1. STUDENT MANUAL
    1. Syllabus
    2. Exercises
    3. Assignments
    4. Labworks
2. LECTURE NOTES
    1. Introduction to Computers and Programming
    2. Basic Elements in C++
    3. Completing the Basics
    4. Selection Statements
    5. Repetition Statements, Arrays and Structured Programming
    6. Functions and Pointers
    7. Introduction to Classes
    8. Object Manipulation - Inheritance

Syllabus

LETTER TO STUDENTS

This course and this student manual reflect a collective effort by your constructor, the Vietnam Education Foundation, The Massachusetts Institute of Technology (MIT) Open Courseware Project and faculty colleagues within Vietnam and the United States who served as reviewers of drafts of this student manual. This course is an important component of our academic program. Although it has been offered for more than three years, this latest version represents an attempt to expand the ranges of sources of information and instruction so that the course continues to be up-to-date and the methods well suited to what is to be learned.

You will be asked from time to time to offer feedback on how the student manual is working and how the course is progressing. Your comments will inform the development team about what is working and what requires attention. Our goal is to help you learn what is important about this particular field and to eventually succeed as a professional applying what you learn in this course.

Thank you for your cooperation. I hope you enjoy the course.

COURSE INFORMATION

Course name: Programming Fundamentals In C++ (501125)

Semester: Spring Semester 2008

Institute: Faculty of Computer Science And Engineering, Hochiminh City University of Technology, Vietnam.

Credit Hours: 3

Instructor: Dr. Duong Tuan Anh, Associate Professor

Office Location: Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology, 268 Ly Thuong Kiet, Dist. 10, Ho Chi Minh City.

Office hours: 14:00 – 17:00 Wednesday or by appointment

Tel: 8647256/Ext. 5841;

Fax: (848) 8645137.

Email:dtanh@cse.hcmut.edu.vn.

Website: http://www.dit.hcmut.edu.vn/~dtanh/

Teaching Assistants:

Mr. Nguyen Quoc Viet Hung (nqvhung@cse.hcmut.edu.vn)

Mr. Ly Hoang Hai (lhhai@cse.hcmut.edu.vn)

Mr. Nguyen Xuan Minh (nxminh@cse.hcmut.edu.vn)

Mr. Nguyen Van Doan (nvdoan@cse.hcmut.edu.vn)

COURSE DESCRIPTION

This course is a comprehensive introductory course that is intended for students who have no background in computer programming. This course provides basic knowledge and skills on programming with two important programming paradigms: structured programming and object-oriented programming. The course covers structured programming paradigm in depth and introduces just some basics on object-oriented programming. The programming language used in this programming course is C++.

COURSE OBJECTIVES

Upon successful completion, students will be able to:

1. Design algorithmic solutions to problems.
2. Translate a specified algorithm into correct self-documented C++ code using generally accepted programming style. In accomplishing this translation, the student shall be able to apply the structured programming mechanisms of C++ including sequence, selection, iteration, recursion, pointers and arrays.
3. Acquire an understanding of basic object-oriented concepts and the issues involved in effective class design.
4. Write C++ programs that use:

    - arrays
    - structures
    - pointers
    - object-oriented concepts such as information hiding, constructors, destructors, inheritance.

INTRUCTOR OBJECTIVES

In order to meet the course objectives, the students are expected to fulfill the following requirements:

- Class attendance must be more than 75%.
- For each chapter, the students should complete at home a sufficient amount of exercises given for the chapter. All the exercises are given in Programming Fundamentals in C++ - Lecture Notes and Exercises, available at the instructor’s website.
- Lab work participation is mandatory. At each lab session, the students must complete a sufficient amount of assignments assigned for that lab session. Lab assistants will evaluate the performance of each student at the end of each lab session.
- There are two programming projects each student is required to produce solutions. These projects aim to train the student’s creativity and problem-solving skills. Project 1 aims to apply all the knowledge in the six first chapters and Project 2 aims to apply those of 2 last chapters. Due date of each project must be respected. Unless it is extremely exceptional, late submission is not accepted.

COURSE DURATION

This course is one semester long, meeting from February 18th through May 29th. Our semester is 16 weeks long. The course meets for three period lecture session and two period lab session. It consists of 42 periods for lectures and 28 periods for lab works (1 period = 45 minutes).

COURSE OUTLINE

Chapter 1: Introduction to Computer and Programming

1. Hardware and software
2. Programming languages
3. Problem solving and software development
4. Algorithms

Chapter 2: Basic Elements in C++

1. Program structures
2. Data types, and operators
3. Variables and variable declarations
4. Integer quantifiers
5. Focus on problem-solving

Chapter 3: Completing the Basics

1. Assignment operators
2. Formatting numbers for program output
3. Using mathematical library functions
4. Program input using the cin object
5. Symbolic constants

Chapter 4: Selection Structures

1. Selection criteria
2. The if-else statement
3. Nested if statement
4. The switch statement

5. The enum specifier

Chapter 5: Repetition Structures – Arrays and Structured Programming

1. Basic loop structures
2. while loops
3. Interactive while loops
4. for loops
5. Nested loops
6. do-while loops
7. Structured programming with C++
8. Arrays
9. Structures

Chapter 6: Functions and Pointers

1. Function and parameter declarations
2. Returning values
3. Variable scope
4. Variable storage class
5. Passing by reference
6. Recursion
7. Passing arrays to functions
8. Pointers
9. The typedef declaration

Chapter 7: Introduction to Classes

1. Classes
2. Information hiding
3. Member functions
4. Dynamically Memory Allocation with operators new and delete
5. Pointers as class members

Chapter 8: Object Manipulation - Inheritance

1. Advanced constructors

2. Destructors
3. Constant objects
4. Inheritance

LAB WORK

This course maintains a laboratory for its students. During semester, a group of lab assistants hang out in lab to answer students' questions and help them in debugging. There are 10 lab sessions in the course and the first lab session starts at the third week of the semester. For each lab session, which is 3 period long, the students are requested to finish at least some required assignments. Notice that lab assistants will grade the performance of each student at the end of each lab session. Lab assistants inspect the working of student programs and ask questions on their program codes in order to grade the student lab performance in each lab session.

Lab work materials

Lab assignments for all lab sessions are given in **Programming Fundamentals in C++ -Laboratory Manual,** which is available in the instructor's web site.

System requirement

This course is designed to be delivered in an environment supporting a C++ compiler. There is supplementary information included about the Visual C++ 6.0 development environment.

INSTRUCTIONAL METHODS

To facilitate the goal of making C++ accessible as a first-level programming course, the following instructional methods are used in this course.

End-of-Chapter Exercises. Every chapter in the Lecture Notes contains several diverse skill-builder and programming exercises. Students are encouraged to do all the exercises after each chapter. Solutions to some of them are provided by instructor or teaching assistants in the class.

Focus on Problem Solving. In the Chapter 2, 3, 4 and 7, each chapter contains a Focus on Problem Solving section with one complete problem per chapter. Each application is used to demonstrate effective problem solving within the context of a complete program solution. Two programming projects – one for structured programming paradigm and one for object-oriented programming paradigm will ask each student to write larger programs in order to help students to enhance their problem-solving skills.

Pseudocode Descriptions. Pseudocode is stressed throughout the course. Flowchart symbols are described, but are only used when visually presenting flow-of-control constructs.

Gentle Introduction to Object-Oriented Programming. In the course, there are two chapters (7and 8) that provide a "mini-course" introduction to basic object-oriented programming concepts and design techniques. Object oriented design techniques are illustrated in details in a Focus on Problem Solving section of Chapter 7 with one complete application. For more advanced features of object-oriented programming in C++, another course named "Object-Oriented Programming" which is offered one semester later will cover.

EVALUATION AND GRADING SYSTEMS

The final grade of each student shall be calculated by means of a weighted average as follows:

Lab works and **programming projects**: **30%**

Midterm examination: **20%**

Final examination: **50%**

Lab work and project evaluation:

Lab work evaluation is based on lab work performance of all 10 lab sessions. Project evaluation is based on the working of 2 programming projects. Sample topics for programming projects are given in the last pages

of Programming Fundamentals in C++ -Laboratory Manual, which is available in the instructor's web site.

Lab assignments and projects are evaluated using the following grading criteria:

-correctness: 40%

- appropriate use of arrays/structures/pointers/functions and/or classes: 25%

- program structure (including efficiency): 10%

- program style: 10%

- documentation: 10%

- format of output: 5%

Scores are given in the range from 0 to 10 (rounded to 0.5) with the following interpretation:

Score range	Percentage	Group
9 -10	90%-100%	Excellent
7.5 - 9	75%-89%	Good
6 – 7.5	60%-74%	Average
5 - 6	50%-60%	Fair
< 5	<50%	Fail

Grading system

Exams

Exams are closed-book exams. Exams include one or more of: short answer, multiple choice, trace the given code, debug the given code or given a

problem description, produce a solution in the form of a short program or a function(s).

The problems given in the exams are like those on exercises and lab work. The students who spend more time and effort in doing exercises and lab assignments will certainly have better performance in the exams. Exam scores are given in the range from 0 to 10.

COURSE MATERIALS

Lecture Notes, Exercises, Laboratory Manual

Available at the instructor's website.

Text book:

[1] G. J. Bronson, Program Development and Design Using C++, 3nd Edition, Brooks/COLE Thomson Learning, 2006.

(All lecture notes and exercises used in this course are mainly from this textbook. Students can find more C++ language features, problem-solving guidelines as well as good exercises and assignments from this book for further self-study.)

Reference books:

[2] H. M Deitel and P. J. Deitel, C++ How to Program – 3rd Edition, Prentice-Hall, 2001.

(This book contains a lot of good C++ programming examples. All examples in this book are very well explained.)

[3] J. Farrel, Object-Oriented Programming Using C++, 2nd Edition, Course Technology/Thomson Learning, 2001.

(If students want to go in depth in object-oriented programming with C++, they will find this book very helpful.)

[4] D. Gosselin, Microsoft Visual C++ 6.0, Course Technology/Thomson Learning, 2001.

(This book is a supplementary text for the students who want to know more about Microsoft Visual C++ 6.0 programming environment.)

CALENDAR

There are 16 sessions which compose of 14 lectures, 1 mid-term exam and 1 final exam.

Week #	*Topics*	*Reading Assignments from the Textbook*	*Notes*
1 18-Feb	Ch 1: Introduction to computer and programming	Ch 1	
2 25-Feb	Ch 2: Basic elements in C++	Ch 2	
3 3-Mar	Ch 3: Completing the basics	Ch 3	Lab session 1
4 10-Mar	Completing the basics (cont.) Ch 4: Selection statements	Ch 3	Lab session 2
5 17-Mar	Selection statements (cont.)	Ch 4	Lab session 3
6 24-Mar	Ch 5: Repetition statements	Ch 5	Lab session 4
7 31-Mar	Repetition statements (cont.)	Ch 5	Lab session 5
8 7-Apr			**Midterm exam**
9 14-Apr	Ch 6: Functions and Pointers	Ch 6	
10 21-Apr	Functions and Pointers (cont.)	Ch 6	Lab session 6
11 28-Apr	Functions and Pointers (cont.)	Ch 6	Lab session 7
12 5-May	Ch 7: Introduction to classes	Ch 8	Due date: Project 1 (9-May)
13 12-May	Introduction to classes (cont.)	Ch 8	Lab session 8
14 19-May	Ch 8: Object Manipulation-Inheritance	Ch 10	Lab session 9
15 26-May	Object Manipulation – Inheritance (cont.)	Ch 10	Lab session 10 Due date: Project 2 (30-May)
16 2-June			**Final Exam**

Course calenda

END-OF-COURSE EVALUATION

To continuously improve course content and design, students are requested to complete end-of-course evaluation form. The student comments will be carefully reviewed and used to improve the quality of the course. Please take a moment to complete the following end-of-course feedback form.

OVERALL

1. Rate your overall satisfaction with the course.

_____Very Satisfied

_____Satisfied

_____Neutral

_____Dissatisfied

_____Very Dissatisfied

1. Rate the course in terms of meeting your educational needs.

_____Very Satisfied

_____Satisfied

_____Neutral

_____Dissatisfied

_____Very Dissatisfied

1. Based on the knowledge/skill you require to do your job, how would you want to change this course?

COURSE CONTENT

1. Rate the course in terms of meeting the stated objectives.

_____Very Satisfied

_____Satisfied

_____Neutral

_____Dissatisfied

_____Very Dissatisfied

1. Rate the course content (e.g., relevance, structure, level of details).

_____Very Satisfied

_____Satisfied

_____Neutral

_____Dissatisfied

_____Very Dissatisfied

1. Rate the participative activities (e.g., labs, exercises, projects).

_____Very Satisfied

_____Satisfied

_____Neutral

_____Dissatisfied

_____Very Dissatisfied

1. Rate the course materials (e.g., lecture notes, slides, lab manual, textbook).

_____Very Satisfied

_____Satisfied

_____Neutral

_____Dissatisfied

_____Very Dissatisfied

1. What topics in the course you like best? Why?
2. What topics in the course you like least? Why?
3. Course content comments:

INSTRUCTION

1. Rate the instructor’s subject knowledge.

_____Very Satisfied

_____Satisfied

_____Neutral

_____Dissatisfied

_____Very Dissatisfied

1. Rate the instructor’s effectiveness (e.g., question handling, presentation, and ability to explain ideas).

_____Very Satisfied

_____Satisfied

_____Neutral

_____Dissatisfied

_____Very Dissatisfied

1. Instruction comments:

-

-

FACILITIES AND EQUIPMENT

1. Rate the classroom facilities provided (e.g. ventilation, lightning, temperature, space, projector).

_____Very Satisfied

_____Satisfied

_____Neutral

_____Dissatisfied

_____Very Dissatisfied

1. Rate the computer lab equipment provided (e.g., computers, software, Internet access).

_____Very Satisfied

_____Satisfied

_____Neutral

_____Dissatisfied

_____Very Dissatisfied

1. Facilities and equipment comments:

SUPPORT

1. Rate the lab/teaching assistant support. (enthusiasm, effectiveness, knowledge, skill, communication ability).

_____Very Satisfied

_____Satisfied

_____Neutral

_____Dissatisfied

_____Very Dissatisfied

1. Support comments.

Thank you for your comments.

Exercises

Exercise 1

1. Design an algorithm in flowchart to find the smallest number in a group of three real numbers.

2. Design an algorithm in flowchart to solve the quadratic equation: ax^2 + bx + c =0 with the inputs a, b, and c.

3. A set of linear equations:

aX + bY = c

dX + eY = f

can be solved using Cramer's rule as:

X = (ce – bf)/(ae – bd)

Y = (af – cd)/(ae – bd)

Design an algorithm in flowchart to read in a, b, c, d, e and f and then solve for X and Y.

4. Design an algorithm in flowchart to read in a group of N numbers and compute the average of them, where N is also an input.

5. Design an algorithm in flowchart to read in a group of N numbers and compute the sum of them, where N is also an input.

Exercise 2

1. Read the following C++ program for understanding. Add some suitable declarations to complete the program and run it.

```
#<include<iostream.h>
```

```
void main(){

...

units = 5;

price = 12.5;

idnumber = 12583;

cost = price*units;

cout << idnumber << " " << units << " " << price
<< " " << cost << endl;

tax = cost*0.06;

total = cost + tax;

cout << tax << " " << total << endl;

}
```

2. Write and run a program to print your first name on the first line, your middle name on the second line and your last name on the third line using only cout statement.

3. Write and run a program that performs the following steps:

- Assigning value to the radius r.
- Calculating the circumference using the formula: C = 2*pi*r.
- Displaying the circumference.

4. Given an initial deposit of money, denoted as P, in a bank that pays interest annually, the amount of money at a time N years later is given by the formula:

Amount = P*(1 + R)^N

where R is the interest rate as a decimal number (e.g., 6.5% is 0.065). Using this formula, design, write and run a program that determines the amount of money that will be available in 4 years if $10.000 is deposited in a bank that pays 7% interest annually.

5. Write and run a program that performs the following steps:

- Assigning value to a Fahrenheit temperature f.
- Calculating the equivalent Celsius temperature C using the formula:

C = (5/9)(f –32).

- Displaying the Celsius temperature C.

Exercise 3

1. Write and run a program that performs the following steps:

- Reading the radius r from the keyboard.
- Calculating the circumference using the formula: C = 2*pi*r.
- Displaying the circumference C.

2. Write and run a program that performs the following steps:

- Reading a Fahrenheit temperature f from the keyboard.
- Calculating the equivalent Celsius temperature C using the formula:

C = (5/9)(f –32).

- Displaying the Celsius temperature C.

3. Write and run a program that reads two integers through the keyboard and performs simple arithmetic operations (i.e., addition, subtraction, multiplication and division) and displays the results.

4. Write and run a program that reads a floating-point number through the keyboard and performs its square root and displays the result with 4 digits of precision to the right of the decimal point.

5. Given an initial deposit of money, denoted as P, in a bank that pays interest annually, the amount of money at a time N years later is given by the formula:

Amount = P*(1 + R)^N

where R is the interest rate as a decimal number (e.g., 6.5% is 0.065). Write and run a program that performs the following steps:

- Reading the values of P and R from the keyboard.
- Calculating the amount of money that will be available in 5 years.
- Displaying the result.

6. Write and run a program that reads the name, age, sex, height and weight of a student and displays with proper heading for each variable.

7. Write a program to accept a single character from the keyboard. Using **cout** statement to display the character or keystroke and its decimal, hexadecimal and octal values. Display in the following format:

CHARACTER	DECIMAL	HEXADECIMAL	OCTAL
----	----	----	---

Exercise 4

1. Write and run a program that reads a character and writes out a name corresponding to the character:

```
if the character is ‘F’ or ‘f’ then the name is ‘Frank’
```

```
if the character is ‘C’ or ‘c’ then the name is ‘Christine’
```

`if the character is ‘A’ or ‘a’ then the name is ‘Amanda’`

`if the character is ‘B’ or ‘b’ then the name is ‘Bernard’`

`otherwise, the name is just the character.`

2. Write and run a program that reads an angle (expressed in degrees) and states in which quadrant the given angle lies. An angle A is said to be in the

- first quadrant if it is in the range 0 <= A < 90
- second quadrant if it is in the range 90 <= A < 180
- third quadrant if it is in the range 180 <= A < 270
- and fourth quadrant if it is in the range 270 <= A < 360.

3. Write and run a program that reads a month from the user and displays the number of days in that month.

4. Write and run a program that reads a numeric grade from a student and displays a corresponding character grade for that numeric grade. The program prints ‘A’ for exam grades greater than or equal to 90, ‘B’ for grades in the range 80 to 89, ‘C’ for grades in the range 70 to 79, ‘D’ for grades in the range 60 to 69 and ‘F’ for all other grades.

5. Write and run a program that reads a marriage code (one character) and writes out a message corresponding to the character:

`if the character is ‘M’ or ‘m’ then the message is “Individual is married”`

`if the character is ‘D’ or ‘d’ then the message is “Individual is divorced”`

`if the character is ‘W’ or ‘w’ then the message is “Individual is widowed”`

`otherwise, the message is “An invalid code was entered”.`

(Hint: use switch statement).

6. Chapter 4 contains the example program which can solve quadratic equations. Modify that program so that when the discriminant del is negative, the imaginary roots are calculated and displayed. For this case, the two roots of the equation are:

x1 = (-b)/(2a) + sqrt(-del)/(2a)i

x2 = (-b)/(2a) + sqrt(-del)/(2a)i

where i is the imaginary number symbol for the square root of –1. (Hint: Calculate the real and imaginary parts of each root separately).

7. Write and run a program that gives the user only three choices: convert from Fahrenheit to Celsius, convert from Celsius to Fahrenheit, or quit. If the third choice is selected, the program stops. If one of the first two choices is selected, the program should prompt the user for either a Fahrenheit or Celsius temperature, as appropriate, and then compute and display a corresponding temperature. Use the conversion formulas:

F = (9/5)C + 32

C = (5/9)(F-32)

8. Write a C++ program to calculate the square root and the reciprocal of a user-entered number. Before calculating the square root, you should validate that the number is not negative, and before calculating the reciprocal, check that the number is not zero.

Exercise 5

1. Write and run a program that reads a positive integer value for K and then computes K! = 1*2*3*…*(K-1)*K and displays the result out.

2. Write and run a program that computes x raised to the power n by repetitive multiplication. Then modify your program to calculate x raised to the power (-n).

3. Write and run a program to tabulate sin(x), cos(x) and tan(x) for x = 5, 10, 15,…,85 degrees. Notice that you have to convert x from degrees to radians before using standard functions sin(x), cos(x), tan(x).

4. Write and run a program to read a list of real numbers and then find the number of positive values and the number of negative ones among them. The number of entries is also entered by the user.

5. Write and run a program to compute the sum of square of the integers from 1 to N, where N is an input parameter.

6. Write and run a program to compute the value of pi, using the series for approximating:

pi/4 = 1-1/3+1/5-1/7+ ... + (-1)^n/(2*n+1)

Hint: Use a **while** loop that terminates when the difference between two successive approximations is less than 1.0E-6.

7. The value of Euler's number, e, can be approximated using the formula:

e = 1 + 1/1! +1/2! + 1/3! +1/4! + … +1/n!

Using this formula, write a program that approximates the value of e using a **while** loop that terminates when the difference between two successive approximations is less than 1.0E-6.

8. The Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13,…, where the first two terms are 0 and 1, and each term thereafter is the sum of the two preceding terms, that is, Fib(n) = Fib(n-1) + Fib(n-2). Using this information, write a program that calculates the nth number in a Fibonacci sequence, where n is entered into the program by the user.

9. Write and run a program that inputs an array of N real numbers, and then computes the average value of the array elements. N should be an input

parameter.

10. Write and run a program that inputs an array of N real numbers, and then finds the largest element in the array. N should be an input parameter.

11. Write and run a program that inputs an integer matrix of order n and transposes it and then prints it out. Transposing a square matrix means:

```
a(i,j) <==> a(j,i) for all i, j.
```

12. The Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13,…, where the first two terms are 0 and 1, and each term thereafter is the sum of the two preceding terms. Write a program that computes and stores the Fibonacci sequence in an integer array F, such that F[i] will store the ith number in a Fibonacci sequence. The size of the array is an input parameter which is entered by the user.

13. Write a program that stores the following hourly rates in an array name hourly_rates: 9.5, 6.4, 12.5, 5.5, 10.5. Your program should also create two arrays named working_hours and wages, each capaple of storing five double-precision numbers. Using a for loop and a cin statement, have your program accept five user-input numbers into working_hours array when the program is run. Your program should store the product of the corresponding values in the hourly_rates and working_hours arrays in the wages array (for example, wages[1] = hourly_rate[1]*working_hours[1]) and display the output as a table consisting of three columns.

14. Write and run a program that reads three strings and prints them out in an alphabetical order. (Hint: Use the strcmp() function).

15. Modify the program in Section **Array of Structures** to compute and display the average rate of the five first employees.

16. The following program reads a set of name, roll number, sex, height and weight of the students from the keyboard using a structure within an array.

```
#include<iostream.h>
```

```
#include<string.h>
const int MAX = 100
struct student{
char name[20];
long int rollno;
char sex;
float height;
float weight;
};
void main(){
student cls[MAX];
int i,n;
cout << " How many names ? \n";
cin >> n;
for( i = 0; i <= n-1; ++i){
cout << "record = "<< i+1 << endl;
cout << "name : "; cin>> cls[i].name;
cout << "rollno : "; cin>> cls[i].rollno;
cout << "sex : "; cin>> cls[i].sex;
cout << "height : "; cin>> cls[i].height;
```

```
cout << “weight : “; cin>> cls[i].weight;

cout >> endl;

}

……

}
```

Include into the above program the code that performs two tasks:

a. displaying data of n students in the following format:

Name	Rollno	Sex	Height	Weight
------	--------	----	-------	--------
------	--------	----	-------	--------

b. computing and displaying the average of heights and the average of weights of the students.

Exercise 6

1. a. Write a function **inorder** that determines whether the three characters are in alphabetic order or not. It returns **true** if the three arguments are in order or **false** otherwise.

b. Write a complete program that reads three characters and calls the function **inorder** to report whether they are in alphabetic order, loops until reading “***”.

2. a. Write a function **IntSquare** that computes the greatest integer so that its square is less than or equal to a given number.

b. Write a complete program that reads an integer n and invokes the function **IntSquare** to compute the greatest integer so that its square is less than or equal to n.

3. a. Write a function that computes the fourth root of its integer argument k. The value returned should be a double. (Hint: Use the library function sqrt()).

b. Write a complete program that reads an integer n and invokes the function to compute the fourth root of n.

4. a. Write a function is_prime(n) that returns true if n is a prime or false, otherwise.

b. Write a complete program that reads an integer n and invokes the function to check whether n is prime.

5. We can recursively define the number of combinations of m things out of n, denote C(n, m), for n >= 1 and 0 <= m <= n, by

C(n,m) = 1 if m = 0 or m=n

C(n, m) = C(n-1, m) + C(n-1, m-1) if 0 < m < n

1. Write a recursive function to compute C(n, m).
2. Write a complete program that reads two integers N and M and invokes the function to compute C(N, M) and prints the result out.

6. Given a function as follows:

```
int cube(int a)
{
a = a*a*a;
return a;
}
```

1. Write a complete program that reads an integer n and invokes the function to compute its cube.

2. Rewrite the function so that the parameter is passed by reference. It is named by cube2. Write a complete program that reads an integer n and invokes the function cube2 to compute its cube, prints the result out and then displays the value of n. What is the value of n after the function call?

7. a. Write a function that can find the largest element in the array that is passed to the function as a parameter.

b. Write a program that inputs an array and invokes the above function to find the largest element in the array and print it out.

8. Given the following function that can find a specified value in an array. If the search is successful, this function returns the position of the specified value in the array, otherwise, it returns –1.

```
int linearSearch(int array[], int key, int sizeofArray)

{

for(int n = 0; n< sizeofArray; n++)

if (array[n] = = key)

return n;

return –1;

}
```

Write a program that performs the following steps:

- Input the integer array of N elements. (N is also an input data).
- Input a value you want to search on the array.
- Invoke the function linearSearch to find the element in the array which is equal to the specified value(assume that position is k).
- Remove that element at location k from the array.

9. A palindrome is a string that reads the same both forward and backward. Some examples are

"ABCBA" "RADAR" "otto" "i am ma i" "C"

Given the following function that returns true if the parameter string is a palindrome or false, otherwise.

```
bool palindrome(char strg[])

{

int len, k, j;

len = strlen(strg);

k = len/2;

j = 0;

bool palin = true;

while ( j < k && palin)

if (strg[j] != strg[len -1-j])

palin = false;

else

++ j;

return (palin)

}
```

Write a C++ program that reads several strings, one at a time and checks if each string is a palindrome or not. Use a while loop in your program. The loop terminates when the user enters a string starting with a '*'.

10. Given the following program:

```
#include<iostream.h>

int main()

{

const int NUMS = 5;

int nums[NUMS] = {16, 54, 7, 43, -5}

int i, total = 0;

int *nPt;

nPt = nums;

for( i = 0; i<NUMS; i++)

total = total + *nPt++;

cout << total<< endl;

return 0;

}
```

a. Explain the meaning of the program.

b. Run to determine the result of the program.

11. Given the following function that can find the largest element in an integer array. Notice that the function uses pointer arithmetic:

```
int findMax(int * vals, int numEls)

{

int j, max = *vals;

for (j = 1; j < numEls; j++)

if (max < *(vals + j))

max = *(vals + j);

return max;

}
```

Write a C++ program that inputs an integer array and invokes the above function to find the largest element in that array and displays the result out.

12. a. Write a function named days() that determines the number of days from the date 1/1/1900 for any date passed as a structure. Use the Date structure:

```
struct Date

{

int month;

int day;

int year;

}
```

In writing the days() function, use the convention that all years have 360 days and each month consists of 30 days. The function should return the number of days for any Date structure passed to it.

b. Rewrite the days() function to receive a pointer to a Date structure rather than a copy of the complete structure.

c. Include the function written in b) in a complete C++ program.

Exercise 7

1. Given the class **student** which is defined as follows:

```
class student{
private:
long int rollno;
int age;
char sex;
float height;
float weight;
public:
void getinfo();
void disinfo();
};
void student::getinfo()
{
cout << " Roll no :";
cin >> rollno;
```

```
cout << " Age :";
cin >> age;
cout << " Sex:";
cin >> sex;
cout << " Height :";
cin >> height;
cout << " Weight :";
cin >> weight;
}
void student::disinfo()
{
cout<<endl;
cout<< " Roll no = "<< rollno << endl;
cout<< " Age =" << age << endl;
cout<< " Sex =" << sex << endl;
cout<< " Height =" << height << endl;
cout<< " Weight =" << weight << endl;
}
```

Write the main program that creates an array of **student** objects, accepts the data of n students and displays the data of all elements in this array. The value of n is also entered by the user.

Remember to organize the program into one interface file and one implementation file and run them again.

2. Construct a class named **Account** consisting of four private data members: name of the customer, account number, balance and rate. The public member functions include a constructor and five other member functions that are listed as follows.

- to make a deposit
- to withdraw an amount from the balance
- to get the rate of interest
- to know the balance amount

Include the class **Account** within a complete program. The program should declare two objects of type **Account** and accept and display data for the objects to verify the operation of the member functions.

3. Given the class **equation** which is defined as follows:

```
class equation{

private:

float a;

float b;

float c;

public:

void getinfo(float a, float b, float c);

void display( );

void equal(float a, float b);

void imag( );
```

```
void real(float a, float b, float det);

}; // end of class declaration section

// beginning of implementation section

void equation::getinfo(float aa, float bb, float
cc)

{

a = aa; b = bb; c = cc;

}

void equation::display( )

{

cout << endl;

cout << " a = "<< a << "\t";

cout << " b = " << b << "\t";

cout << " c = " << c << endl;

}

void equation::equal(float a, float b)

{

float x;

x = -b/(2*a);

cout << " roots are equal = "<< x << endl;
```

```
}
void equation::imag(){
cout << " roots are imaginary \n";
}
void equation::real(float a, float b, float det)
{
float x1, x2, temp;
tem = sqrt(det);
x1 = (-b+temp)/(2*a); x2 = (-b-temp)/(2*a);
cout << " roots are real \n";
cout << " x1 = " << x1 << endl;
cout << " x2 = " << x2 << endl;
}
```

Write a main program that accepts the coefficients of a given quadratic equation and makes use of the member functions of the class equation to solve the equation.

Remember to organize the program into one interface file and one implementation file and run them again.

4. Construct a class named **IntArray** consisting of two private data members: a pointer to the beginning of the array, and an integer representing the size of the array. The public functions include a constructor and member functions that show all elements in the IntArray, and search a specified integer in the array.

Include the class **IntArray** within a complete program. The program should declare one object of type **IntArray** and accept and display data for the object to verify operation of the member functions.

5. Construct a class named **Student** consisting of an integer student idenfification number, an array of five floating point grades, and an integer representing the total number of grades entered. The constructor for this class should initialize all Student data members to zero. Included in the class should be member functions to

- enter a student ID number,

- enter a single test grade and update the total number of grades entered, and

- compute an average grade and display the student ID following by the average grade.

Include the class **Student** within a complete program. The program should declare two objects of type **Student** and accept and display data for the two objects to verify operation of the member functions.

6. Construct a class named **Rectangle** that has floating-point data members named **length** and **width**. The class should have a constructor that sets each data member to 0, member functions named **perimeter()** and **area()** to calculate the perimeter and area of a rectangle, respectively, a member function named **getdata()** to get a rectangle's length and width, and a member function named **showdata()** that displays a rectangle's length, width, perimeter, and area.

Include the **Rectangle** class within a working C++ program.

7. Given the class **point** which is defined as follows:

```
class point {

private:

int x,y;
```

```
public:
point( int xnew, int ynew);
void getdata();
void display();
};
point::point(int xnew, ynew) //constructor
{
x = xnew;
y = ynew;
}
void point::getdata()
{
cout << “Enter an integer value \n”;
cin >> x;
cout << “Enter an integer value \n”;
cin >> y;
}
void point::display()
{
cout << “Entered numbers are \n”;
```

```
cout << " x = " << x << '\t' << " y = " << y << endl;

}
```

Write a main program that creates two **point** objects on the heap, displays these objects and then destroys the two objects.

8. Given the following program:

```
#include <iostream.h>

void main( )

{

int *ptr_a = new int[20];

int *ptr_n = new int;

int i;

cout << " How many numbers are there ? \n";

cin >> *ptr_n;

for (i = 0; i <= *ptr_n – 1; ++i){

cout << " element : ";

cin >> ptr_a[i];

}

cout << " Contents of the array \n";

for (i = 0; i <= *ptr_n – 1; ++i){

cout << ptr_a[i];
```

```
cout << “\t”;

}

delete ptr_n;

delete [] ptr_a;

}
```

a. Explain the meaning of the program.

b. Run to determine the result of the program.

Exercise 8

1. Construct a class named **Complex** that contains two double-precision floating point data members named **real** and **imag**, which will be used to store the real and imaginary parts of a complex number. The function members should include a constructor that provides default values of 0 for each data member and a display function that prints an object’s data values.

Include the above class in a working C++ program that creates and displays the values of two **Complex** objects.

2. Construct a class named **Car** that contains the following three data members: a floating point variable named **engineSize** , a character variable named **bodyStyle** and an integer variable named **colorCode**. The function members should include (1) a constructor that provides default values of 0 for each numeric data members and ‘X’ for each character variable; and (2) a display function that prints the engine size, body style, and color code.

Include the class **Car** in a working C++ program that creates and displays two car objects.

3. Given the following class:

```
class Auto {
```

```
public:
Auto(char*, double);
displayAuto();
private:
char* szCarMake;
double dCarEngine;
};
Auto::Auto(char* szMake, double dEngine){
szCarMake = new char[25];
strcpy(szCarMake, szMake);
dCarEngine = dEngine;
}
Auto::displayAuto(){
cout<< “The car make: “<< szCarMake<< endl;
cout<< “The car engine size: “<< dCarEngine<<
endl;
}
void main(){
Auto oldCar(“Chevy”, 351);
Auto newCar(oldCar);
```

```
oldCar.displayAuto();

newCar.displayAuto();

}
```

1. Add an appropriate destructor to the class **Auto**.
2. Include the class **Auto** in a working C++ program that creates two **Auto** objects on the heap, displays them and destroys them.

4. Given the following class:

```
class Employee{

public:

Employee(const char*, const char*);

char *getFirstName() const;

char *getLastName() const;

private:

char *firstName;

char *lastName;

};

Employee::Employee(const char *first, const char
*last){

firstName = new char[strlen(first)+1];

strcpy(firstName, first);

lastName = new char[strlen(last)+1];
```

```
strcpy(lastName, last);
}
char *Employee::getFirstName() const{
return firstName;
}
char *Employee::getLastName() const{
return lastName;
}
void main(){
Employee *e1Ptr = new Employee(“Susan”, “Baker”);
Employee *e2Ptr = new Employee(“Robert”, “Jones”);
cout << “\n Employee 1: “
<< e1Ptr->getFirstName()
<< “ “ << e1Ptr->getLastName()
<< “\n Employee 2: “
<< e2Ptr->getFirstName()
<< “ “ << e2Ptr->getLastName()<< endl;
delete e1Ptr;
delete e2Ptr;
}
```

Add an appropriate destructor to the class **Employee**.

