CS215

Programming using C++ II

Outline

- Fundamentals of C++
- I/O Streams
- Class & inheritance
- Overloading & overriding
- Templates, Error handling,…

Programming Methodologies

- Programming Methodologies
  - Structured Programming: Dividing a problem into smaller subproblems is called structured design. Each subproblem is then analyzed, and a solution is obtained to solve the subproblem. The structured-design approach is also called modular programming.
  - Object-Oriented Programming (OOP): based on Object-oriented design (OOD)is a widely used programming methodology. In OOD, the first step in the problem-solving process is to identify the components called objects, which form the basis of the solution, and to determine how these objects interact with one another.

Basic Elements of C++

Become familiar with the basic components of a C++ program, including functions, special symbols, and identifiers.

- Explore simple data types

```
#include<iostream>
using namespace std;
int main() {
    int  i=0;
    double x=2.3;
    char s[]="Hello";

    cout<< i <<endl;
    cout<< x <<endl;
    cout<< s <<endl;
    return 0;
}
```

Data Attributes

All data in a C++ program can be described by its data attributes. The data attributes are:

- **Data Type**: By the data type of a variable is meant:
  - How much memory does the variable require
  - How is that memory structured
  - How is that memory accessed by functions and operators.
- **Storage Class:** By storage classis meant of memory (e.g. registers, stack or heap) is assigned to the variable. In C++ there are five storage classes:
  - Automatic is defined on the stack
  - Register is defined in the CPU's registers or defaults to the stack
  - Static is defined on the heap
  - External is defined on the heap
  - Dynamically allocated is defined on the heap.

Data Attributes

- **Duration or Lifetime:** By duration or lifetime of a variable is meant how long a variable will be assigned its memory location. The lifetimes of a C++ variable are:
  - life of the program
  - life of a function
  - life of an object of a class
  - life of a block of code.
- **Scope or Visibility:** By the scope of a variable is meant where the variable is visible or where can it be accessed. In C++ there are five scopes of a variable:
  - block
  - function
  - file
  - class or class object
  - Program.

Type Conversion (Casting)

The cast operator, also called type conversion or type casting is used to convert one type to an acceptable other type. , takes the following form:

static_cast<dataTypeName>(expression)

- Examples:

```
int i=5;
float f;
f = static_cast<float>(i)/2;
```

Expression	Evaluates to
`static_cast<int>(7.9)`	7
`static_cast<int>(3.3)`	3
`static_cast<double>(25)`	25.0
`static_cast<double>(5+3)`	`=static_cast<double>(8)` = 8.0
`static_cast<double>(15) / 2`	= 15.0 / 2 (because `static_cast<double>` (15) = 15.0) = 15.0 / 2.0 = 7.5
`static_cast<double>(15 / 2)`	`=static_cast<double>(7)` (because 15 / 2 = 7) = 7.0

`static_cast<int>(7.8 +`
`static_cast<double>(15) / 2)`
	`=static_cast<int>(7.8+7.5)`
	`=static_cast<int>(15.3)`
	= 15

`static_cast<int>(7.8 +`
`static_cast<double>(15 / 2))`
	`=static_cast<int>(7.8 + 7.0)`
	`=static_cast<int>(14.8)`
	= 14

Passing Arguments

- **Passing Arguments by Value**: Passing by value has several undesirable characteristics to include variable:
  - does not change the argument value
  - may place excessive demand on the stack
  - slows the program when a large variable or a large number of variables must be passed to the stack.
- **Passing and Returning by Reference or Pointer**: A reference of a variable has been discussed in the notes above. A reference of a variable is another name for the variable. References may be used as arguments of functions called passing by reference as in the following functional prototype:
  - modification of multiple variables
  - faster function execution
  - less stack memory use.

Dynamic Arrays

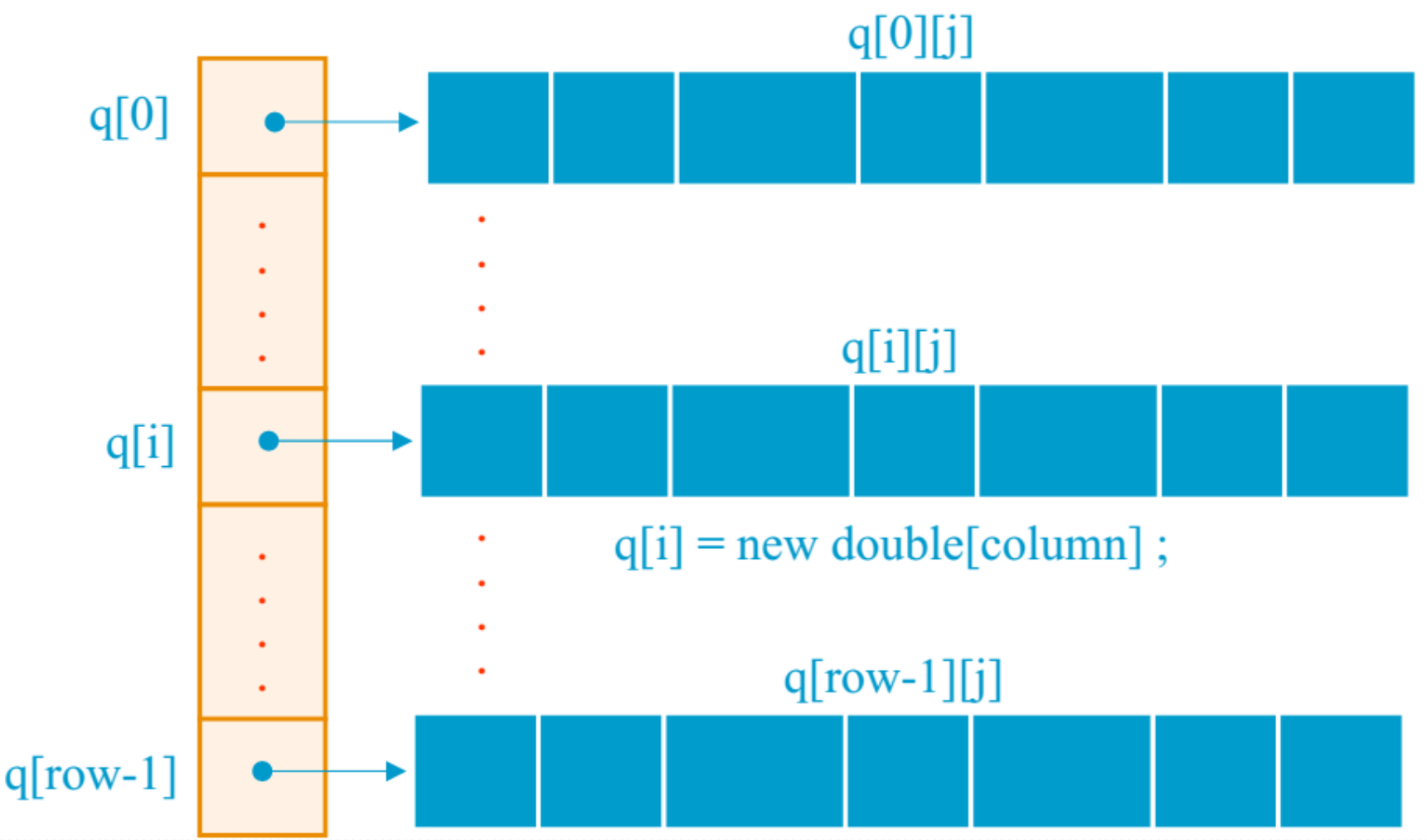
- One Dimensional Array

```
int *p ;
p = new int[N] ;
delete []p ;
```

