Introduction to C++ Programming and Data Structures: Notes

Holger Findling

My CI Group

Introduction to C++ Programming and Data Structures

Table of Contents

Chapter 1: Introduction to C++ Programming 1

Overview 1
Programming Environments 2

Creating a Windows Console Application 3

Chapter 2: Fundamentals in C++ Programming 6

Data Types 7
C/ C++ Statements 7
String Operations 8
Conditional Statements 11
C/ C++ keywords 12
Operator Summary 13

Chapter 3: Control Loops 14

For Loop 14
While Loop 14

Chapter 4: State Machines 16

Switch 16

Chapter 5: Data Structures 17
Chapter 6: Linear Arrays 18

Arrays 18
Vector 23

Chapter 7: Functions 26
Chapter 8: Pointers 31

Pointers 31
Pointers and Functions 32
File Operation 34
Writing to Text File 34
Reading from Text File 35
Writing Data to Binary File 36
Read Data from Binary File 37
Time 38

Chapter 9: Recursion 40

Factorial 40
Power Function 41

Chapter 10: Object Oriented Programming 42

Creating Objects 49

Chapter 11: Inheritance 52
Chapter 12: Linked Lists 53

Example of a Singly Linked List 56
Traversing to the end of the Linked List 59
Inserting an element into the Linked List 60
Deleting an element from the Linked List 61

Chapter 13: Stacks and Queues 63
Chapter 14: Binary Trees 64
The Terminology of Trees 64
Binary Trees 65
Binary Search Trees 66
Binary Expression trees 67
Vector implementation 67
Depth first search for tree in figure #1 68
Breadth first search for tree in figure #1 68
Chapter 15: Graphs 69

Chapter 1: Introduction to C++ Programming

Overview

The initial development of the C programming language evolved around 1969 and 1973. Dennis Ritchie originally designed the language on a UNIX operating system. In 1983 the C programming language was formerly defined by the American National Standards Institute (ANSI). Although the language is not a high level programming language, it gained popularity very fast.

The C programming language served two purposes:

- It provided a vehicle for the programmer to specify actions to be executed at a higher level than assembly.
- It provided a set of concepts for the programmer to use when thinking about what can be done.

The first purpose ideally requires that a language is close to the machine so that all important aspects of a machine are handled simply and efficiently in a way that it is reasonably obvious to the programmer. The C programming language was primarily designed with this in mind.

Bjarne Stroustrup designed the C++ programming language and its first use was realized in July 1983. The name C++ was coined by Rick Mascitti and signifies the evolutionary nature of the changes from C. Bjarne Stroustrup stated that he created the language so that he did not have to program in Assembly or C. Although many new programming languages emerged since 1983, the C++ programming language still dominates in the programming arena.

The main purpose behind developing the C++ programming language was to make writing good programs easier and more pleasant for the individual programmer. C++ is a general programming language with a bias towards system programming and it consists of the following attributes.

- It is better C
- Supports data abstraction
- Supports object-oriented programming
- Supports generic programming

Programming Environments

One of the most popular development environments is the Microsoft .Net Framework. The .Net Framework is a multi-language development environment for building, deploying, and running XML Web services and applications. It is comprised of three parts:

- **Common Language Runtime** – The runtime is responsible for managing memory allocation, starting up and stopping processes and threads, and enforcing security policy.
- **Unified Programming Classes** – The framework provides developers with a unified, object-oriented, and extensible set of class libraries (APIs). All .Net programming languages have access to the framework and the developer can choose the language best suited for a particular task.
- **ASP.net** – ASP builds on the programming classes of the .Net Framework. It provides a Web application model with a set of controls and infrastructure that make it simple to build ASP Web applications.

When software is developed in the UNIX environment a Makefile must be created in order to compile the software. The Makefile is typically created by a senior programmer due to its complexity. An example of a generic Makefile is provided below.

Developing software in the Microsoft Windows operating system environment allows programmers to use the .Net development environment. In this development environment the Makefile is automatically created, and therefore much easier to maintain and modify.

#Makefile

```
CC = g++
INSTALL = ./ -c
CFLAGS = -Wall
SRCS = Main.cpp MyFile_1.cpp MyFile_2.cpp
OBJS = Main.o MyFile_1.o MyFile_2.o
LIBS = -lm
all:    DS
DS:     $(OBJS)
        $(CC) $(CFLAGS) -o $@ $(OBJS) $(LIBS)
install: all
        $(INSTALL) DS $(INSTALL)DS
clean:
        rm -f *.o
```

Compile from the command line:

> make clean
> make Makefile

Creating a Windows Console Application

Start Visual Studios.

- Select: File → New → Project
- Select Templates → Visual C++ → CLR
- Select CLR Console Application
- Enter the Project Name and Location.
  Name: Project1
  Location: C:\
- Check Create directory for solution
- Select: ok

Compile the project after it is created.

- Select: Build → Rebuild Solution

When compiling is completed the Output window should display:

```
1>Build succeeded.
1>
1>Time Elapsed 00:00:09.05
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
```

The Solution Explorer window shows the files that were created for this solution.