5. Given the following program which defines and uses the class Ob. What is the output of the **main()** function?

```
class Ob

{

private:

int field;

public:

Ob();

~Ob();

}

Ob::Ob()

{

field = 0;

cout << “Object created. “<< endl;

}

Ob::~Ob()

Cout << “Object detroyed. “<< endl;

}

void main()
```

```
{
Ob obj[6];
}
```

6. Given a base class named **Circle** which is defined as follows.

```
const double PI = 3.14159;
class Circle
{
protected:
double radius;
public:
Circle(double = 1.0); // constructor
double calcval();
};
// implementation section for Circle
// constructor
Circle::Circle(double r)
{
radius = r;
}
// calculate the area of a circle
```

```
double Circle::calcval()

{

return(PI * radius * radius);

}
```

a. Derive from the base class another class named **Cylinder** which contains an additional data member to store **length**. The only additional function members of **Sphere** should be a constructor and a **calcval()** function that returns the volume of the cylinder (Note: Volume = length*(pi*radius^2)).

b. Define another derived class name **Sphere** from the base **Circle** class. The only additional class members of **Sphere** should be a constructor and a **calcval()** function that returns the volume of the spere. (Note: Volume = 4/3*pi*radius^3).

c. Write the **main()** function in such a way that your program call all of the member functions in the **Cylinder** class and **Sphere** class.

7. Create a base class named **Point** that consists of an x and y coordinate. From this class, derive a class named **Circle** that has an additional data member named **radius**. For this derived class, the x and y data members represents the center coordinates of a circle. The function members of the first class should consist of a constructor and a **distance()** function that returns the distance between two points, where

distance = square-root((x2 – x1)^2 + (y2 – y1)^2)

Additionally, the derived class should have a constructor, an override **distance()** function that calculates the distance between circle centers, and a function named **area** that returns the area of a circle.

Include the two classes in a complete C++ program. Have your program call all of the member functions in each class. In addition, call the base class **distance** function with two **Circle** objects and explain the result returned by the function.

8.Create a base class named **Rectangle** that contains **length** and **width** data members. From this class, derive a class named Box having an additional data member named **depth**. The function members of the base **Rectangle** class should consist of a constructor and an **area** function. The derived **Box** class should have a constructor and an override function named **area** that returns the surface area of the box and a **volume()** function.

Include the two classes in a complete C++ program. Have your program call all of the member functions in each class and explain the result when the **area()** function is called using a **Box** object.

9. Modify the following code so that the overridden **setSound()** function executes correctly with the statements in the **main()** method.

```
class Instruments

{

public:

Instruments();

void setSound(char*);

char* getSound();

protected:

char* szSound;

}

Instruments:: Instruments(){

szSound = new char[25];

}
```

```
void Instruments::setSound(char* szSound){
szSound = “quiet”;
}
char* getSound(){
return szSound;
}
class Percussion: public Instruments
{
public:
Percussion();
void setSound(char*);
}
Percussion::Percussion(){
cout << “I repeat, the drum goes “
<< getSound() << endl;
}
void Percussion::setSound(char* szSound){
szSound = “boom”;
}
void main(){
```

```
Instruments* drums = new Percussion;

drums-> setSound();

cout<< “The drum goes “ << drums->getSound() <<
endl;

}
```

Assignments

1) Here is a challenging problem for those who know a little calculus. The Newton-Raphson method can be used to find the roots of any equation y(x) = 0.

In this method, the $(i+1)$st approximation, x_{i+1}, to a root of $y(x) = 0$ is given in terms of the i-th approximation, x_i, by the formula:

$$x_{i+1} = x_i - y(x_i)/y'(x_i)$$

For example, if $y(x) = 3x^2 + 2x - 2$, then $y'(x) = 6x + 2$, and the roots are found by making a reasonable guess for a first approximation x_1 and iterating using the equation

$$x_{i+1} = x_i - (3x_i^2 + 2x_i - 2)/(6x_i - 2)$$

Newton-Raphson method

1. Using the Newton-Raphson method, find the two roots of the equation 3x^2 + 2x –2 = 0. (Hint: Stop the loop when the difference between the two approximations is less than 0.00001.)
2. Extend the program written for a) so that it finds the roots of any function y(x) = 0, when the function for y(x) and the derivative of y(x) are placed in the code.

2) a. See the definition of determinant of a matrix as below.

The determinant of a 2×2 matrix

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

is $a_{11}a_{22} - a_{21}a_{12}$. Similarly, the determinant of a 3×3 matrix

$$\begin{vmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{vmatrix} =$$

$$a_{11}*D_1 - a_{21}*D_2 + a_{31}*D_3$$

where D_1 is determinant of $\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix}$

D_2 is determinant of $\begin{vmatrix} a_{12} & a_{13} \\ a_{32} & a_{33} \end{vmatrix}$

and D_3 is determinant of $\begin{vmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{vmatrix}$

Determinant of matrix

Using this information, write and test two functions, named det2() and det3(). The det2() function should accept the four coefficients of a 2 X 2 matrix and return its determinant. The det3() function should accept the nine coefficients of a 3 X 3 matrix and return its determinant by calling det2() to calculate the required 2 X 2 determinants.

b. Write and run a C++ program that accepts the nine coefficients of a 3 X 3 matrix in one main() function, passes these coefficients to det3() to calculate its determinant and then displays the calculated determinant.

3) Your professor has asked you to write a C++ program that can be used to determine grades at the end of the semester. For each student, who is identified by an integer number between 1 and 60, four examination grades must be kept. Additionally, two final grade averages must be computed. The first grade average is simply the average of all four grades. The second

grade average is computed by weighting the four grades as follows: the first grade gets a weight of 0.2, the second grade gets a weight of 0.3,the third grade a weight of 0.3 and the fourth grade a weight of 0.2; that is, the final grade is computed as:

0.2* grade1 + 0.3*grade2 + 0.3*grade3 + 0.2*grade4

Using this information, you are to construct a 60 X 6 two-dimensional array, in which the the first four columns for the grades, and the last two columns for the computed final grades. The output of the program should be a display of the data in the completed array.

For test purposes, the professor has provided the following data:

Student	grade 1	grade 2	grade 3	grade 4
1	100	100	100	100
2	100	0	100	0
3	82	94	73	86
4	64	74	84	94
5	64	74	84	94
6	94	84	74	64

Test data

4) A magic square is an n X n matrix in which each of the integer values from 1 to n*n appears exactly once and all column sums, row sums and diagonal sums are equal. For example, the following is a 3 X 3 magic square in which each row, each column, and each diagonal adds up to 15.

8	1	6
3	5	7
4	9	2

Magic matrix

1. Write a function that accepts a two-dimensional array and integer n and checks if the n X n matrix stored in the array is a magic square. The function should return true if it is a magic square and false if it isn't. And also design it to return the "magic sum" of the magic square (sum of each row = sum of each column = sum of each diagonal).
2. Write the driver program that:

    - Let you enter the elements of a matrix.
    - Calls a matrix-printer function to display it.
    - Calls your magic-square-checker function to check if it is a magic square and displays an appropriate message (including the magic sum if it is a magic square.)

5) A prime integer is any integer that is evenly divisible only by itself and 1. The Sieve of Eratosthenes is a method of finding prime numbers. It operates as follows:

1. Create an array with all elements initialized to 1 (true). Array elements with prime subscripts will remain 1. All other array elements will eventually be set to zero.
2. Starting with array subscript 2 (subscript 1 must be prime), every time an array element is found whose value is 1, loop through the remainder of the array and set to zero every element whose subscript is a multiple of the subscript for the element with value 1. For array subscript 2, all elements beyond 2 in the array that are multiple of 2 will be set to zero (subscript 4, 6, 8, 10,etc.); for array subscript 3, all elements beyond 3 in the array that are multiple of 3 will be set to zero (subscripts 6, 9, 12, 15, etc.); and so on.

When this process is complete, the array elements that are still set to one indicate that the subscript is a prime number. These subscripts can then be printed.

Write and run a C++ program that uses an array of 1000 elements to determine and print the prime numbers between 1 and 999. Ignore element 0 of the array.

6) The Colossus Airlines fleet consists of one plane with a seating capacity of 12. It makes one flight daily. Write a seating reservation program with the following features:

a. The program uses an array of 12 structures. Each structure should hold a seat identification number, a marker that indicates whether the seat is assigned and the name of the seat holder. Assume that the name of a customer is not more than 20 characters long.

b. The program displays the following menu:

1. Show the number of empty seats
2. Show the list of empty seats.

1. Show the list of customers together with their seat numbers in the order of the seat numbers
2. Assign a customer to a seat
3. Remove a seat assignment
4. Quit

c. The program successfully executes the promises of its menu. Choices (4) and (5) need additional input from the user, which is done inside the respective functions.

d. After executing a particular function, the program shows the menu again, except for choice (6).

7) In this problem, a customer calls in an order for bicycles, giving his or her name, address, number of bicycles desired, and the kind of bicycle. For now, all bicycles on one order must be the same kind. Mountain bikes cost $269.95 each and street bikes $149.50. The total bill is to be calculated for the order. Additionally, based on the user's knowledge of the customer, the customer is classified as either a good or bad credit risk. Based on the input data, the program is to prepare shipping instructions listing the customer's name, address, number and type of bikes, and the total amount due. Based on the creditworthiness of the customer, the program must indicate on the shipping instructions if this is a C.O.D. (cash on delivery) shipment or whether the customer will be billed separately.

The input and output requirements of this problem are relatively simple. On the input side, the items that must be obtained are:

1. Customer's name
2. Customer's address
3. Number of bicycles ordered
4. Type of bicycle (mountain or street)
5. Creditworthiness of the customer (good or bad)

For output, a set of shipping instructions is to be generated. The instructions must contain the first four input items, the total cost of the order, and the type of billing. The total cost is obtained as the number of bicycles ordered (input item 3) times the appropriate cost per bicycle, and the type of billing is determined by the creditworthiness of the customer (input item 5). If the customer is creditworthy, a bill be sent; otherwise requires cash payment on delivery. The customer record layout is as follows.

Field No.	*Field contents*	*Field type*
1	Customer name	Character [50]
2	Customer address	Character [50]
3	Bicycle ordered	Integer
4	Bicycle type	Character – M or S
5	Creditworthiness	Character – Y or N
6	Dollar value of order	Float

Layout of result

8) The Hanoi Tower is a puzzle consisting of a number of disks placed on three columns.

The disks all have different diameters and holes in the middle so they will fit over the columns (see Figure 1). All the disks start out on column A. The object of the puzzle is to transfer all the disks from column A to column C. Only one disk can be moved at a time, and no disk can be placed on a disk that is smaller that itself.

The solution to this puzzle is easily expressed as a recursive procedure where each n disk solution is defined in terms of an n-1 disk solution. To see how this works, first consider a one-disk puzzle. Clearly, this has a simple solution, where we move the disk from column A to column C.

Now consider the n-disk problem. Moving n disks can be viewed in terms of moving only n –1 disks (hence, the recursion) as follows:

a) Move n-1 disks from column A to column B, using column C as a temporary holding area.

b) Move the last disk (the largest) from A to from C.

c) Move the n-1 disks from column B to column C, using column A as a temporary holding area.

The process ends when the last task involving n =1 disk, i.e., the base case. This is accomplished by trivially moving the disk.

Write a program to solve the Hanoi Tower puzzle. Use a recursive function with four parameters:

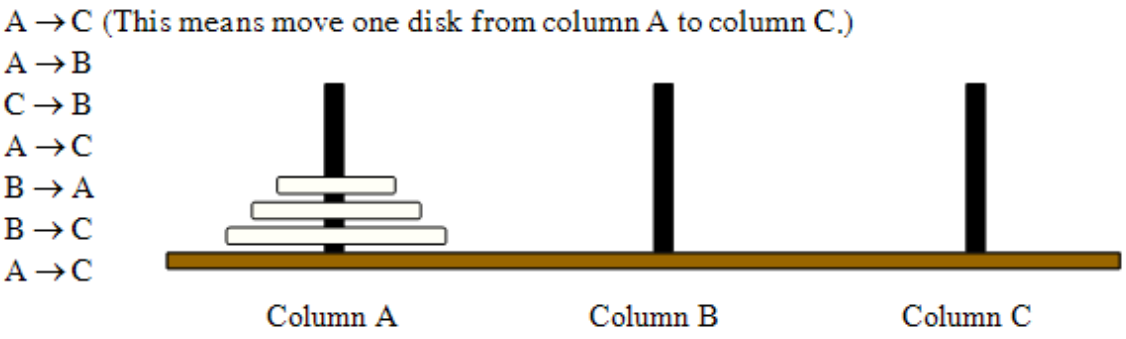
a) The number of disks to be moved

b) The column on which these disks are initially threaded.

c) The column to which this stack of disks is to be moved

d) The column to be used as a temporary holding area.

Your program should print the precise instructions it will take to move the disks from a starting column to the destination column. For example, to move a stack of three disks from column A to column C, your program should print the following series of moves:



Hanoi-Tower puzzle

9) a. Create a class named Fractions having two integer data members named for a fraction's numerator and denominator. The class's default constructor should provide both data members with default values of 1 if no explicit user initialization is provided. Additionally, provide member functions for displaying an object's data values. Also provide the class with the member functions that are capable of adding, subtracting, multiplying, and dividing two Fraction objects according to the following formulas:

Sum of two fractions: a/b + c/d = (ad +cb)/bd

Difference of two fractions: a/b – c/d = (ad – cb)/bd

Product of two fractions: (a/b)X (c/d) = ac/bd

Division of two fractions: (a/b)/(c/d) = ad/cb

b. Include the class written for a) in a working C++ program that tests each of the class's member functions.

10) A stack is an ordered collection of data items in which access is possible only at one end, called the top of the stack with the following basic operations:

1. Push: add an element to the top of the stack.

2. Pop: remove and return the top element of the stack.

3. Check if the stack is empty

4. Check if the stack is full.

It would be nice to have a stack class, because we could then use it to easily develop some applications which need stack data structure.

a. For the moment, assume that we need to define an integer stack class. Since a stack must stores a collection of integers, we can use an integer array to model the stack and a “pointer” named Top to indicate the location of the top of the stack. The array should have a fixed size. So we can begin the declaration of the class by selection two data members:

- Provide an integer array data member to hold the stack elements (the size of the array is a constant).
- Provide an integer data member to indicate the top of the stack.As for the member functions of the stack class, we have to define 5 member functions: the constructor which creates an empty stack and four other member functions (push, pop, check if the stack is empty, check if the stack is full).

b. After defining the stack class, write a main() function which does the following tasks:

- Creating one stack object.
- Pushing into the stack object ten integer elements which take values from 1 to 10 with the increment of 1.
- Popping one by one element from the stack object and displaying it out, repeating this action until the stack becomes empty.

c. Next, apply the stack data structure in solving the following problem: write a program that accepts a string from the user and prints the string backward. (Hint: Use a character stack)

Labworks

TABLE OF CONTENTS

Lab Session 10:

Inheritance

Programming Project Topic Examples

LAB SESSION 1: INTRODUCTION TO C++

1. OBJECTIVE

The objectives of Lab 1 are (1) to known how to run a simple C++ program; (2) to know the basic data types and operators; (3) to learn how to use variable declarations and assignment statements.

2. EXPERIMENT

2.1) Test the following program:

```
#include <iostream.h>

int main()

{

const float PI=3.14159;

float radius = 5;

float area;

area = radius * radius * PI; // Circle area calculation

cout << “The area is “ << area << “ with a radius of 5.\n”;
```

```
radius = 20; // Compute area with new radius.

area = radius * radius * PI;

cout << “The area is “ << area << “ with a radius
of 20.\n”;

return 0;

}
```

1. Run the above program
2. Use #define to define the constant PI
3. Declare the constant PI in the file “mydef.h”, and then use the #include directive to insert the header file in the above program.

2.2) Debug the following code segment.

```
#include <iostream.h>

int main()

{

const int age=35;

cout << age << “\n”;

age = 52;

cout << age << “\n”;

return 0;

}
```

2.3) What is the result of each following expression:

1. 1 + 2 * 4 / 2

2. (1 + 2) * 4 / 2
3. 1 + 2 * (4 / 2)
4. 9 % 2 + 1
5. (1 + (10 - (2 + 2)))

2.4) Run the following programs and explain their results.

a.

```
void main()
{
short i = -3;
unsigned short u;
cout<< sizeof(i)<< &i;
cout<<sizeof(u)<<&u;
cout << (u = i) << "\n";
}
```

b.

```
void main()
{
byte i = 125*4/10;
cout << i << "\n";
}
```

2.5) Write a program that inputs two time points and display the difference between them.

2.6) Run the following programs and explain their results:

a.

```
#include <iostream.h>

int main()

{

int f, g;

g = 5;

f = 8;

if ((g = 25) || (f = 35))

cout << “g is “ << g << “ and f got changed to “
<< f;

return 0;

}
```

b.

```
#include <iostream.h>

void main()

{

if (!0)

{ cout << “C++ By Example \n”; }

int a = 0;
```

```
if ( a !=0 && 2/a >0 )

cout<< “hello”;

}
```

2.7) Write a program that inputs the three grades for mathematics, physics and chemistry. And then it displays the average of the three grades in the following format:

```
Math              8.5
Physics             9
Chemistry          10
---------------------------
AVG             9.17
```

LAB SESSION 2: SELECTION STRUCTURES

1. OBJECTIVE

The objective of Lab 2 is to practice C++’s selection structures, such as:

- if
- if … else
- switch

2. EXPERIMENT

2.1) Run the following program:

```
// BEEP : ‘\x07’

#include <iostream.h>
```

```
#define BEEP cout << “\a \n”
int main()
{
int num;
cout << “Please enter a number “;
cin >> num;
if (num == 1)
{ BEEP; }
else if (num == 2)
{ BEEP; BEEP; }
else if (num == 3)
{ BEEP; BEEP; BEEP; }
else if (num == 4)
{ BEEP; BEEP; BEEP; BEEP; }
else if (num == 5)
{ BEEP; BEEP; BEEP; BEEP; BEEP; }
return 0;
}
```

2.2) Run the following program:

```
#include <iostream.h>
```

```
#define BEEP cout << “\a \n”
int main()
{
int num;
cout << “Please enter a number “;
cin >> num;
switch (num)
{
case (1): { BEEP;
break; }
case (2): { BEEP; BEEP;
break; }
case (3): { BEEP; BEEP; BEEP;
break; }
case (4): { BEEP; BEEP; BEEP; BEEP;
break; }
case (5): { BEEP; BEEP; BEEP; BEEP; BEEP;
break; }
}
return 0;
```

```
}
```

2.3) Remove the **break** statements from the above program and then try it again and explain the result.

2.4) Use **switch** statement to rewrite the following code segment:

```
if (num == 1)

{cout << “Alpha”; }

else if (num == 2)

{ cout << “Beta”; }

else if (num == 3)

{ cout << “Gamma”; }

else

{ cout << “Other”; }
```

2.5) Write a program that inputs the integer variable n consisting of 3 digits and displays it in ascending order of digits.

Example: n = 291. It should be displayed as 129.

2.6) Write a program that inputs a date with correct month, year and day components, and then checks if the year is a leap year or not. Show the result on the screen.

2.7) Write a program that can calculate the fee for a taxi ride. The formula is as follows:

- The first kilometer costs 5000.
- Each next 200m costs 1000.
- If the distance is more than 30km then each next kilometer adds 3000 to the fee.

The program has to input the total distance (in km) and calculate the charge.

2.8) Write a program that inputs a date consisting of day, month, and year components. Check if the date is valid or not and if it is, determine what its previous day is. Example: if the date is 01/01/2003 then its previous day is 31/12/2002.

LAB SESSION 3: REPETITION STRUCTURES

1. OBJECTIVE

The objectives of Lab 3 are to practice the C++'s repetition structures, such as:

- for
- while
- do…while

2. EXPERIMENT

2.1) Determine the result of the following code segment. Explain this result.

```
int a = 1;

while (a < 4)

{

cout << “This is the outer loop\n”;

a++;

while (a <= 25)

{
```

```
break;
cout << “This prints 25 times\n”;
}
}
```

2.2) Test the following program.

```
#include <iostream.h>
#include <iomanip.h>
void main()
{
float total_grade=0.0;
float grade_avg = 0.0;
float grade;
int grade_ctr = 0;
do
{
cout << “What is your grade? (-1 to end) “;
cin >> grade;
if (grade >= 0.0)
{
total_grade += grade; // Add to total.
```

```
grade_ctr ++;

} // Add to count.

} while (grade >= 0.0); // Quit when -1 entered.

grade_avg = (total_grade / grade_ctr);

cout << “\nYou made a total of “ <<
setprecision(1) <<

total_grade << “ points.\n”;

cout << “Your average was “ << grade_avg << “\n”;

if (total_grade >= 450.0)

cout << “** You made an A!!”;

return 0;

}
```

2.3) Test the following program:

```
#include <iostream.h>

void main()

{

int outer, num, fact, total;

cout << “What factorial do you want to see? “;

cin >> num;

for (outer=1; outer <= num; outer++)
```

```
{

total = 1;

for (fact=1; fact<= outer; fact++)

{ total *= fact; }

}

cout << “The factorial for “ << num << “ is “ << total;

return 0;

}
```

2.4) Determine the result of each following code segment:

a.

```
for (ctr=10; ctr>=1; ctr-=3)

{ cout << ctr << “\n”; }
```

b.

```
n =10;

i=1;

for (i = 0 ; i < n ; i++)

cout<< ++i<<endl;
```

c.

```
for (i=1; i<=10; i++);
```

```
for (j=1; j<=5; j++)
{
if ( i == j )
continue;
else ( i>j)
break;
else
cout << i << j;
cout<< endl;
}
```

d.

```
i=1;
start=1;
end=5;
step=1;
for (; start>=end;)
{
cout << i << “\n”;
start+=step;
end--;
```

```
}
```

2.5) Write a C++ program to convert Celsius degrees to Fahrenheit. The Celsius degrees increase from 5 to 50 with the increment of 5 degrees. The resultant table should be in the following form with appropriate headings:

Celsius degrees	Fahrenheit degrees
5	xxxx
10	xxxx
15	xxxx
20	xxxx

Fahrenheit = (9.0 / 5.0) * Celsius + 32.0;

2.6) Write a C++ program to compute the sum and average of N single-precision floating point numbers entered by the user. The value of N is also entered by the user.

2.7) The value of Euler constant, e, is approximated by the following series:

e = 1 + 1/1! +1/2! + 1/3! + 1/4! + 1/5! + …

Using this series, write a C++ program to compute e approximately. Use **while** structure that terminates when the difference between two successive approximations becomes less than 1.0E-6.

2.8) Write a C++ program to calculate and display the amount of money accumulated in a saving bank account at the end of each year for 10 years when depositing 1000$ in the bank. The program has to display the amount of money when interest rates change from 6% to 12% with the increment 1%. Thus, you should use two nested loops: the outer loop iterates according to interest rates and the inner loop iterates according to the years.

2.9) Write a C++ program that inputs a positive integer n and lists out n first prime numbers.

2.10) Write a C++ program to display an equilateral triangle with the height h (h is entered from the user). Example: h = 4

LAB SESSION 4: ARRAYS

1. OBJECTIVE

The objective of Lab session 4 is to get familiar with arrays, one-dimensional and two-dimensional arrays.

2. EXPERIMENT

2.1) Test the following program.

```
#include <iostream.h>

const int NUM = 8;

void main()

{

int nums[NUM];

int total = 0; // Holds total of user’s eight numbers.
```

```
int ctr;

for (ctr=0; ctr<NUM; ctr++)

{

cout << “Please enter the next number...”;

cin >> nums[ctr];

total += nums[ctr];

}

cout << “The total of the numbers is “ << total <<
“\n”;

return;

}
```

2.2) If the array weights is declared as in the following statement, then what the value of weights[5] is ?

int weights[10] = {5, 2, 4};

2.3) Given the statement:

```
char teams[] = {‘E’,’a’,’g’,’l’,’e’,’s’,’\0', ’R’,
‘a’,’m’,’s’,’\0'};
```

which of the following statement is valid?

a. cout << teams;

b. cout << teams+7;

c. cout << (teams+3);

d. cout << teams[0];

e. cout << (teams+0)[0];

f. cout << (teams+5);

2.4) Given the array declaration:

```
int grades[3][5] =
{80,90,96,73,65,67,90,68,92,84,70, 55,95,78,100};
```

Determine the value of the following subscripted variables:

a. grades[2][3]

b. grades[2][4]

c. grades[0][1]

2.5) Write a C++ program that inputs an array consisting of n single-precision floating point numbers and finds the smallest element in the array. (n is an integer that is entered by the user).

2.6) Write a C++ program that inputs an integer array and finds the last element in the array.

2.7) Write a C++ program that inputs an integer array and then inserts the integer value X in the first position in the array.

2.8) Write a C++ program that inputs an integer array and checks if all the elements in the array are unique (i.e. we can not find any pair of elements that are equal to each other).

2.9) Write a C++ program that inputs a matrix and then transposes it and displays the transposed matrix. Transposing a square matrix is to swap:

a(i,j) <==> a(j,i) for all i, j

For example, given the matrix

```
1  3  5
2  7  9
4  1  6
```

After transposing it, it becomes as follows:

```
1  2  4
3  7  1
5  9  6
```

2.10) Write a C++ program that inputs an integer n and two square matrices with order of n. Then the program calculates the multiplication of the two matrices and displays the resultant matrix.

LAB SESSION 5: STRINGS AND STRUCTURES

1. OBJECTIVE

The objectives of Lab session 5 are (1) to get familiar with strings; (2) to get familiar with structures; (3) to practice on processing arrays of structures.

2. EXPERIMENT

2.1) Write a C++ program that accepts a string of characters from a terminal and displays the hexadecimal equivalent of each character.

(Hint: Use the cin.getline() function to input a string)

2.2) Write a C++ program that accepts a two strings of characters from a keyboard and displays the concatenation of the two strings.

(Hint: Use the cin.getline() function to input a string)

2.3) Write and run a program that reads three strings and prints them out in an alphabetical order.

(Hint: Use the strcmp() function).

2.4) Write a C++ program that accepts a string of characters from a terminal and converts all lower-case letters in the string to upper-case letters.

2.5) a. Using the data type

```
struct MonthDays
{
char name[10];
int days;
};
```

define an array of 12 structures of type MonthDays. Name the array months and initialize the array with the names of the 12 months in a year and the number of days in each month.

b. Include the array created in a) in a program that displays the names and number of days in each month.

2.6) a. Declare a data type named Car, which is a structure consisting of the following information for each car:

Car_Number	Kms_driven	Litres_used
25	1450	62
36	3240	136
44	1792	76
52	2360	105
68	2114	67

b. Using the data type defined in a) write a C++ program that inputs the data given in the above table into an array of 5 structures., and then computes and displays a report consisting of 3 columns: car-number, kms-driven and gas-litres-used.

LAB SESSION 6: FUNCTIONS

1. OBJECTIVE

The objectives of Lab session 6 is to practice on C++ functions.

2. EXPERIMENT

2.1) Given the following function:

```
int square(int a)
{
a = a*a;
return a;
}
```

a. Write a C++ program that reads an integer n and invokes the function to compute its square and displays this result.

b. Rewrite the function so that the parameter is passed by reference. It is named by square2. Write a C++ program that reads an integer x and invokes the function square2 to compute its square and displays this result and then displays the value of x. What is the value of x after the function call? Explain this value.

2.2) Read the following function that can compute the largest integer which square is less than or equal to a given integer.

```
int Intqrt(int num)

{

int i;

i = 1;

do

++ i

while i*i <= num;

return(i –1);

}
```

Write a C++ program that inputs an interger n and invokes the function Intqrt to compute the largest integer which square is less than or equal to n.

2.3) a. Write a function that can find the average of all the elements in a double precision floating point array that is passed to the function as a parameter.

b. Write a C++ program that inputs a double precision floating point array and invokes the above function to find the average of all elements in the array and displays it out.

2.4) Write a C++ function that checks if a square matrix with order of n is symmetric or not. And then write a C++ program that inputs a square matrix with order of n and then checks if it is symmetric or not.

2.5) A palindrome is a string that reads the same both forward and backward. Some examples are

"ABCBA" "RADAR" "otto" "i am ma i" "C"

Given the following function that returns **true** if the parameter string is a palindrome or **false**, otherwise.

```
bool palindrome(char strg[])
{
int len, k, j;
len = strlen(strg);
k = len/2;
j = 0;
bool palin = true;
while ( j < k && palin)
if (strg[j] != strg[len -1-j])
palin = false;
else
++ j;
```

```
return (palin)

}
```

Write a C++ program that reads several strings, one at a time and checks if each string is a palindrome or not. Use a **while** loop in your program. The loop terminates when the user enters a string starting with a '*'.

2.6) Write a boolean function that can checks if all the elements in integer array are unique (i.e. we can not find any pair of elements that are equal to each other).

Write a C++ program that inputs an integer array and invokes the function to check if all the elements in integer array are unique.

2.7) Given the following formula for computing the number of combinations of m things out of n, denote C(n, m).

C(n, m) = 1 if m = 0 or m=n

C(n, m) = C(n-1, m) + C(n-1, m-1) if 0 < m < n

1. Write a recursive function to compute C(n,m).
2. Write a complete C++ program that reads two integers N and M and invokes the function to compute C(N,M) and prints the result out.

2.8) The greatest common divisor of two positive integers is the largest integer that is a divisor of both of them. For example, 3 is the greatest common divisor of 6 and 15, and 1 is the greatest common divisor of 15 and 22. Here is a recursive function that computes the greatest common divisor of two positive integers:

```
int gcd(int p, int q)

{

int r ;

if (( r = p%q == 0)
```

```
return q ;

else

return gcd(q, r) ;

}
```

a. First write a C++ program to test the function.

b. Write and test an equivalent iterative function.

LAB SESSION 7: POINTERS

1. OBJECTIVE

The objectives of Lab session 7 are to get familiar with how to use pointers in C++.

2. EXPERIMENT

2.1) Run the following program to determine the size of two data types long and byte.

```
# include <iostream.h>

void main()

{

byte* a;

long* b;

cout<<sizeof(a)<<endl;
```

```
cout<<sizeof(b)<<endl;
}
```

2.2) Given the following code segment:

```
float pay;
float *ptr_pay;
pay=2313.54;
ptr_pay = &pay;
```

determine the values of the following expressions:

a. pay

b. *ptr_pay

c. *pay

d. &pay

2.3) Read for understanding the following program:

```
#include<iostream.h>
void main()
{
int a;
int *aPtr; // aPtr is a pointer to an integer
a = 7;
aPtr = &a; //aPtr set to address of a
```

```
cout << “The address of a is “ << &a
<< “\nThe value of aPtr is “ << aPtr;
cout << “\n\nThe value of a is “<< a
<< “\nThe value of *aPtr is “ << *aPtr
<< endl;
}
```

Run the program and explain the output.

2.4) Given the following array and pointer declarations:

```
int ara[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *ip1, *ip2;
```

Determine which of the following assignments are valid.

a. ip1 = ara;

b. ip2 = ip1 = &ara[3];

c. ara = 15;

d. *(ip2 + 2) = 15; // Assuming ip2 and ara are equal.

2.5) Run the following program and explain the output.

```
#include <iostream.h>
void swap(int a[], int *c1, int *c2, int *d1, int
*d2);
void main()
```

```
{
int a[2], c1, c2,d1,d2;
int *x1, *x2, *y1, *y2;
a[0] = 1 ; a[1] =2;
c1 = 1; c2 =2;
d1 = 1; d2 =2;
x1 = &c1; x2 = &c2;
y1 = &d1; y2 = &d2;
swap(a, x1,x2,y1,y2);
cout<<a[0]<<a[1]<<" "
<< *x1<<*x2<<" "
<<*y1<<*y2;
swap(a, x1,x2,y1,y2);
cout<<a[0]<<a[1]<<" "
<< *x1<<*x2<<" "
<<*y1<<*y2;
}
void swap(int a[], int *c1, int *c2, int *d1, int
*d2)
{
```

```
a[0] = 2 ; a[1] =1;

*c1=2, *c2 =1;

int* temp = d1;

d1 =d2;

d2 = temp;

}
```

2.6) Write a declaration to store the following values into an array named rates: 12.9, 18.6, 11.4, 13.7, 9.5, 15.2, 17.6. Include the declaration in a program that displays the values in the array using pointer notation.

2.7) Given the following function that can find the largest element in an integer array. Notice that the function scans the elements in the array using pointer arithmetic:

```
int findMax(int * vals, int numEls)

{

int j, max = *vals;

for (j = 1; j < numEls; j++)

if (max < *(vals + j))

max = *(vals + j);

return max;

}
```

Write a C++ program that inputs an integer array and invokes the above function to find the largest element in that array and displays the result out.

2.8) In the following program, the function str_output() can display a string which is passed to it as a a pointer parameter:

```
#include<iostream.h>

#include<string.h>

#define MAX 80

void str_output(char *);

int main()

{

char a[MAX], b[MAX];

cin.getline(a, MAX, '\n');

str_output(a);

cout << endl;

strcpy(b,a);

str_output(b);

cout<< endl;

return 0;

}

void str_output(char *ptr)

{

……..
```

```
}
```

a. Complete the function str_output() which displays each element in the string using pointer notation.

b. Run to test the whole program.

2.9) a. Write a function named days() that determines the number of days from the date 1/1/1900 for any date passed as a structure. Use the Date structure:

```
struct Date
{
int month;
int day;
int year;
}
```

In writing the days() function, use the convention that all years have 360 days and each month consists of 30 days. The function should return the number of days for any Date structure passed to it.

b. Rewrite the days() function to receive a pointer to a Date structure rather than a copy of the complete structure.

c. Include the function written in b) in a complete C++ program.

LAB SESSION 8: INTRODUCTION TO CLASSES

1. OBJECTIVE

The objectives of Lab session 8 are (1) to get familiar with how to define object classes; (2) to practice to write constructors and (3) to learn how to

dynamically allocate/deallocate memory on the heap.

2. EXPERIMENT

2.1) a. Read to understand the following program which uses the class student. Organize the program in one source program and run it on a C++ environment.

```
#include<iostream.h>

class student{

private:

long int rollno;

int age;

char sex;

float height;

float weight;

public:

void getinfo();

void disinfo();

};

void student::getinfo()

{

cout << " Roll no :";
```

```
cin >> rollno;
cout << " Age :";
cin >> age;
cout << " Sex:";
cin >> sex;
cout << " Height :";
cin >> height;
cout << " Weight :";
cin >> weight;
}
void student::disinfo()
{
cout<<endl;
cout<< " Roll no = "<< rollno << endl;
cout<< " Age =" << age << endl;
cout<< " Sex =" << sex << endl;
cout<< " Height =" << height << endl;
cout<< " Weight =" << weight << endl;
}
void main()
```

```
{

student a;

cout << " Enter the following information " <<
endl;

a.getinfo();

cout << " \n Contents of class "<< endl;

a.disinfo();

}
```

b. Reorganize the program into an interface file and an implementation file and then run the program.