- Two Dimensional Array

```
double ** q ;
q = new double*[row] ; // matrix size is rowxcolumn
for(int i=0; i<row; i++)
    q[i] = new double[column] ;
.....
for(int i=0; i<row; i++)
     delete []q[i] ;
delete []q ;
```

Dynamic Arrays



Function Declarations and Definitions

Example

```
char  grade (int, int, int); // declaration
int main() {
:
}
// Function definition
char  grade (int exam1, int exam2, int finalExam) {
.
 .   // body of function
.
}
```

Passing Arguments By values

Parameters Passing: Consider swap() function

```
void swap(int a, int b){
        int temp = a ;
        a = b ;
        b = temp ;
}
int main(){
        int i=3,j=5 ;
        swap(i,j) ;
        cout << i << " " << j << endl ;
}
```

Passing Arguments By Pointers

Parameters Passing: Consider swap() function

```
void swap(int* a, int* b){
        int temp = *a ;
        *a = *b ;
        * = temp ;
}
int main(){
        int i=3, j=5 ;
        swap(&i, &j) ;
        cout << i << " " << j << endl ;
}
```

Passing Arguments By References

Parameters Passing: Consider swap() function

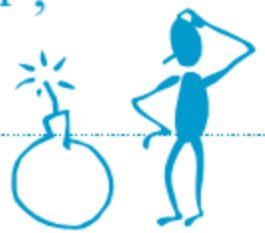
```
void swap(int &a, int &b){
        int temp = a ;
        a = b ;
        b = temp ;
}
int main(){
        int i=3,j=5 ;
        swap(i, j) ;
        cout << i << " " << j << endl ;
}
```

```
int squareByValue(int a){
        return (a*a) ;
}
```

```
void squareByReference(int& a){
        a *= a ;
}
```

```
int main(){
        int x=2,y=3,z=4 ;
        squareByPointer(&x) ;
        cout << x << endl ;
        squareByReference(y) ;
        cout << y << endl ;
        z = squareByValue(z) ;
        cout << z << endl ;
}
```

```
void squareByPointer(int *aPtr){
        *aPtr = *aPtr**aPtr ;
}
```



4
9
16

Return by Reference

- By default in C++, when a function returns a value; An alternative to "return by value" is "return by reference".

For example:

```
const int& max( const int a[], int length) { // Returns an integer reference
    int i=0;  // index of the largest element
    for (int j=0 ; j<length ; j++)
            if (a[j] > a[i]) i = j;
    return a[i]; // returns referenceto a[i]
}
int main() {
    int array[ ] = {12, -54 , 0 , 123, 63}; // An array with 5 elements
    int largest; // A variable to hold the largest elem.
    largest = max(array,5); // find the largest element
    cout << "Largest element is " << largest << endl;
}
```

Return by Reference

- Never return a local variable by reference : Since a function that uses "return by reference" returns an actual memory address, it is important that the variable in this memory location remain in existence after the function returns. When a function returns, local variables go out of existence and their values are lost.

```
int&  f( ) {                    // Return by reference
    int i;                      // Local variable. Created in stack
    :
    return  i;                  // ERROR! i does not exist anymore.
}
```

Passing Arguments

- **Functions with Default Values**: Default arguments are created in the signature to the right of the mandatory arguments by:
  - indicating the data type of the argument
  - naming the variable immediately to the right of the data type
  - listing the assignment operator (=) after the default variable
  - listing the default value to be passed to the variable after the assignment operator.

Functions with Default Values

- For example, suppose that the function: f(…)had the following prototype:

```
int f(int a, int b, int c=6, int d=10) {
      . . .
 }
```

then some possible calls for f()are:

- f(5,6); // both defaults used
- f(5,6,7); // 1st default is overridden
- f(5,6,7,8); // both defaults are overridden

Functions with Default Values

- In calling a function argument must be given from left to right without skipping any parameter .

```
void f(int i, int j=7) ; // right
void g(int i=3, int j) ; // wrong
void h(int i, int j=3,int k=7) ; // right
void m(int i=1, int j=2,int k=3) ; // right
void n(int i=2, int j,int k=3) ; // right?wrong
```

Function Name Overloading

- Function name overloading permits the programmer to use the same function name when the functions have a different signature. This is most useful when the different functions have the same objectives. For example:

- int MAX(int a, int b) { return (a>b?a:b); }

- long MAX(long a, long b) { return (a>b?a:b); }

- float MAX(float a, float b) { return (a>b?a:b); }

- double MAX(double a, double b) { return (a>b?a:b); }

Function Name Overloading

```
double average(const double a[],int size) ;
double average(const int a[],int size) ;
double average(const int a[], const double b[],int size) ;
.
.
.
int main() {
      int w[5]={1, 2, 3, 4, 5} ;
      double x[5]={1.1, 2.2, 3.3, 4.4, 5.5} ;
      cout << average(w, 5) << endl;
      cout << average(x, 5) << endl;
      cout << average(w, x, 5) << endl;
     return 0 ;
}
```

Function Templates

- Example

```
template <typename T>
void printArray(const T *array, const int size) {
    for(int i=0; i < size; i++)
        cout << array[i] << "";
    cout << endl ;
}

int main() {
    int a[3]={1, 2, 3} ;
    double b[5]={1.1, 2.2, 3.3, 4.4, 5.5} ;
    char c[7]={'a', 'b', 'c', 'd', 'e' , 'f', 'g'} ;
    printArray(a, 3) ;
    printArray(b, 5) ;
    printArray(c, 7) ;
}
```

Programmer Defined Data Types

- **Enumerated Data Types:** The general construct of enumerated data types is:

```
enum tag { member_list };
```

where tag is the name of the enumerated data type. For example:

```
enum Boolean {FALSE, TRUE};
Boolean flag = TRUE;

.....
if (flag)
    cout << "The goal was met.\n";
else
    cout << "The goal was not met.\n";
```

Enumerated Data Types

However, no two enumerated data types may use the same name for its enumerators. That is the following two enumerated data type definitions could not be in the same program:

```
enum Stype {RED, COUPE, SUDAN, SUV};
enum Color {WHITE, GREEN, RED, YELLOW, BLUE}
```

because they both have the enumerator: RED.

- By default the enumerators of an enumerated data type are constant integers that begin with the first enumerator having (by default) the value 0 and each consecutive enumerator has the integral value one more than the previous enumerator. However these rules may be overridden as can be seen in the following examples:

```
enum Colors {BLACK=1, RED, YELLOW, GREEN, BLUE};
```

Here BLACK is 1, RED is 2, YELLOW is 3, GREEN is 4 and BLUE is 5.