Header Files:
resource.h
stdafx.h

Resource Files:
app.ico
app.rc

Source Files:
assemblyInfo.cpp
project1.cpp
stdafx.cpp

ReadMe.txt

Notice, the name of the Solution and the name of the file where function main() resides are identical. In this example main() resides in the file project1.cpp.

Function main()

When creating a Project or Solution the function main() indicates the beginning of the program. All executable code written in the C++ programming language requires the function main().

Function main() can have 3 arguments, argc, argv, and envp.

argc An integer that contains the count of arguments that follow in argv. The count is always equal or greater than 1.

argv An array of strings storing the command-line arguments entered by the user of the program.

argv[0] is the command with which the program is invoked.
argv[1] is the first command-line argument, and so on, until argv [argc], which is always NULL.

Example:

```
void main (int argc, char *argv[ ], char *envp[ ] )
{
  ...
  code;
  ...
  return;
}
```

```
Command line:>  C:\MyProgram  Findling 112223333        <enter>
argc = 3
argv[0] = "C:\Test\MyProgram"
argv[1] = "Findling"
argv[2] = "112223333"
```

When a program is executed it receives two collections of data, arguments and the environment. The environment is obtained using the argument called **envp**.

envp is an array of strings representing the variables set in the user's environment.

In UNIX **environ** and **envp** contain the same data, so **envp** is typically omitted and environ is declared external.

Example:

```
extern char **environ;
void main (int argc, char *argv[ ] )
{
   ...
   code;
   ...
   return;
}
```

Chapter 2: Fundamentals in C++ Programming

Let's create our first program Hello World, example 2.1.

Example 2.1

```
#include "stdafx.h"
#include "stdio.h"

using namespace System;

void main(void)
{
   char buffer[] = "Hello World.";
   printf ("%s", buffer);

   return;
}
```

The program starts with two compiler directives, #include. The # symbol instructs the compiler to perform an action before compiling any additional code. In this example #include directs the compiler to include the files stdafx.h and stdio.h. These two header files contain function definitions that we may need when creating our program. The file stdio.h provides the definition for function printf() which displays our data to the Window console.

After the #include directives follow the namespace declarations. A namespace declaration allows programmers to use particular library items associated with that namespace. A typical namespace declaration is

```
using namespace System;
using namespace std;
```

The line of code *char buffer[] = "Hello World."* creates a variable named buffer and initializes it with the string "Hello World." All string content must be encapsulated with quotes " ". If we know that the content of buffer should not change during program execution, then we should declare it as a constant. Declaring a variable as a constant avoids changing it accidentally. const and char are keywords in the C/C++ programming language.

const char buffer[] = "Hello World.";

Each statement in the C/ C++ programming language must be terminated with a semicolon. An expression becomes a statement when it is followed by a semicolon.

Braces { and } are used to group declarations and statements together into a compound statement or block. Braces that surround the statements of a function are an obvious example. There is no semicolon placed after the closing brace that ends a block.

Data Types

Fundamental data types in the C/C++ programming language are divided into three categories, which are integral types, floating point numbers, and void, shown in Table 2.1. Integral types are comprised of whole numbers, and floating types express numbers with a fractional part. The data type void is typically used to declare a generic pointer. No variable of type void can be declared.

Data types can be declared as signed or unsigned. Declaring a variable as signed int is identical to declaring a variable of type int. Unless a variable is declared unsigned it is assumed to be signed.

```
int ans = 0;
signed int ans = 0;
unsigned int ans = 0;
```

Table 2.1 – Fundamental Data Types

Category	Type	Size	Range
Integral	char	1 byte	-128 to 127
Integral	bool	1 byte	<true, false>
Integral	short	2 bytes	-32,768 to 32,767
Integral	int	4 bytes	-2,147,483,648 to 2,147,483,647
Integral	long	4 bytes	-2,147,483,648 to 2,147,483,647
Integral	long long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Floating	float	4 bytes	3.4 E +/- 38 (7 digits)
Floating	double	8 bytes	1.7 E +/- 308 (15 digits)
Void	void		

C/ C++ Statements

A C++ statement must be terminated with a semicolon. One can place many statements on a single line, each separated by a semicolon. However it is a good coding practice to place each statement on a separate line. It makes the code more readable, therefore easier to understand and debug.

A declaration is an instantiation of a variable of specified data type. At the time of execution of the program the declaration statements do not incur any computational time. For example *int x;* is a declaratory statement and no value is assigned at this time. The memory allocation is determined by the linker and if necessary resolved by the Common Language Runtime (CLR). The assignment statement *int x = 0;* declares the variable x of data type int and assigns the value 0.

There are numerous ways to express an assignment statement, each is equivalent and generates the same result. Suppose we wanted to increment a variable called num. The

statements below increment the variable by 1. In general programmers prefer the shortest style.

```
num = num + 1;
num += 1;
++num;
num++;
```

The statements *++num;* and *num++;* are not identical and care should be taking to use it correctly. In example 2.2 ans is equal 130 and in example #2.3 ans is equal 120. In example 2.2 ans is incremented before the multiplication operator is applied and in example 2.3 num is incremented after the multiplication operator is applied.

