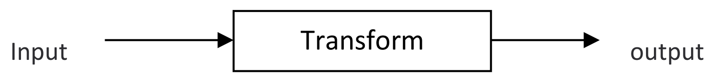
ABSTRACTION

Abstraction is a technique for managing complexity. It works by suppressing the more complex details below the current level of abstraction. details may abstracted away, (information hiding) leaving just the definition or programmer interface (the what as opposed to the how).



The Black Box model

Abstraction in computing has evolved on many levels and has been going on ever since those first "virtual machines" which led to the "FOR- TRAN machines" and other high-level, automatic coding systems. It has been going on since the first COBOL compiler modeled an abstract machine that understood English¬ like commands and worked with data files. After all, a computer is a nice little gadget, but it is not exactly a machine, in the sense of doing anything useful, without appropriate software. In that sense, any software provides the computer user with a machine abstraction and, indeed, it has been so taken over the years. Thus, programming languages and their processors become abstract, high-level, even programmer-friendly, "computers," which may be used without regard for such machine-level clerical details as keeping track of memory locations, program instruction counters, etc.

Even the evolution of programming languages has followed a course of continuing abstraction of these languages and systems away from the level of the machine and towards the level of the human user.

Abstraction can apply to control (operations) or to data:

- Control abstraction is the abstraction of program operations and involves the use of subprograms like functions
- Data abstraction allows for the organizing of data in meaningful ways. This includes the notion of a datatype. Data-level abstractions include data types, which are generally built into the language and abstract data types. An abstract data type facility may be used to implement data structures as true user-defined data types by hiding the details of the implementation (the how) behind the wall of a user interface (the what). In the object-oriented programming paradigm, the object is a data abstraction. (Also, the data models of database management systems are data-level abstractions.)

An object is a way to combine both data abstractions and control abstraction.

We have already seen a lot of abstraction in C++. Functions, classes, data types, etc.

Another way to look at it:

		Do I see it?	
		Yes	No
Is it there?	Yes	REAL	TRANSPARENT
	No	VIRTUAL	ABSTRACT

Let’s explain these terms:
Real.
Transparent.
Virtual.
Abstract.

C++ is a fairly strongly types language and we have to implement abstraction with virtual functions. This is the way we handle true polymorphism in C++.

Consider the following example of a virtual function.

```
//modified from LaFore p. 460
//virtualfunction.cpp

#include <iostream>
#include <string>
using namespace std;

class person {
      protected:
      string name;
   public:
      void EnterName(){cout << "Enter Name:  "; cin >> name;}
      void printName(){cout << "Name is:  " << name << endl;}
      virtual void EnterData() = 0;       //pure virtual function
      virtual bool isOutstanding() = 0; //pure virtual function
};
//*****************************************************************
class student : public person {
      private:
      float gpa;
   public:
      void EnterData() {
            person::EnterName(); //call to method in base class
            cout << "Enter student's GPA: "; cin >> gpa;
            }
      bool isOutstanding(){return (gpa > 3.5) ? true : false;}
};
//*****************************************************************
class professor : public person {
      private:
      int numPubs;
   public:
      void EnterData(){
            person::EnterName();
            cout << "Enter number of professor's publications: ";
            cin >> numPubs;
            }
      bool isOutstanding() {return (numPubs > 100) ? true : false;}
};
//*****************************************************************
int main(){
      person* persPtr[100];
   int n = 0;
   char choice;
   cout << "Any data to enter? (y/n)"; cin >> choice;
   while (choice=='y') {
      cout << "Entering student or professor? (s/p):  ";
      cin >> choice;
      if (choice == 's') persPtr[n] = new student;
            else persPtr[n] = new professor;
      persPtr[n]->EnterData();
      n++;
      cout << "\nEnter another? (y/n):  "; cin >> choice;
      }
   for (int i=0; i<n; i++) {
      persPtr[i]->printName();
      if (persPtr[i]->isOutstanding())
            cout << "This person is outstanding!" << endl;
      }

   cout << "\n\n\nPress any key to close console window:  ";
   char c; cin >> c;
      return 0;
}
```

```
Any data to enter? (y/n)y
Entering student or professor? (s/p):  s
Enter Name:  VinceMcMahon
Enter student's GPA: 2.2

Enter another? (y/n):  y
Entering student or professor? (s/p):  p
Enter Name:  AndretheGiant
Enter number of professor's publications: 250

Enter another? (y/n):  y
Entering student or professor? (s/p):  p
Enter Name:  ElvisCole
Enter number of professor's publications: 23

Enter another? (y/n):  y
Entering student or professor? (s/p):  s
```

```
Enter Name:  DuncanMcLeod
Enter student's GPA: 3.9

Enter another? (y/n):  n
Name is:  VinceMcMahon
Name is:  AndretheGiant
This person is outstanding!
Name is:  ElvisCole
Name is:  DuncanMcLeod
This person is outstanding!
```

Constructors cannot be “virtual” since they cannot be overridden. An explicit destructor *can* be virtual.

How does C++ implement “virtual”? With **late binding**.

Late Binding ≡ dynamic binding, i.e., during runtime
Early Binding ≡ static binding, i.e., during compile

In this case we don’t know what class the contents of a pointer might be until execution.

Virtual functions with late binding is the way C++ carries out **polymorphism**.

Polymorphism is often associated with inheritance. In C++ a class containing a pure virtual method (function) is termed an abstract class, and there must exist a derived class for creating objects (instantiation).