Enumerated Data Types

```
enum Position {NORTH, EAST=90, SOUTH=180, WEST=270};
enum Base {FIRST, SECOND=200, THIRD};
enum Value {ZERO, NULL=0, ONE, NUMERO_UNO=1};
```

- Operations on Enumeration Types

```
Enum sports {BASKETBALL, FOOTBALL, HOCKEY, BASEBALL,
SOCCER, VOLLEYBALL};
sports popularSport, mySport;
popularSport = FOOTBALL; // assignment
popularSport =static_cast<sports>(popularSport + 1); //addition
FOOTBALL <= SOCCER // is legal -> true
for(mySport = BASKETBALL; mySport <= SOCCER;
        mySport =static_cast<sports>(mySport + 1))
        . . .
```

Structures

- Definition of a structure is:

```
struct Tag {
    member list
};
```

Note: Currently it is recommended that the structure's name start with a cap.

- For example:

```
struct Date {
    int theMonth;
    int theDay;
    int theYear;
};
```

```
Date invoiceDate, shippingDate;  // declaration
```

Structures

- Example 2:

```
struct Stuff {
    char name[30];
    float price;
    };

Stuff item1 = {"Widget", 5.75};  // declaration and initialization
Stuff item2;
item2 = item1;

cout << "The name of item 1 is " << item1.name << endl
     << "The price of item 1 is " << item1.price << endl
     << "The name ofitem 2 is " << item2.name << endl
     << "The price of item 2 is " << item2.price << endl;
```

Structures

- Example 3:

```
enum workType {salaried, hourly};
struct Date {
        int theMonth, theDay, theYear;
};
struct Employee {
        Date hireDate;
        workType salaryType;
        har employeeLastName[15],
        employeeFirstName[10];
        float unitPay;
};

Employee oldStaff = { {6,12,70}, salaried, "Thomas", "John", 350.00};
```

```
struct EmployeeType {
  string firstname;
  string middlename;
  string lastname;
  string empID;
  string address1;
  string address2;
  string city;
  string state;
  string zip;
  int hiremonth;
  int hireday;
  int hireyear;
  int quitmonth;
  int quitday;
  int quityear;
  string phone;
  string cellphone;
  string fax;
  string pager;
  string email;
  string deptID;
  double salary;
};
```

```
Struct NameType {
   string first;
   string middle;
   string last;
};
Struct AddressType {
   string address1;
   string address2;
   string city;
   string state;
   string zip;
};
Struct DateType {
   int month;
   int day;
   int year;
};
Struct  ContactType {
   string phone;
   string cellphone;
   string fax;
   string pager;
   string email;
};
```

```
Struct EmployeeType {
    NameType name;
    string empID;
    AddressType address;
   DateType hireDate;
   DateType quitDate;
   CntactType contact;
   string deptID;
   doubles alary;
};
```

Structures

- Declaration array of structure

  Employee emply[100];

- References to attributes of structure

  emply[i].name.first , 0≤ i < 100
  emply[i].address.city, 0≤ i < 100
  emply[i].hireDate.month, 0≤ i < 100
  emply[i].salary, 0≤ i < 100

Operator Overloading

- In C++ it is possible to overload the built-in C++ operators such as +, -, = and ++ so that they too invoke different functions, depending on their operands.
- That is, the + in a+b will add the variables if a and b are integers, but will call a different function if a and b are variables of a user defined type.
- You can't overload operators that don't already exist in C++
- Overloaded operators make your programs easier to write, read, and maintain.
- Functions of operators have the name operator and the symbol of the operator. For example the function for the operator + will have the name operator+

Operator Overloading

```
struct Scomplex {
    float real,img;
};
SComplex operator+(SComplex v1, SComplex v2) {
    SComplex result;
    result.real=v1.real+v2.real;
    result.img=v1.img+v2.img;
    return result;
}
main() {
    SComplex c1={1, 2},c2 ={5, 1};
    SComplex c3;
    c3 = c1 + c2; // c1+(c2)
}
```

I/O stream

- Standard Input / Output
- File Input / Output

I/O Streams

- Instead of library functions (printf, scanf), in C++ library objects are used for IO operations.
- When a C++ program includes the iostream header, four objects are created and initialized:
  - ► cin handles input from the standard input, the keyboard.
  - ►cout handles output to the standard output, the screen.
  - ►cerr handles unbuffered output to the standard error device, the screen
  - ►clog handles buffered error messages to the standard error device

- To use cin and cout, every C++ program must use the preprocessor directive:

  #include<iostream>

  using namespace std;

I/O Streams

- The syntax of an input statement usingcinand the extraction operator>>is:

```
cin >> variable >> variable...;
```

cin >> payRate;

cin >> hoursWorked;

Or

cin >> payRate >> hoursWorked;

Example-1

- Suppose you have the following variable declarations:

```
int a, b;
double z;
char ch;
```

- The following statements show how the extraction operator>>works.

Statement	Input	Value Stored in Memory
`cin >> ch;`	`A`	`ch = 'A'`
`cin >> ch;`	`AB`	`ch = 'A', 'B'` is held for later input
`cin >> a;`	`48`	`a = 48`
`cin >> a;`	`46.35`	`a = 46, .35` is held for later input
`cin >> z;`	`74.35`	`z = 74.35`
`cin >> z;`	`39`	`z = 39.0`
`cin >> z >> a;`	`65.78 38`	`z = 65.78, a = 38`

Example-2

- Suppose you have the following variable declarations:

```
int a, b;
double z;
char ch;
```

- The following statements show how the extraction operator>>works.

Statement	Input	Value Stored in Memory
`cin >> a >> ch >> z;`	`57 A 26.9`	`a = 57, ch = 'A', z = 26.9`
`cin >> a >> ch >> z;`	`57  A` `26.9`	`a = 57, ch = 'A', z = 26.9`

`cin >> a >> ch >> z;`	`57`
	`A`
	`26.9`
		`a = 57, ch = 'A', z = 26.9`

`cin >> a >> ch >> z;`	`57A26.9`	`a = 57, ch = 'A', z = 26.9`

Example-3

- Suppose you have the following variable declarations:

```
int a, b;
double z;
char ch;
```

Statement	Input	Value Stored in Memory
`cin >> z >> ch >> a;`	`36.78B34`	`z = 36.78, ch = 'B', a = 34`
`cin >> z >> ch >> a;`	`36.78` `B34`	`z = 36.78, ch = 'B', a = 34`
`cin >> a >> b >> z;`	`11 34`	`a = 11, b = 34,` computer waits for the next number
`cin >> a >> z;`	`78.49`	`a = 78, z = 0.49`
`cin >> ch >> a;`	`256`	`ch = '2', a = 56`
`cin >> a >> ch;`	`256`	`a = 256,` computer waits for the input value for `ch`
`cin >> ch1 >> ch2;`	`A B`	`ch1 = 'A', ch2 = 'B'`

I/O Streams – the get Function

- The syntax of cin, together with the get function to read a character, follows:

```
cin.get(varChar);
```

- consider the following statement:

cin >> ch1 >> ch2 >> num;

can be replaced by

cin.get(ch1);

cin.get(ch2);

cin >> num;

I/O Streams – the get Function

- The syntax of cin, together with the get function to read string as array of characters asfollows:

```
cin.get(varChar, size);
```

- consider the following statement:

```
char x[20];
getline(x, 20);  // or
cin.get(x, 20);
```

I/O Streams –the ignore Function

- When you want to process only partial data (say, within a line), you can use the stream function ignore to discard a portion of the input. The syntax to use the function ignore is:

```
cin.ignore(intExp, chExp);
```

consider the following statement:

cin.ignore(100, '\n');

- ignores either the next 100 characters or all characters until the newline character is found, whichever comes first.