Example 2.2

```
int ans = 0;
int num = 12;
ans = ++num * 10;
```

Example 2.3

```
int ans = 0;
int num = 12;
ans = num++ * 10;
```

String Operations

Let's modify program Hello World and not initialize the buffer at the time of the declaration. The initialization of buffer is performed using function strcpy(). Include the header file string.h to use the various string functions.

Example 2.4

```
#include "stdafx.h"
#include "stdio.h"
#include "string.h"

using namespace System;

void main(void)
{
   char buffer[32];
   strcpy (buffer, "Hello World.");

   printf ("%s", buffer);
   return;
}
```

The compiler may generate a warning stating that function strcpy() is deprecated. This implies a newer version of strcpy() is available. By convention the deprecated function can be replaced by appending _s to the function name.

```
Project1.cpp(12): warning C4996: 'strcpy': This function or variable may be unsafe.
Consider using strcpy_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See
online help for details.
```

Modify the code to read:

```
char buffer[32];
strcpy_s (buffer, "Hello World.");
```

The length of a string can be determined using function strlen(). “Hello World.” is 12 characters long.

```
int length;
length = strlen (buffer);
```

Two strings can be concatenated using function strcat_s(). In the example below two constant strings, Agent and Name are joined together. Both strings are separated by a space. When concatenating two strings the programmer must ensure the buffer is large enough to hold the content of both strings. Every string must be terminated by a null terminater ‘\0, which is the last element in the string. Therefore, buffer must be declared at least 12 characters long. It is a programming practice to make the size of the buffer a function of the power of 2. Example char buffer [8, 16, 32, etc …].

Example 2.5

```
void main(void)
{
   const char Agent[] = "Agent";
   const char Name[] = "Smith”;
   char buffer[32];

   strcpy_s(buffer, Agent);
   strcat_s(buffer, " ");
   strcat_s(buffer, Name);

   printf ("%s", buffer);
   return;
}
```

A simpler approach to concatenating multiple strings is to use function sprintf(). Notice the space between the two strings is created by the space between both “%s %s”. Function sprintf() uses the same formatting style as function printf().

Note, function printf() writes to the Window console, sprintf() writes to a string, and fprintf() writes to a text file.

Example 2.6

```
void main(void)
{
   const char Agent[] = "Agent";
   const char Name[] = "Smith";
   char buffer[32];

   sprintf_s (buffer, "%s %s", Agent, Name);

   printf ("%s", buffer);
   return;
}
```

The content of a string can be reversed using function strrev(). The output of buffer is "htimS tnegA".

Example 2.7

```
void main(void)
{
   const char Agent[] = "Agent";
   const char Name[] = "Smith";
   char buffer[32];

   sprintf_s (buffer, "%s %s", Agent, Name);
   strrev(buffer);

   printf ("%s", buffer);
   return;
}
```

Two strings can be compared to determine if they are identical using function strcmp(). The function returns zero if both strings are identical; otherwise a value greater than zero indicates that the first character that does not match has a greater value in the first parameter versus the second. A negative value indicates the opposite. In the example below isEqual is assigned the value 1 since “S” has a greater value than “A”.

```
int isEqual = strcmp(Name, Agent);
```

Typically, a string comparison is performed using a conditional statement. Note, the function strcmp() returns a 0 (false) when both strings are identical and the ! symbol performs a boolean negation. ! false = true.

```
if (!strcmp(Name, Agent)) {
   // The strings match
}
```

Conditional Statements

A conditional statement can assume various syntactical forms shown below. In the case where a conditional statement executes only one statement the opening and closing brackets are optional. Conditional statements alter the flow of executions. If a condition is evaluated true, then the statements associated with the condition are executed; otherwise the statements are skipped.

If

```
if (expression)
    statement;
```

```
if (expression)
{
    statement;
}
```

If-Else

```
if (expression)
    statement;
else
    statement;
```

If - Else If - Else

```
if (expression)
    statement;
else if (expression)
    statement;
else if (expression)
    statement;
else
    statement;
```

Suppose we wanted to verify if a string spells backwards the same as forward, which is the definition of a palindrome. In the example below the content of the buffer is copied to char reverseBuf[32] using function strcpy(). Then the content in reverseBuf is reversed using function strrev(). The conditional statement verifies if the two strings contain the same content using function strcmp(). Since the string "111000111" is a palindrome strcmp() returns a 0 (false) which is inverted by the ! operator to 1 (true) satisfying the conditional statement.

