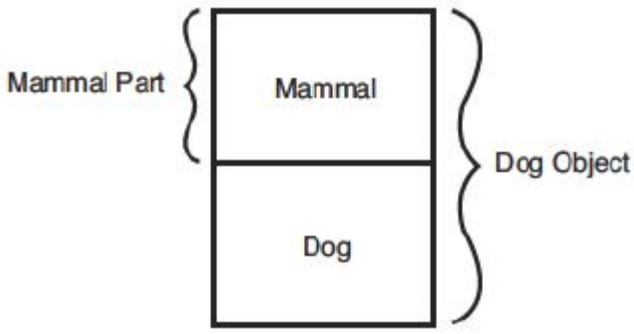
Polmorphism is an unusual word that means the ability to take many forms. It comes from the roots poly, which means many, and morph, which means form. You are dealing with Mammal in its many forms.

You can use polymorphism to declare a pointer to Mammal and assign to it the address of a Dog object you create on the heap.
Mammal* pMammal = new Dog;

You then can use this pointer to invoke any member function on Mamaml. What you would like is for those functions that are overriden in Dog to call the correct function.
Virtual member functions let you do that.

- **How Virtual Member Functions Work**

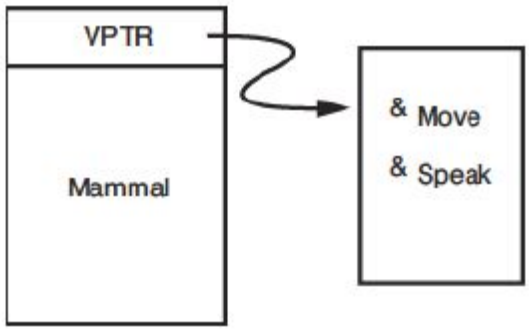
When a derived object, such as a Dog object, is created, first the constructor for the base class is called, and then the constructor for the derived class is called.



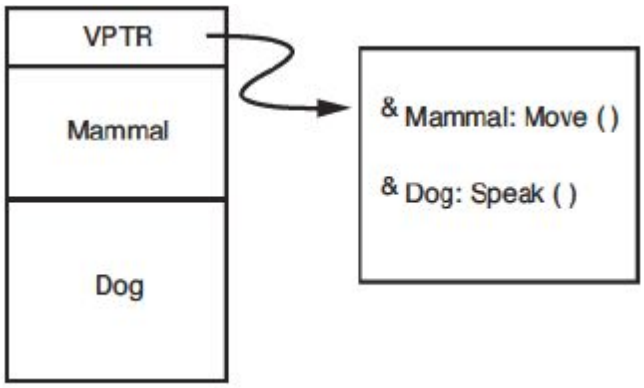
When a virtual function is created in an object, the object must keep track of that function. Many compilers build a virtual function table, called a v-table. One of these is kept for each

type, and each object of that type keeps a virtual table pointer (called a vptr ot v-pointer), which points to that table.

Each object's vptr points to the v-table that, in turn, has a pointer to each of the virtual member functions.When the Mammal part of the Dog is created, the vptr is initialized to point to the virtual methods for Mammal.



When the Dog constructor is called and the Dog part of this object is added, the vptr is adjusted to point to the virtual function overrides (if any) in the Dog object.



When a pointer to a Mammal is used, the vptr continues to point to the correct function, depending on the real type of the object.

The rule of thumb is this: If any of the functions in your class are virtual, the destructor should also be virtual.

Create a clone member function in the base class and make it virtual. A clone function creates a new copy of the current object and returns that object.

After you declare any functions virtual, you've paid most of the price of the v-table (although each entry does add a small memory overhead). At that point, you want the destructor to be vietual, and the assumption will be that all other functions progably will also be virtual.

Polymorphism enables the same interface to be implemented with different member functions in a set of classes related by inheritance. Polymorphism achieves an important goal in object-oriented programming by letting similar objects handle related functionality by reusing an interface.

- **Dynamic_cast operator**

Here's how it works: If you have a pointer to a base class, such as Mammal, and you assign to it a pointer to a derived class, such as Cat, you can use the Mammal pointer

polymorphically to access virtual functions. Then, if you need to get at the Cat object to call, for example, the purr() method, you create a Cat pointer using the dynamic_cast operator to do so.

abstract data type(ADT)
An abstract data type represents a concept (like shape) rather than an object (like circle). In C++, an ADT is always the base class to other classes, and it is not valid to make an instance of an ADT.

- **Pure Virtual Functions**

C++ supports the creation of abstract data types with pure virtual functions. A pure cirtual function is a virtual function that must be overriddent in the derived class. A virtual function is made pure by initializing it with 0, as in the following:

virtual void draw() =0;

Any class with one or more pure virtual functions is an ADT, and it is illegal to instantiate an object of a class that is an ADT. Trying to do so causes a compile-time error. Putting a pure virtual function in your class signals two things to clients of your class:

Don’t make an object of this class; derive from it.
Make sure to override the pure virtual function.