Example

```
int a, b;
```

and the input:

```
25 67 89 43 72
12 78 34
```

Now consider the following statements:

```
cin >> a;
cin.ignore(100, '\n');
cin >> b;
```

The first statement, `cin >> a;`, stores `25` in `a`. The second statement, `cin.ignore(100, '\n');`, discards all of the remaining numbers in the first line. The third statement, `cin >> b;`, stores `12` (from the next line) in `b`.

Exercise

Consider the declaration:

```
char ch1, ch2;
```

and the input:

```
Hello there. My name is Mickey.
```

a. Consider the following statements:

```
cin >> ch1;
cin.ignore(100, '.');
cin >> ch2;
```

The putback and peek Functions

- The syntax to use the function putback is:

```
istreamVar.putback(ch);
```

- The syntax to use the function peek is:

```
ch = istreamVar.peek();
```

```
main() {
    char ch;
    cout << "Enter a string:  ";
    cin.get(ch);
    cout << endl;
    cout << "After first cin.get(ch); "<< "ch = " << ch << endl;
   cin.get(ch);
   cout << "After second cin.get(ch); "<< "ch = " << ch << endl;
   cin.putback(ch);
   cin.get(ch);
   cout << "After putback and then "<< "cin.get(ch); ch = " << ch << endl;
   ch = cin.peek();
   cout << "After cin.peek(); ch = "<< ch << endl;
   cin.get(ch);
   cout << "After cin.get(ch); ch = "<< ch << endl;
}
```

The putback and peek Functions

Sample Run:In this sample run, the user input is shaded.

Enter a string: abcd

After first cin.get(ch); ch = a

After second cin.get(ch); ch = b

After putback and then cin.get(ch); ch = b

After cin.peek(); ch = c

After cin.get(ch); ch = c

The clearFunction

- The syntax to use the function clear is:

```
istreamVar.clear();
```

- The function clear to restore the input stream to a working state.
- After using the function clear to return the input stream to a working state, you still need to clear the rest of the garbage from the input stream. This can be accomplished by using the function ignore.

```
main(){
   int a;
   cout << "Please enter a float number: ";
   while (!(cin >> a)) { // bad input
         cin.clear();
         while (cin.get() != '\n') continue;  // or cin.ignore(100, '\n');
         cout << "Bad input; Please enter a number: ";
   }
   cout << a << endl;
   system ("pause");
}
```

Output and Formatting Output

- setprecision Manipulator: The general syntax of the setprecision manipulator is:

```
setprecision(n)
```

cout << setprecision(2);

- Fixed Manipulator: The general is:

cout << fixed;

- showpoint Manipulator: The general syntax is:

cout << showpoint;

- Setw Manipulator: example

cout << setw(5) << x << endl;

File Processing

- To create, read, write and update files.
- Sequential file processing.
- Binary file processing.

I/O File Streams

- I/O file streams require the inclusion the header:

  #include<fstream>

- Two typed of file C++ can deal with:
  - Text I/O streams.
  - Binary I/O streams.

- Using these constructors as in the following examples:

  ```
  ifstream theFile("test.txt");  // or  "test.bin"
  ostream theFile("test.txt");
  fstream theFile("test.txt", mode);
  ```

- The variable mode may include one or more of the following bits:

I/O File Streams

- The variable mode may include one or more of the following bits:

Mode	Purpose
ios::app	used for appending to end of file
ios::ate	used to move to end of file after open
ios::binary	used to change from text to binary file
ios::in	used to permit input from file
ios::nocreate	used to open if the file already exists
ios::noreplace	used to open if the file does not exists
ios::out	used to permit output to the file
ios::trunc	used to remove all data from a file

```
#include<fstream>
#include <iostream>
using namespace std;
struct employees {
      int empno;
      char name[10];
      double basic;
      };

employees emp;
void getdata(){
    cout<<endl<<"enter emp no: ";
cin>>emp.empno;
    cout<<endl<<"enter emp name: ";
cin>>emp.name;
    cout<<endl<<"enter basic pay: ";
cin>>emp.basic;
}
main(){
    ofstream outfile("asciifile.txt");
    for(int i=1; i <=3; i++) {
       getdata();
       outfile << emp.empno << " " <<
emp.name << " " << emp.basic << endl;
    }
    outfile.close();
    system("pause");
}
```

```
. . .
main(){
  ifstream infile("asciifile.txt");
  if(!infile){
       cout << "File did not open" << endl;
       system("pause");
       exit(1);
  }
  infile >> emp.empno >> emp.name >> emp.basic;
  while (infile) {
     cout<<endl<<"employee number: "<<emp.empno;
     cout<<endl<<"emp ıname "<<emp.name;
     cout<<endl<<"basic pay "<<emp.basic;
     cout<<endl;
     infile >> emp.empno >> emp.name >> emp.basic;
    }
 infile.close();
 system("pause");
}
```

```
#include<fstream>
#include<iostream>
#include<string>
using namespace std;
void main(){
string line;
// The following 2 lines open the files TESTING.TXT in same folder at the program.
/ / reads text file one line at a time and displays it on screen
    ifstream infile;
    infile.open("TESTING.TXT");
    if(!infile){
       cout << "Error opening the file." << endl ;
       exit(1);
    }
    while(infile){
       getline(infile,line);
       cout << line << endl;
    }
    infile.close();
    cout << endl ;
}
```

```
#include<fstream>
#include<iostream>
using namespace std;
main(){ // This program read text file one character at a time and
            // displays it on screen
   char ch;
   infile("readtextFile.TXT", ios::in);
   if(!infile){
      cout << "Error opening the file." << endl;
      system("pause");
      exit(1);
   }
   infile.get(ch);
   while(infile){
     cout << ch;
     infile.get(ch);
   }
   infile.close();
   cout << endl;
}
```

I/O File Streams

- When you attempt to open a file, you should test to see if it opened as in the following where a File is an object of one of the classes in the header fstream:

```
if(!aFile.is_open( ))
{
    cerr << "The file can not open"
    exit(1);
}
```

or

```
if(!aFile)
{
    cerr << "The file can not open"
    exit(1);
}
```

I/O File Streams

- As you are reading data in from a file, you must take care that there is still data to be read. This could be done with a while()loop either using the name of the object like: a File or by using the method: eof()as in the following examples:

```
while(aFile)
{
        ......

}
```

or

```
while(!aFile.eof())
{
        ..........
}
```

I/O File Streams

- When done using a file close the file by using the method: close()as in the following examples:

```
aFile.close();
```