Example 2.8

```
void main(void)
{
   char buffer[] = "111000111";
   char reverseBuf[32];

   strcpy(buffer, reverseBuf);
   strrev(reverseBuf);

   if (!strcmp(buffer, reverseBuf))
   {
      printf ("The sting is a palindrome);
   }
   else
   {
      printf ("The sting is not palindrome);
   }

   return;
}
```

C/ C++ keywords

A C++ keyword is also called a reserved word, and it is a special item of text that the compiler expects to be used in a particular way.

bool	break	case	catch	char
class	const	const_cast	continue	default
delete	deprecated	dllexport	dllimport	do
double	dynamic_cast	else	enum	extern
explicit	false	float	for	friend
goto	if	inline	int	long
mutable	naked	namespace	new	noinline
noreturn	nothrow	novtable	operator	private
property	protected	public	register	reinterpret_cast
return	selectany	signed	short	sizeof
static	static_cast	struct	switch	throw
true	try	typedef	typeid	union
unsigned	using	uuid	virtual	volatile
void	wchar_t	while		

Operator Summary

Scope resolution	class_name:: member
Scope resolution	namespace_name:: member
Global	:: name
Member selection	object.member
Member selection	object->member
Post increment	lvalue++
Post decrement	lvalue- -
Size of object	sizeof expr
Size of type	sizeof (type)
Create (allocate)	new (type)
Destroy	delete pointer
Multiply	expr * expr
Divide	expr / expr
Modulo (remainder)	expr % expr
Shift left	expr << expr
Shift right	expr >> expr
Less than	expr < expr
Less than or equal	expr <= expr
Greater than	exp > expr
Greater than or equal	exp>= expr
Equal	expr == expr
Not equal	expr != expr
Bitwise AND	expr & expr
Logical AND	expr && expr
Logical OR	expr || expr
Bitwise OR	expr | expr

Chapter 3: Control Loops

For Loop

```
for (int i  = 0; i  < 10; i++)
{
    Statements;
}
Statements;
```

int i =0 creates an incrementer for the loop and initializes it to zero.

i < 10 directs the loop to be executed 10 times since i started at zero. When i is equal 10 the condition is false and the for loop exits. Execution continues with the statement following the for loop.

i++ directs the for loop to increment i each time one iteration through the for loop completes.

An alternative implementation of the for loop is possible, and it resembles the while loop.

```
int i  = 0;
for ( ; i  < 10; )
{
    Statements;
    i++;
}
Statements;
```

While Loop

```
int i = 0;
while ( i <10 )
{
    Statements;
    i++;
}
Statements;
```

Chapter 4: State Machines

Switch

The switch statement is a multi-way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly. One should use a switch statement instead of a conditional if – else if – else statement, because the computer hardware (CPU) is designed to be more efficient using a switch. However, not all conditional statements can be implemented using a switch.

```
switch (expression)
{
    case const-expression: statements;
    case const-expression: statements;
    default: statements;
}
```

Each case is labeled by one or more integer-valued constants or constant expressions.
If a case matches the value of the expression, the execution starts with that case.

The case labeled default is executed if no case statement applies. The default case is optional in a switch; however it is coding standard to provide a default case. The none usage of a default state should be indicated as shown below.

```
default :
{
        // intentionally left blank, for future code enhancement.
}
```

Chapter 5: Data Structures

Every data type has two defining characteristics:

1. The domain of the type
2. A collection of allowable operations on those values.

A linear data structure is one whose components are ordered in the following way:

1. There is a unique first component.
2. There is a unique last component.
3. Every component, except the first, has a unique predecessor.
4. Every component, except the last, has a unique successor.

A non-linear data structure is an unordered collection of items. There is no designated first or last component. Figure 5.1 shows the collection of Data Structures.

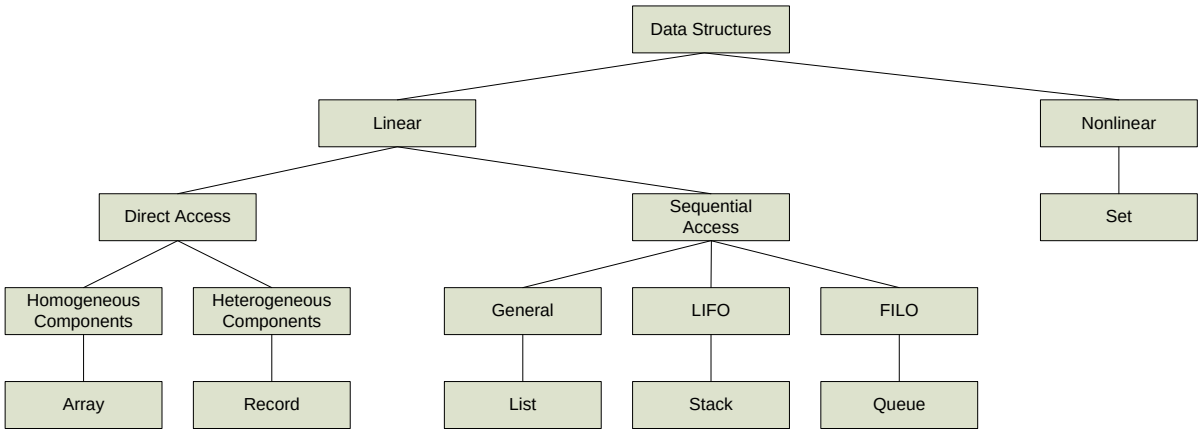


Figure 5.1 – Data Structures.