2.2) Given the class student as defined in 2.1.a. Write a complete C++ program in which the main() function creates an array of size 10 to store student objects and prompts the user to enter the data for the student objects in the array and then displays the objects in the array.

2.3) Given the class student as defined in 2.1.a. Write a complete C++ program in which the main() function performs the following tasks:

- to declare a run-time-allocated array on the heap to contain the student objects.
- to prompt the user to enter an integer n and allocate a memory area on the heap to store n student objects.
- to prompt the user to enter the student objects and store them in the array and display all the objects on the screen.
- to deallocate the memory area for the array on the heap.

2.4) Test the following program:

```
class Int{
```

```
private:
int idata;
public:
Int(){
idata=0;
cout<<"default constructor is called"<<endl;
}
Int(int d){
idata=d;
cout<<"constructor with argument is called"<<endl;
}
void showData(){
cout<<"value of idata: "<<idata<<endl;
}
};
void main()
{
Int i;
Int j(8);
Int k=10;
```

```
Int *ptrInt = new Int();

ptrInt->showData();

delete ptrInt;

}
```

What are the outputs of the program? Explain the results.

What is the purpose of the statement delete ptrInt; in the program?

2.5) Define a class named Rectangle which contains two single-precision floating point data members: length and width. The class has some member functions:

- A constructor with no parameters that assigns 0 to two data members of the created object.
- A constructor with two single-precision floating-point parameters which assigns two parameters to the two data members of the created object.
- Function perimeter() to compute the perimeter of the rectangle.
- Function area() to compute the area of the rectangle.
- Function getdata( ) to prompt the user to enter the length and width for a rectangle.
- Function showdata( ) to display length, width, perimeter and area of a rectangle.

Include the class Rectangle in a complete C++ program. The main() function of this program creates two Rectangle objects using the two constructors respectively and displays the data of the two objects to check the working of all the member functions.

Then modify the program by replacing the two above constructor functions by a constructor with default arguments.

2.6) Define a class named CStudent which consists of the following data members:

- Student id-number (integer).
- An array of size 5 to contains at most 5 grades (single-precision floating point numbers).
- An integer to indicate the number of entered grades.

The class also has the following member functions:

- the constructor which assigns the initial values 0 to all data members of each Cstudent objects.
- A function to get a student-id-number.
- A function to get one grade and update the total of the entered grades.
- A function to compute the average of all entered grades of a student.
- A function to display student-id-number, and the average grade of that student.Include the class CStudent in a complete C++ program. This program creates one CStudent object, inputs the data for the object and displays the object's data to verify the workings of the member functions.

2.7) Test the following program which uses a run-time allocated array.

```
#include <iostream.h>

void main()

{

int num;

cout<< "Please enter the numbers of input: "

cin>>num;

int a = new int [num];

int total = 0; // Holds total of user's eight numbers.

int ctr;
```

```
for (ctr=0; ctr<num; ctr++)

{

cout << “Please enter the next number...”;

cin >> a[ctr];

total += a[ctr];

}

cout << “The total of the numbers is “ << total <<
“\n”;

return;

delete [] a;

}
```

2.8) Given a class named IntArray that contains two private data members: a pointer to the beginning of the array, and an integer representing the size of the array. The public functions include a constructor and member functions that show every element in the IntArray, and show the first IntArray element only. The definition of the class IntArray is as follows:

```
// IntArray.h

class IntArray

{

private:

int* data; //pointer to the integer array

int size;
```

```
public:
IntArray(int* d, int s);
void showList();
void showFirst( );
};
// IntArray.cpp
IntArray::IntArray(int* d, int s)
{
data = d;
size = s;
}
void IntArray::showList()
{
cout<<"Entire list:" <<endl;
for(int x = 0; x< size; x++)
cout<< data[x]<<endl;
cout<< "----------------------"<< endl;
}
void IntArray::showFirst()
{
```

```
cout<< "First element is ";

cout << data[0]<< endl;

}
```

a. Add to the class IntArray one more member function named findMax which returns the largest element in the array.

b. Write a main program that instantiates one array of integers and then displays the array, the first element and the largest element of the array.

LAB SESSION 9: OBJECT MANIPULATION

1. OBJECTIVE

The objectives of Lab session 9 are (1) to learn to write parameterized constructor, constructor with default arguments and (2) to practice destructors.

2. EXPERIMENT

2.1) Run the following program in a C++ environment.

```
class Int{

private:

int idata;

public:

Int(){

idata=0;
```

```
cout<<"default constructor is called"<<endl;
}
Int(int d=9){
idata=d;
cout<<"constructor with argument is called"<<endl;
}
void showData(){
cout<<"value of idata: "<<idata<<endl;
}
};
void main()
{
Int i;
Int j(8);
Int k=10;
}
```

a. Explain why the program incurs a compile-time error.

b. Modify the program in order to remove the above error. Run the modified program.

2.2) Run the following program in a C++ environment.

```
class Vector{
private:
int *value;
int dimension;
public:
Vector(int d=0){
dimension=d;
if (dimension==0)
value=NULL;
else{
value=new int[dimension];
for (int i=0; i<dimension; i++)
value[i]=0;
}
}
void showdata(){
for (int i=0; i<dimension; i++)
cout<<value[i];
cout<<endl;
}
```

```
~Vector(){
if (value!=NULL)
delete value;
}
};
void main()
{
Vector v(5);
v.showdata();
Vector v2(v);
v2.showdata();
}
```

a. Explain why the program incurs one memory error at run-time.

b. Now add the following code segment in the Vector class definition.

```
Vector(const Vector& v){
dimension = v.dimension;
value=new int[dimension];
for (int i=0; i<dimension; i++)
value[i]=v.value[i];
}
```

Check if the program still incurs the memory error or not. Explain why.

2.3) a. Test the following program in a Visual C++ environment:

```
class some{ // code segment a
public:
~some() {
cout<<"some's destructor"<<endl;
}
};
void main() {
some s;
s.~some();
}
```

What is the output of the above program during execution? Explain the output.

b. Test the following program in Visual C++ environment:

```
class some{ // code segment b
int *ptr;
public:
some(){
ptr= new int;
```

```
}
~some(){
cout<<"some's destructor"<<endl;
if (ptr!=NULL){
cout<<"delete heap memory"<<endl;
delete ptr;
}
}
};
void main()
{
some s;
// s.~some();
}
```

What is the output of the above program during execution? Explain the output.

c. In the main() function of the program in b, if we remove the two slashes (//) before the statement s.~some(); then what the result is when the program is executed ? Explain why.

2.4) Given the class definition as follows:

```
class Auto {
```

```
public:
Auto(char*, double);
displayAuto(char*, double);
private:
char* szCarMake;
double dCarEngine;
};
Auto::Auto(char* szMake, double dEngine){
szCarMake = new char[25];
strcpy(szCarMake, szMake);
dCarEngineSize = dCarEngine;
}
Auto::displayAuto(){
cout<< “The car make: “<< szCarMake<< endl;
cout<< “The car engine size: “<< dCarEngine<<
endl;
}
void main(){
Auto oldCar(“Chevy”, 351);
Auto newCar(oldCar);
```

```
oldCar.displayAuto();

newCar.displayAuto();

}
```

1. Add an appropriate copy constructor to the Auto class .
2. Add an appropriate destructor to the Auto class .

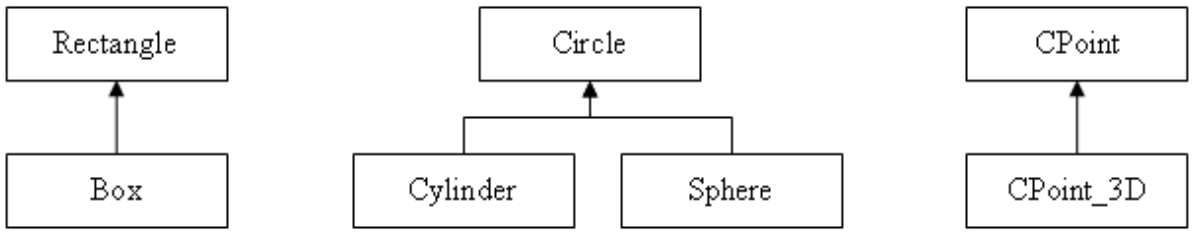
c. Run all the modifications in a C++ environment.

LAB SESSION 10: INHERITANCE

1. OBJECTIVE

The objectives of Lab session 10 are (1) to get familiar with class inheritance; (2) to learn the workings of the constructor and destructor of a derived class.

This lab session will use the inheritance hierarchies in the following figure.



2. EXPERIMENT

2.1) Given the Point class defined as follows.

```
class Point{
```

```
private:
int color;
protected:
double x;
double y;
public:
Point(double x=0, double y=0){
this->x=x; this->y=y;
}
void move(double dx, double dy){
x=x+dx;
y=y+dy;
}
~Point(){
cout<<"Destructor Point called";
}
};
```

Derive from the class Point, another class named Point_Derive1 class, which is defined as follows.

```
class Point_Derive1:public Point{
```

```
private:

double z;

public:

Point_Derive1();

void move(double dx, double dy, double dz);

~Point_Derive1();

};
```

a. List out the data members and member functions of the Point_Derive1 class. And determine the access specifier of each of these members.

Derive from the class Point, another class named Point_Derive2 class, which is defined as follows.

```
class Point_Derive2:protected Point{

private:

double z;

public:

Point_Derive1();

void move(double dx, double dy, double dz);

~Point_Derive1();

};
```

b. List out the data members and member functions of the Point_Derive2 class. And determine the access specifier of each of these members.

Derive from the class Point, another class named Point_Derive3 class, which is defined as follows.

```
class Point_Derive3:private Point{

private:

double z;

public:

Point_Derive1();

void move(double dx, double dy, double dz);

~Point_Derive1();

};
```

c. List out the data members and member functions of the Point_Derive3 class. And determine the access specifier of each of these members.

2.2) Given the following program in which the Cylinder class is a subclass derived from the Circle class

```
#include <iostream.h>

#include <math.h>

const double PI = 2.0 * asin(1.0);

// class declaration

class Circle

{

protected:
```

```
double radius;
public:
Circle(double = 1.0); // constructor
double calcval();
};
// implementation section for Circle
// constructor
Circle::Circle(double r)
{
radius = r;
}
// calculate the area of a circle
double Circle::calcval()
{
return(PI * radius * radius);
}
// class declaration for the derived class
// Cylinder which is derived from Circle
class Cylinder : public Circle
{
```

```
protected:

double length; // add one additional data member
and

public: // two additional function members

Cylinder(double r = 1.0, double l = 1.0) :
Circle(r), length(l) {}

double calcval();

};

// implementation section for Cylinder

double Cylinder::calcval() // this calculates a
volume

{

return length * Circle::calcval(); // note the
base function call

}

int main()

{

Circle circle_1, circle_2(2); // create two Circle
objects

Cylinder cylinder_1(3,4); // create one Cylinder
object

cout << "The area of circle_1 is " <<
circle_1.calcval() << endl;
```

```
cout << "The area of circle_2 is " <<
circle_2.calcval() << endl;

cout << "The volume of cylinder_1 is " <<
cylinder_1.calcval() << endl;

circle_1 = cylinder_1; // assign a cylinder to a
Circle

cout << "\nThe area of circle_1 is now " <<
circle_1.calcval() << endl;

return 0;

}
```

a. Run the program in a Visual C++ environment and determine the output of the program.

b. Modify the above program by deriving a subclass named Sphere from the base class Circle. Member functions of Sphere class include a constructor and a function named calcval() which returns the volume of the sphere. (The formula to compute the volume of a sphere is (4/3)*pi* R*R*R). And modify the main() function in order that it invokes all the member functions of the Sphere class.

2.3) Define a base class named Rectangle that contains two data members length and width. From this class, derive a subclass named Box which includes one more data member, depth. Two member functions of the base class are the constructor and a function named area() which returns the area of the rectangle. The derived class Box should have its own constructor and an overriding function area() which returns the surface area of the box and the function volume() that returns the volume of the box.

Write a complete C++ program which invokes all the member functions of the two above classes. Besides, the main() function also invokes the area() function of the base class to apply to a Box object. Explain the result of this function call.

2.4) a. Construct a class named CPoint that consists of the following data members and member functions:

- x, y coordinates
- a constructor with two parameters representing x, y coordinates (their default values are allowed to be 0)
- a member function named display, which prints out two coordinates on the screen.
- A member function named getInfo, which accepts two input data from the user for x, y coordinates.
- Two member functions named setX, setY to update values to x, y, respectively.
- Two member functions named getX, getY to retrieve values from x, y, respectively.
- A member function named distance, which accepts as parameter an object of class CPoint and calculates the distance from the object invoking the function to the parameter object.

b. Construct a derived class named CPoint_3D from the class CPoint, described as follows:

- There is one more data member: coordinate z.
- There are overriding member functions in CPoint_3D for all corresponding member functions in the class CPoint.

c. Include the class constructed in a) and b) in a working C++ program. Have your program call all of the member functions in the CPoint_3D class.

PROGRAMMING PROJECT TOPIC EXAMPLES

PROJECT 1

This project aims to review all the chapters from Chapter 1 (Introduction to Computers and Programming) to Chapter 6 (Functions and Pointers) which focus on structured programming.

An example of Project 1 can be described as follows:

The Colossus Airlines fleet consists of one plane with a seating capacity of 12. It makes one flight daily. Write a seating reservation program with the following features:

- The program uses an array of 12 structures. Each structure should hold a seat identification number, a marker that indicates whether the seat is assigned and the name of the seat holder. Assume that the name of a customer is not more than 20 characters long.

- The program displays the following menu with six choices:

    - Show the number of empty seats
    - Show the list of empty seats.
    - Show the list of customers together with their seat numbers in the order of the seat numbers
    - Assign a customer to a seat
    - Remove a seat assignment
    - Quit

- The program successfully executes the promises of its menu. Choices (4) and (5) need additional input from the user which is done inside the respective functions.
- After executing a particular function, the program shows the menu again, except for choice (6).

PROJECT 2

This project aims to review all the chapters from Chapter 7 (Introduction to Classes) to Chapter 8 (Object Manipulation - Inheritance) which focus on the basics of object-oriented programming.

An example of Project 2 can be described as follows:

A **stack** is an ordered collection of data items in which access is possible only at one end, called the top of the stack with the following basic operations:

- Push: add an element to the top of the stack.
- Pop: remove and return the top element of the stack.
- Check if the stack is empty
- Check if the stack is full.

It would be nice to have a **stack** class, because we could then use it to easily develop some applications which need stack data structure.

For the moment, assume that we need to define an **integer stack** class. Since a stack must stores a collection of integers, we can use an integer array to model the stack and a "pointer" named **Top** to indicate the location of the top of the stack. The array should have a fixed size. So we can begin the declaration of the class by selection two data members:

- Provide an integer array data member to hold the stack elements (the size of the array is a constant).
- Provide an integer data member to indicate the top of the stack.

As for the member functions of the stack class, we have to define 5 member functions: the constructor which creates an empty stack and four other member functions (push, pop, check if the stack is empty, check if the stack is full).

After defining the stack class, write a main() function which does the following tasks:

- Creating one stack object.
- Pushing into the stack object ten integer elements which take values from 1 to 10 with the increment of 1.
- Popping one by one element from the stack object and displaying it out, repeating this action until the stack becomes empty.

Next, apply the stack data structure in solving the following problem: write a program that accepts a string from the user and prints the string backward. (Hint: Use a character stack)

Introduction to Computers and Programming
This chapter discusses what computers are, how they work and how they are programmed. This chapter also includes an introduction to problem solving and program development.

Hardware and Software

A **computer** is a device capable of performing computations and making logical decisions at speeds millions and even billions of times faster than human beings can. For example, many of today's personal computers can perform hundreds of millions of additions per second.

Computers process data under the control of sets of instructions called **computer programs**. These computer programs guide the computer through orderly sets of actions specified by people called computer programmers.

A computer is comprised of various devices (such as the keyboard, screen, "mouse", disks, memory, CD-ROM and processing units) that are referred to as **hardware**. The computer programs that run on a computer are referred to as **software**.

Computer Hardware

Almost every computer may be seen as being divided into six logical units. Figure 1 illustrates the main computer components.

Input Unit

This unit obtains information from various input devices and places this information at the disposal of the other units so that the information may be processed. The information is entered into computers today through keyboards and mouse devices.

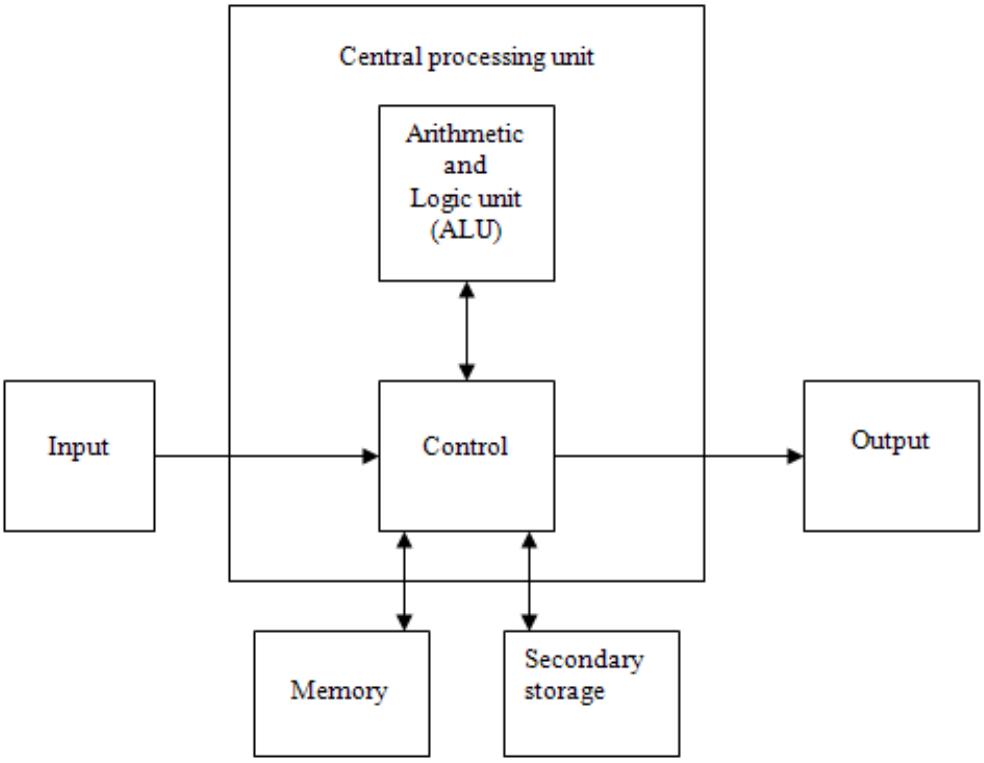
Output Unit

This unit takes information that has been processed by the computer and places it on various output devices to make information available for use outside the computer. Most output from computer today is displayed on screens, printed on paper, or used to control other devices.

Memory Unit

The memory unit stores information. Each computer contains memory of two main types: **RAM** and **ROM**.

RAM (random access memory) is volatile. Your program and data are stored in RAM when you are using the computer.



Basic hardware units of a computer

ROM (read only memory) contains fundamental instructions that cannot be lost or changed by the user. ROM is non-volatile.

Arithmetic and Logic Unit (ALU)

The ALU performs all the arithmetic and logic operations. Ex: addition, subtraction, comparison, etc.

Central Processing Unit (CPU)

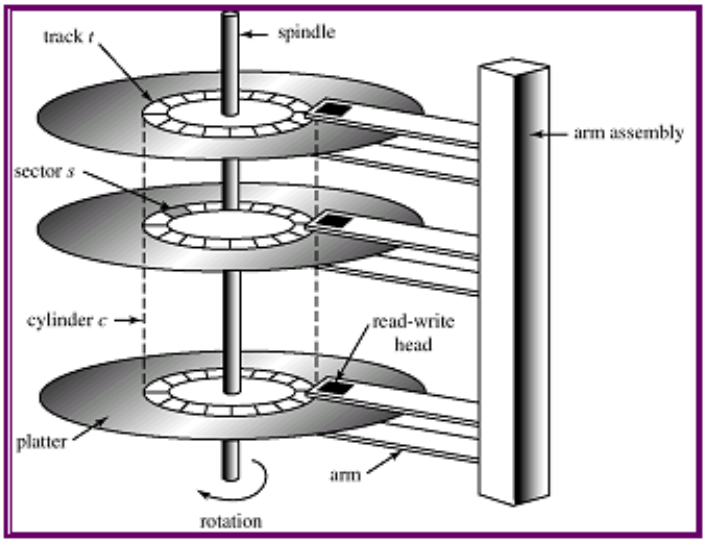
The unit supervises the overall operation of the computer. The CPU tells the input unit when information should be read into the memory unit, tell the ALU when information from the memory should be used in calculations and tells the output unit when to send information from the memory unit to certain output devices.

Secondary Storage

Secondary storage devices are used to be permanent storage area for programs and data.

Virtually all secondary storage is now done on magnetic tapes, magnetic disks and CD-ROMs.

A magnetic hard disk consists of either a single rigid platter or several platters that spin together on a common spindle. A movable access arm positions the read and write mechanisms over, but not quite touching, the recordable surfaces. Such a configuration is shown in Figure 2.



The internal structure of a magnetic hard disk drive

Computer Software

A **computer program** is a set of instructions used to operate a computer to produce a specific result.

Another term for a program or a set of programs is **software**, and we use both terms interchangeably throughout the text.

Writing computer programs is called **computer programming**.

The languages used to create computer programs are called **programming languages**.

To understand C++ programming, it is helpful to know a little background about how current programming languages evolved.

Machine and Assembly Languages

Machine languages are the lowest level of computer languages. Programs written in machine language consist of entirely of 1s and 0s.

Programs in machine language can control directly to the computer's hardware.

00101010 000000000001 000000000010

10011001 000000000010 000000000011

A machine language instruction consists of two parts: an instruction part and an address part.

The **instruction part** (opcode) is the leftmost group of bits in the instruction and tells the computer the operation to be performed.

The **address part** specifies the memory address of the data to be used in the instruction.

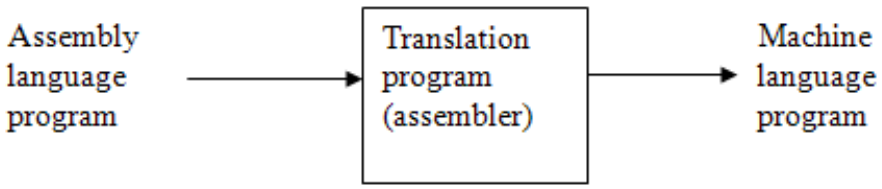
Assembly languages perform the same tasks as machine languages, but use symbolic names for opcodes and operands instead of 1s and 0s.

LOAD BASEPAY

ADD OVERPAY

STORE GROSSPAY

Since computers can only execute machine language programs, an assembly language program must be translated into a machine language program before it can be executed on a computer.



Assembly translation

Machine languages and assembly languages are called low-level languages since they are closest to computer hardware.

High-level Programming Languages

High-level programming languages create computer programs using instructions that much easier to understand than machine or assembly language instructions.

Programs written in a high-level language must be translated into a low level language using a program called a **compiler**.

A compiler translates programming code into a low-level format.

High-level languages allow programmers to write instructions that look like every English sentences and commonly used mathematical notations.

Each line in a high-level language program is called a **statement**.

Ex: Result = (First + Second)*Third.

Once a program is written in a high-level language, it must also be translated into the machine language of the computer on which it will be run. This translation can be accomplished in two ways.

When each statement in a high-level source program is translated individually and executed immediately upon translation, the programming

language used is called an **interpreted language**, and the program doing the translation is called an **interpreter**.

When all of the statements in a high-level source program are translated as a complete unit before any one statement is executed, the programming language used is called is called a **compiled language**. In this case, the program doing the translation is called a **compiler**.

Application and System Software

Two types of computer programs are: application software and system software.

Application software consists of those programs written to perform particular tasks required by the users.

System software is the collection of programs that must be available to any computer system for it to operate.

The most important system software is the **operating system**. Examples of some well-known operating systems include MS-DOS, UNIX, and MS WINDOWS. Many operating systems allow user to run multiple programs. Such operating systems are called **multitasking systems**.

Beside operating systems, language translators are also system softwares.

High-Level Programming Languages

Because of the difficulty of working with low-level languages, high-level languages were developed to make it easier to write computer programs. High level programming languages create computer programs using instructions that are much easier to understand than machine or assembly language code because you can use words that more clearly describe the task being performed. Examples of high-level languages include FORTRAN, COBOL, BASIC, PASCAL, C, C++ and JAVA.

C and C++ are two separate, but related programming languages. In the 1970s, at Bell Laboratories, Dennis Ritchie and Brian Kernighan designed the C programming language. In 1985, at Bell Laboratories, Bjarne Stroutrup created C++ based on the C language. C++ is an extension of C that adds object-oriented programming capabilities.

What is Syntax?

A programming language's **syntax** is the set of rules for writing grammatically correct language statements. In practice this means a C statement with correct syntax has a proper form specified for the compiler. As such, the compiler accepts the statement and does not generate an error message.

The C Programming Language

C was used exclusively on UNIX and on mini-computers. During the 1980s, C compilers were written for other platforms, including PCs.

To provide a level of standardization for C language, in 1989, ANSI created a standard version of C that is called ANSI C.

One main benefit of the C language is that it is much closer to assembly language other than other types of high-level programming languages.

The programs written in C often run much faster and more efficiently than programs written in other types of high-level programming language.

The C++ Programming Language

C++ is an extension of C that adds object-oriented programming capabilities. C++ is a popular programming language for writing graphical programs that run on Windows and Macintosh.

The standardized version of C++ is commonly referred to as ANSI C++.

The ANSI C and ANSI C++ standards define how C/C++ code can be written.

The ANSI standards also define **run-time libraries**, which contains useful functions, variables, constants, and other programming items that you can add to your programs.

The ANSI C++ run-time library is also called the Standard Template Library or Standard C++ Library.

Structured Programming and Object Oriented Programming

During the 1960s, many large software development effects encountered severe difficulties. Software schedules were typically late, costs greatly exceeded budgets and finished products were unreliable. People began to realize that software development was a far more complex activity than they had imagined. Research activity in the 1960s resulted in the evolution of structured programming – a discipline approach to writing programs that are clearer than unstructured programs, easier to test and debug and easier to modify. Chapter 5 discusses the principles of structured programming. Chapters 2 through 6 develop many structured programs.

One of the more tangible results of this research was the development of the Pascal programming language by Niklaus Wirth in 1971. Pascal was designed for teaching structured programming in academic environments and rapidly became the preferred programming languages in most universities.

In the 1980s, there is a revolution brewing in the software community: **object- oriented programming**. Objects are essentially reusable software components that model items in the real world. Software developers are discovering that using a modular, object-oriented design and implementation approach can make software development groups much

more productive than with previous popular programming techniques such as structured programming.

Object-oriented programming refers to the creation of reusable software objects that can be easily incorporated into another program. An **object** is programming code and data that can be treated as an individual unit or component. **Data** refers to information contained within variables, constants, or other types of storage structures. The procedures associated with an object are referred as **functions** or **methods**. Variables that are associated with an object are referred to as **properties** or **attributes**. Object-oriented programming allows programmers to use programming objects that they have written themselves or that have been written by others.

Problem Solution and Software Development

No matter what field of work you choose, you may have to solve problems. Many of these can be solved quickly and easily. Still others require considerable planning and forethought if the solution is to be appropriate and efficient.

Creating a program is no different because a program is a solution developed to solve a particular problem. As such, writing a program is almost the last step in a process of first determining what the problem is and the method that will be used to solve the problem.

One technique used by professional software developers for understanding the problem that is being solved and for creating an effective and appropriate software solution is called the software development procedure. The procedure consists of three overlapping phases

- Development and Design

- Documentation

- Maintenance

As a discipline, software engineering is concerned with creating readable, efficient, reliable, and maintainable programs and systems.

Phase I: Development and Design

The first phase consists of four steps:

1. Analyze the problem

This step is required to ensure that the problem is clearly defined and understood. The person doing the analysis has to analyze the problem requirements in order to understand what the program must do, what outputs are required and what inputs are needed. Understanding the problem is very important. Do not start to solve the problem until you have understood clearly the problem.

2. Develop a Solution

Programming is all about solving problems. In this step,

you have to develop an algorithm to solve a given problem. **Algorithm** is a sequence of steps that describes how the data are to be processed to produce the desired outputs.

An algorithm should be (at least)

–complete (i.e. cover all the parts)

–unambiguous (no doubt about what it does)

–finite (it should finish)

3. Code the solution

This step consists of translating the algorithm into a computer program using a programming language.

4. Test and correct the program

This step requires testing of the completed computer program to ensure that it does, in fact, provide a solution to the problem. Any errors that are found during the tests must be corrected.

Figure below lists the relative amount of effort that is typically expended on each of these four development and design steps in large commercial programming projects.

Step	Effort
Analyze the problem	10%
Develop a solution	20%
Code the solution	20%
Test and correct the program	50%

Four development and design steps in commercial programming projects

Phase II: Documentation

Documentation requires collecting critical documents during the analysis, design, coding, and testing.

There are five documents for every program solution:

- Program description
- Algorithm development and changes
- Well-commented program listing
- Sample test runs
- User’s manual

Phase III: Maintenance

This phase is concerned with the ongoing correction of problems, revisions to meet changing needs and the addition of new features. Maintenance is often the major effort, and the longest lasting of the three phases. While development may take days or months, maintenance may continue for years or decades.

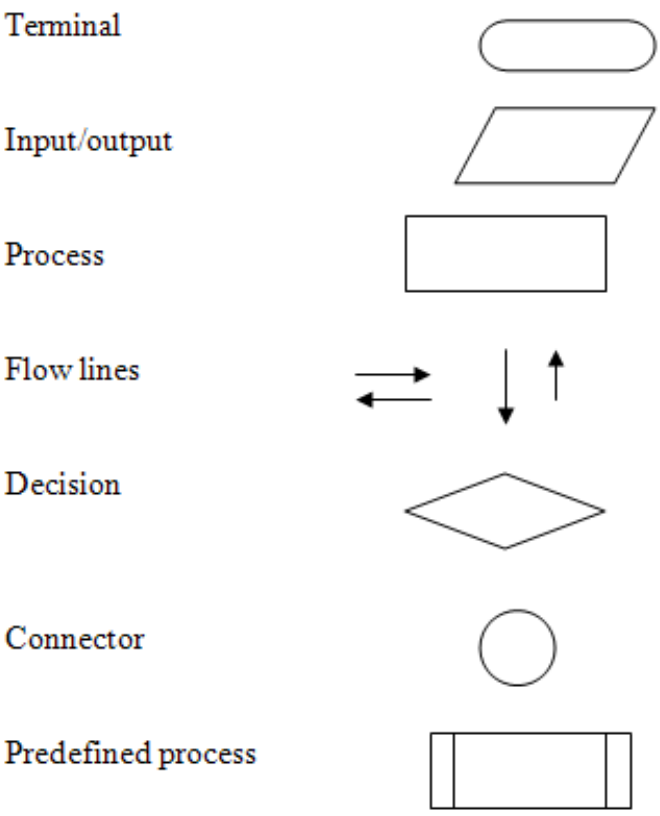
Algorithms

An algorithm is defined as a step-by-step sequence of instructions that describes how the data are to be processed to produce the desired outputs. In essence, an algorithm answers the question: “What method will you use to solve the problem?”

You can describe an algorithm by using flowchart symbols. By that way, you obtain a flowchart which is an outline of the basic structure or logic of the program.

Flowchart Symbols

To draw flowchart, we employ the symbols shown in the Figure below.



Flowchart symbols

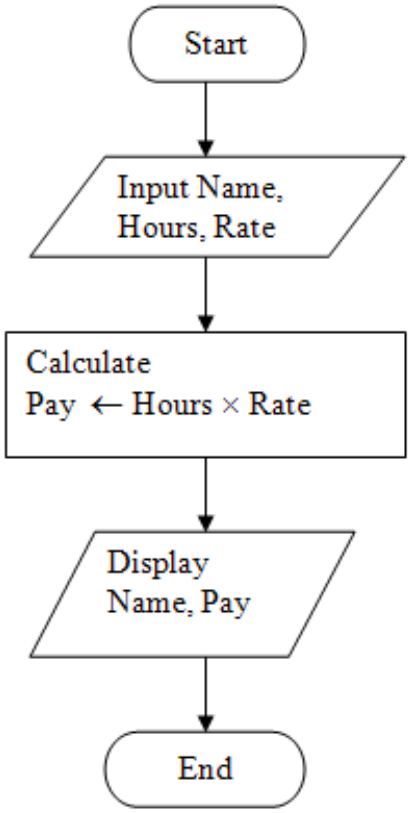
The meaning of each flowchart symbol is given as follows.

Symbol name	Description
Terminal	Indicates the beginning or end of an algorithm
Input/Output	Indicates an input or output operation
Process	Indicates computation or data manipulation
Flow lines	Connects the flowchart symbols and indicates the logic flow.
Decision	Indicates a program branch point
Connector	Indicates an entry to, or exit from another part of the flowchart or a connection point
Predefined process	Indicates a predefined process, as in calling a function

Description of flowchart symbols

To illustrate an algorithm, we consider the simple program that computes the pay of a person. The flowchart for this program is given in the Figure below.

Note: Name, Hours and Pay are **variables** in the program.



A sample flowchart

Algorithms in pseudo-code

You also can use English-like phases to describe an algorithm. In this case, the description is called pseudocode. Pseudocode is an artificial and informal language that helps programmers develop algorithms. Pseudocode has some ways to represent sequence, decision and repetition in algorithms. A carefully prepared pseudocode can be converted easily to a corresponding C++ program.

Example: The following set of instructions forms a detailed algorithm in pseudocode for calculating the payment of person.