Any class that derives from an ADT inherits the pure virtual function as pure, and so must override every pure virtual function if it wants to instantiate objects.

- **summary**

An anstract data type is a class that cannot be implemented as an object. Instead, it defines common member variables and functions for its derived classes.

A function becomes a pure virtual function by adding = 0 to the end of its declartion. If a class contains at least one pure function, the class is an abstract data type.

The compliler will not allow objects of an abstract data type to be instantiated.

15. Storing Information in Linked Lists

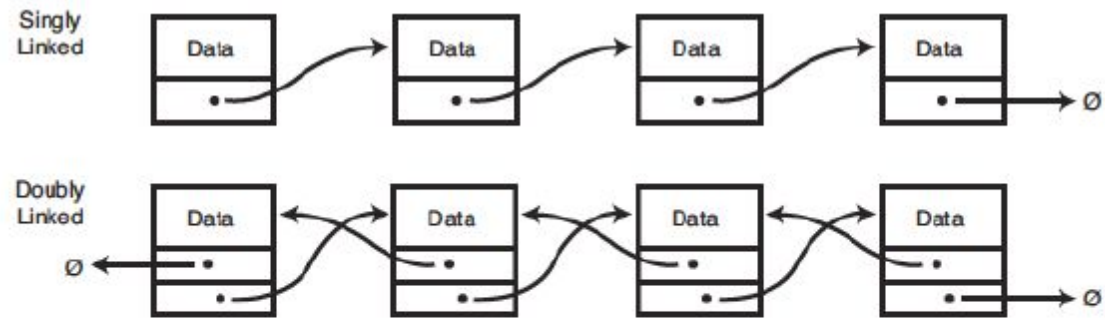
- **What a linked list is**

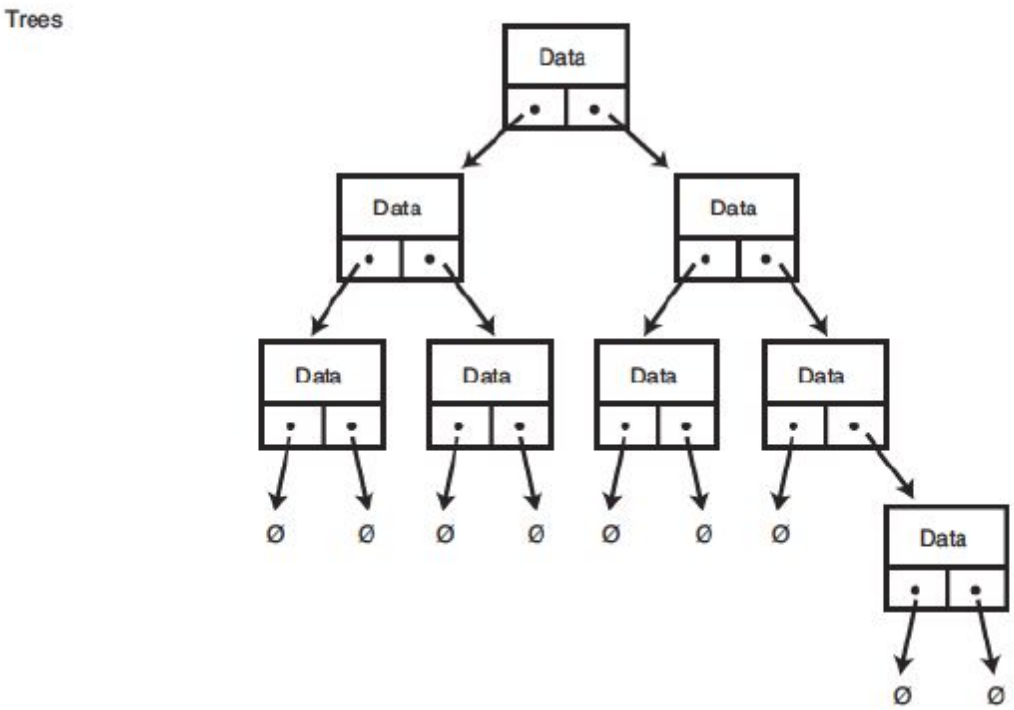
Arrays are much like Tupperware products. a fixed size. pick too large container, waste space. too small one, not enough room to hold what you need to store.

A linked list is a data structure that consists of small continers that connect together. The idea is to write a class that holds one object of your data-such as one Cat or one Rectangle-and knows how to point to the next container in the list.

Singly linked
Doubly linked

Trees





- **Liked Lists as Objects**

In object-oriented programming, each infividual object is given a narrow and well-definedset of responsibilities. The linked list is responsible for maintaining the head node. The HeadNode immediately passes any new data on to whatever it currently points to, without regard for what that might be.

The TailNode, whenever it is handed data, creates a new node and inserts it. It knows only one thing: If this came to me, it gets inserted right before me.