Chapter 6: Linear Arrays

Arrays

The linear array is the data structure most familiar to programmers. Components are called array elements, and individual elements can be accessed using an index.

Arrays have the distinguishing characteristics:

1. Direct access to a component
2. Homogenous components

A linear array of type double is declared:

double Array [10];

The array can also be initialized at the time of declaration.

int Array [4] = {0, 1, 2, 2 };

The compiler can automatically determine the size of the array from the initialization statement.

double Array [] = {1.0, 1.4, 1.5, 4.0 };

In example 6.1 an array of 10 elements is created and it's elements are initialized using a for loop. The size of the array is defined using the compiler directive #define. It is good coding practice to globally define sizes of arrays. If the size of the array has to change, then the code modification takes place in one place which is less error prone. By convention the globally defined variable is capitalized.

Example 6.1 – Initializing a Linear Array

```
#define SIZE 10
...

int Array [SIZE];
for (int i = 0; i < SIZE; i++)
{
     Array[i] = 0;
 }
```

In example 6.2 an array of 100 elements are created and initialized using the loop incrementer. The average value is calculated following the initialization.

Example 6.2 – Calculating the Average Sum

```
#define SIZE 100
...

int Array [SIZE];
for (int i = 0; i < SIZE; i++)
{
     Array[i] = i;
 }

int sum = 0;
for (int i = 0; i < SIZE; i++)
{
     sum += Array[i];
 }

double avg = sum / SIZE;
printf ("The average = %f \n", avg);
```

Example 6.3 shows copying the content of one array to another array.

Example 6.3 – Copying a Linear Array

```
#define SIZE 100
...

int Array [SIZE];
for (int i = 0; i < SIZE; i++)
{
     Array[i] = i;
 }

int Clone[SIZE];
for (int i = 0; i < SIZE; i++)
{
     Clone[i] = Array[i];
 }
```

Example 6.4 – Σ

Implement the sum = $\sum_{i=0}^{n-1} i$

The close form notation for sum = ½ n (n-1).

```
int sum = 0;
int n = 10;
for (int i = 0; i < n; i++)
{
  sum += i;
}
printf ("n = %d  sum = %d \n", n, sum);
```

Example 6.5 - Σi

Implement the sum = $\sum_{i=1}^{n} i$

The close form notation for sum = ½ n (n+1).

```
int sum = 0;
int n = 10;
for (int i = 1; i <= n; i++)
{
  sum += i;
}
printf ("n = %d  sum = %d \n", n, sum);
```

Example 6.6 – Printing the Sequence Σ

Let's modify Example 6.4 and print the sums of n = 0 to 10 to the console.

```
int sum = 0;
int qty = 10;

for (int n = 1; n <= qty; n++)
{
  sum = 0;
  for (int i = 0; i < n; i++)
  {
    sum += i;
  }
  printf ("n = %d  sum = %d \n", n, sum);
}
```

Example 6.7 – Fibonacci Sequence

The Fibonacci sequence is defined f(n) = f(n-1) + f(n-2). The first few members of the sequence are 0, 1, 1, 2, 3, 5, 8, 13, 21, … . We can generate the Fibonacci sequence using a for loop and capture the result to a linear array.

```
#include "stdafx.h"
#include "stdio.h"

using namespace System;

#define SIZE 20

void main(void)
{
   int Fib[SIZE];
  Fib[0] = 0;
  Fib[1] = 1;

  for (int i = 2; i < SIZE; i++)
  {
    Fib[i] = Fib[i-1] + Fib[i-2];
  }

  return;
}
```

Example 6.8 - Searching a Linear Array

Search Problem:

Design an algorithm that searches a linear array for a number x.
Assume that numbers stored in the array are sorted in ascending order before the search begins.

Discussion:

Suppose we are searching for the number 3.
We could compare number 3 with each value stored in the array and return the index if the number is found.

Although such algorithm implementation is correct, it is inefficient to compare every value in the array. If the number we are looking for is in the last element of the array, then we would have to inspect every element in the array. This type algorithm in its worst case requires n comparisons.

Suppose we divide the array in half. If the number is in the lower half, then we discard the upper half of the array. Otherwise we discard the lower half.

Continue dividing the array until 1 element remains. If that number matches, return the index to the array. In the worst case the number can be found in computational time lg(n) versus n.

```
#include "stdafx.h"
#include "stdio.h"

using namespace System;

#define SIZE 100

void main(void)
{
  const int num = 20;

  //*************************************************
  // Create and initialize the array.
  //*************************************************
  int Array[SIZE];
  for (int i = 0; i < SIZE; i++)
  {
    Array[i] = i;
  }

  //*************************************************
  // Divide the array in half until the number is found.
  //*************************************************
  int start = 0;
  int end = SIZE - 1;
  int middle;

  while (true)
  {
    middle = ((end - start) / 2) + start;
    if (num == Array[middle])
      break;
    else if (end == start || end < start)
    {
      middle = -1;
      break;
    }
    else if (num < Array[middle])
       end = middle - 1;
    else if (num > Array[middle])
       start = middle + 1;

  } // End while

  //*************************************************
  // Print the result
  //*************************************************
  if (middle >= 0)
      printf ("index = %d  number = %d \n", middle, Array[middle]);
  else
    printf ("Number not found \n");
```

```
  return;
}
```

Vector

A linear array is a very powerful tool in algorithm design. However it has a disadvantage; the size of the array is fixed and must be declared at compile time. Dynamically resizing the array is not possible.