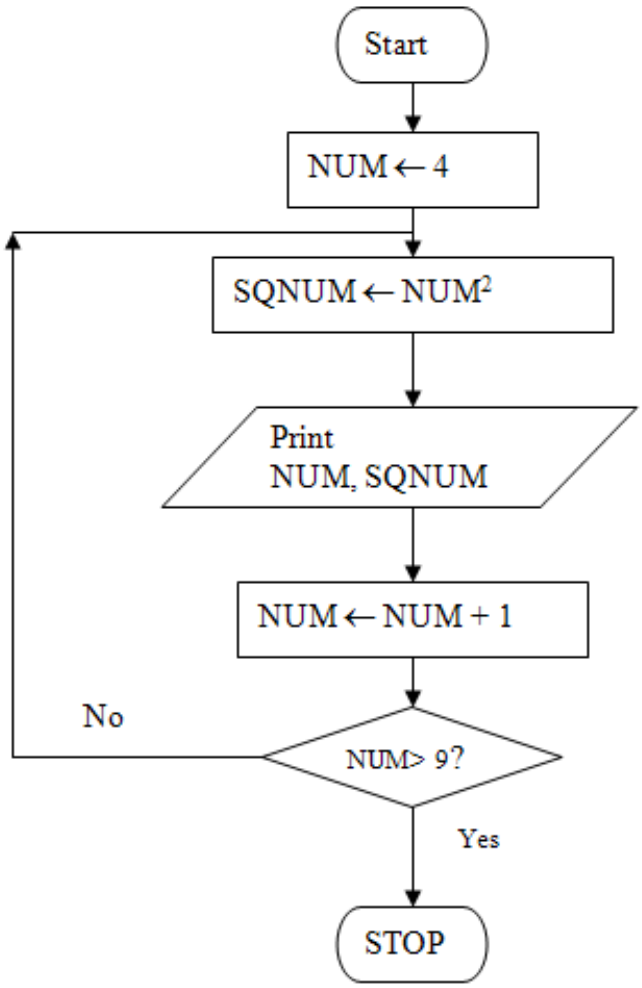
```
Input the three values into the variables Name,
Hours, Rate.

Calculate Pay = Hours  Rate.

Display Name and Pay.
```

Loops in Algorithms

Many problems require repetition capability, in which the same calculation or sequence of instructions is repeated, over and over, using different sets of data.

Example 1.1. Write a program to do the task: Print a list of the numbers from 4 to 9, next to each number, print the square of the number.

The flowchart for the algorithm that solves this problem is given in Figure below. You will see in this figure the flowchart symbol for decision and the flowline that can connect backward to represent a loop.



Flowcharts of example 1.1

Note:

1. In the flowchart, the statement

NUM = NUM + 1

means “old value of NUM + 1 becomes new value of NUM ”.

The above algorithm can be described in pseudocode as follows:

```
NUM = 4
```

```
do

SQNUM = NUM*NUM

Print NUM, SQNUM

NUM = NUM + 1

while (NUM <= 9)
```

You can compare the pseudo-code and the flowchart in Figure above to understand the meaning of the **do… while** construct used in the pseudo-code.

Flowchart versus pseudocode

Since flowcharts are inconvenient to revise, they have fallen out of favor by programmers. Nowadays, the use of pseudocode has gained increasing acceptance.

Only after an algorithm has been selected and the programmer understands the steps required can the algorithm be written using computer-language statements. The writing of an algorithm using computer-language statements is called **coding** the algorithm, which is the third step in our program development process.

Basic Elements in C++
This chapter gets you started immediately writing some simple C++ programs and helps you to understand some basic elements in any high level programming language such as data types, arithmetic operators, output statements and assignment statements.

Program Structures

Modular Programs

A large program should be organized as several interrelated segments, arranged in a logical order: The segments are called modules. A program which consists of such modules is called a modular program.

In C++, modules can be classes or functions.

We can think of a function as a program segment that transforms the data it receives into a finished result.

Each function must have a name. Names or identifiers in C++ can made up of any combination of letters, digits, or underscores selected according to the following rules:

- Identifiers must begin within an uppercase or lowercase ASCII letter or an underscore (_).
- You can use digits in an identifier, but not as the first character. You are not allowed to use special characters such as $, &, * or %.
- Reserved words cannot be used for variable names.

Examples:

DegToRadintersectaddNums

FindMax1_densityslope

Examples of invalid identifiers:

1AB3

E%6

while

Note: C++ is a case-sensitive language (i.e. upper and lower case characters are treated as different letters).

The main() function

The main() function is a special function that runs automatically when a program first executes.

All C++ programs must include one main() function. All other functions in a C++ program are executed from the main() function.

The first line of the function, in this case int main() is called a function header line.

The function header line contains three pieces of information:

1. What type of data, if any, is returned from the function.
2. The name of the function
3. What type of data, if any, is sent into the function.

```
int main()

{

program statements in here

return 0;

}
```

Note: The line

return 0;

is included at the end of every main function. C++ keyword return is one of several means we will use to exit a function. When the return statement is used at the end of main as shown here, the value 0 indicates that the program has terminates successfully.

The cout Object

The cout object is an output object that sends data given to it to the standard output display device.

To send a message to the cout object, you use the following pattern:

cout << “text”;

The insertion operator, <<, is used for sending text to an output device.

The text portion of the statement is called a text string. Text string is text that is contained within double quotation marks.

Consider the following program.

Example

```
#include <iostream.h>

int main()

{

cout << "Hello world!”;

return 0;

}
```

The **output** of the above program:

Hello world!

Preprocessor Directives

Before you can use any runtime libraries in your program, you must first add a header-file into your program, using the #include statement. A header file is a file with an extension of .h that is included as part of a program and notifies the compiler that a program uses run-time libraries.

One set of classes you will use extensively in the next few chapters is the iostream classes. The iostream classes are used for giving C++ programs input capabilities and output capabilities.

The header file for the iostream class is iostream.h.

The #include statement is one of the several preprocessor directives that are used with C++.

The preprocessor is a program that runs before the compiler. When it encounters an #include statement, the preprocessor places the entire contents of the designated file into the current file.

Preprocessor directives and include statements allow the current file to use any of the classes, functions, variables, and other code contained within the included file.

Example: To include the iostream.h file you use the following statement:

```
#include <iostream.h>
```

An i/o manipulator is a special function that can be used with an i/o statement. The endl i/o manipulator is part of the iostream classes and represents a new line character.

Example:

```
cout << “Program type: console application” << endl;

cout << “Create with: Visual C++ “<< endl;

cout << “Programmer: Don Gesselin” << endl;
```

All statements in C++ must end with a semicolon. Large statements can span multiple lines of code.

Example:

```
cout << “Program type: console application,”

<< “Create with: Visual C++ “

<< “Programmer: Don Gesselin”;
```

Comments

Comments are lines that you place in your code to contain various type of remarks. C++ support two types of comments: line and block.

C++ line comments are created by adding two slashes (//) before the text you want to use as a comment.

Block comments span multiple lines. Such comments begin with /* and end with the symbols */.

Example:

```
void main()

{

/*
```

```
This line is part of the block comment.

This line is also part of the block

comment.

*/

cout << “Line comment 1 “;

cout << “Line comment 2 “;

// This line comment takes up an entire line.

}
```

All programs should contain comments. They are

remarks, insights, wisdom in code without affecting the program. The compiler ignores comments

.

Data Types and Operators

Data Types

A data type is the specific category of information that a variable contains.

There are three basic data types used in C++: integers, floating point numbers and characters.

Integers

An integer is a positive or negative number with no decimal places.

- 259 -13 0 200

Floating Point Numbers

A floating point number contains decimal places or is written using exponential notations.

-6.16 -4.4 2.7541 10.5

Exponential notation, or scientific notation is a way of writing a very large numbers or numbers with many decimal places using a shortened format.

2.0e11 means 2*1011

C++ supports three different kinds of floating-point numbers:

- float (i.e. single precision numbers),
- double (i.e. double precision numbers)
- long double.

A double precision floating-point number can contain up to 15 significant digits.

The Character Data Type

To store text, you use the character data type. To store one character in a variable, you use the char keyword and place the character in single quotation marks.

Example:

char cLetter = ‘A’;

Escape Sequence

The combination of a backlash (\) and a special character is called an escape sequence. When this character is placed directly in front of a select group of character, it tells the compiler to escape from the way these characters would normally be interpreted.

Examples:

\n : move to the next line

\t : move to the next tab

The bool Data Type

The C++ bool type can have two states expressed by the built-in constants true (which converts to an integral one) and false (which converts to an integral zero). All three names are keywords. This data type is most useful when a program must examine a specific condition and, as a result of the condition being either true or false, take a prescribed course of action.

Arithmetic Operators

Most programs perform arithmetic calculations. Arithmetic operators are used to perform mathematical calculations, such as addition, subtraction, multiplication, and division in C++.

Operator	Description
+	Add two operands
-	Subtracts one operand from another operand
*	Multiplies one operand by another operand
/	Divides one operand by another operand
%	Divides two operands and returns the remainder

Arithmetic operators

A simple arithmetic expression consists of an arithmetic operator connecting two operands in the form:

```
operand operator operand
```

Examples:

3 + 7

18 – 3

12.62 + 9.8

12.6/2.0

Example

```
#include <iostream.h>

int main()

{

cout << "15.0 plus 2.0 equals " << (15.0 + 2.0) <<
'\n'

<< "15.0 minus 2.0 equals " << (15.0 - 2.0) <<
'\n'

<< "15.0 times 2.0 equals " << (15.0 * 2.0) <<
'\n'

<< "15.0 divided by 2.0 equals " << (15.0 / 2.0)
<< '\n';

return 0;

}
```

The **output** of the above program:

15.0 plus 2.0 equals 17

15.0 minus 2.0 equals 13

15.0 times 2.0 equals 30

15.0 divided by 2.0 equals 7.5

Integer Division

The division of two integers yields integer result. Thus the value of 15/2 is 7.

Modulus % operator produces the remainder of an integer division.

Example:

9%4 is 1

17%3 is 2

14%2 is 0

Operator Precedence and Associativity

Expressions containing multiple operators are evaluated by the priority, or precedence, of the operators.

Operator precedence defines the order in which an expression evaluates when several different operators are present. C++ have specific rules to determine the order of evaluation. The easiest to remember is that multiplication and division happen before addition and subtraction.

The following table lists both precedence and associativity of the operators.

Operator	Associativity
unary -	Right to left
* / %	Left to right
+ -	Left to right

Precedence and associativity of the operators

Example: Let us use the precedence rules to evaluate an expression containing operators of different precedence, such as 8 + 5*7%2*4. Because the multiplication and modulus operators have a higher precedence than the addition operator, these two operations are evaluated first (P2), using their left-to-right associativity, before the addition is evaluated (P3). Thus, the complete expression is evaluated as:

```
8 + 5 * 7 % 2 * 4 =
  8 + 35 % 2 * 4 =
     8 + 1 * 4   =
      8 + 4      = 12
```

Expression evaluation

Expression Types

An expression is any combination of operators and operands that can be evaluated to yield a value. An expression that contains only integer values as operands is called an integer expression, and the result of the expression

is an integer value. Similarly, an expression containing only floating-point values (single and double precision) as operands is called a floating-point expression, and the result of the expression is a floating point value (the term real expression is also used).

Variables and Declaration Statements

One of the most important aspects of programming is storing and manipulating the values stored in **variables**. A variable is simply a name chosen by the programmer that is used to refer to computer storage locations. The term **variable** is used because the value stored in the variable can change, or vary.

Variable names are also selected according to the rules of identifiers:

- Identifiers must begin with an uppercase or lowercase ASCII letter or an underscore (_).
- You can use digits in an identifier, but not as the first character. You are not allowed to use special characters such as $, &, * or %.
- Reserved words cannot be used for variable names.

Example: Some valid identifiers

my_variable

Temperature

x1

x2

_my_variable

Some invalid identifiers are as follows:

%x1%my_var@x2

We should always give variables meaningful names, from which a reader might be able to make a reasonable guess at their purpose. We may use comments if further clarification is necessary.

Declaration Statements

Naming a variable and specifying the data type that can be stored in it is accomplished using **declaration statement**. A declaration statement in C++ programs has the following syntax:

```
type name;
```

The type portion refers to the data type of the variable.

The data type determines the type of information that can be stored in the variable.

Example:

```
int sum;
```

```
long datenem;
```

```
double secnum;
```

Note:

1. A variable must be declared before it can be used.
2. Declaration statements can also be used to store an initial value into declared variables.

Example:

```
int num = 15;
```

```
float grade1 = 87.0;
```

Variable declarations are just the instructions that tell the compiler to allocate memory locations for the variables to be used in a program.

A variable declaration creates a memory location but it is undefined to start with, that means it's empty.

Example

```
#include <iostream.h>

int main()

{

float price1 = 85.5;

float price2 = 97.0;

float total, average;

total = price1 + price2;

average = total/2.0; // divide the total by 2.0

cout << "The average price is " << average << endl;

return 0;

}
```

The **output** of the above program:

The average price is 91.25

Let notice the two statements in the above program:

```
total = price1 + price2;
```

```
average = total/2.0;
```

Each of these statements is called an assignment statement because it tells the computer to assign (store) a value into a variable. Assignment statements always have an equal (=) sign and one variable name on the left of this sign. The value on the right of the equal sign is assigned to the variable on the left of the equal sign.

Display a Variable's Address

Every variable has three major items associated with it: its data type, its actual value stored in the variable and the address of the variable. The value stored in the variable is referred to as the variable's contents, while the address of the first memory location used for the variable constitutes its address.

To see the address of a variable, we can use **address operator**, &, which means "the address of ". For example, &num means the address of num.

Example

```
#include <iostream.h>

int main()

{

int a;

a = 22;

cout << "The value stored in a is " << a << endl;

cout << "The address of a = " << &a << endl;

return 0;
```

```
}
```

The **output** of the above program:

The value stored in a is 22

The address of a = 0x0065FDF4

The display of addresses is in hexadecimal notation.

Integer Quantifiers

Portable languages like C++ must have flexible data type sizes. Different applications might need integers of different sizes. C++ provides **long integer**, **short integer**, and **unsigned integer** data types. These three additional integer data types are obtained by adding the quantifier **long**, **short** or **unsigned** to the normal integer declaration statements.

Example:

```
long int days;
```

```
unsigned int num_of_days;
```

The reserved words **unsigned int** are used to specify an integer that can only store nonnegative numbers.

The **signed** and **unsigned** quantifiers tell the compiler how to use the sign bit with integral types and characters (floating-point numbers always contain a sign). An unsigned number does not keep track of the sign and thus has an extra bit available, so it can store positive numbers twice as large as the positive numbers that can be stored in a signed number.

Data type	Storage	Number Range
short int	2 bytes	-32768 to 32767
unsigned int	2 bytes	0 to 65535
long int	4 bytes	-2,147,483,648 to -2,147,483,647

Integer types with quantifiers

When you are modifying an **int** with **short** or long, the keyword **int** is optional.

Now all the built-in data types provide by C++ are given in the following list, ordered descendingly by the size of the data types.

Data types

---

long double

double

float

unsigned long

long int

unsigned int

int

short in

char

Data Type Conversions

An expression containing both integer and floating point operands is called a mixed mode expression.

Example:

```
int a;

float x = 2.5;

a = x + 6; // x + 6 is a mixed mode expression
```

Note: We should **avoid** mixed-mode expression.

Examples:

```
char Ch;

int In1 = 129, In2, In3;

double Real1 = 12.34, Real2;
```

What happens with the following mixed mode assignments?

```
Ch = In1/2 + 1; // Right side = 65; assigns ‘A’ to Ch

In2 = Ch + 1; // Right side = 66; assigns 66 to In2

Real2 = In1/2; // Right side = 64; assigns 64.0 to Real2

In3 = Real1/2.0 // Right side = 6.17; truncates this value and assigns 6 to In3
```

The general rules for converting integer and floating point operands in mixed mode arithmetic expressions were presented as follows:

1. If both operands are either character or integer operands:

    - when both operands are character, short or integer data types, the result of the expression is an integer value.
    - when one of the operand is a long integer, the result is a long integer, unless one of the operand is an unsigned integer. In the later case, the other operand is converted to an unsigned integer value and the resulting value of the expression is an unsigned value.

2. If any one operand is a floating point value:

    - when one or both operands are floats, the result of the operation is a float value;
    - when one or both operands are doubles, the result of the operation is a double value;
    - when one or both operands are long doubles, the result of the operation is a long double value;

Notice that converting values to lower types can result in incorrect values. For example, the floating point value 4.5 gives the value 4 when it is converted to an integer value. The following table lists the built-in data types in order from “highest type” to “lowest type”.

Determining Storage Size

C++ provides an operator for determining the amount of storage your compiler allocates for each data type. This operator, called the sizeof() operator.

Example:

sizeof(num1)

sizeof(int)

sizeof(float)

The item in parentheses can be a variable or a data type.

Example

```
// Demonstrating the sizeof operator
#include <iostream.h>
int main()
{
char c;
short s;
int i;
long l;
float f;
double d;
long double ld;
cout << "sizeof c = " << sizeof(c)
<< "\tsizeof(char) = " << sizeof( char )
<< "\nsizeof s = " << sizeof(s)
<< "\tsizeof(short) = " << sizeof( short )
<< "\nsizeof i = " << sizeof (i)
<< "\tsizeof(int) = " << sizeof( int )
<< "\nsizeof l = " << sizeof(l)
```

```
<< "\tsizeof(long) = " << sizeof( long )
<< "\nsizeof f = " << sizeof (f)
<< "\tsizeof(float) = " << sizeof(float)
<< "\nsizeof d = " << sizeof (d)
<< "\tsizeof(double) = " << sizeof(double)
<< endl;
return 0;
}
```

The **output** of the above program:

sizeof c = 1sizeof(char) = 1

sizeof s = 2sizeof(short) = 2

sizeof i = 4sizeof(int) = 4

sizeof l = 4sizeof(long) = 4

sizeof f = 4sizeof(float) = 4

sizeof d = 8sizeof(double) = 8

Focus on Problem Solving

In this section, the software development procedure presented in the previous chapter is applied to a specific programming problem. This procedure can be applied to any programming problem to produce a completed program and forms the foundation for all programs developed in this text.

Problem: Telephone Switching Networks

A directly connected telephone network is one in which all telephones in the network are connected directly and do not require a central switching station to establish calls between two telephones.

The number of direct lines needed to maintain a directly connected network for n telephones is given by the formula:

lines = n(n-1)/2

For example, directly connecting four telephones requires six individual lines.

Using the formula, write a C++ program that determines the number of direct lines for 100 telephones and the additional lines required if 10 new telephones were added to the network. Use our top-down software development procedure.

Step 1: Analyze the Problem

For this program, two outputs are required: the number of direct lines required for 100 telephones and the additional lines needed when 10 new telephones are added to the existing network. The input item required for this problem is the number of telephones, which is denoted as n in the formula.

Step 2: Develop a Solution

The first output is easily obtained using the given formula lines = n(n-1)/2. Although there is no formula given for additional lines, we can use the given formula to determine the total number of lines needed for 110 subscribers. Subtracting the number of lines for 100 subscribers from the number of lines needed for 110 subscribers then yields the number of

additional lines required. Thus, the complete algorithm for our program, in pseudocode, is:

```
Calculate the number of direct lines for 100
subscribers.

Calculate the number of direct lines for 110
subscribers.

Calculate the additional lined needed, which is
the

difference between the second and the first
calculation.

Display the number of lines for 100 subscribers.

Display the additional lines needed.
```

Step 3: Code the Solution

The following program provides the necessary code.

```
#include <iostream.h>

int main()

{

int numin1, numin2, lines1, lines2;

numin1 = 100;

numin2 = 110;

lines1 = numin1*(numin1 – 1)/2;
```

```
lines2 = numin2*(numin2 – 1)/2;

cout << “The number of initial lines is “ <<
lines1 << “.\n”;

cout << “There are “ << lines2 – lines1

<< “ additional lines needed.\n”;

return 0;

}
```

Step 4: Test and Correct the Program

The following **output** is produced when the program is compiled and executed:

The number of initial lines is 4950.

There are 1045 additional lines needed.

Because the displayed value agrees with the hand calculation, we have established a degree of confidence in the program.

Completing the Basics
In the last chapter, we explored how results are displayed and how numerical data are stored and processed using variables and assignment statements. In this chapter, we complete our introduction to the basics of C++ by presenting additional processing and input capabilities.

Assignment Operators

Assignment operator (=) is used for assignment a value to a variable and for performing computations.

Assignment statement has the syntax:

```
variable = expression;
```

Expression is any combination of constants, variables, and function calls that can be evaluated to yield a result.

Example:

```
length = 25;
```

```
cMyCar = “Mercedes”;
```

```
sum = 3 + 7;
```

```
newtotal = 18.3*amount;
```

```
slope = (y2 – y1)/(x2 – x1);
```

The order of events when the computer executes an assignment statement is

- Evaluate the expression on the right hand side of the assignment operator.

- Store the resultant value of the expression in the variable on the left hand side of the assignment operator.

Note:

1. It’s important to note that the equal sign in C++ does not have the same meaning as an equal sign in mathematics.

2. Each time a new value is stored in a variable, the old one is overwritten.

Example

```
// This program calculates the volume of a cylinder,

// given its radius and height

#include <iostream.h>

int main()

{

float radius, height, volume;

radius = 2.5;

height = 16.0;

volume = 3.1416 * radius * radius * height;

cout << "The volume of the cylinder is " << volume << endl;

return 0;

}
```

The **output** of the above program:

The volume of the cylinder is 314.16

We can write **multiple assignments**, such as a = b = c = 25;. Because the assignment operator has a right-to-left associativity, the final evaluation

proceeds in the sequence

c = 25;

b = 25;

c = 25;

Data Type Conversion across Assignment Operator

Note that data type conversion can take place across assignment operators, that is, the value of the expression on the right side of the assignment operator is converted to the data type of the variable to the left side of the assignment operator.

For example, if temp is an integer variable, the assignment temp = 25.89 causes the integer value 25 to be stored in the integer variable temp.

Assignment Variations

C++ also use a shorthand notation to perform an operation and an assignment at the same time. This is denoted by an operator followed by an equal sign. For example, to add 4 to the variable x and assign x to the result, you say: x += 4. Figure 1 illustrates assignment operator and all assignment variations.

Operator	Example	Meaning
=	iNum1 = iNum2	
+=	iNum1 += iNum2	iNum1 = iNum1 + iNum2
-=	iNum1 -= iNum2	iNum1 = iNum1 - iNum2
*=	iNum1 *= iNum2	iNum1 = iNum1 * iNum2
/=	iNum1 /= iNum2	iNum1 = iNum1 / iNum2
%=	iNum1 %= iNum2	iNum1 = iNum1 % iNum2

Variations of assignment

Assignment statements such as sum += 10 or its equivalent, sum = sum + 10, are very common in C++ programming.

Increment and decrement operators

For the special case in which a variable is either increased or decreased by 1, C++ provides two unary operators: **increment operator** and **decrement operator**.

Operator	Description
++	Increase an operand by a value of one
--	Decrease an operand by a value of one

Increment operator and decrement operator

The increment (++) and decrement (--) unary operators can be used as prefix or postfix operators to increase or decrease value.

A **prefix** operator is placed **before** a variable and returns the value of the operand after the operation is performed.

A **postfix** operator is placed **after** a variable and returns the value of the operand before the operation is performed.

Prefix and postfix operators have different effects when used in a statement

b = ++a;

will first increase the value of a to 6, and then assign that new value to b. It is equivalent to

a = a +1;

b = a;

On the other hand, execution of the statement

b = a++;

will first assign the value of 5 to b, and then increase the value of a to 6. It is now equivalent to

b = a;

a = a + 1;

The decrement operators are used in a similar way.

Example

```
// Preincrementing and postincrementing

#include <iostream.h>

int main()

{
```

```
int c;
c = 5;
cout << c << endl; // print 5
cout << c++ << endl; // print 5 then postincrement
cout << c << endl << endl; // print 6
c = 5;
cout << c << endl; // print 5
cout << ++c << endl; // preincrement then print 6
cout << c << endl; // print 6
return 0; // successful termination
}
```

The **output** of the above program:

5

5

6

5

6

6

Formatting Number for Program Output

Besides displaying correct results, a program should present its results attractively with good formats.

Stream Manipulators

Stream manipulator functions are special stream functions that change certain characteristics of the input and output.

The main advantage of using manipulator functions is they facilitate the formatting of the input and output streams.

- setw(): The setw() stands for set width. This manipulator is used to specify the minimum number of the character positions on the output field a variable will consume.

- setprecision(): The setprecision() is used to control the number of digits of an output stream display of a floating point value. Setprecision(2) means 2 digits of precision to the right of the decimal point.

To carry out the operations of these manipulators in a user program, you must include the header file <iomanip.h>.

Example

```
#include <iostream.h>

#include <iomanip.h>

int main()

{

cout << setw(3) << 6 << endl

<< setw(3) << 18 << endl

<< setw(3) << 124 << endl
```

```
<< "---\n"

<< (6+18+124) << endl;

return 0;

}
```

The **output** of the above program:

```
  6
 18
124
---
148
```

Output of above program

Example:

```
cout << “|” << setw(10)

<< setioflags(ios::fixed)<< setprecision(3) <<
25.67<<”|”;
```

causes the printout

```
|   25.670|
```

Output of above code segment

- setiosflags: This manipulator is used to control different input and output settings.

setioflag(ios::fixed) means the output field will use conventional fixed-point decimal notation.

setiosflag(ios::showpoint) means the output field will show the decimal point for floating point number.

setiosflag(ios::scientific) means the output field will use exponential notation.

Note: In the absence of the ios::fixed flag, a floating point number is displayed with a default of 6 significant digits. If the integral part of the number requires more than 6 digits, the display will be in exponential notation.

Below are some other format flags for use with setiosflags().

Flag	Meaning
ios::showpos	display a leading + sign when the number is positive.
ios::dec	display in decimal format
ios::oct	display in octal format
ios::left	left-justify output
ios::right	right-justify output
ios::hex	display in hexadecimal format

Flags for use with setiosflags()

Example

```
// This program will illustrate output conversions

#include <iostream.h>

#include <iomanip.h>

int main()

{

cout << "The decimal (base 10) value of 15 is " <<
15 << endl

<< "The octal (base 8) value of 15 is "

<< setiosflags(ios::oct) << 15 << endl

<< "The hexadecimal (base 16) value of 15 is "

<< setiosflags(ios::hex) << 15 << endl;

return 0;
```

```
}
```

The **output** of the above program:

The decimal (base 10) value of 15 is 15

The octal (base 8) value of 15 is 17

The hexadecimal (base 16) value of 15 is f

To designate an octal integer constant, the number must have a leading **0**. Hexadecimal number are denoted using a leading **0x**.

Example

```
// Octal and hexadecimal integer constant

#include <iostream.h>

int main()

{

cout << "The decimal value of 025 is " << 025 <<
endl

<< "The decimal value of 0x37 is "<< 0x37 << endl;

return 0;

}
```

The **output** of the above program:

The decimal value of 025 is 21

The decimal value of 0x37 is 55

Using Mathematical Library Functions

Although addition, subtraction, multiplication and division are easily accomplished using C++'s arithmetic operators, no such operators exist for finding the square root of a number or determining trigonometric values. To facilitate such calculations, C++ provides standard library functions that can be included in a program.

Functions are normally called by writing the name of the function, followed by a left parenthesis, followed by the **argument** (or a comma-separated list of arguments) of the function, followed by a right parenthesis. For example, a programmer desiring to calculate and print the square root of 900.0 might write:

```
cout << sqrt(900.0);
```

When this statement is executed, the math library function **sqrt** is called to calculate the square root of the number contained in the parentheses (900.0). The number 900.0 is the argument of the **sqrt** function. The preceding statement would print 30. The **sqrt** function takes an argument of type double and returns a result of type double.

If your program uses mathematic function sqrt(), it should have the preprocessor command **#include<math.h>** in the beginning of the program. This makes a mathematical library accessible. Below are some commonly used mathematical functions provided in C++.

Function Name	Description	Return Value
abs(a)	Absolute value	Same data type as argument
log(a)	Natural logarithm	double
sin(a)	sine of a (a in radians)	double
cos(a)	cosine of a (a in radians)	double
tan(a)	tangent of a (a in radians)	double
log10(a)	common log (base 10) of a	double
pow(a1,a2)	a1 raised to the a2 power	double
exp(a)	e^a	double
sqrt(a)	square root of a	double

Mathematical functions

Except abs(a), the functions all take an argument of type double and return a value of type double.

Example

```
// this program calculates the area of a triangle
// given its three sides
#include <iostream.h>
#include <math.h>
int main()
{
double a,b,c, s;
a = 3;
b = 4;
```

```
c = 5;

s = (a+b+c)/2.0;

area = sqrt(s*(s-a)*(s-b)*(s-c));

cout << "The area of the triangle = " << area << endl;

return 0;

}
```

The **output** of the above program:

The area of the triangle = 6.0

Casts

We have already seen the conversion of an operand's data type within mixed-mode expressions and across assignment operators. In addition to these implicit data type conversions that are automatically made within mixed-mode expressions and assignment, C++ also provides for explicit user-specified type conversion. This method of type conversion is called **casting**. The word **cast** is used in the sense of "casting into a mold."

Casting or **type casting**, copies the value contained in a variable of one data type into a variable of another data type.

The C++ syntax for casting variables is

```
variable = new_type( old_variable);
```

where the new_type portion is the keyword representing the type to which you want to cast the variable.

Example:

```
int iNum = 100;
```

```
float fNum;
```

```
fNum = float (inum);
```

If you do not explicitly cast a variable of one data type to another data type, then C++ will try to automatically perform the cast for you.

Program Input Using the CIN Object

So far, our programs have been limited in the sense that all their data must be defined within the program source code.

We will now learn how to write programs which enable data to be entered via the keyboard, while the program is running.

Such programs can be made to operate upon different data every time they run, making them much more flexible and useful.

Standard Input Stream

The **cin** object reads in information from the keyboard via the standard input stream.

The **extraction operator** (>>) retrieves information from the input stream.

When the statement cin >> num1; is encountered, the computer stops program execution and accepts data from the keyboard. The user responds by typing an integer (or float) and then pressing the Enter key (sometimes called the Return key) to send the number to the computer. When a data item is typed, the cin object stores the integer (or float) into the variable listed after the >> operator.

The **cin** and **cout** stream objects facilitate interaction between the user and the computer. Because this interaction resembles a dialogue, it is often

called conversational computing or interactive computing.

Example

```
#include <iostream.h>

int main()

{

int integer1, integer2, sum; // declaration

cout << "Enter first integer\n"; // prompt

cin >> integer1; // read an integer

cout << "Enter second integer\n"; // prompt

cin >> integer2; // read an integer

sum = integer1 + integer2;

cout << "Sum is " << sum << endl;

return 0; // indicate that program ended
successfully

}
```

The **output** of the above program:

Enter the first integer

45

Enter the second integer

72

Sum is 117

Example

```
#include <iostream.h>

int main()

{

int integer1, integer2, sum; // declaration

cout << "Enter two integers\n"; // prompt

cin >> integer1 >> integer2; // read two integers

sum = integer1 + integer2;

cout << "Sum is " << sum << endl;

return 0;

}
```

The **output** of the above program:

Enter two integers

45 72

Sum is 117

Symbolic Constants

C++ introduces the concept of a named constant that is just like a variable, except that its value cannot be changed. The qualifier **const** tells the compiler that a name represents a constant. Any data type, built-in or user-

defined, may be defined as const. If you define something as const and then attempt to modify it, the compiler will generate an error.

To define a constant in a program, we use const declaration qualifier.

Example:

```
const float PI = 3.1416;

const double SALESTAX = 0.05;

const int MAXNUM = 100;
```

Once declared, a constant can be used in any C++ statement in place of the number it represents.

Example

```
// this program calculates the circumference of a
circle

// given its radius

#include <iostream.h>

int main()

{

const float PI = 3.1416

float radius, circumference;

radius = 2.0;

circumference = 2.0 * PI * radius;

cout << "The circumference of the circle is "
```

```
<< circumference << endl;

return 0;

}
```

The **output** of the above program:

The circumference of the circle is 12.5664

Focus on Problem Solving

In this section, we present a programming problem to further illustrate both the use of cin statements to accept user input data and the use of library functions for performing calculations.

Problem: Approximating the Exponential Function

The exponential function e^x, where e is known as Euler's number (and has the value 2.718281828459045…) appears many times in descriptions of natural phenomena. The value of e^x can be approximated using the following series:

1 + x/1 + x^2/2 + x^3/6 + x^4/24 + x^5/120 + x^6/720 + …

Using this polynomial as a base, assume you are given the following assignment: Write a program that approximates e raised to a user input value of x using the first four terms of this series. For each approximation, display the value calculated by C++'s exponential function, exp(), the approximate value, and the absolute difference between the two. Make sure to verify your program using a hand calculation. Once the verification is complete, use the program to approximate e^4.

Using the top-down development procedure, we perform the following steps.

Step 1: Analyze the Problem

The statement of the problem specifies that four approximations are to be made using one, two, three, and four terms of the approximating polynomial, respectively. For each approximation, three output values are required: the value of produced by the exponential function, the approximate value, and the absolute difference between the two values. The structure of the required output display is as below (in symbolic form).

e^x	Approximation	Difference
library function value	1st approximate value	1st difference
library function value	2nd approximate value	2nd difference
library function value	3rd approximate value	3rd difference
library function value	4th approximate value	4th difference

Realizing the each line in the display can only be produced by executing a cout statement, it should be clear that four such statements must be executed. Additionally, since each output line contains three computed values, each cout statement will have three items in its expression list.

The only input to the program consists of the value of x. This will, of course, require a single prompt and a cin statement to input the necessary value.

Step 2: Develop a Solution

Before any output items can be calculated, the program needs to prompt the user for a value of x and then accept the entered value. The output display consists of two title lines followed by four lines of calculated data. The title lines can be produced using two cout statements. Now let’s see how the data being displayed are produced.

The first item on the first data output line illustrated in Table 3.4 can be obtained using the exp() function. The second item on this line, the

approximation to ex , can be obtained by using the first term in the polynomial that was given in the program specification. Finally, the third item on the line can be calculated by using the abs() function on the difference between the first two lines. When all of these items are calculated, a single cout statement can be used to display the three results on the same line.

The second output line illustrated in Table 3.4 displays the same type of items as the first line, except that the approximation to ex requires the two of two terms of the approximating polynomial. Notice also that the first item on the second line, the value obtained by the exp() function, is the same as the first item on the first line. This means that this item does not have to recalculated; the value calculated for the first line can simply be displayed a second line. Once the data for the second line have been calculated, a single cout statement can again be used to display the required values.

Finally, only the second and third items on the last two output lines shown in Figure 1 need to be recalculated because the first item on these lines is the same as previously calculated for the first line. Thus, for this problem, the complete algorithm described in pseudocode is:

```
Display a prompt for the input value of x.

Read the input value.

Display the heading lines.

Calculate the exponential value of x using the
exp() function.

Calculate the first approximation.

Calculate the first difference.

Print the first output line.

Calculate the second approximation.
```

```
Calculate the second difference.

Print the second output line.

Calculate the third approximation.

Calculate the third difference.

Print the third output line.

Calculate the fourth approximation.

Calculate the fourth difference.

Print the fourth output line.
```

To ensure that we understand the processing used in the algorithm, we will do a hand calculation. The result of this calculation can then be used to verify the result produced by the program that we write. For test purposes, we use a value of 2 for x, which causes the following approximations:

Using the first term of the polynomial, the approximation is

e^2 = 1

Using the first two terms of the polynomial, the approximation is

e^2 = 1 + 2/1 = 3

Using the first three terms of the polynomial, the approximation is

e^2 = 3 + 2^2/2 = 5

Using the first four terms of the polynomial, the approximation is

e^2 = 5 + 2^3/6 = 6.3333

Notice that the first four terms of the polynomial, it was not necessary to recalculate the value of the first three terms; instead, we used the previously

calculated value.