Binary Streams and Class I/O

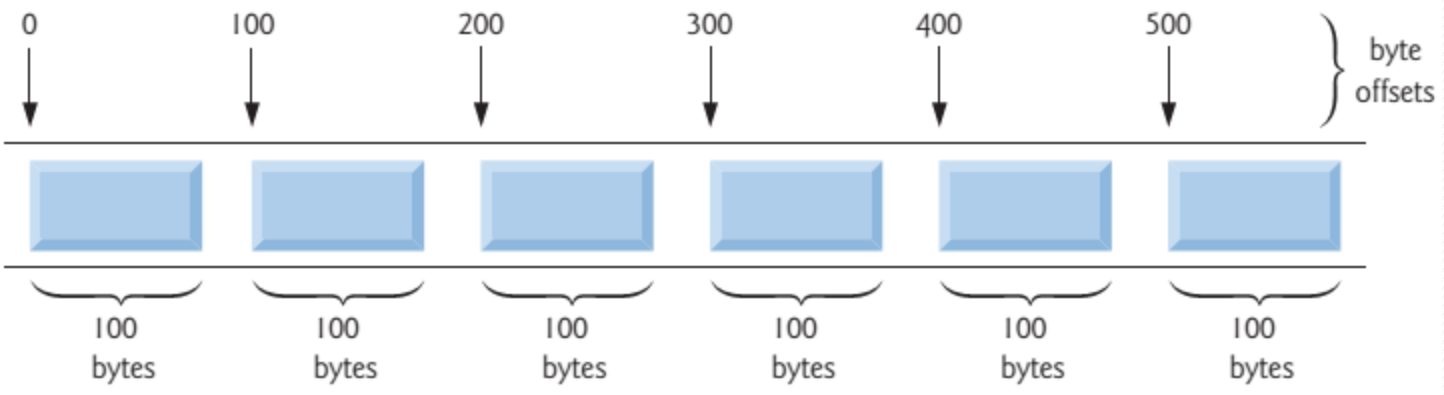
- ofstream outfile("asciifile.bin");

or

- fstream outfile("asciifile.bin", ios::out);

Binary Streams

- view of a Binary file composed of fixed-length records(each record, in this case, is 100bytesl ong)



```
#include<fstream>
#include <iostream>
using namespace std;
struct employees {
      int empno;
      char name[10];
      double basic;
      };

employees emp;
void getdata(){
    cout<<endl<<"enter empno: ";
cin>>emp.empno;
    cout<<endl<<"enter empname: ";
cin>>emp.name;
    cout<<endl<<"enter basic pay: ";
cin>>emp.basic;
}
main(){
    ofstream outfile("asciifile.bin");
    for(int i=1; i <=3; i++) {
       getdata();
       outfile.write((char *)&emp,
sizeof(emp));
    }
    outfile.close();
    system("pause");
}
```

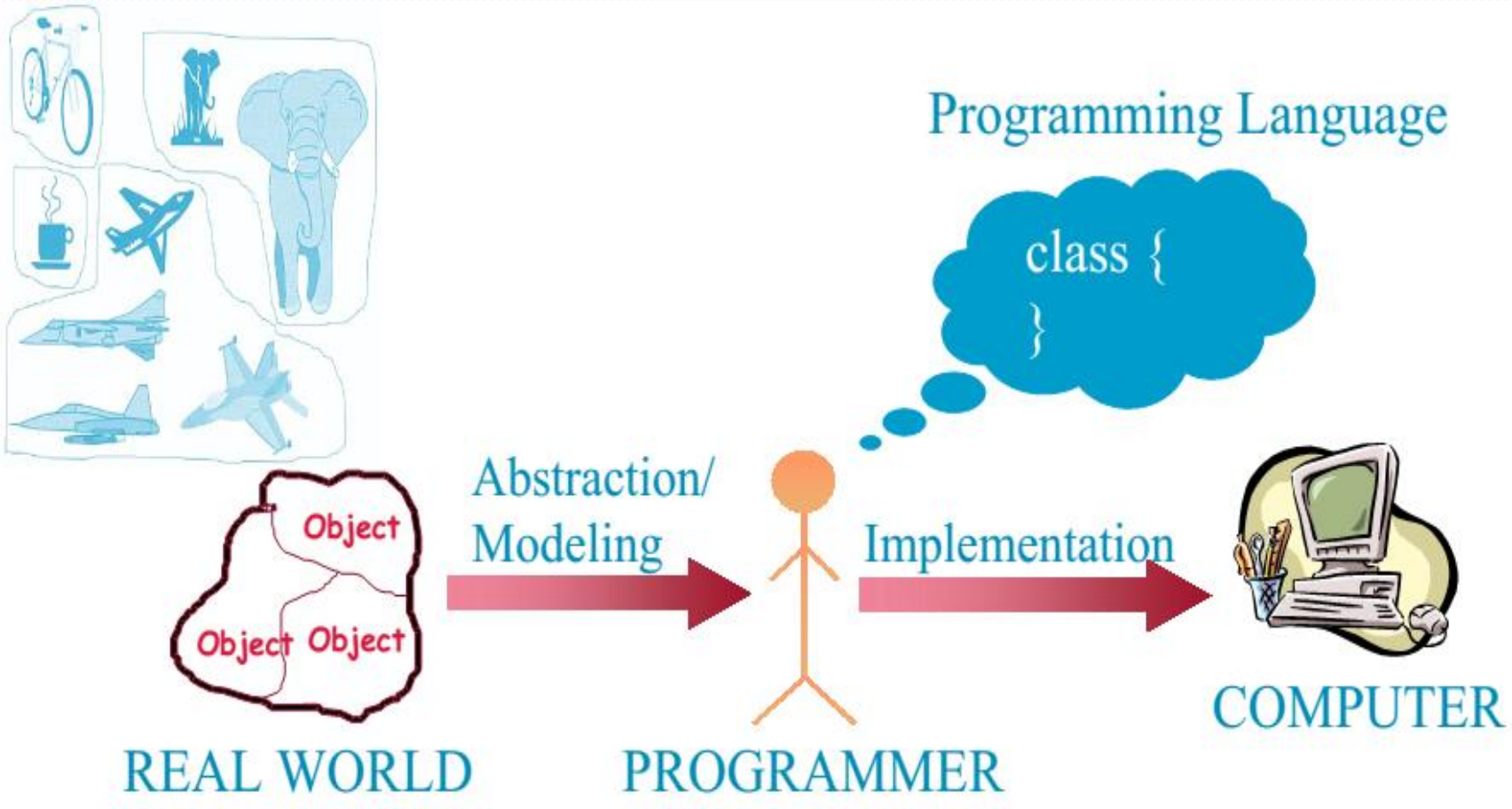
```
. . .
main(){
  ifstream infile("asciifile.txt");
  if(!infile){
       cout << "File did not open" << endl << endl;
       system("pause");
       exit(1);
  }
  infile.read((char*)&emp, sizeof(emp));
  while (infile) {
     cout<<endl<<"employee number: "<<emp.empno;
     cout<<endl<<"emp ıname "<<emp.name;
     cout<<endl<<"basic pay "<<emp.basic;
     cout<<endl;
     infile.read((char*)&emp, sizeof(emp));     }
 infile.close();
 system("pause");
}
```

Object oriented Programming (OOP)

-Unified Modeling Language (UML) notation.

- Implementation (Class)

Object Oriented Design (OOD)



Object Oriented Design (OOD)

- What kinds of things become objects in object-oriented programs?
  - Human entities: Employees, customers, salespeople, worker, manager
  - Graphics program: Point, line, square, circle, …
  - Mathematics: Complex numbers, matrix
  - Computer user environment: Windows, menus, buttons
  - Data-storage constructs: Customized arrays, stacks, linked list.