The standard C++ library provides numerous containers that provide alternatives to arrays. The facilities of the standard library are defined in the namespace std.

<vector>	one-dimensional array of T
<list>	doubly-linked list of T
<deque>	double-ended queue of T
<queue>	queue of T
<stack>	stack of T
<map>	associative array of T
<set>	set of T
<bitset>	array of booleans

A ***vector*** is similar to linear arrays and stores its elements in contiguous memory. The size of the vector can be changed dynamically.

A vector has numerous functions that allow data manipulation.
Listed are a few functions below:

at	Returns a reference to the element at a specified location in the vector.
back	Returns a reference to the last element of the vector.
begin	Returns a random-access iterator to the first element in the container.
capacity	Returns the number of elements that the vector could contain without allocating more storage.
Clear	Erases the elements of the vector.
empty	Tests if the vector container is empty.
erase	Removes an element or a range of elements in a vector from a specified position.
front	Returns a reference to the first element in a vector.
insert	Inserts an element or a number of elements into the vector at a specified position.

Example 6.9

Create a vector and resize it.

```
#include "stdafx.h"
#include "stdio.h"
#include <vector>

using namespace System;
using namespace std;

#define SIZE  1000

int main ()
{
   ...

   vector<double> Array (SIZE);
   Array.resize (2000);

    ...
    return 0;
}
```

Example 6.10:

Create a vector and initialize it using a for loop.

```
#include "stdafx.h"
#include "stdio.h"
#include <vector>

using namespace System;
using namespace std;

#define SIZE 20

void main(void)
{
 vector<double> Array(SIZE);

 for (int i = 0; i < SIZE; i++)
 {
   Array[i] = i;
   printf("%f \n", Array[i]);
 }

 return;
}
```

Example 6.11

Create a vector and access the elements using an iterator. Accessing the content stored in the vector requires dereferencing the iterator*; i.e.*, *iter.

```
#include "stdafx.h"
#include "stdio.h"
#include <vector>

using namespace System;`
using namespace std;

#define SIZE  20

void main(void)
{
  vector<double> Array(SIZE);
  vector<double>::iterator iter;
  double value = 0;

  for (iter = Array.begin(); iter != Array.end(); ++iter)
  {
    *iter = value++;
  }

  For (iter = Array.begin(); iter != Array.end(); ++iter)
  {
    printf ("%f \n", *iter);
  }

  return;
}
```

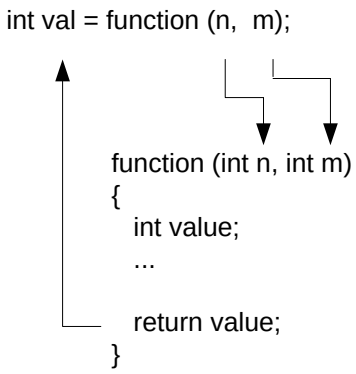
Chapter 7: Functions

A function provides a convenient way to encapsulate some computation. It provides for the decomposition of a large problem into smaller sub-problems, which are easier to manage. Functions can be used without worrying about its implementation. With properly designed functions, it is possible to ignore how a job is done, knowing what is done is sufficient.

A function definition has the form:

```
return-type function-name (parameter declarations, if any)
{
   declarations
   statements;
}
```

In C/C++, all function arguments are passed by value. This means the function is given the values of the arguments, and they are stored in temporary variables. The scope of the temporary variable is good for the lifespan of the function.



All functions should have a function prototype. The compiler verifies that a function call is compliant with its function protocol, and generates an error if it is not compliant. Typically, the function protocol is implemented at the beginning of a cpp file. The name of the parameters is not required; however it is good practice to include them.

```
// Function Prototype
int function (int n, int m);
```

Example 7.1

Create a function that returns the power given the base and a number.
i.e, p = 2^3 => 2 * 2 * 2

```
#include <stdio.h>

// Function Prototype
int power (int base, int n);

void main()
{
  for (int n = 0; n < 10; ++n)
  {
    printf ("%d %d %d \n", n, power (2, n), power (3, n));
  }

  return;
}

// Function power()
int power (int base, int n)
{
   int i;
   int p = 1;

   for (i = 1; i <= n; ++i)
  {
    p = p * base;
  }

  return p;
}
```

Example 7.2

Create an array of size 1000 and populate it with 1000 random generated numbers of data type double. Print to console the sum, mean, and standard deviation.

Convert the random generated data to data type integer with the range [0 .. 99]. The conversion can be performed using "number = (int) rand() % 100". The result shall be stored in a new array of data type int. Print to console the sum, mean, and standard deviation.