Step 3: Code the Algorithm

The following program represents a description of the selected algorithm in C++.

```
// This program approximates the function e raised to the x power

// using one, two, three, and four terms of an approximating polynomial.

#include <iostream.h>

#include <iomanip.h>

#include <math.h>

int main()

{

double x, funcValue, approx, difference;

cout << “\n Enter a value of x: “;

cin >> x;

// print two title lines

cout << “ e to the x Approximation Difference\n”

cout << “------------ ---------------------- ----------\n”;

funcValue = exp(x);
```

```
// calculate the first approximation

approx = 1;

difference = abs(funcValue – approx);

cout << setw(10) << setiosflags(iso::showpoint) <<
funcValue

<< setw(18) << approx

<< setw(18) << difference << endl;

// calculate the first approximation

approx = 1;

difference = abs(funcValue – approx);

cout << setw(10) << setiosflags(iso::showpoint) <<
funcValue

<< setw(18) << approx

<< setw(18) << difference << endl;

// calculate the second approximation

approx = approx + x;

difference = abs(funcValue – approx);

cout << setw(10) << setiosflags(iso::showpoint) <<
funcValue

<< setw(18) << approx

<< setw(18) << difference << endl;
```

```
// calculate the third approximation

approx = approx + pow(x,2)/2.0;

difference = abs(funcValue – approx);

cout << setw(10) << setiosflags(iso::showpoint) <<
funcValue

<< setw(18) << approx

<< setw(18) << difference << endl;

// calculate the fourth approximation

approx = approx + pow(x,3)/6.0;

difference = abs(funcValue – approx);

cout << setw(10) << setiosflags(iso::showpoint) <<
funcValue

<< setw(18) << approx

<< setw(18) << difference << endl;

return 0;

}
```

In reviewing the program, notice that the input value of x is obtained first. The two title lines are then printed prior to any calculations being made. The value of ex is then computed using the exp() library function and assigned to the variable funcValue. This assignment permits this value to be used in the four difference calculations and displayed four times without the need for recalculation.

Since the approximation to the ex is “built up” using more and more terms of the approximating polynomial, only the new term for each approximation is calculated and added to the previous approximation. Finally, to permit the same variables to be reused, the values in them are immediately printed before the next approximation is made.

Step 4: Test and Correct the Program

The following is the sample run produced by the above program is:

```
Enter a value of x: 2
        e to the x      Approximation       Difference
        ----------      -----------------   ------------
        7.389056        1.000000            6.389056
        7.389056        3.000000            4.389056
        7.389056        5.000000            2.389056
        7.389056        6.333333            1.055723
```

A sample run produced by the above program

The first two columns of output data produced by the sample run agree with our hand calculation. A hand check of the last column verifies that it also correctly contains the difference in values between the first two columns.

Selection Statements
The flow of control refers to the order in which a program's statements are executed. Unless directed otherwise, the normal flow of control for all programs is sequential. This means that statements are executed in sequence, one after another, in the order in which they are placed within the program. Selection, repetition and function invocation structures permit the flow of control to be altered in defined ways. This chapter introduces to you C++'s selection statements. Repetition and invocation techniques are presented in the next two chapters.

Selection Criteria

Relational Operators

Relational operators are used to compare two operands for equality and to determine if one numeric value is greater than another. A Boolean value of true or false is returned after two operands are compared. The list of relational operators is given below.

Operator	Description
==	equal
!=	not equal
>	greater than
<	less than
<=	less than or equal
>=	greater than or equal

Relational operators

Example:

```
a = = b
```

```
(a*b) != c
```

```
s == 'y'
```

```
x<= 4
```

The value of a relational expression such as a > 40 depends on the value stored in the variable a.

Logical Operators

Logical operators, AND, OR and NOT are used for creating more complex conditions. Like relational operators, a Boolean value of true or false is returned after the logical operation is executed.

When the AND operator, &&, is used with two simple expressions, the condition is true only if both individual expressions are true by themselves.

The logical OR operator, ||, is also applied with two expressions. When using the OR operator, the condition is satisfied if either one or both of the two expressions are true.

The NOT operator,!, is used to change an expression to its opposite state; thus, if the expression has any nonzero value (true),! expression produces a zero value (false). If an expression is false,! expression is true (and evaluates to false).

Example:

```
(age > 40) && (term < 10)
```

```
(age > 40) || (term < 10)
```

```
!(age > 40)
```

```
( i==j) || (a < b) || complete
```

The relational and logical operators have a hierarchy of execution similar to the arithmetic operators. The following table lists the precedence of these operators in relation to the other operators we have used.

Level	Operator	Associativity
1.	! unary - ++ --	Right to left
2.	* / %	Left to right
3.	+ -	Left to right
4.	< <= > >=	Left to right
5.	== !=	Left to right
6.	&&	Left to right
7.	||	Left to right
8.	= += -= *= /=	Right to left

Associativity of operators

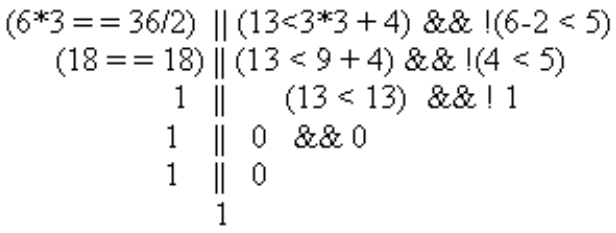
Example: Assume the following declarations:

```
char key = 'm';

int i = 5, j = 7, k = 12;

double x = 22.5;
```

Expression	Equivalent expression	Value	Interpretation
i + 2 == k-1	(i + 2) == (k – 1)	0	false
'a' + 1 == 'b'	('a' + 1) == 'b'	1	true
25 >= x + 1.0	25 >= (x + 1.0)	1	true
key – 1 > 20	(key – 1) > 20	0	false

Results of expressions

By evaluating the expressions within parentheses first, the following compound condition is evaluated as:



Evaluation process

The bool Data Type

As specified by the ANSO/ISO standard, C++ has a built-in Boolean data type, bool, containing the two values true and false. As currently implemented, the actual values represented by the bool values, true and false, are the integer values 1 and 0, respectively. For example, consider the following program, which declares two Boolean variables:

Example

```
#include<iostream.h>

int main()

{

bool t1, t2;

t1 = true;
```

```
t2 = false;

cout << “The value of t1 is “<< t1

<< “\n and the value of t2 is “<< t2 << endl;

return 0;

}
```

The **output** of the program is:

The value of t1 is 1

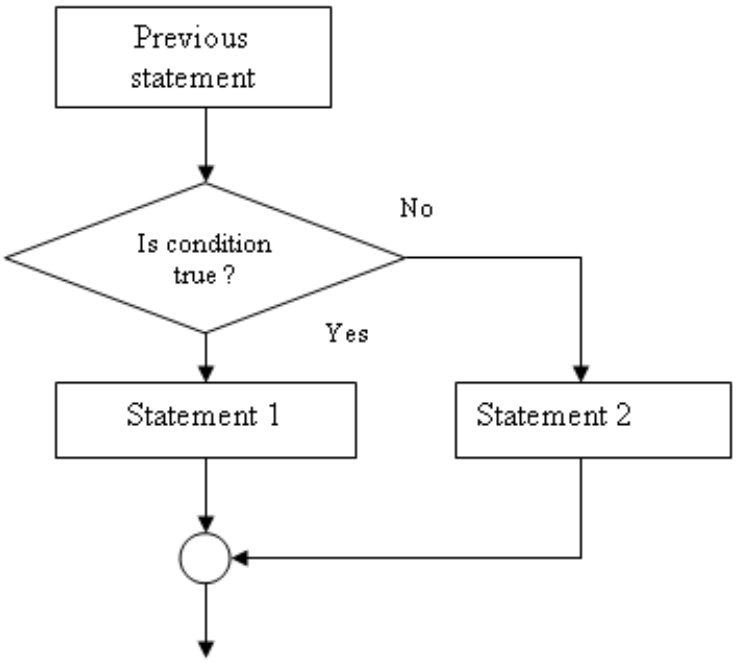
and the value of t2 is 0

The If-Else Statement

The **if-else** statement directs the computer to select a sequence of one or more statements based on the result of a comparison.

The syntax for an if .. else statement:

```
if (conditional expression) {

statements;

}

else {

statements;

}
```



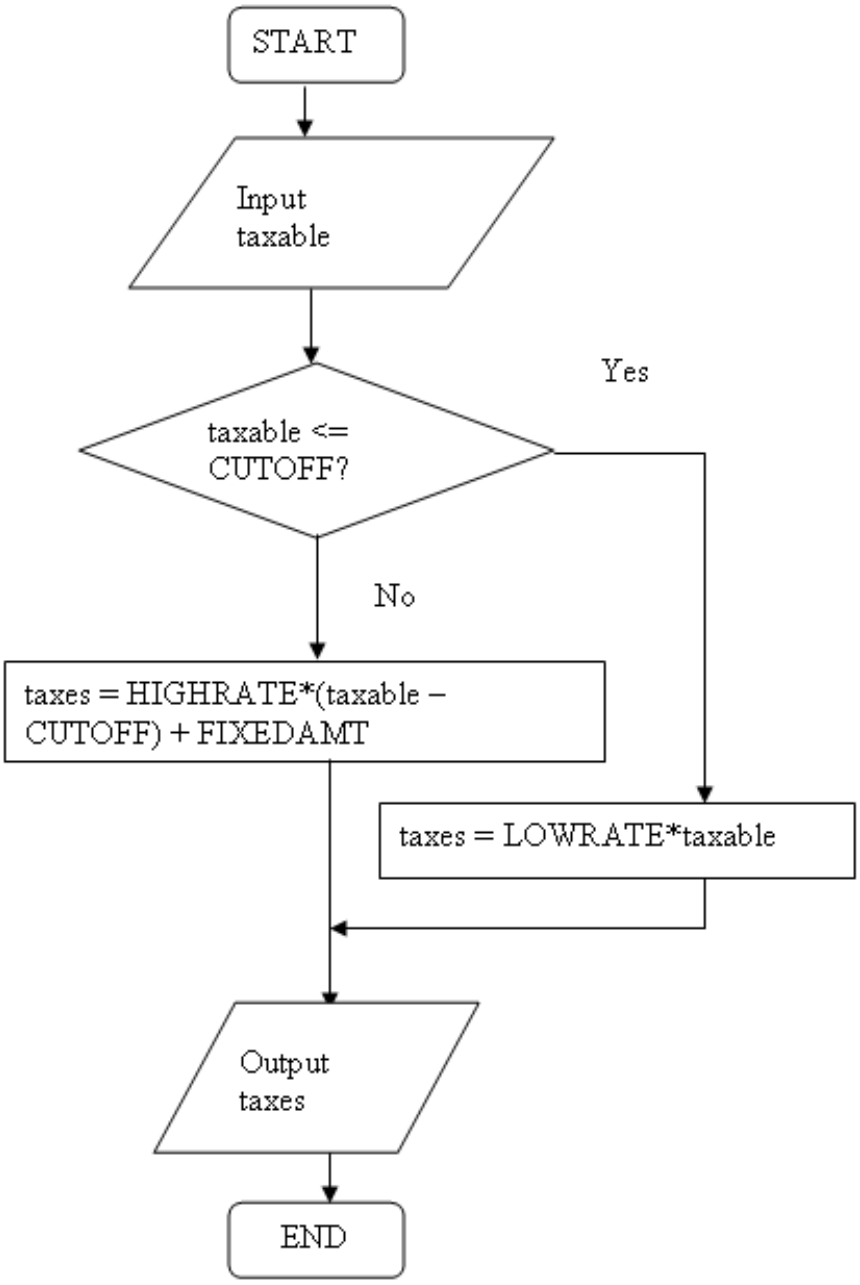
Flowchart of statement

Example 1

We construct a C++ program for determining income taxes. Assume that these taxes are assessed at 2% of taxable incomes less than or equal to $20,000. For taxable income greater than $20,000, taxes are 2.5% of the income that exceeds $20,000 plus a fixed amount of $400. (The flowchart of the program is given in Figure 2.)

```
#include <iostream.h>

#include <iomanip.h>

const float LOWRATE = 0.02; // lower tax rate

const float HIGHRATE = 0.025; // higher tax rate
```

```
const float CUTOFF = 20000.0; // cut off for low
rate

const float FIXEDAMT = 400; // fixed dollar amount
for higher rate amounts

int main()

{

float taxable, taxes;

cout << "Please type in the taxable income: ";

cin >> taxable;

if (taxable <= CUTOFF)

taxes = LOWRATE * taxable;

else

taxes = HIGHRATE * (taxable - CUTOFF) + FIXEDAMT;

// set output format

cout << setiosflags(ios::fixed)

<< setiosflags(ios::showpoint)

<< setprecision(2);

cout << "Taxes are $ " << taxes << endl;

return 0;

}
```



Flowchart of example

The **results** of the above program:

Please type in the taxable income: 10000

Taxes are $ 200

and

Please type in the taxable income: 30000

Taxes are $ 650

Block Scope

All statements within a compound statement constitute a single block of code, and any variable declared within such a block only is valid within the block.

The location within a program where a variable can be used formally referred to as the **scope** of the variable.

Example:

```
{ // start of outer block

int a = 25;

int b = 17;

cout << “The value of a is “ << a

<< “ and b is “ << b << endl;

{ // start of inner block

float a = 46.25;

int c = 10;

cout << “ a is now “ << a
```

```
<< “b is now “ << b
<< “ and c is “ << c << endl;
}
cout << “ a is now “ << a
<< “b is now “ << b << endl;
} // end of outer block
```

The **output** is

The value of a is 25 and b is 17

a is now 46.25 b is now 17 and c is 10

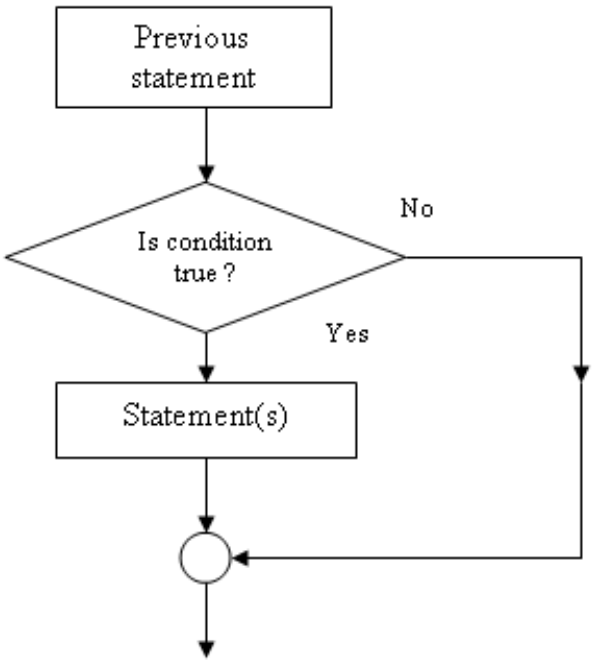
a is now 25 b is now 17

One-way Selection

A useful modification of the if-else statement involves omitting the else part of the statement. In this case, the if statement takes a shortened format:

```
if (conditional expression) {
statements;
}
```

The flow chart of one-way if statement is as below.



Flowchart of statement

Example

The following program displays an error message for the grades that is less than 0 or more than 100.

```
#include <iostream.h>

int main()

{

int grade;

cout << "\nPlease enter a grade: ";

cin >> grade;
```

```
if(grade < 0 || grade > 100)

cout << " The grade is not valid\n";

return 0;

}
```

Nested If Statement

An if-else statement can contain simple or compound statements. Any valid C++ statement can be used, including another if-else statement. Thus, one or more if-else statements can be included within either part of an if-else statement. The inclusion of one or more if statement within an existing if statement is called a nested if statement.

The if-else Chain

When an if statement is included in the else part of an existing if statement, we have an if-else chain.

```
if (expression-1)

statement-1

else if (expression-2)

statement-2

else

statement-3
```

Example

The following program calculates the monthly income of a computer salesperson using the following commission schedule:

Monthly Sales	Income
Greater than or equal to $50,000	$375 plus 16% of sales
Less than $50,000 but greater than or equal to $40,000	$350 plus 14% of sales
Less than $40,000 but greater than or equal to $30,000	$325 plus 12% of sales
Less than $30,000 but greater than or equal to $20,000	$300 plus 9% of sales
Less than $20,000 but greater than or equal to $10,000	$250 plus 5% of sales
Less than $10,000	$200 plus 3% of sales

Layout of result

```
#include <iostream.h>

#include <iomanip.h>

int main()

{

float monthlySales, income;

cout << "\nEnter the value of monthly sales: ";

cin >> monthlySales;

if (monthlySales >= 50000.00)

income = 375.00 + .16 * monthlySales;

else if (monthlySales >= 40000.00)

income = 350.00 + .14 * monthlySales;
```

```
else if (monthlySales >= 30000.00)
income = 325.00 + .12 * monthlySales;
else if (monthlySales >= 20000.00)
income = 300.00 + .09 * monthlySales;
else if (monthlySales >= 10000.00)
income = 250.00 + .05 * monthlySales;
else
income = 200.00 + .03 * monthlySales;
// set output format
cout << setiosflags(ios::fixed)
<< setiosflags(ios::showpoint)
<< setprecision(2);
cout << "The income is $" << income << endl;
return 0;
}
```

The **output** of the program:

Enter the value of monthly sales: 36243.89

The income is $4674.27

The Switch Statement

The **switch** statement controls program flow by executing a set of statements depending on the value of an expression.

Note: The value of expression must be an integer data type, which includes the char, int, long int, and short data types.

The syntax for the switch statement:

```
switch(expression){

case label:

statement(s);

break;

case label;

statement(s);

break;

default:

statement(s);

}
```

The expression in the switch statement must evaluate to an integer result. The switch expression's value is compared to each of these case values in the order in which these values are listed until a match is found. When a match occurs, execution begins with the statement following the match.

If the value of the expression does not match any of the case values, no statement is executed unless the keyword **default** is encountered. If the value of the expression does not match any of the case values, program execution begins with the statement following the word default.

The break statement is used to identify the end of a particular case and causes an immediate exit from the switch statement. If the break statements are omitted, all cases following the matching case value, including the default case, are executed.

Example

```
#include <iostream.h>

int main()

{

int iCity;

cout << "Enter a number to find the state where a city is located. "<< endl;

cout << “1. Boston” << endl;

cout << "2. Chicago" << endl;

cout << "3. Los Angeles” << endl;

cout << "4. Miami” << endl;

cout << "5. Providence” << endl;

cin >> iCity;

switch (iCity)

{

case 1:

cout << "Boston is in Massachusetts " << endl;

break;
```

```
case 2:

cout << "Chicago is in Illinois " << endl;

break;

case 3:

cout << "Los Angeles is in California " << endl;

break;

case 4:

cout << "Miami is in Florida " << endl;

break;

case 5:

cout << "Providence is in Rhode Island " << endl;

break;

default:

cout << “You didn’t select one of the five cities” << endl;

} // end of switch

return 0;

}
```

The **output** of the above program:

Enter a number to find the state where a city is located.

1. Boston

2. Chicago

3. Los Angeles

4. Miami

5. Providence

3

Los Angeles is in California

The **switch** statement is a clean way to implement multi-way selection (i.e., selecting from among a number of different execution paths), but it requires an expression that evaluates to an integral value at compile-time.

When writing a switch statement, you can use multiple case values to refer to the same set of statements; the default label is optional. For example, consider the following example:

```
switch(number)

{

case 1:

cout << “Have a Good Morning\n”;

break;

case 2:

cout << “Have a Happy Day\n”;

break;

case 3:
```

```
case 4:

case 5:

cout << “Have a Nice Evening\n”;

}
```

The Enum Specifier

An enumerated data type is a way of attaching names to numbers, thereby giving

more meaning to anyone reading the code. The **enum** specifier creates an enumerated data type, which is simply a user-defined list of values that is given its own data type name. Such data types are identified by the reserved word **enum** followed by an optional user-selected name for the data type and a listing of acceptable values for the data type.

Example:

```
enum day { mon, tue, wed, thu, fri, sat, sun};

enum color {red, green, yellow};
```

Any variable declared to be of type color can take only a value of red or green or yellow. Any variable declared to be of type day can take only a value among seven given values.

The statement

```
enum day a, b,c;
```

declares the variables a, b, and c to be of type day.

Internally, the acceptable values of each enumerated data type are ordered and assigned sequential integer values beginning with 0. For example, for the values of the user-defined type color, the correspondences created by

the C++ compiler are that red is equivalent to 0, green is equivalent to 1, and yellow is equivalent to 2. The equivalent numbers are required when inputting values using cin or displaying values using cout.

Example

```
#include <iostream.h>

int main()

{

enum color{red, green, yellow};

enum color crayon = red;

cout << “\nThe color is “ << crayon << endl;

cout << “Enter a value: “;

cin >> crayon;

if (crayon == red)

cout << “The crayon is red.” << endl;

else if (crayon == green)

cout << “The crayon is green.” << endl;

else if (crayon== yellow)

cout << “The crayon is yellow.” << endl;

else

cout << “The color is not defined. \n” << endl;
```

```
return 0;

}
```

The **output** of the above program:

The color is 0

Enter a value: 2

The crayon is yellow.

Focus on Problem Solving

Two major uses of C++'s **if** statements are to select appropriate processing paths and to prevent undesirable data from being processed at all. In this section, an example of both uses is provided.

Problem: Solving Quadratic Equations

A quadratic equation is an equation that has the form ax2 + bx + c = 0 or that can be algebraically manipulated into this form. In this equation, x is the unknown variable, and a, b and c are known constants. Although the constants b and c can be any numbers, including 0, the value of the constant a cannot be 0 (if a is 0, the equation becomes a linear equation in x). Examples of quadratic equations are:

5x^2 + 6x + 2 = 0

x^2 - 7x + 20 = 0

34x^2 + 16 = 0

In the first equation, a = 5, b = 6, and c = 2; in the second equation, a = 1, b = -7, and c = 20; and in the third equation, a = 34, b = 0 and c = 16.

The real roots of a quadratic equation can be calculated using the quadratic formula as:

delta = b2 – 4ac

root1 = (-b + squared-root(delta))/(2a)

root2 = (-b - squared-root(delta))/(2a)

Using these equations, we will write a C++ program to solve for the roots of a quadratic equation.

Step 1: Analyze the Problem

The problem requires that we accept three inputs – the coefficients a, b and c of a quadratic equation – and compute the roots of the equation using the given formulas.

Step 2: Develop a Solution

A first attempt at a solution is to use the user-entered values of a, band c to directly calculate a value for each of the roots. Thus, our first solution is:

Display a program purpose message.

Accept user-input values for a, b, and c.

Calculate the two roots.

Display the values of the calculated roots.

However, this solution must be refined further to account for a number of possible input conditions. For example, if a user entered a value of 0 for both a and b, the equation is neither quadratic nor linear and has no solution (this is referred to as a degenerate case).

Another possibility is that the user supplies a nonzero value for b but make a 0. In this case, the equation becomes a linear one with a single solution of

–c/b. A third possibility is that the value of the term b^2 – 4ac, which is called the discriminant, is negative. Since the square root of a negative number cannot be taken, this case has no real roots. Finally, when the discriminant is 0, both roots are the same (this is referred to as the repeated roots case).

Taking into account all four of these limiting cases, a refined solution for correctly determining the roots of a quadratic equation is expressed by the following pseudocode:

```
Display a program purpose message.

Accept user-input values for a, b, and c.

If a = 0 and b = 0 then

Display a message saying that the equation has no solution.

Else if a = 0 then

calculate the single root equal to –c/b.

display the single root.

Else

Calculate the discriminant.

If the discriminant > 0 then

Solve for both roots using the given formulas.

Display the two roots.

Else if the discriminant < 0 then

Display a message that there are no real roots.
```

```
Else

Calculate the repeated root equal to –b/(2a).

Display the repeated root.

Endif.

Endif.
```

Notice in the pseudocode that we have used nested if-else structures. The outer if-else structure is used to validate the entered coefficients and determine that we have a valid quadratic equation. The inner if-else structure is then used to determine if the equation has two real roots (discriminant > 0), two imaginary roots (discriminant < 0) or repeated roots (discriminant =0).

Step 3 : Code the Algorithm

The equivalent C++ code corresponding to our pseudocode is listed as the following program

```
// This program can solve quadratic equation

#include <iostream.h>

#include <math.h>

#include <iomanip.h>

int main()

{

double a, b, c, del, x1, x2;
```

```
cout << “This program calculates the roots of a\n”;

cout << “ quadratic equation of the form\n”;

cout << “ 2\n”;

cout << “ ax + bx + c = 0\n\n”;

cout << “Enter values for a, b, and c: “;

cin >> a >> b >> c;

if ( a == 0.0 && b == 0.0)

cout << “The equation is degenerate and has no roots.\n”;

else if (a == 0.0)

cout << “The equation has the single root x = “

<< -c/b << endl;

else

{

del = b*b – 4.0*a*c;

if (del > 0.0)

{

x1 = (-b + sqrt(del))/(2*a);

x2 = (-b – sqrt(del))/(2*a);

cout << "The two roots are “
```

```
<< x1 << " and " << x2 << endl;

}

else if (del <0)

cout << "Both roots are imaginary.\n";

else

cout << "Both roots are equal to " << -b/(2*a) <<
endl;

}

return 0;

}
```

Step 4: Test and Correct the Program

Test values should include values for a, b and c that result in two real roots, plus limiting values for a and b that result in linear equation (a = 0, b != 0), a degenerate equation (a = 0, b = 0), and a negative and 0 discriminant. Two such test runs of the above program follow:

This program calculates the roots of a

quadratic equation of the form

ax^2 + bx + c = 0

Please enter values for a, b and c: 1 2 -35

The two real roots are 5 and –7

and

This program calculates the roots of a

quadratic equation of the form

ax^2 + bx + c = 0

Please enter values for a, b and c: 0 0 16

This equation is degenerate and has no roots.

We leave it as an exercise to create test data for the other limiting cases checked for by the program.

Repetition Statements, Arrays and Structured Programming
This chapter explores the different methods programmers use to construct repeating sections of code and how that code can be implemented in C++. More commonly, a section of code that is repeated is referred to as a loop, because after the last statement in the code is executed, the program branches, or loops, back to the first statement and start another repetition through the code. Each repetition is also referred to as an iteration.

Basic Loop Structures

The real power of a program is realized when the same type of operation must be made over and over.

Constructing a repetitive section of code requires that four elements be present. The first necessary element is a repetition statement. This repetition statement defines the boundaries containing the repeating section of code and also controls whether the code is executed or not. C++ provides three different forms of repetition statements:

1. **while** structure
2. **for** structure
3. **do-while** structure

Each of these statements requires a condition that must be evaluated, which is the second required element for constructing repeating sections of code. Valid conditions are similar to those used in selection statements. If the condition is true, the code is executed; otherwise, it is not.

The third required element is a statement that initially sets the condition. This statement must always be placed before the condition is first evaluated to ensure correct loop execution the first time the condition is evaluated.

Finally, there must be a statement within the repeating section of code that allows the condition to become false. This is necessary to ensure that, at some point, the repetition stop.

The condition being tested can be evaluated at either (1) the beginning or (2) the end of the repeating section of code.

If the test occurs at the beginning of the loop, the type of loop is called a **pre-test loop** or entrance-controlled loop. If the test occurs at the end of the loop, the type of loop is called **a post-test loop** or exit-controlled-loop.

In addition to where the condition is tested (pretest or posttest), repeating sections of code are also classified. In a fixed count loop, the condition is used to keep track of how many repetitions have occurred. In this kind of loops, a fixed number of repetitions are performed, at which point the repeating section of code is exited.

In many situations, the exact number of repetitions are not known in advance or the items are too numerous to count beforehand. In such cases, a variable condition loop is used. In a variable condition loop, the tested condition does not depend on a count being achieved, but rather on a variable that can change interactively with each pass through the loop. When a specified value is encountered, regardless of how many iterations have occurred, repetitions stop.

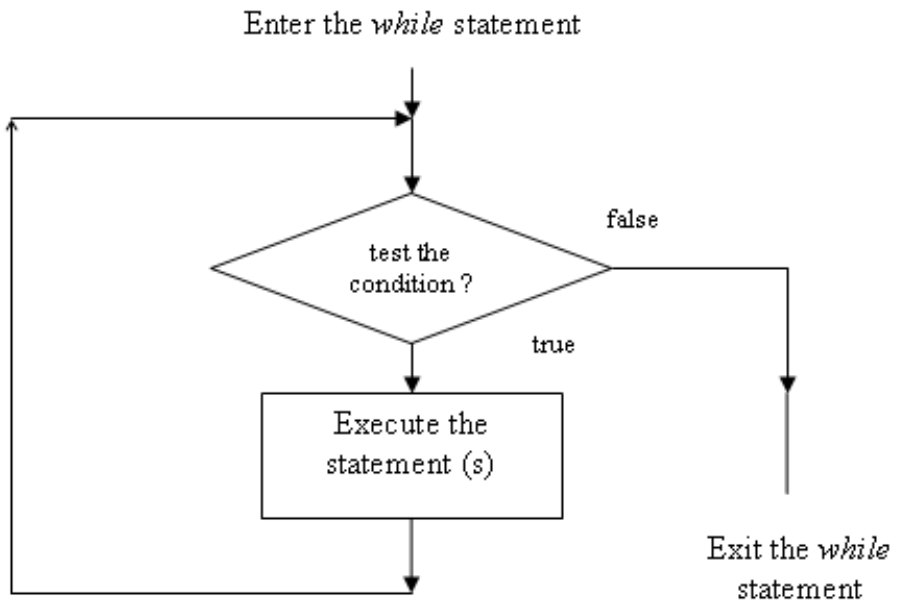
While Loops

The **while** statement is used for repeating a statement or series of statements as long as a given conditional expression is evaluated to true.

The syntax for the while statement:

```
while( condition expression){

statements;

}
```

The flow chart of the while statement is given below.



Flow chart of the while statement

Example

```
// this program computes the sum of 10 first integers starting from 1

#include <iostream.h>

int main()

{

const int N = 10

int sum = 0;

int count = 1; // initialize count

while (count <= N){
```

```
sum = sum + count;

count++; // increment count

}

cout << “The sum is “ << sum << endl;

return 0;

}
```

The **output** of the above program:

The sum is 55

In the above program, the loop incurs a counter-controlled repetition. Counter-controlled repetition requires:

1. the name of a control variable (the variable count in this case);
2. the initial value of the control variable (count is initialized to 1 in this case)
3. the condition that tests for the final value of the control variable (i.e., whether looping should continue) ;
4. the increment (or decrement) by which the control variable is modified each time through the loop.

Example

```
#include <iostream.h>

int main()

{

int i;
```

```
i = 10;

while (i >= 1)

{

cout << i << " ";

i--; // subtract 1 from i

}

return 0;

}
```

The **output** of the above program:

1. 9 8 7 6 5 4 3 2 1

Interactive While Loops

Combining interactive data entry with the repetition capabilities of the while statement produces very adaptable and powerful programs.

Example

```
// Class average program with counter-controlled repetition

#include <iostream.h>

int main()

{

int total, // sum of grades
```

```
gradeCounter, // number of grades entered

grade, // one grade

average; // average of grades

// initialization phase

total = 0;

gradeCounter = 1; // prepare to loop

while ( gradeCounter <= 10 ) { // loop 10 times

cout << "Enter grade: "; // prompt for input

cin >> grade; // input grade

total = total + grade; // add grade to total

gradeCounter = gradeCounter + 1; // increment
counter

}

// termination phase

average = total / 10; // integer division

cout << "Class average is " << average << endl;

return 0;

}
```

The following is a **sample run** of the above program:

Enter grade: 98

Enter grade: 76

Enter grade: 71

Enter grade: 87

Enter grade: 83

Enter grade: 90

Enter grade: 57

Enter grade: 79

Enter grade: 82

Enter grade: 94

Class average is 81

Sentinels

In programming, data values used to indicate either the start or end of a data series are called sentinels. The sentinel values must be selected so as not to conflict with legitimate data values.

Example

```
#include <iostream.h>

int main()

{

float grade, total;

grade = 0;
```

```
total = 0;

cout << "\nTo stop entering grades, type in any
number less than 0.\n\n";

cout << "Enter a grade: ";

cin >> grade;

while (grade >= 0 )

{

total = total + grade;

cout << "Enter a grade: ";

cin >> grade;

}

cout << "\nThe total of the grades is " << total
<< endl;

return 0;

}
```

The following is a **sample run** of the above program:

To stop entering grades, type in any number less than 0.

Enter a grade: 95

Enter a grade: 100

Enter a grade: 82

Enter a grade: -2

The total of the grades is 277

break statement

The **break** statement causes an exit from the innermost enclosing loop statement.

Example:

```
while( count <= 10)

{

cout << “Enter a number: “:

cin >> num;

if (num > 76){

cout << “you lose!\n”;

break;

}

else

cout << “Keep on trucking!\n”;

count++;

}

//break jumps to here
```

The **break** statement violates pure structured programming principles because it provides a second, nonstandard exit from a loop.

However, it is useful and valuable for breaking out of loops when an unusual condition is detected.

continue Statements

The **continue** statement halts a looping statement and restarts the loop with a new iteration.

```
while( count < 30)
{
cout << “Enter a grade: “;
cin >> grade;
if (grade < 0 || grade > 100)
continue;
total = total + grade;
count++;
}
```

In the above program, invalid grades are simply ignored and only valid grades are added to the total.

The null statement

All statements must be terminated by a semicolon. A semicolon with nothing preceding it is also a valid statement, called the **null** statement. Thus, the statement

;

is a null statement.

Example:

```
if (a > 0)

b = 7;

else ;
```

The **null** statement is a do-nothing statement.

For Loops

The **for** statement is used for repeating a statement or series of statements as long as a given conditional expression evaluates to true.