Object Oriented Design (OOD)

- Object Oriented Design or OOD to better describe the design of the solutions to today's problems.
- This type of programming is called Object Oriented Programming or OOP.
- Several different tools have been created to support this approach to programming. One of these tools is called the Unified Modeling Language or UML.

Object Oriented Design (OOP)

- Object-oriented programming(OOP) is a particular conceptual approach to designing programs and C++ has enhanced C with features that ease the way to applying that approach. The following are the most important OOP features:
  - Abstraction
  - Encapsulation and data hiding
  - Polymor phism
  - Inheritance
  - Reusability of code

Object Oriented Design (OOP)

- When you approach a programming problem in an object oriented language, you no longer ask how the problem will be divided into functions, but how it will be divided into objects.
- Thinking in terms of objects rather than functions has a helpful effect on how easily you can design programs.
- Because the real world consists of objects and there is a close match between objects in the programming sense and objects in the real world.

What is an Object?

- Many real-world objects have both a state(characteristics that can change) and abilities(things they can do).
- Real-world object=State (properties)+ Abilities (behavior)
- Programming objects = Data + Functions
- The match between programming objects and real-world objects is the result of combining data and member functions.
- How can we define an object in a C++ program?

What is an Object?

- Many real-world objects have both a state(characteristics that can change) and abilities(things they can do).
- Real-world object=State (properties)+ Abilities (behavior)
- Programming objects = Data + Functions
- The match between programming objects and real-world objects is the result of combining data and member functions.
- How can we define an object in a C++ program?

Classes and Objects

- Class is a new data type which is used to define objects. A class serves as a plan, or a template. It specifies what data and what functions will be included in objects of that class. Writing a class doesn't create any objects.
- A class is a description of similar objects. A class is a grouping of data and functions. A class is very much like a structure type as used in ANSI-C, it is only a pattern to be used to create a variable which can be manipulated in a program.
- Objects are instances of classes. An object is an instance of a class, which is similar to a variable defined as an instance of a type. An object is what you actually use in a program.

Classes Vs. Structures

- lassand structkeywords have very similar meaning in
- the C++.
- They both are used to build object models.
- The only difference is their default access mode.
- The default access mode for class is private.
- The default access mode for struct is public.

Class

- The class definition contains declarations of any data that may be stored in the objects. This data is called data members or attributes (and sometimes fields). Today the term attributes is more frequently used.
- The definition may also contain the declarations or definitions of functions that describe the way the objects may be manipulated. These functions are called member functions, operations or methods. Today the name method is used more frequently.

Class

- Informally in C++ the class definition includes the following:
  - The keyword class
  - A tag or name of the class
  - A pair of beginning and ending braces {and} the last of which is followed by a Semicolon
  - A collection of zero or more attribute declarations where each section is headed by the access section specifier.

- The access section specifiers are the keywords:
  - Private
  - protected
  - Public

Class Definition Time

```
class Time {
    private:
            int hour;
            int minute;
            int second ;
    public:
    // Get Functions
    int GetHour(){returnhour;} ;
    int GetMinute(){returnminute;} ;
    int GetSecond(){return second;} ;
    // Set Functions
    void SetTime(int h,int m,int s){hour=h;minute=m;second=s;};
    void SetHour(inth){hour= (h>=0 && h<24) ? h : 0;} ;
    void SetMinute(intm){minute= (m>=0 && m<60) ? m : 0;} ;
    void SetSecond(ints){second= (s>=0 && s<60) ? s : 0;} ;
    void PrintTime();
};
```

Class

- Variable (Object) Declaration: Variables that are of a class data type are called instances or objects of the class. Objects may be defined similar to how variables of other data types are defined as the following:

```
Time myTime;
Time yourTime;
```

- Defining Array of Objects

```
Time time[10];
```

Unified Modeling Language (UML)

- A class and its members can be described graphically using a notation known as the Unified Modeling Language (UML) notation.

Unified Modeling Language (UML)

- A class and its members can be described graphically using the Unified Modeling Language (UML) notation.

class Time
-hour: int - minute: int - second: int
+ GetHour() + GetMinute() + GetSecond() + SetTime(int , int, int) + SetHour(int) + SetMinute(int) + SetSecond(int) + PrintTime();

A +(plus) sign in front of a member name indicates that this member is a public member;
A -(minus) sign indicates that this is a private member.

```
class Circle {
    public:
        void setRadius(double);
        double getRadius();
        double calculateCircumference();
        double calculateArea();
    private:
        double radius; // Radius of this circle
        const double PI = 3.14159;
};
void Circle::setRadius(double rad) {
        radius = rad;
}
double Circle::getRadius() {
        return radius;
}
double Circle::calculateCircumference() {
        return (2 * PI * radius);
}
double Circle::calculateArea() {
        return(PI * radius * radius);
}
```

Class

- In the following, you should assume that a Circle object named myCircle has been created in a program that uses the Circle class, and radius is given a value as shown in the following code:

```
Circle myCircle;
myCircle.setRadius(3.0);
```

- What is the output when the following line of C++ code executes?

```
cout << "The circumference is : " << myCircle.calculateCircumference();
```

- Is the following a legal C++ statement? Why or why not?

```
cout << "The area is : " << calculateArea();
```

- Consider the following C++ code. What is the value stored in the myCircle object's attribute named radius?

```
myCircle.setRadius(4.0);
```

```
class TComplex{
        float real,img;
    public:
        TComplex(float, float); // constructor
        void print() const;  // const method
        void reset() {real=img=0;} // non-const method
         TComplex add(const ComplexT&)
};
void TComplex::print() const { // const method
        std::cout << "complex number= " << real << ", " << img; }
ComplexT ComplexT::add(const ComplexT& z) {
        ComplexT result; // local object
        result.re = re + z.re;
        result.im = im + z.im;
        return result;}
main() {
        const TComplex cz(0,1); // constant object
        TComplex ncz(1.2,0.5)  // non-constant object
        cz.print(); // OK
        cz.reset(); // Error !!!
        ncz.reset(); // OK
}
```

```
class TComplex{
        float real,img;
    public:
        TComplex(float, float); // constructor
        void print() const;  // const method
        void reset() {real=img=0;} // non-const method
        TComplex operator+(TComplex&);// header of operator+
functio
};
void TComplex::print() const { // const method
        std::cout << "complex number= " << real << ", " << img; }
TComplex TComplex::operator+(TComplex& z)
        ComplexT result; // local object
        result.re = re + z.re;
        result.im = im + z.im;
        return result;}
main() {
         TComplex z1,z2,z3 ;
         z3=z1+z2;
....
}
```

Inheritance

- OOP provides a way to modify a class without changing its code.
- This is achieved by using inheritance to derive a new class from the old one.
- The old class (called the base class) is not modified, but the new class (the derived class) can use all the features of the old one and additional features of its own.

```
class <derived> : public <base> {
    <member-declarations>
```