Generate histogram information (count) of the integer array. The histogram shall be comprised of 10 bins.

Bin 0 = 0 to 9
Bin 1 = 10 to 19
Bin 2 = 20 to 29

Bin 3 = 30 to 39
Bin 4 = 40 to 49
Bin 5 = 50 to 59
Bin 6 = 60 to 69
Bin 7 = 70 to 79
Bin 8 = 80 to 89
Bin 9 = 90 to 99

Print to console the histogram information.

```
#include "stdafx.h"
#include "stdio.h"
#include "stdlib.h"
#include "memory.h"
#include "math.h"

#define SIZE 1000

//*****************************************************
// Function Prototypes
//*****************************************************
void RandomGenerator(double* RndNum);
int histogram(int num);

using namespace System;

//*********************************************************************
// Function: main()
//*********************************************************************
void main(void)
{
   // Create an array of type double
   double arrayDbl[SIZE];
   memset(arrayDbl, 0, sizeof(double) * SIZE);

   // Populate array with random numbers
   RandomGenerator(arrayDbl);

   // Calculate the sum, mean, and standard deviation
   double sum = 0.0;
   for (int i = 0; i < SIZE; i++)
   {
      sum += arrayDbl[i];
   }
   double mean = sum / SIZE;

   double temp = 0.0;
   for (int i = 0; i < SIZE; i++)
   {
      double dif = arrayDbl[i] - mean;
      temp += (dif * dif);
   }
   double variance = temp / SIZE;
   double stdDev = sqrt(variance);

   // Display to console the sum mean and standard deviation
   printf ("Sum = %f, mean = %f, stdDev = %f \n\n", sum, mean, stdDev);
```

```
    // Create an array of type int and populate it with data from arrayDbl
    // Modify the data to limit the range 0 .. 99

    int arrayInt[SIZE];
    for (int i = 0; i < SIZE; i++)
    {
       arrayInt[i] = (int) arrayDbl[i] % 100;
    }

    // Calculate the sum, mean, and standard deviation
    sum = 0.0;
    for (int i = 0; i < SIZE; i++)
    {
       sum += arrayInt[i];
    }
    mean = sum / SIZE;

    temp = 0.0;
    for (int i = 0; i < SIZE; i++)
    {
       double dif = arrayInt[i] - mean;
       temp += (dif * dif);
    }
    variance = temp / SIZE;
    stdDev = sqrt(variance);

    // Display to console the sum mean and standard deviation
    printf ("Sum = %f, mean = %f, stdDev = %f \n\n", sum, mean, stdDev);

    // Create an array of data type int
    // This array will serves as our histogram with 10 bins
    int histo[10];
    memset(histo, 0, sizeof(int) * 10);

    // Calculate the histogram
    int totalCnt = 0;
    for (int i = 0; i < SIZE; i++)
    {
       int bin = histogram(arrayInt[i]);
       histo[bin] += 1;
       totalCnt++;
    }

    // Print histogram information to console
    printf("Total Cnt = %d \n", totalCnt);
    printf("Bin 0 = %d \n", histo[0]);
    printf("Bin 1 = %d \n", histo[1]);
    printf("Bin 2 = %d \n", histo[2]);
    printf("Bin 3 = %d \n", histo[3]);
    printf("Bin 4 = %d \n", histo[4]);
    printf("Bin 5 = %d \n", histo[5]);
    printf("Bin 6 = %d \n", histo[6]);
    printf("Bin 7 = %d \n", histo[7]);
    printf("Bin 8 = %d \n", histo[8]);
    printf("Bin 9 = %d \n", histo[9]);

}
```

```
//*********************************************************************
// Function: histogram()
//*********************************************************************

int histogram(int num)
{
   int bin = -1;

   if (num >= 0 && num < 10)
      bin = 0;
   else if (num >= 10 && num < 20)
      bin = 1;
   else if (num >= 20 && num < 30)
      bin = 2;
   else if (num >= 30 && num < 40)
      bin = 3;
   else if (num >= 40 && num < 50)
      bin = 4;
   else if (num >= 50 && num < 60)
      bin = 5;
   else if (num >= 60 && num < 70)
      bin = 6;
   else if (num >= 70 && num < 80)
      bin = 7;
   else if (num >= 80 && num < 90)
      bin = 8;
   else if (num >= 90 && num < 100)
      bin = 9;

   return bin;
}

//*********************************************************************
// Function: RandomGenerator()
//*********************************************************************
void RandomGenerator(double* RndNum)
{
   for (int i = 0; i < SIZE; i++)
   {
      *RndNum = rand();
      RndNum++;
   }

   return;
}
```

Chapter 8: Pointers

Pointers

A pointer is a variable that contains the address of a variable.

	Variable	Memory Content	Address
int var1 = 4;	var1	4	0x0001
int var2 = 3;	var2	3	0x0002
		\0	0x0003
		\0	0x0004
int *pVar1 = &var1;	pVar1	0x0001	0x0005
int *pVar2 = &var2;	pVar2	0x0002	0x0006

Initializing a pointer to nullptr, assigns the value 0x0000. Address 0x0000 in the physical memory space does not exist; therefore we cannot store anything in address 0x0000. Such an attempt causes a segmentation fault. We use the nullptr to indicate that no valid memory is assigned to the pointer.