One of the main differences between while statement and for statement is that in addition to a conditional expression, you can also include code in the for statement

- to initialize a counter variable and
- changes its value with each iteration

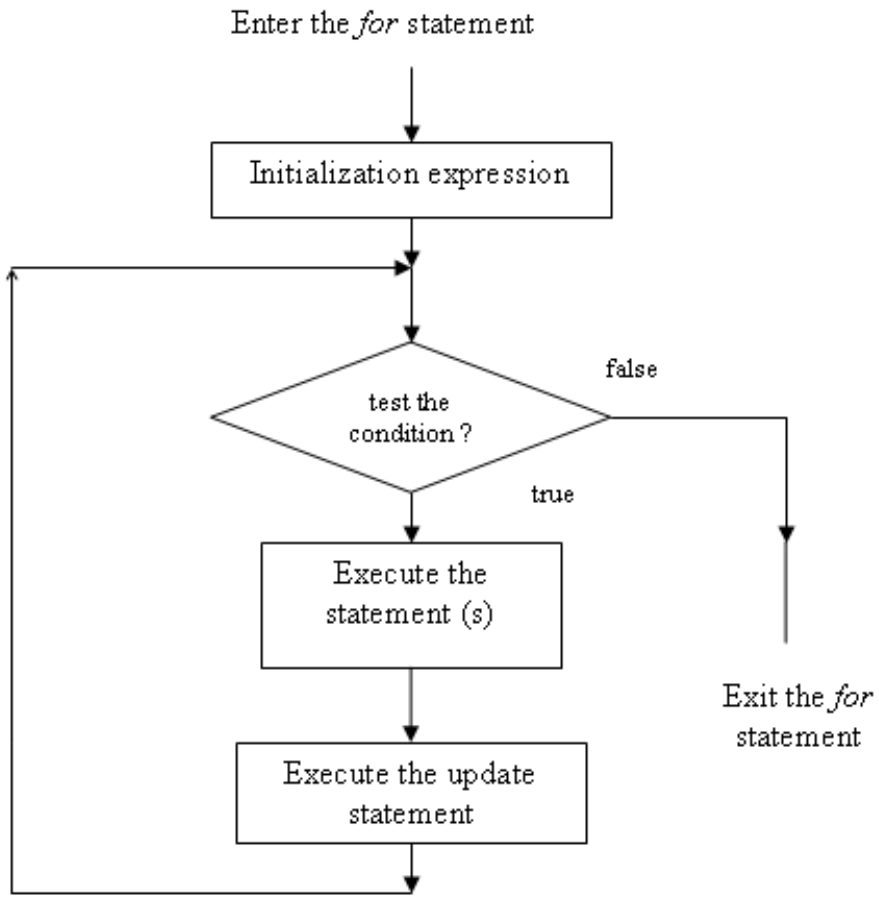
The syntax of the for statement:

```
for ( initialization expression; condition; update
statement){

statement(s);

}
```

In its most common form, the initialization expression consists of a single statement used to set the starting value of a counter variable, the condition contains the maximum or minimum value of the counter variable can have

and determines when the loop is finished, and the update statement provides the increment value that is added to or subtracted from the counter variable each time the loop is executed.

The flowchart of the for statement is given below.



Flow chart of the for statement

Example

```
#include <iostream.h>
```

```
int main()

{

int sum = 0;

for (int number = 2; number <= 100; number += 2)

sum += number;

cout << “Sum is “ << sum << endl;

return 0;

}
```

The **output** of the above program:

Sum is 2550

Example

In this example, we have to solve the following problem:

A person invests $1000.00 in a saving account with 5 percent interest. Assuming that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:

a = p(1 + r)^n

where p is the original amount invested, r is the annual interest rate and n is the number of years and a is the amount on deposit at the end of the nth year.

```
#include <iostream.h>

#include <iomanip.h>
```

```
#include <math.h>

int main()

{

double amount,

principal = 1000.0,

rate = 0.05;

cout << "Year” << setw(21)

<< "Amount on deposit" << endl;

cout << setiosflags(ios::fixed | ios::showpoint)
<< setprecision(2);

for (int year = 1; year <= 10; year++)

{

amount = principal*pow(1.0 + rate, year);

cout << setw(4) << year

<< setw(21) << amount << endl;

}

return 0;

}
```

The **output** of the above program:

YearAmount on deposit

1 1050.00

1. 1102.50
2. 1157.62
3. 1215.51
4. 1276.28
5. 1340.10
6. 1407.10
7. 1477.46
8. 1551.33
9. 1628.89

Nested Loops

In many situations, it is convenient to use a loop contained within another loop. Such loops are called **nested loops**.

Example

```
#include <iostream.h>

int main()

{

const int MAXI = 5;

const int MAXJ = 4;

int i, j;

for(i = 1; i <= MAXI; i++) // start of outer loop

{

cout << "\ni is now " << i << endl;
```

```
for(j = 1; j <= MAXJ; j++) // start of inner loop

cout << " j = " << j; // end of inner loop

} // end of outer loop

cout << endl;

return 0;

}
```

The **output** of the above program:

i is now 1

j = 1 j = 2 j = 3 j = 4

i is now 2

j = 1 j = 2 j = 3 j = 4

i is now 3

j = 1 j = 2 j = 3 j = 4

i is now 4

j = 1 j = 2 j = 3 j = 4

i is now 5

j = 1 j = 2 j = 3 j = 4

Do-While Loops

The **do..while** statement executes a statement or statements once, then repeats the execution as long as a given conditional expression evaluates to true.

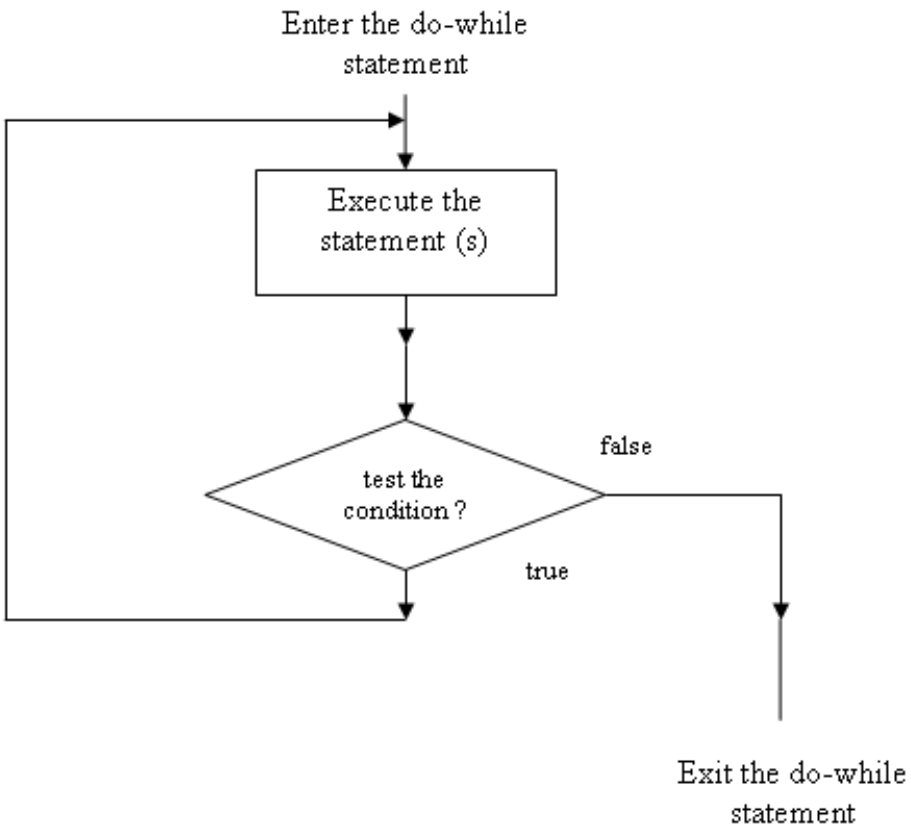
The do..while statement is used to create post-test loops.

The syntax for the do..while statement:

```
do {
statements;
} while (conditional expression);
```

Example:

```
do {
cout<< “\nEnter an identification number:”;
cin >> idNum;
} while (idNum < 1000|| idNum> 1999);
```



Flow chart of the do...while statement

Here, a request for a new id-number is repeated until a valid number is entered.

```
do {

cout<< “\nEnter an identification number:”;

cin >> idNum;

if (idNum < 1000 || idNum > 1999)

{

cout << “An invalid number was just entered\n”;
```

```
cout << “Please reenter an ID number /n”;

}

else break;

} while (true);
```

Structured Programming with C++

The goto Statement

In C++, **goto** statement – an unconditional branch, is just a legacy code from C language. The result of the goto statement is a change in the flow of control of the program to the first statement after the **label** specified in the goto statement.

Example:

```
start: // label

if (cout > 10) go to end;

…

…

go to start;

end: cout << endl;
```

The goto statement can lead to programs that are more difficult to debug, maintain, and modify.

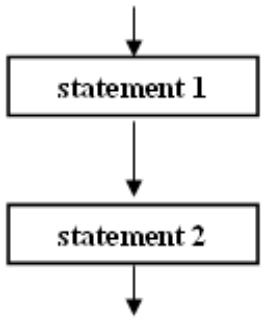
Structured Programming

During the 1960s, it became clear that the indiscriminate use of transfers of control through goto statements was the root of much difficulty experienced by programmer groups. The notion of so-called structured programming became almost synonymous with "**goto elimination**."

Bohm and Jacopini's work demonstrated that all programs could be written in terms of only three control structures:

- sequence structure
- selection structure
- repetition structure

The sequence structure is built into C++. Unless directed otherwise, the computer executes C++ statements one after the other in the order in which they are written. Below is a sequence structure.



Sequence
Structure

C++ provides three types of selection structures:

- if statement (single-selection structure)

- if-else statement (double-selection structure)

- switch statement. (multiple-selection structure)

C++ provides three types of repetition structures:

- while statement

- do-while statement

- for statement

So C++ has only seven control structures: sequence, three types of selection and three types of repetition. Each C++ program is formed by combining as many of each type of control structures as is appropriate for the algorithm the program implements.

We will see that each control structure has only one **entry point** and one **exit point**. These single-entry/single-exit control structures make it easy to build programs.

One way to build program is to connect the exit point of one control structure to the entry point of the next. This way is called control-structure-stacking.

Another way is to place one control structure inside another control structure. This way is called **control-structure-nesting**.

Consistent applying reasonable indentation conventions throughout your programs greatly improves program readability. We suggest a fixed-size tab of about ¼ inch or three blanks per indent.

For example, we indent both body statements of an if..else structure as in the following statement:

```
if (grade >= 60)

cout << “Passed”;

else
```

```
cout << “Failed”;
```

Top-down Stepwise Refinement

Using good control structures to build programs is one of the main principles of structured programming. Another principle of structured programming is top-down, stepwise refinement.

Consider the following problem:

Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.

We begin with a pseudocode representation of the top:

Determine the class average for the exam

Now we begin the refinement process. We divide the top into a series of smaller tasks and list these in the order in which they need to be performed. This results in the following first refinement.

First Refinement:

Initialize variables

Input, sum and count the exam grades

Calculate and print the class average

Here only the sequence structure has been used.

To proceed to the next level of refinement, we need some variables and a repetition structure. We need a running total of the numbers, a count of how many numbers have been processed, a variable to receive the value of each grade as it is input and a variable to hold the calculated average. We need a loop to calculate the total of the grades before deriving the average. Because we do not know in advance how many grades are to be processed,

we will use sentinel-controlled repetition. The program will test for the sentinel value after each grade is input and will terminate the loop when the sentinel value is entered by the user. Now we come to the pseudocode of the second refinement.

Second Refinement:

```
Input the first grade(possibly the sentinel)

While the user has not as yet entered the sentinel

Add this grade into the running total

Input the next grade(possibly the sentinel)

Calculate and print the class average
```

The pseudocode statement

```
Calculate and print the class average
```

can be refined as follows:

```
If the counter is not equal to zero

set the average to the total divided by the
counter

print the average

else

Print “No grades were entered”.
```

Notice that we are being careful here to test for the possibility of division by zero – a fatal error, if undetected, would cause the program to fail. Now we come to the pseudocode of the third refinement.

Third Refinement:

```
Initialize total to zero

Initialize counter to zero

Input the first grade

While the user has not as yet entered the sentinel

Add this grade into the running total

Add one to the grade counter

Input the next grade

If the counter is not equal to zero

set the average to the total divided by the
counter

print the average

else

Print “No grades were entered”.
```

Final step: After coding, we come to the following C++ program.

```
#include <iostream.h>

#include <iomanip.h>

int main()

{

int total, // sum of grades

gradeCounter, // number of grades entered
```

```
grade; // one grade

double average; // number with decimal point for
average

// initialization phase

total = 0;

gradeCounter = 0;

// processing phase

cout << "Enter grade, -1 to end: ";

cin >> grade;

while ( grade != -1 ) {

total = total + grade;

gradeCounter = gradeCounter + 1;

cout << "Enter grade, -1 to end: ";

cin >> grade;

}

// termination phase

if ( gradeCounter != 0 ) {

average = double ( total ) / gradeCounter;

cout << "Class average is " << setprecision( 2 )

<< setiosflags( ios::fixed | ios::showpoint )
```

```
<< average << endl;

}

else

cout << "No grades were entered" << endl;

return 0;

}
```

Arrays

An **array** is an advanced data type that contains a set of data represented by a single variable name.

An element is an individual piece of data contained in an array.

Array Declaration

The syntax for declaring an array is

```
type name[elements];
```

Array names follow the same naming conventions as variable names and other identifiers.

Example:

```
int MyArray[4];

char StudentGrade[5];
```

The declaration int MyArray[3]; tells the compiler to reserve 4 elements for integer array MyArray.

The numbering of elements within an array starts with an index number of 0. An index number is an element's numeric position within an array. It is also called a subsript.

Each individual element is referred to as an indexed variable or a subscripted variable because both a variable name and an index or subscrip value must be used to reference the element.

Example:

StudentGrade[0] refers to the first element in the StudentGrade array.

StudentGrade[1] refers to the second element in the StudentGrade array.

StudentGrade[2] refers to the third element in the StudentGrade array.

StudentGrade[3] refers to the fourth element in the StudentGrade array.

StudentGrade[4] refers to the fifth element in the StudentGrade array.

Subscripted variables can be used anywhere scalar variables are valid. Examples using the elements of the MyArray array are:

```
MyArray[0] = 17;
```

```
MyArray[1] = MyArray[0] – 11;
```

```
MyArray[2] = 5*MyArray[0];
```

```
MyArray[3] = (MyArray[1] + MyArray[2] –3)/2;
```

```
Sum = MyArray[0] +MyArray[1] +MyArray[2] +
MyArray[3];
```

Example

```
#include <iostream.h>
```

```
int main(){

char StudentGrade[5]= {‘A’, ‘B’, ‘C’, ‘D’, ‘F’};

for ( int i = 0; i < 5; i++)

cout << StudentGrade[i] << endl;

return 0;

}
```

The **output** is:

A

B

C

D

F

Example

```
// Compute the sum of the elements of the array

#include <iostream.h>

int main()

{

const int arraySize = 12;

int a[ arraySize ] = { 1, 3, 5, 4, 7, 2, 99, 16,
45, 67, 89, 45 };
```

```
int total = 0;

for ( int i = 0; i < arraySize; i++ )

total += a[ i ];

cout << "Total of array element values is " <<
total << endl;

return 0 ;

}
```

The **output** of the above program is as follows :

Total of array element values is 383

Multi-Dimensional Arrays

The C++ language allows arrays of any type, including arrays of arrays. With two bracket pairs we obtain a two-dimensional array. The idea can be iterated to obtain arrays of higher dimension. With each bracket pair we add another array dimension.

Some examples of array declarations

```
int a[1000]; // a one-dimensional array

int b[3][5]; // a two-dimensional array

int c[7][9][2]; // a three-dimensional array
```

In these above example, b has 3 X 5 elements, and c has 7 X 9 X 2 elements. Starting at the base address of the array, all the array elements are stored contiguously in memory.

For the array b, we can think of the array elements arranged as follows:

	col 1	col2	col3	col4	col5
row 1	b[0][0]	b[0][1]	b[0][2]	b[0][3]	b[0][4]
row 2	b[1][0]	b[1][1]	b[1][2]	b[1][3]	b[1][4]
row 3	b[2][0]	b[2][1]	b[2][2]	b[2][3]	b[2][4]

Multi-dimensional array

Example

This program checks if a matrix is symmetric or not.

```
#include<iostream.h>

const int N = 3;

int main()

{

int i, j;

int a[N][N];

bool symmetr = true;

for(i= 0; i<N; i++)

for (j = 0; j < N; j++)

cin >> a[i][j];

for(i= 0; i<N; i++)

for (j = 0; j < N; j++)
```

```
cout << a[i][j]<< endl;
for(i= 0; i<N; i++){
for (j = 0; j < N; j++)
if(a[i][j] != a[j][i]){
symmetr = false;
break;
}
if(!symmetr)
break;
}
if(symmetr)
cout<<"\nThe matrix is symmetric"<< endl;
else
cout<<"\nThe matrix is not symmetric"<< endl;
return 0;
}
```

Strings and String Built-in Functions

In C++ we often use character arrays to represent strings. A string is an array of characters ending in a null character (‘\0’). A string may be assigned in a declaration to a character array. The declaration

```
char strg[] = “C++”;
```

initializes a variable to the string “C++”. The declaration creates a 4-element array strg containing the characters ‘C’, ‘+’, ‘+’ and ‘\0’. The null character (\0) marks the end of the text string. The declaration determines the size of the array automatically based on the number of initializers provided in the initializer list.

C++ does not provide built-in operations for strings. In C++, you must use a string built-in functions to manipulate char variables. Some commonly used string functions are listed below.

Function	Description
strcat()	Append one string to another
strchr()	Find the first occurrence of a specified character in a string
strcmp()	Compare two strings
strcpy()	Replaces the contents of one string with the contents of another
strlen()	Returns the length of a string

String functions

The strcpy() function copies a literal string or the contents of a char variable into another char variable using the syntax:

```
strcpy(destination, source);
```

where destination represents the char variable to which you want to assign a new value to and the source variable represents a literal string or the char variable contains the string you want to assign to the destination.

The strcat() function combines two strings using the syntax:

```
strcat(destination, source);
```

where destination represents the char variable whose string you want to combine with another string. When you execute strcat(), the string represented by the source argument is appended to the string contained in the destination variable.

Example:

```
char FirstName[25];

char LastName[25];

char FullName[50];

strcpy(FirstName, “Mike”);

strcpy(LastName, “Thomson”);

strcpy(FullName, FirstName);

strcat(FullName, “ “);

strcat(FullName, LastName);
```

Two strings may be compared for equality using the strcmp() function. When two strings are compared, their individual characters are compared a pair at a time. If no differences are found, the strings are equal; if a difference is found, the string with the first lower character is considered the smaller string.

The functions listed in Figure 2 are contained in the string.h header file. To use the functions, you must add the statement #include<string.h> to your program.

Example

```
#include<iostream.h>
```

```
#include<string.h>

int main()

{

char FirstName[25];

char LastName[25];

char FullName[50];

strcpy(FirstName, "Mike");

strcpy(LastName, "Thomson");

strcpy(FullName, FirstName);

strcat(FullName, " ");

strcat(FullName, LastName);

cout << FullName << endl;

int n;

n = strcmp(FirstName, LastName);

if(n<0)

cout<< FirstName << " is less than "<<
LastName<<endl;

else if(n ==0)

cout<< FirstName << " is equal to "<<
LastName<<endl;

else
```

```
cout<< FirstName << " is greater than "<<
LastName<<endl;

return 0;

}
```

The **output** of the program:

Mike Thomson

Mike is less than Thomson

How to input a string

Inputting a string from a keyboard requires the string I/O library function cin.geline(). The cin.getline() function has the syntax:

```
cin.getline(str, terminatingLength,
terminatingChar)
```

where str is a string or character pointer variable, terminatingLength is an integer constant or variable indicating the maximum number of input characters that can be input, and terminatingChar is an optional character constant or variable specifying the terminating character. If this optional third argument is omitted, the default terminating character is the newline (‘\n’) character.

The function call stops reading characters when the terminatingChar key is pressed or until terminatingLength characters have been read, whichever comes first.

Example

```
#include<iostream.h>

int main()

{
```

```
char Text[40];

cin.getline(Text, 40, '\n');

cout << Text << endl;

return 0;

}
```

The cin.getline() function continously accepts and stores characters typed at the keyboard into the character array named Text until either 39 characters are entered (the 40th character is then used to store the end-of-string marker, \0), or the ENTER key is detected.

Structures

A **structure**, or **struct**, is an advanced, user-defined data type that uses a single variable name to store multiple pieces of related information.

The individual pieces of information stored in a structure are referred to as **elements**, field, or members.

You define a structure using the syntax:

```
struct struct_name{

data_type field_name;

data_type field_name;

……..

} variable_name;
```

For example, the statement

```
struct emloyee{
```

```
char idnum[5];

char name[40];

long salary;

};
```

declares the form of a structure named employee and reserves storage for the individual data items listed in the structure. The employee structure consists of three data items or fields.

And the statement

```
struct emloyee{

char idnum[5];

char name[40];

long salary;

} Emp;
```

declares that Emp is a structure variable which has the form of the structure employee.

To access the field inside a structure variable, you append a period to the variable name, followed by the field name using the syntax:

```
variable.field;
```

When you use a period to access a structure fields, the period is referred to as the **member selection operator**.

Example

```
#include <iostream.h>
```

```
struct Date // this is a global declaration
{
int month;
int day;
int year;
};
int main()
{
Date birth;
birth.month = 12;
birth.day = 28;
birth.year = 1986;
cout << "\nMy birth date is "
<< birth.month << '/'
<< birth.day << '/'
<< birth.year % 100 << endl;
return 0;
}
```

The **ouput** of the above program is:

My birth date is 12/28/86

Arrays of Structures

The real power of structures is realized when the same structure is used for lists of data. Declaring an array of structures is the same as declaring an array of any other variable type.

Example

The following program uses array of employee records. Each of employee record is a structure named PayRecord. The program displays the first five employee records.

```
#include <iostream.h>

#include <iomanip.h>

const int MAXNAME = 20;

// maximum characters in a name

struct PayRecord // this is a global declaration

{

long id;

char name[MAXNAME];

float rate;

};

int main()

{

const int NUMRECS = 5;

// maximum number of records
```

```
int i;
PayRecord employee[NUMRECS] = {
{ 32479, "Abrams, B.", 6.72 },
{ 33623, "Bohm, P.", 7.54},
{ 34145, "Donaldson, S.", 5.56},
{ 35987, "Ernst, T.", 5.43 },
{ 36203, "Gwodz, K.", 8.72 }
};
cout << endl; // start on a new line
cout << setiosflags(ios::left);
// left justify the output
for ( i = 0; i < NUMRECS; i++)
cout << setw(7) << employee[i].id
<< setw(15) << employee[i].name
<< setw(6) << employee[i].rate << endl;
return 0;
}
```

The **output** of the program is:

32479	Abrams, B.	6.72
33623	Bohm, P.	7.54
34145	Donaldson, S.	5.56
35987	Ernst, T.	5.43
36203	Gwodz, K	8.72

Output of program

Functions and Pointers

Most computer programs that solve real-world problems are much larger than the programs in the first few chapters. Experience has shown that the best way to develop and maintain a large program is to construct it from smaller pieces or program units each, of which is more manageable than the original program. Generally, user-defined program units are called subprograms. This technique is called divide-and-conquer. This chapter deals with the method of declaration of the user-defined function and its use in C++. Besides, this chapter also covers pointer data type which is one of the strength of the C++ language.

Function and Parameter Declarations

In C++ all **subprograms** are referred to as **functions**. A function allows you to treat a related group of C++ statements as a single unit. The programmer can write functions to define specific tasks that could be used at many points in a program.

Functions allow the programmer to modulize a program. All variables declared in function definitions are **local variables** – they are known only in the function in which they are defined. Most functions have a list of parameters that provide the means for communicating information between functions.

There are several motivations for dividing a program into functions. The **divide-and-conquer** approach makes program development more manageable. Another motivation is software reusability – using existing functions as building blocks to create new programs. Software reusability is a major factor in object-oriented programming. With good function naming and definition, programs can be created from standardized functions that accomplish specific tasks. A third motivation is to avoid repeating code in a program. Packing code as a function allows the code to be executed from several location in a program simply by calling the function.

Defining a Function

The lines that compose a function within a C++ program are called a function definition. The syntax for defining a function is

```
data_type name_of_function (parameters){

statements;

}
```

A function definition consists of four parts:

- A reserved word indicating the return data type of the function's return value.

- The function name

- Any parameters required by the function, contained within parentheses.

- The function's statements enclosed in curly braces { }.

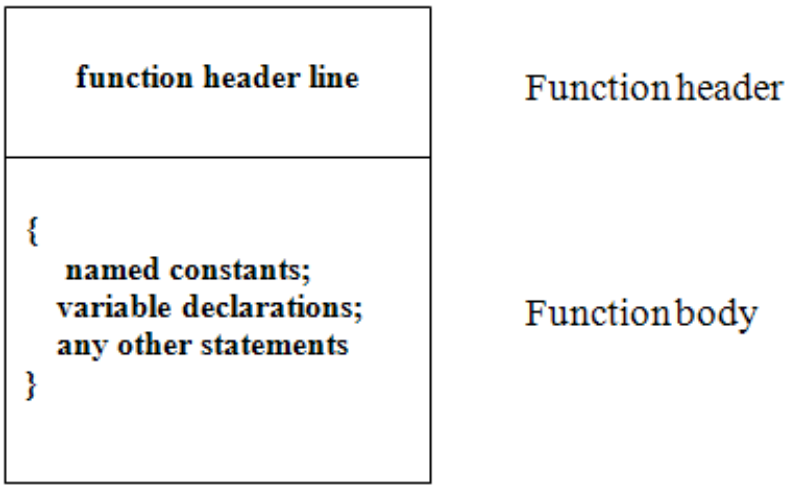
Example: The following function determines the largest integer among three parameters passed to it.

```
int maximum( int x, int y, int z )

{

int max = x;

if ( y > max )

max = y;

if ( z > max )

max = z;

return max;
```

```
}
```

You designate a data type for function since it is common to return a value from a function after it executes.

General format of a function is described below.



General format of a function

Variable names that will be used in the function header line are called formal parameters.

How to Call Functions

To execute a function, you must invoke, or call, it from the **main**() function.

The values or variables that you place within the parentheses of a function call statement are called arguments or actual parameters.

Example: Function maximum is invoked or called in main() with the call

maximum(a, b, c)

Function Prototypes

One of the most important features of C++ is the function prototype. A **function prototype** declares to the compiler that you intend to use a function later in the program. It informs the compiler the name of the function, the type of data returned by the function, the number of parameters the function expects to receive, the types of the parameters and the order in which these parameters are expected. The compiler uses function prototypes to validate function calls.

If you try to call a function at any point in the program prior to its function prototype or function definition, you will receive an error when you compile the project.

Example

```
// Finding the maximum of three integers

#include <iostream.h>

int maximum(int, int, int); // function prototype

int main()

{

int a, b, c;

cout << "Enter three integers: ";

cin >> a >> b >> c;
```

```
// a, b and c below are arguments to the maximum function call

cout << "Maximum is: " << maximum (a, b, c) << endl;

return 0;

}

// Function maximum definition

// x, y and z are parameters to the maximum function definition

int maximum( int x, int y, int z)

{

int max = x;

if ( y > max )

max = y;

if ( z > max )

max = z;

return max;

}
```

The **output** of the above program:

Enter three integers: 22 85 17

Maximum is: 85

Passing by Value

If a variable is one of the actual parameters in a function call, the called function receives a copy of the values stored in the variable. After the values are passed to the called function, control is transferred to the called function.

Example: The expression maximum(a, b, c) calls the function maximum() and causes the values currently residing in the variables a, b and c to be passed to maximum().

The method of passing values to a called function is called pass by value.

Functions with Empty Parameter Lists

Functions may have empty parameter list. The function prototype for such a function requires either the keyword void or nothing at all between the parentheses following the function name.

Example:

int display();

int display(void);

Returning Values

To actually return a value, the function must use a **return** statement, which has the form:

```
return expression;
```

or

```
return(expression);
```

Remember that values passes back and forth between functions must be of the same data type.

When the **return** statement is encountered, the expression is evaluated first. The value of the expression is then automatically converted to the data type declared in the function header before being sent back to the calling function. After the value is returned, program control reverts to the calling function.

Inline functions

For small functions, you can use the inline keyword to request that the compiler replace calls to a function with the function definition wherever the function is called in a program.

Example

```
// Using an inline function to calculate

// the volume of a cube.

#include <iostream.h>

inline double cube(double s) { return s * s * s; }

int main()

{

cout << "Enter the side length of your cube: ";

double side;

cin >> side;

cout << "Volume of cube with side "
```

```
<< side << " is " << cube(side) << endl;

return 0;

}
```

The **output** of the above program:

Enter the side length of your cube: 3.5

Volume of cube with side 3.5 is 42.875

Function Overloading

C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters (at least their types are different). This capability is called **function overloading**.

When an overloaded function is called, the C++ compiler selects the proper functions by examining the number, types and order of the arguments in the call.

Function overloading is commonly used to create several functions of the same name that perform similar tasks but on different data types.

Example:

```
void showabs(int x)

{

if( x < 0)

x = -x;

cout << “The absolute value of the integer is “ <<
x << endl;
```

```
}

void showabs(double x)

{

if( x < 0)

x = -x;

cout << “The absolute value of the double is “ <<
x << endl;

}
```

The function call

showabs(10);

causes the compiler to use the first version of the function showabs.

The function call

showabs(6.28);

causes the compiler to use the second version of the function showabs.

Default Arguments

C++ allows **default arguments** in a function call. Default argument values are listed in the function prototype and are automatically transmitted to the called function when the corresponding arguments are omitted from the function call.

Example: The function prototype

```
void example (int, int = 5, float = 6.78);
```

provides default values for the two last arguments.

If any of these arguments are omitted when the function is actually called, the C++ compiler supplies these default values.

Thus, all following function calls are valid:

```
example(7, 2, 9.3); // no default used

example(7, 2); // same as example(7, 2, 6.78)

example(7); // same as example(7, 5, 6.78)
```

Variable Scope

Scope refers to where in your program a declared variable or constant is allowed used. A variable can be used only when inside its scope.

Global scope refers to variables declared outside of any functions or classes and that are available to all parts of your program.

Local scope refers to a variable declared inside a function and that is available only within the function in which it is declared.

Example:

```
#include <iostream.h>

int x; // create a global variable named firstnum

void valfun(); // function prototype (declaration)

int main()

{
```

```
int y; // create a local variable named secnum

x = 10; // store a value into the global variable

y = 20; // store a value into the local variable

cout << "From main(): x = " << x << endl;

cout << "From main(): y = " << y << endl;

valfun(); // call the function valfun

cout << "\nFrom main() again: x = " << x << endl;

cout << "From main() again: y = " << y << endl;

return 0;

}

void valfun() // no values are passed to this function

{

int y; // create a second local variable named y

y = 30; // this only affects this local variable's value

cout << "\nFrom valfun(): x = " << x << endl;

cout << "\nFrom valfun(): y = " << y << endl;

x = 40; // this changes x for both functions

return;

}
```

The **output** of the above program:

From main(): x = 10

From main(): y = 20

From valfun(): x = 10

From valfun(): y = 30

From main() again: x = 40

From main() again: y = 20

In the above program, the variable x is a global variable because its storage is created by a definition statement located outside a function. Both functions, main() and valfun() can use this global variable with no further declaration needed. The program also contains two separate local variables, both named y. Each of the variables named y is local to the function in which their storage is created, and each of these variables can only be used within the appropriate functions.

Scope Resolution Operator

When a local variable has the same name as a global variable, all uses of the variable's name within the scope of the local variable refer to the local variable.

In such cases, we can still access to the global variable by using **scope resolution operator** (::) immediately before the variable name.

The :: operator tells the compiler to use the global variable.

Example

```
#include <iostream.h>
```

```
float number = 42.8; // a global variable named number

int main()

{

float number = 26.4; // a local variable named number

cout << "The value of number is " << number << endl;

return 0;

}
```

The **output** of the above program:

The value of number is 26.4

Example

```
#include <iostream.h>

float number = 42.8; // a global variable named number

int main()

{

float number = 26.4; // a local variable named number
```

```
cout << "The value of number is " << ::number << endl;

return 0;

}
```

The **output** of the above program:

The value of number is 42.8

Variable Storage Class

The **lifetime** of a variable is referred to as the **storage duration**, or storage class.

The four available storage classes are **auto**, **static**, **extern** and **register**.

If one of these class names is used, it must be placed before the variable's data type in a declaration statement.

Examples:

```
auto int num;

static int miles;

register int dist;

extern float price;

extern float yld;
```

Local Variable Storage Classes

Local variables can only be members of the auto, static, or register storage classes.

If no class description is included in the declaration statement, the variable is automatically assigned to the auto class.

Automatic Variables

The term auto is short for automatic. Automatic storage duration refers to variables that exist only during the lifetime of the command block (such as a function) that contains them.

Example

```
#include <iostream.h>

int funct(int); // function prototype

int main()

{

int count, value; // count is a local auto
variable

for(count = 1; count <= 10; count++)

value = funct(count);

cout << count << '\t' << value << endl;

return 0;

}

int funct( int x)
```

```
{
int sum = 100; // sum is a local auto variable
sum += x;
return sum;
}
```

The **output** of the above program:

1 101

2 102

3 103

4 104

5 105

6 106

7 107

8 108

9 109

10 110

Note: The effect of increasing sum in funct(), before the function's return statement, is lost when control is returned to main().

In some applications, we want a function to remember values between function calls. This is the purpose of the static storage class.

A local static variable is not created and destroyed each time the function declaring the static variable is called. Once created, local static variables remain in existence for the life of the program.

Example

```
#include <iostream.h>

int funct( int); // function prototype

int main()

{

int count, value; // count is a local auto
variable

for(count = 1; count <= 10; count++)

value = funct( count);

cout << count << '\t' << value << endl;

return 0;

}

int funct( int x)

{

static int sum = 100; // sum is a local auto
variable

sum += x;

return sum;

}
```

The **output** of the above program:

1 101

2 103

3 106

4 110

5 115

6 121

7 128

8 136

9 145

10 155

Since sum has a permanent memory space it retains the same value in the period of time between leaving function and again entering it later.

Note:

1. The initialization of static variables is done only once when the program is first compiled. At compile time, the variable is created and any initialization value is placed in it. Thereafter, the value in the variable is kept without further initialization each time is called. (compile-time initialization).
2. All static variables are set to zero when no explicit initialization is given.

Register Variables

Register variables have the same time duration as automatic variables. The only difference between register and automatic variables is where the storage for the variable is located.

Register variables are stored in CPU's internal registers rather than in memory.

Examples:

```
register int time;
```

```
register double difference;
```

Global Variable Storage Classes

Global variables are created by definition statements external to a function. Once a global variable is created, it exists until the program in which it is declared is finished executing.

Global variables may be declared as **static** or **extern** (but not both).

The purpose of the extern storage class is to extend the scope of a global variable beyond its normal boundaries. To understand this, we must notice that the programs we have written so far have always been contained together in one file. Thus, when you have saved or retrieved programs, you have only needed to give the computer a single name for your program. Larger programs typically consist of many functions stored in multiple files and all of these files are compiled separately. Consider the following exaple.

Example:

```
//file1
```

```
int a;
```

```
float c;
```

```
static double d;
.
.
int main()
{
func1();
func2();
func3();
func4();
.
}
int func1();
{
.
.
}
int func2();
{
.
.
```

```
}
//end of file1
//file2
double b;
int func3();
{
.
.
}
int func4();
{
.
.
}
//end of file2
```

Although the variable a has been declared in file1, we want to use it in file2. Placing the statement extern int a in file2, we can extend the scope of the variable a into file2.