Or

```
class <derived> : private <base> {
    <member-declarations>
```

Inheritance

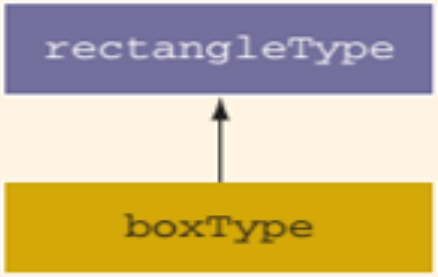
- Suppose we want to create Box class using rectangle class.
- First **Base Class**

RectangleType
-Length: double - width: double
+ setDimension(double, double): void + getLength(): const double + getWidth() : const double + area(): const double + perimeter(): const double + RectangleType() + RectangleType(double, double)

Inheritance

- Suppose we want to create Box class using rectangle class.
- First **Base Class**

BoxType
-height: double
+ setDimension(double, double, double): void + getHeight (): const double + getWidth() : const double + surfaceArea(): const double + volume(): const double + BoxType() + BoxType(double, double, double)



Inheritance

```
Class rectangleType {
    public:
        void setDimension(double l, double w);
        double getLength() const;
        double getWidth() const;
        double area() const;
        double perimeter() const; //circumference
        void print() const;
        rectangleType(doublel = 0,doublew = 0);
        //Constructor with default parameters
    private:
        double length;
        double width;
};

rectangleType::rectangleType (double l,double w) {
     setDimension (l, w);
}
```

Inheritance

- classboxType: public rectangleType {
- public:
- void setDimension(double l, double w, double h);
- double getHeight() const;
- double area() const;
- double volume() const;
- void print() const;
- boxType (doublel = 0, doublew = 0, doubleh = 0);
- //Constructor with default parameters
- private:
- double height;
- };

Inheritance

You can write the definition of the constructor of the class boxType as follows:

```
boxType::boxType(double l, double w, double h)
        : rectangleType(l, w)
{
    if (h >= 0)
        height = h;
    else
        height = 0;
}
```

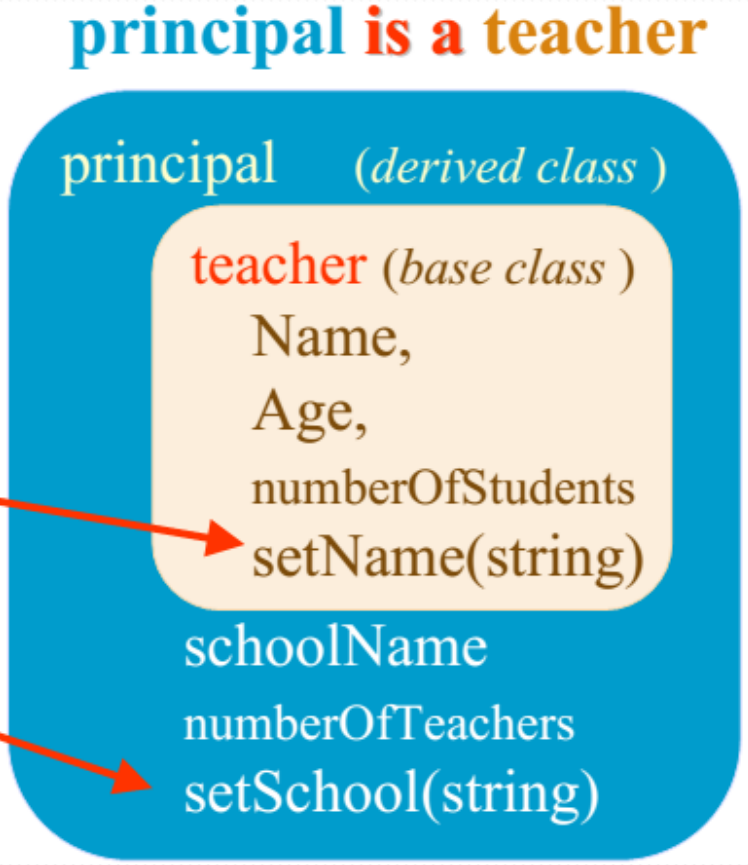
Inheritance

```
class Teacher{ // Base class
    private: // means public for derived class members
            string name;
            int age, numberOfStudents;
    public:
            void setName (const string & new_name){
             name = new_name; }
};

class Principal : public Teacher { // Derived class
            string schoolName; // Additional members
            int numberOfTeachers;
    public:
            void setSchool(const string & s_name){ schoolName = s_name; }
};
```

Inheritance

```
int main()  {
   Teacher t1;
   Principal p1;
   p1.setName(" Principal 1");
   t1.setName(" Teacher 1");
   p1.setSchool(" Elementary School");
   return 0;
}
```



LAB : CREATING A CLASS IN C++

In this lab, you create a programmer-defined class and then use it in a C++ program. The program should create two Rectangle objects and find their area and perimeter. Use the Circle class that you worked with in Exercise 10-1 as a guide.

- Open the class file named Rectangle.cpp using Notepad or the text editor of your choice.
- In the Rectangle class, create two private attributes named length and width. Both Length and width should be data type double.
- Write public set methods to set the values for length and width.
- Write public get methods to retrieve the values for length and width.
- Write a public calculateArea()method and a public calculatePerimeter() method to calculate and return the area of the rectangle and the perimeter of the rectangle.
- Save this class file, Rectangle.cpp, in a directory of your choice and then open the file named MyRectangleClassProgram.cpp.
- In the MyRectangleClassProgram, create two Rectangleobjects named rectangleOne and rectangleTwo using the default constructor as you saw in MyEmployeeClassProgram.cpp.
- Set the length of rectangleOne to 4.0 and the width to 6.0. Set the length of rectangleTwo to 9.0 and the width to 7.0.
- Print the value of rectangleOne's perimeter and area, and then print the value of rectangleTwo's perimeter and area.
- Save MyRectangleClassProgram.cppin the same directory as Rectangle.cpp.
- Compile the source code file MyRectangleClassProgram.cpp.
- Execute the program.
- Record the output below.

Inheritance

HomeWork

- Construct a base class triangle.
- Construct a derived class pyramid using rectangle and triangle base classed.
- Construct a derived class cylinder
- Construct a derived class bag using class box by adding(price, style)

Multiple Inheritance

```
class computer_screen{
    public:
        computer_screen( char *, long, int, int);
        void show_screen (void);
    private:
        char type[32];
        long colors;
        int x_resolution, y_resolution;
};
Class mother_board {
    public:
        mother_board( int, int, int);
        void show_mother_board(void);
    private:
        int processor, speed, Ram;
};
```