Now the scope of the variable a is not only in file1, but also in func3 and func4.

```
//file1
```

```
int a;
float c;
static double d;
.
.
int main()
{
func1();
func2();
func3();
func4();
.
}
extern double b;
int func1();
{
.
.
}
int func2();
```

```
{
.
.
}
//end of file1
//file2
double b;
extern int a;
int func3();
{
.
.
}
int func4();
{
extern float c;
.
.
}
//end of file2
```

Besides, placing the statement extern float c; in func4() extends the scope of this global variable, created in file1, into func4(), and the scope of the global variable b, created in file2, is extended into func1() and func2() by the declaration statement extern double b; placed before func1().

Note:

1. We cannot make static variables external.
2. The scope of a global static variable cannot extend beyond the file in which it is declared.

Pass by Reference Using Reference Parameters

Two ways to invoke functions in many programming languages are: call by value and

call by reference

When an argument is passed **call by value**, a copy of the argument's value is made and passed to the called function. Changes to the copy do not affect the original variable's value in the caller.

With **call-by-reference**, the caller gives the called function the ability to access the caller's data directly, and to modify that data if the called function chooses so.

To indicate that the function parameter is **passed-by-reference**, simply follow the parameter's type in the function prototype of function header by an ampersand (&).

For example, the declaration

```
int& count;
```

in the function header means "count is a reference parameter to an int".

Example

```
// Comparing call-by-value and call-by-reference
with references.

#include <iostream.h>

int squareByValue( int );

void squareByReference( int & );

int main()

{

int x = 2, z = 4;

cout << "x = " << x << " before squareByValue\n"

<< "Value returned by squareByValue: "

<< squareByValue( x ) << endl

<< "x = " << x << " after squareByValue\n" <<
endl;

cout << "z = " << z << " before squareByReference"
<< endl;

squareByReference( z );

cout << "z = " << z << " after squareByReference"
<< endl;

return 0;

}

int squareByValue( int a )
```

```
{
return a *= a; // caller's argument not modified
}
void squareByReference( int &cRef )
{
cRef *= cRef; // caller's argument modified
}
```

The **output** of the above program:

x = 2 before squareByValue

Value returned by squareByValue: 4

x = 2 after squareByReference

z = 4 before squareByReference

z = 16 after squareByReference

Since cRef is a reference parameter, so, squareByReference() now has direct access to the argument z. Thus, any change to cRef within squareByReference() directly alters the value of z in main(). The assignment of value to cRef within squareByReference() is relected in main() as the altering of z's value.

Recall from Chapter 2 that the ampersand, &, in C++ means "the address of". Additionally, an & symbol used within a declaration refers to "the address of" the preceding data type. Using this information, declaration such as double& num1 and int& secnum are sometimes more clearly understood if they are read backward. Reading the declaration int& cRef in this manner yields the information that "cRef is the address of an int value."

Example

```
// This program can solve quadratic equation

#include <iostream.h>

#include <math.h>

#include <iomanip.h>

in quad( double, double, double, double &, double &);

int main()

{

double a, b, c, x1, x2;

int code;

cout << “Enter the coefficients of the equation: “<< endl;

cin >> a >> b >> c;

code = quad(a, b, c, x1, x2);

if (code == 1 || code == 2)

cout << "x1 = “ << x1 << setw(20) << “x2 = “ << x2 << endl;

else if (code == 3)

cout << “ There is no solution “ << endl;

return 0;
```

```
}

int quad(double a, double b, double c, double
&px1, double &px2)

{

double del;

del = b*b – 4.0*a*c;

if (del == 0.0)

{

px1 = -b/(2*a);

px2 = px1;

return 1;

}

else if (del > 0.0)

{

px1 = (-b + sqrt(del))/(2*a);

px2 = (-b – sqrt(del))/(2*a);

return 2;

}

else

return 3;
```

```
}
```

Note: The called-by-value parameters a, b, and c are used to passed the data from the calling function to the called function, and the two reference parameters px1 and px2 are used to pass the results from the called function to the calling function.

Recursion

In C++, it's possible for a function to call itself. Functions that do so are called seft-referential or **recursive** functions.

Example: To compute factorial of an integer

1! = 1

n! = n*(n-1)!

Example

```
// Recursive factorial function

#include <iostream.h>

#include <iomanip.h>

unsigned long factorial( unsigned long );

int main()

{

for ( int i = 0; i <= 10; i++ )

cout << setw( 2 ) << i << "! = " << factorial( i )
<< endl;

return 0;
```

```
}
// Recursive definition of function factorial
unsigned long factorial( unsigned long number )
{
if (number < 1) // base case
return 1;
else // recursive case
return number * factorial( number - 1 );
}
```

The **output** of the above program:

0! = 1

1! = 1

2! = 2

3! = 6

4! = 24

5! = 120

6! = 720

7! = 5040

8! = 40320

9! = 362880

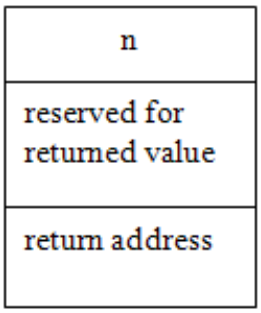
10! = 3628800

How the Computation is Performed

The mechanism that makes it possible for a C++ function to call itself is that C++ allocates new memory locations for all function parameters and local variables as each function is called. There is a **dynamic data area** for each execution of a function. This allocation is made dynamically, as a program is executed, in a memory area referred as the stack.

A **memory stack** is an area of memory used for rapidly storing and retrieving data areas for active functions. Each function call reserves memory locations on the stack for its parameters, its local variables, a return value, and the address where execution is to resume in the calling program when the function has completed execution (return address). Inserting and removing items from a stack are based on last-in/first-out mechanism.

Thus, when the function call factorial(n) is made, a data area for the execution of this function call is pushed on top of the stack. This data area is shown as figure below.



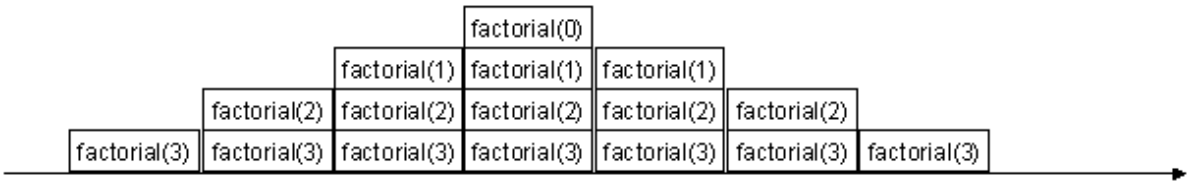
The data area for the first call to factorial

The progress of execution for the recursive function factorial applied with n = 3 is as follows:

```
factorial(3) = 3*factorial(2)
             = 3*(2*factorial(1))
             = 3*(2*(1*factorial(0)))
             = 3*(2*(1*1))
             = 3*(2*1)
             = 3*2
             = 6
```

Progress of execution for
the recursive function

During the execution of the function call factorial(3), the memory stack evolves as shown in the figure below. Whenever the recursive function calls itself, a new data area is pushed on top of the stack for that function call. When the execution of the recursive function at a given level is finished, the corresponding data area is popped from the stack and the return value is passed back to its calling function.



The memory stack for execution of function call factorial(3)

Example: Fibonacci numbers:

F(N) = F(N-1) + F(N-2) for N>= 2

F(0) = F(1) = 1

```
#include<iostream.h>
int fibonacci(int);
int main()
{
int m;
cout <<"Enter a number: ";
cin>> m;
cout <<"The fibonacci of "<<m<<" is: "
<< fibonacci(m)<< endl;
return 0;
}
int fibonacci(int n)
{
if (n<= 1) return 1;
return (fibonacci(n-1)+ fibonacci(n-2));
}
```

The **output** of the above program:

Enter a number: 4

The fibonacci of 4 is: 5

Recursion vs. Iteration

In this section, we compare recursion and iteration and discuss why the programmer might choose one approach over the other in a particular situation.

Both iteration and recursion are based on a control structure: Iteration uses a repetition structure; recursion uses a selection structure. Both iteration and recursion involve repetition: Iteration explicitly used a repetition structure while recursion achieves repetition through repeated function calls. Iteration and recursion both involve a termitation test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized. Iteration with counter-controlled repetition and recursion both gradually approach termination: Iteration keeps modifying a counter variable until the counter assumes a value that makes the loop-continuation condtion falil; recursion keeps producing simpler versions of the original problem until the base case is reached. Both iteration and recursion can occur indefinitely: An infinite loop occurs with iteration if the loop-continuation test never become false; infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case.

Recursion has many inconveniences. It repeatedly invokes the mechanism, and consequently the overhead, of function calls. This can be expensive in both CPU time and memory space. Each recursive call causes another copy of the function (actually only the function’s variables) to be created; this can consume considerable memory. Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted.

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen in preference to

an iterative approach when the recursive approach more naturally reflects the problem and results in a program that is easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution is not apparent.

Passing Arrays to Functions

To pass an array to a function, specify the name of the array without any brackets. For example, if array hourlyTemperature has been declared as

```
int hourlyTemperature[24];
```

The function call statement

```
modifyArray(hourlyTemperature, size);
```

passes the array hourlyTemperature and its size to function modifyArray.

For the function to receive an array through a function call, the function's parameter list must specify that an array will be received.

For example, the function header for function modifyArray might be written as

```
void modifyArray(int b[], int arraySize)
```

Notice that the size of the array is not required between the array brackets.

Example

To illustrate the use of array and function, we set for ourselves the following tasks:

1. Read in the amount to be deposited in the bank, the interest rate and the number of years to deposit.
2. Invoke the function to compute a table which keeps the amount we get after i years of deposit at the i-th component of the array.
3. Display out the above array

```
/* Compute compound interest */

#include<iostream.h>

#include<iomanip.h>

#define YMAX 50

void interest(double, double, int, double []);

int main()

{

double deposit, rate;

int i, years;

double compounded[YMAX];

cout<< “\n ENTER DEPOSIT, INTEREST RATE, NUMBER OF
YEARS \n”;

cin>>deposit>>rate>>years;

cout<<endl;

if(years>YMAX)

cout<<”\n Number of years must be less than

or equal”<<YMAX;

else

{

interest(deposit, rate, years, compounded);
```

```
for( i = 0; i < years; ++i)

cout<< i+1 << setw(25)<< compounded[i]

<< endl;

}

cout<< endl;

return 0;

}

void interest(double deposit, double rate,

int years, double cp[])

{

int i;

for( i = 0; i < years; ++i){

deposit = deposit*(1.0 + rate);

cp[i] = deposit;

}

}
```

Example

In the following program, we have to search an integer array for a given element. We use linear search in which each item in the array is examined sequentially until the desired item is found or the end of the array is reached.

```
#include<iostream.h>
int linearSearch( int [], int, int);
int main()
{
const int arraySize = 100;
int a[arraySize], searchkey, element;
for (int x = 0; x < arraySize, x++)
// create some data
a[x] = 2*x;
cout<< “Enter integer search key: “<< endl;
cin >> searchKey;
element = linearSearch(a, searchKey, arraySize);
if(element !=-1)
cout<<”Found value in element “<< element
<< endl;
else
cout<< “Value not found “ << endl;
return 0;
}
```

```
int linearSearch(int array[], int key, int sizeofArray)

{

for(int n = 0; n< sizeofArray; n++)

if (array[n] = = key)

return n;

return –1;

}
```

Pointers

In this section, we discuss one of the most powerful features of the C++ programming language, the **pointer**. Pointers are among C++'s most different capabilities to master. In section **Pass by Reference**, we saw that references can be used to perform call-by-reference. Pointers enable programs to simulate call-by-reference and to create and manipulate dynamic data structures (i.e., data structures that can grow and shrink).

A **pointer** is a special type of variable that stores the memory address of other variables.

You declare a variable as a pointer by placing the **indirection operator** (*) after the data type or before the variable name.

Examples:

```
int *pFirstPtr;

int *pSecondPtr;
```

You use the **address-of operator** (&) to assign to the pointer variable the memory address of another variable.

Examples:

```
double dPrimeInterest;
```

```
double *pPrimeInterest;
```

```
pPrimeInterest = &dPrimeInterest;
```

Once you assign the memory address of a variable to a pointer, to access or modify the contents of the variable pointed to by the pointer, you precede a pointer name in an expression with the **de-reference** (*) operator.

Example

The program in this example demonstrates the pointer operators. Memory locations are output in this example as hexadecimal integers.

```
#include<iostream.h>

int main()

{

int a;

int *aPtr; // aPtr is a pointer to an integer

a = 7;

aPtr = &a; //aPtr set to address of a

cout << “The address of a is “ << &a

<< “\nThe value of aPtr is “ << aPtr;

cout << “\n\nThe value of a is “<< a

<< “\nThe value of *aPtr is “ << *aPtr
```

```
<< endl;

return 0;

}
```

The **output** of the above program:

The address of a is 0x0065FDF4

The value of aPtr is 0x0065FDF4

The value of a is 7

The value of *aPtr is 7

Notice that the address of a and the value of aPtr are identical in the output, confirming that the address of a is assigned to the pointer variable aPtr.

Calling Functions by Reference with Pointer Arguments

In C++, programmers can use pointers and the dereference operator to simulate call-by-reference. When calling a function with arguments should be modified, the addresses of the arguments are passed. This is normally achieved by applying the address-of operator (&) to the name of the variable whose value will be used. A function receiving an address as an argument must define a pointer parameter to receive the address.

Example

```
// Cube a variable using call-by-reference

// with a pointer argument

#include <iostream.h>

void cubeByReference( int * ); // prototype
```

```
int main()

{

int number = 5;

cout << "The original value of number is " <<
number;

cubeByReference( &number );

cout << "\nThe new value of number is " << number
<< endl;

return 0;

}

void cubeByReference( int *nPtr )

{

*nPtr = (*nPtr) * (*nPtr) * (*nPtr); // cube
number in main

}
```

The **output** of the above propgram:

The original value of number is 5

The new value of number is 125

Pointers and Arrays

Notice that the name of an array by itself is equivalent to the base address of that array. That is, the name z in isolation is equivalent to the expression

&z[0].

Example

```
#include<iostream.h>

int main()

{

int z[] = { 1, 2, 3, 4, 5};

cout << “The value return by ‘z’ itself is

the addr “ << z << endl;

cout << “The address of the 0th element of

z is “ << &z[0] << endl;

return 0;

}
```

The **output** of the above program:

The value return by ‘z’ itself is the addr 0x0065FDF4

The address of the 0th element of z is 0x0065FDF4

Accessing Array Element Using Pointer and Offset

Now, let us store the address of array element 0 in a pointer. Then using the indirection operator, *, we can use the address in the pointer to access each array element.

For example, if we store the address of grade[0] into a pointer named gPtr, then the expression *gPtr refers to grade[0].

One unique feature of pointers is that offset may be included in pointer expression.

For example, the expression *(gPtr + 3) refers to the variable that is three (elements) beyond the variable pointed to by gPtr.

The number 3 in the pointer expression is an offset. So gPtr + 3 points to the element grade[3] of the grade array.

Example

```
#include <iostream.h>

int main()

{

int b[] = { 10, 20, 30, 40 }, i, offset;

int *bPtr = b; // set bPtr to point to array b

cout << "Array b printed with:\n"

<< "Array subscript notation\n";

for ( i = 0; i < 4; i++ )

cout << "b[" << i << "] = " << b[ i ] << '\n';

cout << "\nPointer/offset notation\n";

for ( offset = 0; offset < 4; offset++ )

cout << "*(bPtr + " << offset << ") = "

<< *( bPtr + offset ) << '\n';

return 0;
```

```
}
```

The **output** of the above program is:

Array b printed with:

Array subscript notation

b[0] = 10

b[1] = 20

b[2] = 30

b[3] = 40

Pointer/offset notation

*(bPtr + 0) = 10

*(bPtr + 1) = 20

*(bPtr + 2) = 30

*(bPtr + 3) = 40

Pointers and Strings

In C++ we often use character arrays to represent strings. A string is an array of characters ending in a null character (‘\0’). Therefore, we can scan through a string by using a pointer. Thus, in C++, it is appropriate to say that a string is a **constant pointer** – a pointer to the string’s first character.

A string may be assigned in a declaration to either a character array or a variable of type char *. The declarations

```
char color[] = “blue”;
```

```
char* colorPtr = “blue”;
```

each initialize a variable to the string “blue”. The first declaration creates a 5-element array color containing the characters ‘b’, ‘l’, ‘u’, ‘e’ and ‘\0’. The second declaration creates pointer variable colorPtr that points to the string “blue” somewhere in the memory.

The first declaration determines the size of the array automatically based on the number of initializers provided in the initializer list.

Example

```
/* Printing a string one character at a time using
a non-constant pointer to constant data */

#include<iostream.h>

int main( )

{

char strng[] = “Adams”;

char *sPtr;

sPtr = &strng[0];

cout << “\nThe string is: \n”;

for( ; *sPtr != ‘\0’; sPtr++)

cout << *sPtr << ‘ ‘;

return 0;

}
```

The **output** of the above program:

The string is:

A d a m s

Note: The name of a string by itself is equivalent to the base address of that string.

Passing Structures as Parameters

Complete copies of all members of a structure can be passed to a function by including the name of the structure as an argument to the called function.

Example

```
#include <iostream.h>

struct Employee // declare a global type

{

int idNum;

double payRate;

double hours;

};

double calcNet(Employee); // function prototype

int main()

{

Employee emp = {6782, 8.93, 40.5};

double netPay;
```

```
netPay = calcNet(emp); // pass by value

cout << "The net pay for employee "

<< emp.idNum << " is $" << netPay << endl;

return 0;

}

double calcNet(Employee temp) // temp is of data

// type Employee

{

return (temp.payRate * temp.hours);

}
```

The **output** is:

The net pay for employee 6782 is $361.665

In the above program, the function call

```
calcNet(emp);
```

passes a copy of the complete emp structure to the function calcNet( ). The parameter passing mechanism here is call-by-value.

An alternative to the pass-by-value function call, we can pass a structure by **passing a pointer**. The following example shows how to pass a structure by passing a pointer.

Example

```
#include <iostream.h>
```

```
struct Employee // declare a global type
{
int idNum;
double payRate;
double hours;
};
double calcNet(Employee *); //function prototype
int main()
{
Employee emp = {6782, 8.93, 40.5};
double netPay;
netPay = calcNet(&emp); // pass an address
cout << "The net pay for employee "
<< emp.idNum << " is $" << netPay << endl;
return 0;
}
double calcNet(Employee* pt) //pt is a pointer
{ //to a structure of Employee type
return (pt->payRate * pt->hours);
}
```

The **output** is:

The net pay for employee 6782 is $361.665

The Typedef Declaration Statement

The **typedef** declaration statement permits us to construct alternate names for an existing C++ data type name. The syntax of a typedef statement is:

```
typedef data-type new-type-name
```

For example, the statement:

```
typedef float REAL;
```

make the name REAL a synonym for float. The name REAL can now be used in place of the term float anywhere in the program after the synonym has been declared.

The definition

```
REAL val;
```

is equivalent to

```
float val;
```

Example: Consider the following statement:

```
typedef struct
{
char name[20];
int idNum;
} EMPREC;
```

The declaration

```
EMPREC employee[75];
```

is equivalent to

```
struct
{
char name[20];
int idNum;
} employee[75];
```

Example: Consider the following statement:

```
typedef double* DPTR;
```

The declaration:

```
DPTR pointer1;
```

is equivalent to

```
double* pointer1;
```

Introduction to Classes
Now we begin our introduction to object-oriented programming in C++. Why have we deferred object-oriented programming in C++ until this chapter? The answer is that the objects we will build will be composed in part of structured program pieces, so we need to establish a basis in structured programming first. Let us briefly explain some key concepts and terminology of object orientation. Object-oriented programming (OOP) encapsulates data (attributes) and functions (behavior) into packages called classes; the data and functions of a class are intimately tied together. A class is like a blueprint. Out of a blueprint, a builder can build a house. Out of a class, a programmer can create an object. One blueprint can be reused many times to make many objects of the same class. Classes have the property of information hiding. This means that although class objects may know how to communicate with one another across well-defined interfaces, classes normally are not allowed to know how other classes are implemented – implementation details are hidden within the classes themselves.

Classes

In C++ programming, classes are structures that contain variables along with functions for manipulating that data.

The functions and variables defined in a class are referred to as class members.

Class variables are referred to as **data members**, while class functions are referred to as **member functions**.

Classes are referred to as user-defined data types or programmer-defined data types because you can work with a class as a single unit, or objects, in the same way you work with variables.

When you declare an object from a class, you are said to be instantiating an object.

The most important feature of C++ programming is class definition with the class keyword. You define classes the same way you define structures, and

you access a class's data members using the **member selection operator**.

Example:

```
class Time {

public:

Time();

void setTime( int, int, int );

void printMilitary();

void printStandard();

private:

int hour;

int minute;

int second;

};
```

Once the class has been defined, it can be used as a type in object, array and pointer definitions as follows:

"`Time sunset, // object of type Times`"

"`ArOfTimes[5], // array of Times objects`"

"`*ptrTime; // pointer to a Times objects`"

The class name becomes a new type specifier. There may be many objects of a class, just as there may be many variables of a type such as int. The programmer can create new class types as needed. This is one reason why C++ is said to be an extensible language.

Information Hiding

The principle of **information hiding** states that any class members that other programmers, or clients, do not need to access or know about should be hidden.

Many programmers prefer to make all of their data member private in order to prevent clients from accidentally assigning the wrong value to a variable or from viewing the internal workings of their programs.

Access Specifiers

Access specifiers control a client's access to data members and member functions. There are four levels of access specifiers: public, private, protected, and friend.

The **public** access specifier allows anyone to call a class's function member or to modify a data member.

The **private** access specifier is one of the key elements in information hiding since it prevents clients from calling member functions or accessing data members.

Both public and private specifiers have what is called **class scope**: class members of both access types are accessible from any of a class's member functions.

Example:

```
class Time {

public:

Time();

void setTime( int, int, int );
```

```
void printMilitary();

void printStandard();

private:

int hour;

int minute;

int second;

};
```

Note that the data members hour, minute, and second are preceded by the private member access specifier. A class' private data members are normally not accessible outside the class. The philosophy here is that the actual data representation used within the class is of no concern to the class' clients. In this sense, the implementation of a class is said to be hidden from its clients. Such information hiding promotes program modifiability and simplifies the client's perception of a class.

You can depict the classes graphically in a class diagram as below.

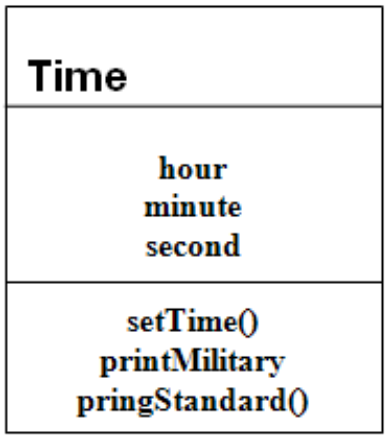


Diagram of class
Time

The diagram shown above follows the format of the Unified Modeling Language (UML). Each class is represented by a box, with the class name in the top portion of the box, any data members that you care to describe in the middle portion of the box, and the member functions (the functions that belong to this object, which receive any messages you send to that object) in the bottom portion of the box.

Interface and Implementation Files

Although the first step in information hiding is to assign private access specifiers to class members, private access specifiers only designate which class members a client is not allowed to call or change. Private access specifiers do not prevent clients from seeing class code. To prevent clients from seeing the details of how your code is written, you place your class's interface code and implementation code in separate files.

The separation of classes into separate **interface** and **implementation** files is considered to be a fundamental software development technique since it allows you to hide the details of how your classes are written and makes it easier to modify programs.

The **interface** code refers to the data member and function member declarations inside a class's braces. Interface code does not usually contain definitions for function members, nor does it usually assigns values to the data members. You create interface code in a header file with an .h extension.

The **implementation** code refers to a class's function definitions and any code that assigns values to a class's data members. In other words, implementation code contains the actual member functions themselves and

assigns values to data members. You add implementation code to standard C++ source files with an extension of .cpp.

As far as clients of a class are concerned, changes in the class' implementation do not affect the client as long as the class' interface originally provided to the client is unchanged. All that a client needs to know to use the class correctly should be provided by the interface.

Preventing Multiple Inclusion

Large class-based programs are sometimes composed of multiple interface and implementation files. With large program, you need to ensure that you do not include multiple instances of the same header file when you compile the program, since multiple inclusion will make your program unnecessary large.

C++ generates an error if you attempt to compile a program that includes multiple instances of the same header file. To prevent this kind of error, most C++ programmers use the #define preprocessor directive with the #if and #endif preprocessor directives in header files.

The #if and #endif preprocessor directives determine which portions of a file to compile depending on the result of a conditional expression.

The syntax for the #if and #endif preprocessor directives:

```
#if conditional expression

statements to compile;

#endif
```

Example:

```
#if !defined(TIME1_H)

#define TIME1_H
```

```
class Time {
public:
Time();
void setTime( int, int, int );
void printMilitary();
void printStandard();
private:
int hour;
int minute;
int second;
};
#endif
```

Note: Common practice when defining a header file's constant is to use the header file's name in uppercase letters appended with H. For example, the constant for the time1.h header file is usually defined as TIME1_H.

Member Functions

In this section, we learn how to write member functions for a class.

Inline functions

Although member functions are usually defined in an implementation file, they can also be defined in an interface file. Functions defined inside the

class body in an interface file are called **inline functions**.

Example:

```
class Stocks {

public:

double getTotalValue(int iShares, double
dCurPrice){

double dCurrentValue;

iNumShares = iShares;

dCurrentPricePerShare = dCurPrice;

dCurrentValue = iNumShares*dCurrentPricePerShare;

return dCurrentValue;

}

private:

int iNumShares;

double dPurchasePricePerShare;

double dCurrentPricePerShare;

};
```

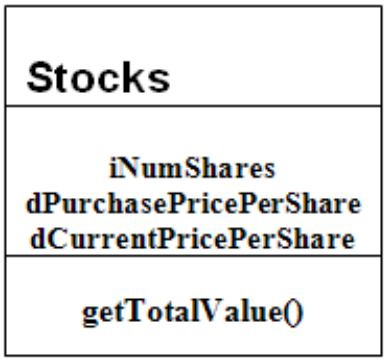


Diagram of class stock

Member functions in Implementation File

Member function definitions are always placed in the implementation file.

In the example below, for the class Stocks, the definition of the member function getTotalValue is placed in the source-code file stocks.cpp in which the main program is also included.

Example

//stocks.h ---------- interface section

```
#if !defined(STOCKS_H)

#define STOCKS_H

class Stocks{

public:
```

```
double getTotalValue(int iShares, double
dCurPrice);

private:

int iNumShares;

double dPurchasePricePerShare;

double dCurrentPricePerShare;

};

#endif
```

// stocks.cpp ---------- implementation section

```
#include “stocks.h”

#include<iostream.h>

double Stocks::getTotalValue(int iShares, double
dCurPrice){

double dCurrentValue;

iNumShares = iShares;

dCurrentPricePerShare = dCurPrice;

dCurrentValue = iNumShares*dCurrentPricePerShare;

return dCurrentValue;

}

int main(){

Stocks stockPick;
```

```
cout << stockPick.getTotalValue(200, 64.25) << endl;

return 0;

}
```

Output of the above program:

12850

Note: The format of member functions included in the implementation section is as follows:

```
return-type Class-name::functionName(parameter-list)

{

function body

}
```

In order for your class to identify which functions in an implementation section belong to it, you precede the function name in the function definition header with the class name and the **scope resolution operator** (::).

Access Functions

Access to a class' private data should be carefully controlled by the use of member functions, called **access functions**. For example, to allow clients to read the value of private data, the class can provide a get function.

To enable clients to modify private data, the class can provide a set function. Such modification would seem to violate the notion of private data. But a set member function can provide data validation capabilities

(such as range checking) to ensure that the value is set properly. A set function can also translate between the form of data used in the interface and the form used in the implementation.

A get function need not expose the data in “raw” format; rather, the get function can edit data and limit the view of the data the client will see.

Example

// time1.h

```
#if !defined(TIME1_H)

#define TIME1_H

class Time {

public:

Time(); // constructor

void setTime( int, int, int ); // set hour,
minute, second

void printMilitary(); // print military time
format

void printStandard(); // print standard time
format

private:

int hour;

int minute;

int second;
```

```
};

#endif;
```

// time1.cpp

```
#include “time1.h”

#include <iostream.h>

// Time constructor initializes each data member
to zero.

// Ensures all Time objects start in a consistent
state.

Time::Time() {

hour = minute = second = 0;

}

void Time::setTime( int h, int m, int s )

{

hour = ( h >= 0 && h < 24 ) ? h : 0;

minute = ( m >= 0 && m < 60 ) ? m : 0;

second = ( s >= 0 && s < 60 ) ? s : 0;

}

void Time::printMilitary()

{

cout << ( hour < 10 ? "0" : "" ) << hour << ":"
```

```
<< ( minute < 10 ? "0" : "" ) << minute;

}

void Time::printStandard()

{

cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour
% 12 )

<< ":" << ( minute < 10 ? "0" : "" ) << minute

<< ":" << ( second < 10 ? "0" : "" ) << second

<< ( hour < 12 ? " AM" : " PM" );

}

// Driver to test simple class Time

int main()

{

Time t; // instantiate object t of class Time

cout << "The initial military time is ";

t.printMilitary();

cout << "\nThe initial standard time is ";

t.printStandard();

t.setTime( 13, 27, 6 );

cout << "\n\nMilitary time after setTime is ";
```

```
t.printMilitary();

cout << "\nStandard time after setTime is ";

t.printStandard();

t.setTime( 99, 99, 99 ); // attempt invalid
settings

cout << "\n\nAfter attempting invalid settings:"

<< "\nMilitary time: ";

t.printMilitary();

cout << "\nStandard time: ";

t.printStandard();

cout << endl;

return 0;

}
```

The **output** of the above program:

The initial military time is 00:00

The initial standard time is 12:00:00 AM

Military time after setTime is 13:37

Standard time after setTime is 1:27:06 PM

After attempting invalid settings:

Military time: 00:00

Standard time: 12:00:00 AM

In the above example, for the class Time we can see that member functions printStandard and printMilitary are two get function and member function setTime is a set function.

Constructor Functions

A **constructor function** is a special function with the same name as its class. This function is called automatically when an object from a class is instantiated.

You define and declare constructor functions the same way you define other functions, although you do not include a return type since constructor functions do not return values.

Example:

```
class Payroll{

public:

Payroll( ){ // constructor function

dFedTax = 0.28;

dStateTax = 0.05;

};

private:

double dFedTax;

double dStateTax;

}
```

You also include just a function prototype in the interface file for the constructor function and then create the function definition in the implementation file.

```
Payroll::Payroll( ){ // constructor function

dFedTax = 0.28;

dStateTax = 0.05;

};
```

Example

```
#include <iostream.h>

#include <iomanip.h>

// class declaration section

class Date

{

private:

int month;

int day;

int year;

public:

Date(int = 7, int = 4, int = 2001); // constructor
with default values

};
```

```
// implementation section

Date::Date(int mm, int dd, int yyyy) //
constructor

{

month = mm;

day = dd;

year = yyyy;

cout << "Created a new data object with data
values "

<< month << ", " << day << ", " << year << endl;

}

int main()

{

Date a; // declare an object

Date b; // declare an object

Date c(4,1,2002); // declare an object

return 0;

}
```

The **output** of the above program:

Created a new data object with data values 7, 4, 2001

Created a new data object with data values 7, 4, 2001

Created a new data object with data values 4,1, 2002

Default constructor refers to any constructor that does not require any parameters when it is called.

In the above example, the prototype Date(int = 7, int = 4, int = 2001) is valid for a default constructor. Here, each argument has been given a default value. Then an object can be declared as type Date without supplying any further arguments.

Although any legitimate C++ statement can be used within a constructor function, such as the cout statement used in above example, it is best to keep constructors simple and use them only for initialization purposes.

Structures vesus Classes

Similarities between Structures and Classes

1. Both can be used to model objects with different attributes represented as data members (also called fields or instance varibles). They can thus be used to process non-homogeneous data sets.
2. They have essentially the same syntax.

Differences between Structures and Classes

1. Member of a structure by default are public. Member of a class by default are private unless explicitly declared to be public
2. A structure consists of only data elements while a class consists of not only data elements but also functions (operations) which are operated on the data elements.

Dynamic Memory Allocation with Operators New and Delete

The **new** and **delete** operators provides a nice means of performing dynamic memory allocation (for any built-in or user-defined type). Consider the following code

```
TypeName *typeNamPtr;
```

```
typeNamePtr = new TypeName;
```

The new operator automatically creates an object of the proper size, calls the constructor for the object and returns a pointer of the correct type.

To destroy the object and free the space for this object in C++ you must use the delete operator as follows:

```
delete typeNamePtr;
```

For built-in data types, we also can use the new and delete operators.

Example 1:

```
int *pPointer;
```

```
pPointer = new int;
```

Example 2:

```
delete pPointer;
```

Example 3: A 10-element integer array can be created and assigned to arrayPtr as follows:

```
int *arrayPtr = new int[10];
```

This array is deleted with the statement

```
delete [] arrayPtr;
```

Stack versus heap

A **stack** is a region of memory where applications can store data such as local variables, function calls, and parameter information.

The programmers have no control over the stack. C++ automatically handles placing and removing data to and from stack.

The **heap** or free store, is an area of memory that is available to application for storing data whose existence and size are not known until run-time.

Notice that when we use **new** operator, we can allocate a piece of memory on the heap and when we use **delete** operator, we can deallocate (free) a piece of memory on the heap. In other words, we can manage the memory allocation on the heap explicitly through new and delete operators.

The syntax for using the new operator is

```
pointer = new data_type;
```

For example, to declare an int pointer iPointer that points to a heap variable, you use the following statements:

```
int* iPointer;
```

```
iPointer = new int;
```

The syntax for using the delete operator is

```
delete pointer_name;
```

For example, to delete the heap memory pointed to by the iPointer pointer, you use the statement delele iPointer;.

Deleting the contents of an array stored on the heap also requires a slightly different syntax. You must append two brackets to the delete keyword using the syntax delete[] array_name; .

Notice that the **delete** operator does not delete the pointer itself. Rather, it deletes the contents of the heap memory address pointed to by a pointer variable. You can reuse the pointer itself after calling the delete operator. The pointer still exists and points to the same heap memory address that it did before calling the delete operator.

Example

```
#include<iostream.h>
int main( )
{
double* pPrimeInterest = new double;
*pPrimeInterest = 0.065;
cout << “The value of pPrimeInterest is: “
<< *pPrimeInterest << endl;
cout << “The memory address of pPimeInterest is:”
<< &pPrimeInterest << endl;
delete pPrimeInterest;
*pPimeInterest = 0.070;
cout << “The value of pPrimeInterest is: “
<< *pPrimeInterest << endl;
cout << “The memory address of pPrimeInterest is:
“
<< &pPrimeInterest << endl;
return 0;
}
```

The **output** of the above program:

The value of pPrimeInterest is: 0.065

The memory address of pPrimeInterest is: 0x0066FD74

The value of pPrimeInterest is: 0.070

The memory address of pPrimeInterest is: 0x0066FD74.

Note: The above program declares the pPrimeInterest pointer on the heap and assigns to it a value of 0.065. Then the delete operator deletes the heap address that stores the value of 0.065. Finally, a new value is added to the heap address. You can see that after the delete statement executes, the pPimeInterest pointer still point to the same memory address.

Example

In the following program, we can create some objects of the class Stocks on the stack or on the heap and then manipulate them.

```
#include<iostream.h>

class Stocks{

public:

int iNumShares;

double dPurchasePricePerShare;

double dCurrentPricePerShare;

};

double totalValue(Stocks* pCurStock){

double dTotalValue;

dTotalValue = pCurStock-
>dCurrentPricePerShar*pCurStock->iNumShares;
```

```
return dTotalValue;

}

int main( ){

//allocated on the stack with a pointer to the
stack object

Stocks stockPick;

Stocks* pStackStock = &stockPick;

pStackStock->iNumShares = 500;

pStackStock-> dPurchasePricePerShare = 10.785;

pStackStock-> dCurrentPricePerShare = 6.5;

cout << totalValue(pStackStock) << endl;

//allocated on the heap

Stocks* pHeapStock = new Stocks;

pHeapStock->iNumShares = 200;

pHeapStock-> dPurchasePricePerShare = 32.5;

pHeapStock-> dCurrentPricePerShare = 48.25;

cout << totalValue(pHeapStock) << endl;

return 0;

}
```

The **output** of the above program:

3250

9650

In the above program, the new operator in the statement:

```
Stocks* pHeapStock = new Stocks;
```

invokes the constructor of the Stocks class to create a Stocks object on the heap and returns a pointer which is assigned to the pointer variable pHeapStock.

Note:

1. The totalValue() function is not a function member of the Stocks class. Rather, it is a function that is available to the entire program.

2. When declaring and using pointers and references to class objects, follow the same rules as you would when declaring and using pointers and references to structures. You can use the **indirect member selection operator** (->) to access class members through a pointer to an object either on stack or on the heap.

As we will see, using new and delete offers other benefits as well. In particular, new invokes the constructor and delete invokes the class'destructor.

Pointers as Class Members

A class can contain any C++ data type. Thus, the inclusion of a pointer variable in a class should not seem surprising.

In some cases, pointers as class members are advantageous. For example, assume that in the class Book we need to store a book title. Rather than using a fixed length character array as a data member to hold each book title, we could include a pointer member to a character array and then allocate the correct size array for each book title as it is needed.

Example

```
#include <iostream.h>

#include <string.h>

// class declaration

class Book

{

private:

char *title; // a pointer to a book title

public:

Book(char * = NULL); // constructor with a default value

void showtitle(); // display the title

};

// class implementation

Book::Book(char *strng)

{

title = new char[strlen(strng)+1]; // allocate memory

strcpy(title,strng); // store the string

}

void Book::showtitle()
```

```
{

cout << title << endl;

return;

}

int main()

{

Book book1("DOS Primer"); // create 1st title

Book book2("A Brief History of Western
Civilization"); // 2nd title

book1.showtitle(); // display book1's title

book2.showtitle(); // display book2's title

return 0;

}
```

The **output** of the above program:

DOS Primer

A Brief History of Western Civilization

The body of the Book() constructor contains two statements. The first statement performs two taks: First, the statement allocates enough storage for the length of the name parameter plus one to accommodate the end-of-string null character, '\n'. Next, the address of the first allocated character position is assigned to the pointer variable title.

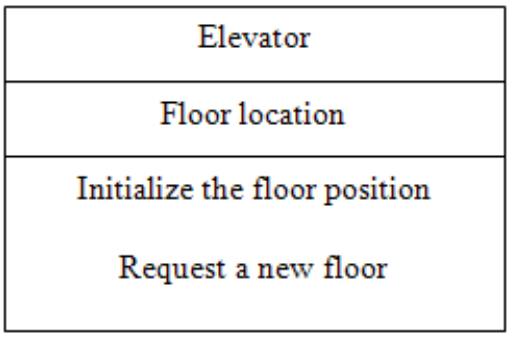
Focus on Problem Solving

Problem: Constructing an Elevator Object

In this application, you are required to simulate the operation of an elevator. Output is required that describes the current floor on which the elevator is stationed or is passing. Additionally, an internal elevator button that is pushed as a request to move to another floor should be provided. The elevator can travel between the first and fifteenth floors of the building in which it is situated.

Analyze the Problem

For this application, we have one object, an elevator. The only attribute of interest is its location. A rider can request a change in the elevator's position (state). Additionally, we must be able to establish the initial floor position when a new elevator is put in service. Figure below illustrates an object diagram that includes both the required attributes and operations.



Object diagram

Figure 7.3 An Elevator Class Diagram

Develop a Solution

For this application, the location of the elevator, which corresponds to its current floor position, can be represented by an integer member variable whose value ranges between 1 and 15. The variable is named currentFloor. The value of currentFloor effectively represents the current state of the elevator. The services that we provide for changing the state of the elevator are an initialization function to set the initial floor position when a new elevator is put in service and a request function to change the elevator's position (state) to a new floor. Putting an elevator in service is accomplished by declaring a single class instance and requesting a new floor position is equivalent to pushing an elevator button.

The response to the elevator button should be as follows:

```
If a request is made for either a nonexistent
floor or the current floor,

Do nothing

Else if the request is for a floor above the
current floor,

Display the current floor number

While not at the designated floor

Increment the floor number

Display the new floor number

End while

Display the ending floor number

Else

Display the current floor number
```

```
While not at the designated floor
Decrement the floor number
Display the new floor number
End while
Display the ending floor number
Endif
```

Coding the Solution

From the design, a suitable class declaration is:

//class declaration

```
class Elevator
{
private:
int currentFloor;
public:
Elevator(int = 1); //constructor
void request(int);
};
```

The two declared public member functions, Elevator() and request(), are used to define the external services provided by each Elevator object. The Elevator() function becomes a constructor function that is automatically

called when an object of type Elevator is created. We use this function to initialize the starting floor position of the elevator. The request() function is used to alter its position. To accomplish these services, a suitable class implementation section is:

// class implementation section

```
Elevator::Elevator(int cfloor)

{

currentFloor = cfloor;

}

void Elevator::request(int newfloor)

{

if (newfloor < 1 || newfloor > MAXFLOOR ||
newfloor == currentFloor)

; // doing nothing

else if (newfloor > currentFloor) // move elevator
up

{

cout << “\nStarting at floor “ << currentFloor <<
endl;

while (newfloor > currentFloor)

{

currentFloor++;
```

```
cout << " Going up – now at floor " <<
currentFloor << endl;

}

cout << " Stopping at floor " << currentFloor <<
endl;

}

else // move elevator down

{

cout << "\nStarting at floor " << currentFloor <<
endl;

while (newfloor < currentFloor)

{

currentFloor--;

cout << " Going down – now at floor " <<
currentFloor << endl;

}

cout << " Stopping at floor " << currentFloor <<
endl;

}

return;

}
```

The constructor function is straightforward. When an Elevator object is declared, it is initialized to the floor specified; if no floor is explicitly given, the default value of 1 is used. For example, the declaration

```
Elevator a(7);
```

Initializes the variable a.currentFloor to 7, whereas the declaration

```
Elevator a;
```

uses the default argument value and initializes the variable a.currentFloor to 1.

The request() function defined in the implementation section is more complicated and provides the class's primary service. Essentially, this function consists of an if-else statement having three parts: If an incorrect service is requested, no action is taken; if a floor above the current position is selected, the elevator is moved up; and if a floor below the current position is selected, the elevator is moved down. For movement up or down, the function uses a while loop to increment or decrement the position one floor at a time and report the elevator's movement using a cout statement. The following program includes this class in a working program.

```
#include <iostream.h>

const int MAXFLOOR = 15;

//class declaration

class Elevator

{

private:

int currentFloor;

public:
```

```
Elevator(int = 1); //constructor

void request(int);

};

// implementation section

Elevator::Elevator(int cfloor)

{

currentFloor = cfloor;

}

void Elevator::request(int newfloor)

{

if (newfloor < 1 || newfloor > MAXFLOOR ||
newfloor == currentFloor)

; // doing nothing

else if (newfloor > currentFloor) // move elevator
up

{

cout << “\nStarting at floor “ << currentFloor <<
endl;

while (newfloor > currentFloor)

{

currentFloor++;
```

```
cout << " Going up – now at floor " <<
currentFloor << endl;

}

cout << " Stopping at floor " << currentFloor <<
endl;

}

else // move elevator down

{

cout << "\nStarting at floor " << currentFloor <<
endl;

while (newfloor < currentFloor)

{

currentFloor--;

cout << " Going down – now at floor " <<
currentFloor << endl;

}

cout << " Stopping at floor " << currentFloor <<
endl;

}

return;

}

int main()
```

```
{
Elevator a; // declare 1 object of type Elevator
a.request(6);
a.request(3);
return 0;
}
```

Test and Correct the Program

Within the main() function, three class function calls are included. The first statement creates an object name a of type Elevator. Since no explicit floor has been given, the elevator begins at floor 1, which is the default constructor argument.

A request is then made to move the elevator to floor 6, which is followed by a request to move to floor 3. The output produced by the program is:

```
Starting at floor 1
  Going Up – now at floor 2
  Going Up – now at floor 3
  Going Up – now at floor 4
  Going Up – now at floor 5
  Going Up – now at floor 6
Stopping at floor 6

Starting at floor 6
  Going Down – now at floor 5
  Going Down – now at floor 4
  Going Down – now at floor 3
Stopping at floor 3
```

Output of program

Note that control is provided by the main() function. This control is sequential, with two calls made to the same object operation, using different argument values. This control is perfectly correct for testing purposes. However, by incorporating calls to request() within a while loop and using the random number function rand() to generate random floor requests, a continuous simulation of the elevator's operation is possible.

Design Techniques of Object-Oriented Programs

The basic requirements of object-oriented programming are evident in even as simple a program as the above program. Before the main() function can be written, a useful class must be constructed. For programs that used objects, the design process is loaded with the requirement that careful consideration of the class – its declaration and implementation – be given. Code contained in the implementation section effectively removes code that would otherwise be part of main()'s responsibility. Thus, any program that uses the object does not have to repeat the implementation details within its main() function. Rather, the main() function is only concerned with sending messages and how the state of the object is retained are not main()'s concern; these details are hidden within the class construction.

Object Manipulation - Inheritance

In the previous chapter, the declaration, initialization, and display of objects were presented. In this chapter, we continue our construction of classes by discussing how to write advanced constructors and destructors. Besides, this chapter also discusses one of C++ most powerful features, inheritance. Inheritance permits the reuse and extension of existing code in a way that ensures the new code does not adversely affect what has already been written.

Advanced Constructors

In the last chapter, we already notice that constructors can provide a mechanism for initializing data members. However, constructors can do more than initializing data members. They can execute member functions and perform other type of initialization routines that a class may require when it first starts.

Parameterized Constructors

Although constructor functions do not return values, they can accept **parameters** that a client can use to pass initialization values to the class.

Example: We can have a constructor function definition in the implementation file as follows:

```
Payroll::Payroll(double dFed, double dState){

dFedTax = dFed;

dStateTax = dState;

};
```

Once you create a parameterized constructor, you have to supply parameters when you instantiate a new object.

Example:

//Payroll.h

```
class Payroll{
public:
Payroll(double, double);
private:
double dFedTax;
double dStateTax;
}
```

//Payroll.cpp

```
#include “Payroll.h
#include <iostream.h>
Payroll::Payroll(double dFred, double dState){
dFedTax = dFed;
dStateTax = dState;
};
int main( ){
Payroll employee; //illegal
……
return 0;
```

```
}
```

Example

The program in this example finds the distance between two points using the pointer to class objects technique.

```
//points

#include <iostream.h>

#include<math.h>

class point {

private:

int x,y;

public:

point( int xnew, int ynew);

inline int getx(){

return(x);

}

inline int gety(){

return(y);

}

double finddist(point a, point b);

};
```

```
point::point(int xnew, ynew) //parameterized
constructor

{

x = xnew;

y = ynew;

}

double point::finddist(point a, point b)

{

double temp;

temp = ((b.y – a.y)*(b.y – a.y) + (b.x – a.x)*(b.x
– a.x));

return(sqrt(temp)):

}

int main()

{

double value;

point aobj(4,3), bobj(0, -1);

value = aobj.finddist(aobj, bobj);

cout << “Distance between two points = “<< value
<< endl;

return 0;
```

```
}
```

The **output** of the above program:

Distance between two points = 5.656855

Constructor functions can be **overloaded**, just like other functions. This means that you can instantiate different versions of a class, depending on the supplied parameters

Being able to overload a constructor function allows you to instantiate an object in multiple ways.

Example:

//Payroll.h

```
class Payroll{

public:

Payroll();

Payroll(double dFed);

Payroll(double dFed, double dState);

private:

double dFedTax;

double dStateTax;

}
```

//Payroll.cpp

```
#include “Payroll.h
```

```
#include <iostream.h>

Payroll::Payroll(){

dFedTax = 0.28;

dStateTax = 0.05;

};

Payroll::Payroll(double dFed){

dFedTax = dFed;

};

Payroll::Payroll(double dFred, double dState){

dFedTax = dFed;

dStateTax = dState;

};

int main( ){

Payroll employeeFL(0.28);

Payroll employeeMA(0.28, 0.0595);

return 0;

}
```

In the above example, the Payroll class has two parameterized constructor functions: one for states that have a state income tax and one for states that do not have a state income tax.

Initialization Lists

Initialization lists, or member initialization lists, are another way of assigning initial values to a class's data members.

An initialization list is placed after a function header's closing parenthesis, but before the function's opening curly braces.

Example: Given the simple constructor that assigns parameter values to the Payroll class.

```
Payroll::Payroll(double dFed, double dState){

dFedTax = dFed;

dStateTax = dState;

};
```

You can use initialization list to rewrite the above constructor function.

```
Payroll::Payroll(double dFed, double dState)

:dFedTax(dFed), dStateTax(dState){

};
```

Parameterized Constructors that Uses Default Arguments

To create a parameterized constructor that uses default arguments, we can put the default values at the constructor prototype.

In the following example, the class Employee has a constructor with the prototype:

```
Employee(const int id = 999, const double hourly =
5.65);
```

This format provides the constructor function with default values for two arguments. When we create an Employee object, the default values in the constructor function prototype are assigned to the class variables.

Example

```
#include<iostream.h>

class Employee{

private:

int idNum;

double hourlyRate;

public:

Employee(const int id = 9999, const double hourly
= 5.65);

void setValues(const int id, const double hourly);

void displayValues();

};

Employee::Employee(const int id, const double
hourly)

{

idNum = id;

hourlyRate = hourly;

}

void Employee::displayValues()
```

```
{

cout<<”Employee #<< idNum<<” rate $”<<

hourlyRate<< “ per hour “<<endl;

}

void Employee::setValues(const int id, const
double hourly)

{

idNum = id;

hourlyRate = hourly;

}

int main(){

Employee assistant;

cout<< “Before setting values with setValues()”<<
endl;

assistant.displayValues();

assistant.setValues(4321, 12.75);

cout<< “After setting values with setValues()”<<
endl;

assistant.displayValues();

return 0;

}
```

The **output** of the above program:

Before setting values with setValues()

Employee #9999 rate $5.65 per hour

After setting values with setValues()

Employee #4321 rate $12.75 per hour

Destructors

A **default destructor** cleans up any resources allocated to an object once the object is destroyed. The default constructor is sufficient for most classes, except when you have allocated memory on the heap.

To delete any heap variables declared by your class, you must write your own destructor function.

You create a destructor function using the name of the class, the same as a constructor function, preceded by a tilde ~. Destructor functions cannot be overloaded. A destructor accepts no parameter and returns no value.

A destructor is called in two ways:

- when a stack object loses scope when the function in which it is declared ends.

- when a heap object is destroyed with the delete operator.

The destructor itself does not actually destroy the object – it performs termination house keeping before the system reclaims the object's memory so that memory may be reused to hold new objects.

Example

//Stocks_02.h

```
class Stocks {
public:
Stocks(char* szName);
~Stocks(); //destructor
void setStockName(char* szName);
char* getStockName();
void setNumShares(int);
int getNumShares(int);
void setPricePerShare(double);
double getPricePerShar();
double calcTotalValue();
private:
char* szStockName;
int iNumShares;
double dCurrentValue;
double dPricePerShare;
};
```

//Stocks.cpp

```
#include “stocks_02.h”
#include <string.h>
```

```
#include <iostream.h>
Stocks::Stocks(char* szName){
szStockName = new char[25];
strcpy(szStockName, szName);
};
Stocks::~Stocks(){
delete[] szStockName;
cout <<”Destructor called” << endl;
}
void Stocks::setNumShares(int iShares){
iNumShares = iShares;
}
int Stocks::getNumShares()const{
return iNumShares;
}
void Stocks::setPricePerShare(double dPrice){
dPricePerShare = dPrice;
}
int Stocks::getPricePerShare() const{
return dPricePerShare;
```

```
}
void Stocks::setStockName(char* szName){
strcpy(szStockName, szName);
}
char* Stock::getStockName() const{
return szStockName;
}
double Stocks::calcTotalValue(){
dCurrentValue = iNumShares*dPricePerShare;
return dCurrentValue;
}
int main(){
Stocks stockPick1(“Cisco”);
stockPick1.setNumShares(100);
stockPick1.setPricePerShare(68.875);
Stocks* stockPick2 = new Stocks(“Lucent”); //heap
object
stockPick2->setNumShares(200);
stockPick2->setPricePerShare(59.5);
cout << “The current value of your stock in “
```

```
<< stockPick1.getStockName() << " is $"
<< stockPick1.calcTotalValue()
<< "." << endl;
cout << "The current value of your stock in "
<< stockPick2->getStockName() << " is $"
<< stockPick2->calcTotalValue()
<< "." << endl;
return 0;
}
```

The **output** of the above program:

The current value of your stock in Cisco is $6887.5

The current value of your stock in Lucent is $11900

Destructor called.

Notice that in the above program, the destructor function is called only once. The stockPick1 object calls the destructor when it is destroyed by the main() function going out of scope. The stockPick2 object does not call the destructor since it is declared on the heap and must be deleted manually.

To delete the stockPick2 object manually, add the statement delete stockPick2; to the main() function as in the following program:

```
int main(){
Stocks stockPick1("Cisco");
stockPick1.setNumShares(100);
```

```
stockPick1.setPricePerShare(68.875);

Stocks* stockPick2 = new Stocks(“Lucent”); //heap
object

stockPick2->setNumShares(200);

stockPick2->setPricePerShare(59.5);

cout << “The current value of your stock in “

<< stockPick1.getStockName() << “ is $”

<< stockPick1.calcTotalValue()

<< “.” << endl;

cout << “The current value of your stock in “

<< stockPick2->getStockName() << “ is $”

<< stockPick2->calcTotalValue()

<< “.” << endl;

delete stockPick2;

return 0;

}
```

The **output** of the above program:

The current value of your stock in Cisco is $6887.5

The current value of your stock in Lucent is $11900

Destructor called.

Destructor called.

Constant Objects

If you have any type of variable in a program that does not change, you should always use the **const** keyword to declare the variable as a constant.

To declare an object as constant, place the const keyword in front of the object declaration.

Example:

```
const Date currentDate;
```

Note: Constant data members in a class can not be assigned values using a standard assignment statement. Therefore, you must use an initialization list to assign initial values to constant data members.

Example:

//Payroll.h

```
class Payroll{

public:

Payroll();

private:

const double dFedTax;

const double dStateTax;

};
```

//Payroll.cpp

```
#include “Payroll.h

#include <iostream.h>

Payroll::Payroll()

:dFedTax(0.28), dStateTax(0.05){

};
```

In contrast, the following code raises several compiler errors since constant data members must be initialized in an initialization list:

Example:

//Payroll.h

```
class Payroll{

public:

Payroll();

private:

const double dFedTax;

const double dStateTax;

};
```

//Payroll.cpp

```
#include “Payroll.h”

#include <iostream.h>

Payroll::Payroll( ){
```

```
dFedTax = 0.28; //illegal

dStateTax = 0.05; //illegal

};
```

Constant Functions

Another good programming technique is to use the **const** keyword to declare **get functions** as constant function. Get functions are function members which do not modify data members.

The const keyword makes your programs more reliable by ensuring that functions that are not supposed to modify data cannot modify data. To declare a function as constant, you add the const keyword after a function's parentheses in both the function declaration and definition.

//Payroll.h

```
double getStateTax(Payroll* pStateTax) const;
```

//Payroll.cpp

```
double Payroll::getStateTax(Payroll* pStateTax)
const {

return pStateTax->dStateTax;

};
```

Inheritance

Inheritance is a form of software reusability in which new classes are created from existing classes by absorbing their attributes and behaviors, and overriding or embellishing these with capabilities the new classes require. Software reusability saves time in programming development. It

encourages the reuse of proven and debugged high quality software, thus reducing problems after a system becomes functional.

Basic Inheritance

Inheritance refers to the ability of one class to take on the characteristics of another class.

Often, classes do not have to be created “from scratch”. Rather, they may be derived from other classes that provide attributes and behaviors the new classes can use. Such software reuse can greatly enhance programmer productivity.

Base Classes and Derived Classes

When you write a new class that inherits the characteristics of another class, you are said to be deriving or subclassing a class.

An inherited class is called the **base class**, or **superclass** and the class that inherits a base class is called a **derived class** or **subclass**.

A class that inherits the characteristics of a base class is said to be extending the base class since you often extend the class by adding your own class members.

When a class is derived from a base class, the derived class inherits all of the base class members and all of its member functions, with the exception of:

- constructor functions
- copy constructor functions
- destructor functions
- overloaded assignment (=) functions

A derived class must provide its own implementation of these functions.

Consider a class originally developed by a company to hold an individual's data, such as an ID number and name. The class, named Person, contains three fields and two member functions.

```
class Person

{

private:

int idnum;

char lastName[20];

char firstName[15];

public:

void setFields(int, char[], char[]);

void outputData( );

};

void Person::setFields(int num, char last[], char
first[])

{

idnum = num;

strcpy(lastName, last);

strcpy(firstName, first);

}

void Person::outputData( )
```

```
{

cout<< “ID#”<< idnum << “ Name: “<< firstName << “ “<< lastName << endl;

}
```

.

The company that uses the Person class soon realizes that the class can be used for all kinds of individuals – customers, full-time employees, part-time employees, and suppliers all have names and numbers as well. Now, the company wants to define the Customer class which inherits the members of the Person class.

The class header declaration for a derived class Customer which inherits the characteristics of the Person class is as follows:

```
class Customer: public Person

{

……// other statements go here

}
```

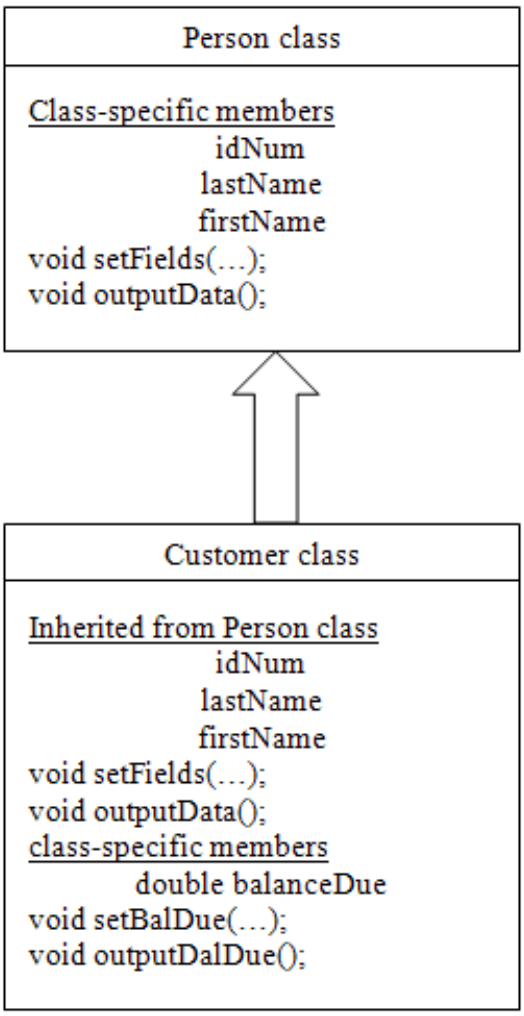
The access modifiers and base class names following the colon in a class’s header declaration statement are called the base list. Here, the public inheritance is used since it is most common.

The Customer class contains all the members of Person because it inherits them. In other words, every Customer object has an idNum, lastName and firstName, just as a Person object does. Additionally, you can define the Customer class to include an additional data member, balanceDue, and two more functions: setBalDue() and outputBalDue().

The base class Person and the derived class Customer can be graphically represented as in the figure below.

The arrow in the above class diagram points from the derived class to the base

class.



Base class and derived class

Once you extend a base class, you can access its class member directly through objects instantiated from the derived class.

Example:

```
int main(){

customer cust;

cust.setField(123, “Richard”, “Leakey”);

cust.outputData( );

return 0;

}
```

The object cust which belongs to the class Customer can call the member functions setFields() and outputData() that belongs to the base class Person.

Example

```
#include<iostream.h>

#include<string.h>

class Person

{

private:

int idnum;

char lastName[20];

char firstName[15];

public:

void setFields(int, char[], char[]);
```

```
void outputData( );

};

void Person::setFields(int num, char last[], char
first[])

{

idnum = num;

strcpy(lastName, last);

strcpy(firstName, first);

}

void Person::outputData( )

{

cout<< “ID#”<< idnum << “ Name: “<< firstName << “
“<< lastName << endl;

}

class Customer:public Person

{

private:

double balanceDue;

public:

void setBalDue;

void outputBalDue( );
```

```
};
void Customer::setBalDue(double bal)
{
balanceDue = bal;
}
void Customer::outputBalDue()
{
cout<< “Balance due $” << balanceDue<< endl;
}
int main()
{
Customer cust;
cust.setFields(215, “Santini”, “Linda”);
cust.outputData();
cust.setBalDue(147.95);
cust.outputBalDue();
return 0;
}
```

The **output** of the above program:

ID#215 Name: Linda Santini

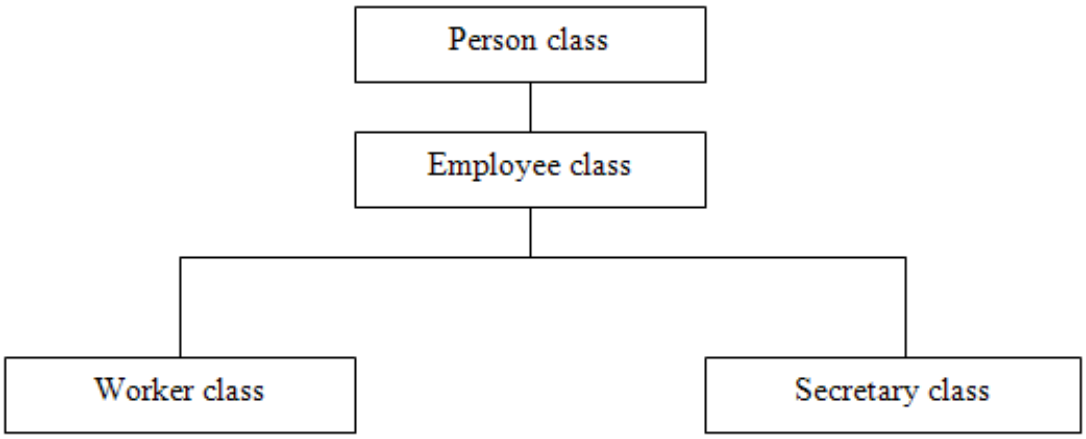
Balance due $147.95

Of course, a Customer object can use its own class' member functions, setBalDue() and outputBalDue(). Additionally, it can use the Person functions, setFields() and outputData(), as if they were its own.

Class Hierarchy

Derived classes themselves can serve as base classes for other derived classes. When you build a series of base classes and derived classes, the chain of inherited classes is known as a class hierarchy.

Figure below shows a simple inheritance hierarchy. A typical company has hundreds of persons. An important subset of these persons is a set of employees. Employees are either workers or secretaries.



Person class hierarchy

Each class in a class hierarchy cumulatively inherits the class members of all classes that precede it in the hierarchy chain.

A class that directly precedes another class in a class hierarchy, and is included in the derived class's base list, is called the direct base class.

A class that does not directly precede another class in a class hierarchy, and that not included in the derived class's base list, is called the indirect base class.

Access Specifiers and Inheritance

Even though a derived class inherits the class members of a base class, the base class's members are still bound by its access specifiers.

Private class members in the base class can be accessed only the base class's member functions.

For example, the idNum data member in the Person class is private. If you write the following member function in the Customer class, which attempts to directly access to the idNum data member, you will get a compiler error.

```
void Customer::outputBalDue(){

cout<< " ID #"<< idNum<< " Balance due $"<< balanceDue<<endl;

}
```

Instead, to access the idNum data member you must call the Person class's outputData() member function, which is public. Alternatively, you can declare the idNum data member with the protected access specifier.

The protected access modifier restricts class member access to

1. the class itself
2. to classes derived from the class, or

The following code shows a modified version of the Person class declaration in which the private access modifier has been changed to protected.

Example:

```
class Person {

protected:

int idNum;

char lastName[20];

char firstName[15];

public:

void setFields(int num, char last[], char
first[]);

void outputData();

};
```

A member function in Customer class that attempts to directly access to the idNum data member will work correctly since the Customer class is a derived class of the Person class and the idNum data member is now declared as protected.

Overriding Base Class Member Functions

Derived classes are not required to use a base class's member functions. You can write a more suitable version of a member function for a derived class when necessary. Writing a member function in a derived class to replace a base class member function is called **function overriding**.

To override a base class function, the derived member function declaration must exactly match the base class member function declaration, including the function name, return type and parameters.

To force an object of a derived class to use the base class version of an overridden function, you precede the function name with the base class name and the scope resolution operator using the syntax:

```
object.base_class::function();
```

Example

In the following code, the base class Person and the derived class Employee have their own function member with the same name setFields().

```
#include<iostream.h>

#include<string.h>

class Person

{

private:

int idnum;

char lastName[20];

char firstName[15];

public:

void setFields(int, char[], char[]);

void outputData( );

};

void Person::setFields(int num, char last[], char first[])

{
```

```
idnum = num;

strcpy(lastName, last);

strcpy(firstName, first);

}

void Person::outputData( )

{

cout<< “ID#”<< idnum << “ Name: “<< firstName << “ “<< lastName << endl;

}
```

// derived class

```
class Employee:public Person

{

private:

int dept;

double hourlyRate;

public:

void setFields(int, char[], char[], int, double);

};

void Employee::setFields(int num, char last[], char first[], int dept, double sal)

{
```

```
Person::setFields(num, last, first);
dept = dep;
hourlyRate = sal;
}
int main()
{
Person aPerson;
aPerson.setFields(123, “Kroening”, “Ginny”);
aPerson.outputData();
cout<< endl<<endl;
Employee worker;
worker.Person::setFields(777,”John”, “Smith”);
worket.outputData();
worker.setFields(987,”Lewis”, “Kathy”, 6, 23.55);
worker.outputData();
return 0;
}
```

The **output** of the above program:

ID # 123 Name: Ginny Kroening

ID # 777 Name: John Smith

ID # 987 Name: Kathy Lewis

In the above program, when you use the Employee class to instantiate an Employee object with a statement such as Employee worker; and then the statement worker.setFields(); uses the child function with the name setFields(). When used with a child class object, the child class function overrides the parent class version. On the other hand, the statement worker.outputData(); uses the parent class function because no child class function has the name outputData().

Overriding a base class member functions with a derived member function demonstrates the concept of polymorphism. Recall that polymorphism permits the same function name to take many forms.

Constructors and Destructors in Derived Classes

When you derive one class from another class, you can think of any instantiated object of the derived class as having two portions:

- the base class portion and
- the derived class portion.

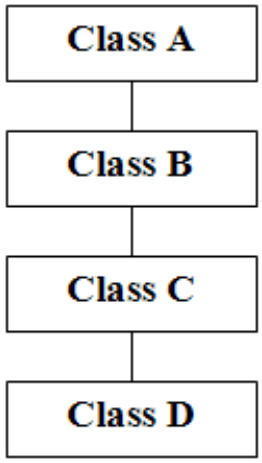
During the instantiating process, the base class portion of the object is instantiated, and then the derived class portion of the object is instantiated.

So, two constructors execute for a single derived class object: the base class constructor and the derived class constructor.

When a derived class object instantiates, constructors begin executing at the top of the class hierarchy. First, the base constructor executes, then any indirect base class's constructors execute. Finally, the derived class' constructor executes.

When an object is destroyed, class destructors are executed in the reverse order. First, the derived class's destructor is called, then the destructors for any indirect base classes, and finally, the destructor for the base class. Figure below illustrates this process using a class hierarchy with four levels.

The order of construction makes sense, since it allows base classes to perform any initialization on class members that may be used by derived classes. And the order of destruction ensures that any base class members required by derived classes are not destroyed until all objects of any derived classes are destroyed first.



Order of Construction
1.Class A
2. Class B
3. Class C
4. Class D

Order of Destruction
1. Class D
2. Class C
3. Class B
4. Class A

Execution of constructors and destructors in a class hierarchy.

Example

```
#include<iostream.h>

#include<string.h>

class Person
```

```
{
private:
int idnum;
char lastName[20];
char firstName[15];
public:
Person();
void setFields(int, char[], char[]);
void outputData( );
};
Person::Person(){
cout << “Base class constructor call “<< endl;
}
void Person::setFields(int num, char last[], char
first[])
{
idnum = num;
strcpy(lastName, last);
strcpy(firstName, first);
}
```

```
void Person::outputData( )

{

cout<< “ID#”<< idnum << “ Name: “<< firstName << “
“<< lastName << endl;

}

class Customer:public Person

{

private:

double balanceDue;

public:

Customer();

void setBalDue;

void outputBalDue( );

};

Customer::Customer(){

cout << “Derived constructor called” << endl;

}

void Customer::setBalDue(double bal)

{

balanceDue = bal;
```

```
}
void Customer::outputBalDue()
{
cout<< “Balance due $” << balanceDue<< endl;
}
int main()
{
Customer cust;
cust.setFields(215, “Santini”, “Linda”);
cust.outputData();
cust.setBalDue(147.95);
cust.outputBalDue();
return 0;
}
```

The **output** of the above program is:

Base class constructor called

Derived class constructor called

ID #215 Name: Linda Santini

Balance due $147.95

The output shows that both the constructor of Person class and the constructor of Customer class involve in creating the object Cust.