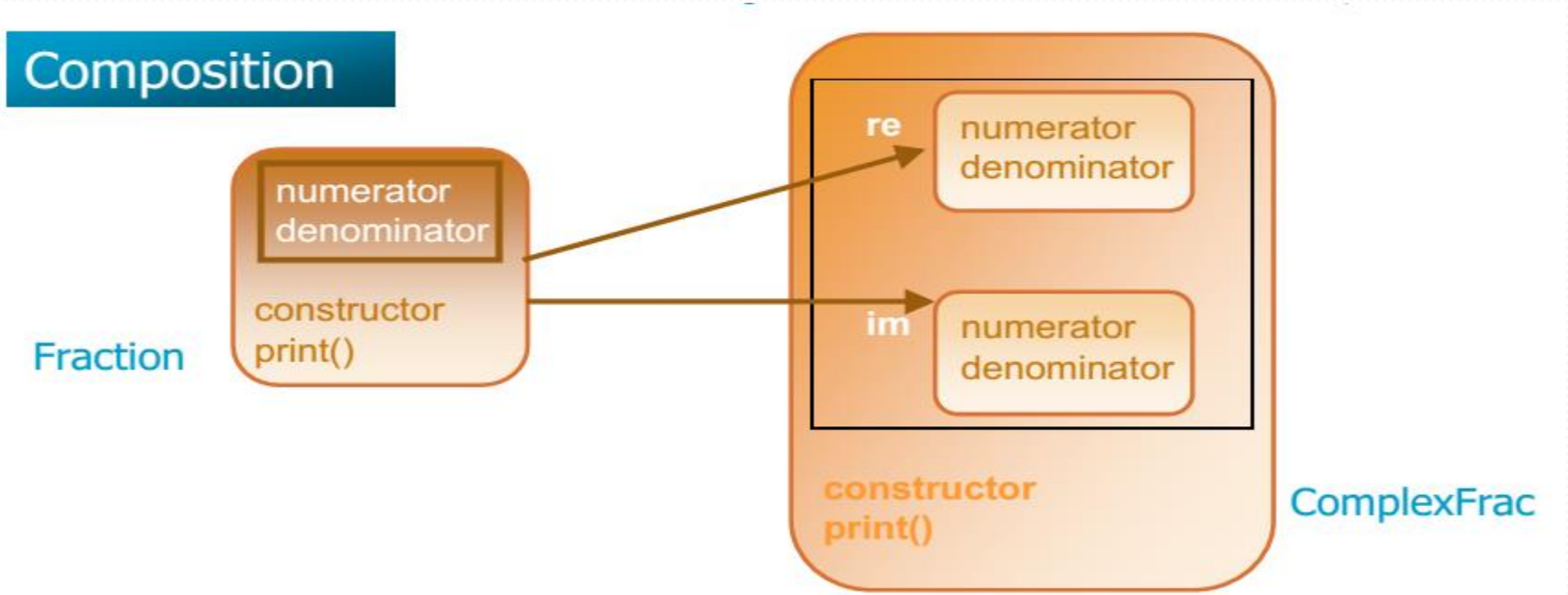
Multiple Inheritance

- Using both base classes in a new derived class.

```
class computer : public computer_screen,  public mother_board
      { public:
           computer( char *, int, float, // own data
            char *, long, int,  int // computer screen data
            int, int, int); // mother board date
            void show_computer (void);
       private:
            char name[64];
            int hard_disk;
            float floppy;
   };
```

Nesting Objects: Classes as Members of Other Classes

- A class may include objects of other classes as its data members.
- Example, a class is designed (ComplexFrac) to define complex numbers. The data members of this class are fractions which are objects of another class (Fraction).



```
class Fraction {  // A class to define fractions
        int numerator, denominator;
    public:
        Fraction(int, int); // CONSTRUCTOR
        void print() const;
};
Fraction::Fraction(int num, int denom) { // CONSTRUCTOR
        numerator = num;
        if (denom==0) denominator = 1;
        else denominator = denom;
        cout << "Constructor of Fraction" << endl;
}
void Fraction::print() const {
        cout << numerator << "/" << denominator << endl;
}
```

```
class ComplexFrac {  // Complex numbers, real and imag. parts are fractions
        Fraction re, im; // objects as data members of another class
    public:
        ComplexFrac(int,int); // Constructor
        void print() const;
};
ComplexFrac::ComplexFrac(int re_in, int im_in): re(re_in, 1) , im(im_in, 1) {
:
}
void ComplexFrac::print() const {
        re.print();
        im.print();
}
int main() {
        ComplexFrac cf(2, 5);
        cf.print();
}
```

Polymorphism

- Polymorphism, which is implemented in C++ by virtual functions.

Class templates

```
class  array{
     puplic:
          array( int);
          long sum(void);
          int average_value(void);
          void show_array(void);
          int add_value(int);
    private:
          int *data, size, index;
};
long array:: sum(void) {
        long sum = 0; int i;
        for (i=0; I < index; i++)    sum += data[i];
        return sum;
}
```

Class templates

```
class  array {
    puplic:
        array( int);
        long sum(void);
        int average_value(void);
        void show_array(void);
        int add_value(int);
   private:
       int *data, size, index;
};
long array:: sum(void) {
     long sum = 0; int i;
      for (i=0; i < index; i++)
           sum += data[i];
      return sum;
}
```

```
Template <class T, class T1>
class  array {
    puplic:
        array( int);
        T1 sum(void);
        T average_value(void);
        void show_array(void);
        int add_value(T);
   private:
        T *data,
        int size, index;
};
Template <class T, class T1>
T1 array<T, T1> :: sum(void) {
      T1 sum = 0; int i;
      for (i=0; i < index; i++)
           sum += data[i];
      return sum;
}
```

vector Class

- Must include:

#include <vector>

```
vector<int> v;
cout << v.capacity() << v.size() ;
v.insert(v.end(),3) ;                          v = (3)
cout << v.capacity() << v.size() ;
v.insert (v.begin(), 2, 5);                    v = (5,5,3)

vector<int> w (4,9);                           w = (9,9,9,9)
w.insert(w.end(), v.begin(), v.end() );        w = (9,9,9,9,5,5,3)
w.swap(v) ;          v = (9,9,9,9,5,5,3)       w=(5,5,3)
```

vector Class

- Must include:

#include <vector>

w.erase(w.begin());	**w = (5,3)**
w.erase(w.begin(),w.end()) ;	
cout << w.empty() ? "Empty" : "not Empty"	**Empty**

vector<int> v;	
v.insert(v.end(),3) ;	**v = (3)**
v.insert(v.begin(),5) ;	**v = (5,3)**
cout << v.front() << endl;	**5**
cout << v.back() ;	**3**
v.pop_back();	
cout << v.back() ;	**5**

```
#include <iostream>
int main() {
   int i;
   char ok=0;
   while(!ok) {                          // cycle until input OK
      cout << "\nEnter an integer: ";
      cin >> i;
      if( cin.good() ) ok=1;                  // if no errors
      else {
        cin.clear();                       // clear the error bits
        cout << "Incorrect input";
        cin.ignore(20, '\n');                // remove newline
      }
   }
   cout << "integer is " << i; // error-free integer
}
```

Error-Status Bits

The stream error-status bits (error byte) are an ios member that report errors that occurred in an input or output operation.

goodbit	No errors (no bits set, value = 0).
eofbit	Reached end of file.
failbit	Operation failed (user error, premature EOF).
badbit	Invalid operation (no associated streambuf).
hardfail	Unrecoverable error.

Various ios functions can be used to read (and even set) these error bits.

int = eof();	Returns true if EOF bit set.
int = fail();	Returns true if fail bit or bad bit or hard-fail bit set.
int = bad();	Returns true if bad bit or hard-fail bit set.
int = good();	Returns true if everything OK; no bits set.
clear(int=0);	With no argument, clears all error bits; otherwise sets specified bits, as in clear(ios::failbit).