In the C/C++ language, there is a strong relationship between pointers and arrays. Any operation that can be achieved by array subscripting can also be accomplished with pointers. The pointer version is faster, but to the beginner a little harder to understand.

The declaration **int array[5]** reserves consecutive memory space for 5 integers.

	Variable	Memory Content	Address
for (int i = 0; i < 5; i++)	array[0]	0	0x0001
{	array[1]	1	0x0002
array[i] = i;	array[2]	2	0x0003
}	array[3]	2	0x0004
	array[4]	4	0x0005
int *pA = nullptr;	pA	'\0	0x0006
int *pEnd = nullptr;	pEnd	'\0	0x0007
pA = array;	pA	0	0x0001
pEnd = array[4];	pEnd	4	0x0005

The notation **array[i]** refers to the ith element of the array. If pA is a pointer to an integer, declared as **int *pA;** then the assignment **pA = &array[0]** sets pointer pA to element zero of the array, that is, pA contains the address of array[0].

The actual data stored in the address pointed to by pA can be accessed by de-referencing the pointer.

```
int value  = *pA;
```

A pointer can be incremented, pA++, which moves the pointer to the next array element.

```
pA = array;       //  *pA is equal 0;  pA is equal 0x0001
pA++;             //  *pA is equal 1;  pA is equal 0x0002

pA += 2;          //  *pA is equal 3; pA is equal 0x0004
pA = nullptr;     //    pA is not assigned to any memory
```

Pointers and Functions

The calling function must provide the ***address*** of the variable (technically a pointer to the variable), and the called function must declare the parameter to be a pointer and access the variable indirectly through it. A pointer is a variable that contains the address of a variable. Pointers are used because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code.

```
int val, Result;
...
function_name (val, &Result);
...
function_name (int val, int *pResult) { ...
```

Since C/C++ passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. If we are required to sort an array of numbers, than the array must be passed to the function by reference; *i.e.*, the calling function passes a pointer holding the address of the beginning of the array.

The called function can now manipulate the data stored in memory indirectly. One major advantage is that data is stored in memory only once. Improved spatial locality reduces disk swapping.

Example:

Write a function that swaps two variables.

```
void main (void)
{
   int a = 1
```

```
   int b = 2;
   swap (&a, &b);
    ...
}


void swap ( int *pA, int *pB)
{
      int temp;

      temp = *pA;
      *pA   = *pB;
      *pB   = temp;

     return;
}
```

File Operation

FILE *fopen (const char *_filename_, const char *_mode_);

Parameters
filename - Filename.
mode - Type of access permitted.

Return Value
Each of these functions returns a pointer to the open file. A null pointer value indicates an error.

Mode
"r" Opens for reading. If the file does not exist or cannot be found, the **fopen** call fails.
"w" Opens an empty file for writing. If the given file exists, its contents are destroyed.
"a" Opens for writing at the end of the file (appending) without removing the EOF marker before writing new data to the file; creates the file first if it doesn't exist.

"r+" Opens for both reading and writing. (The file must exist.)
"w+" Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.

"a+" Opens for reading and appending; the appending operation includes the removal of the EOF marker before new data is written to the file and the EOF marker is restored after writing is complete; creates the file first if it doesn't exist.

Writing to Text File

```
#include "stdafx.h"
#include "stdio.h"

using namespace System;
using namespace std;

void main(void)
{
   // File pointer
   FILE* fptr = nullptr;

   // General text to be stored to file
   char buffer[] = "First Name, Last Name \nAddress \nCity, State, Zip \n";

   // Create the file.
   fptr = fopen("./MyFile.txt", "w");
   if (!fptr)
   {
      printf("Unable create or open file. \n");
      return;
   }
```

```
    // Write data to file
    fprintf(fptr, buffer);

    // Close the file
    fclose(fptr);
    fptr = nullptr;

    return;
}
```

The content of ./DataFile.txt is given below:

First Name, Last Name
Address
City, State, Zip

Reading from Text File

```
#include "stdafx.h"
#include "stdio.h"

using namespace System;
using namespace std;

void main(void)
{
    // File pointer
    FILE* fptr = nullptr;

    fptr = fopen("./MyFile.txt", "r");
    if (!fptr)
    {
        printf("Unable to open file. \n");
        return;
    }

    char buf[120];

    while (fgets(buf, 120, fptr) != nullptr)
    {
        printf("%s", buf);
    }

    fclose(fptr);
    fptr = nullptr;

    return;
}
```

Writing Data to Binary File

There are times when we need to store data to binary file instead of a text file. We use the function fwrite() to write the data to file.

size_t fwrite (const void **buffer*, **size_t** *size*, **size_t** *count*, **FILE** **stream* **);**

Parameters

buffer - Pointer to data to be written.
size - Item size in bytes.
count - Maximum number of items to be written.
stream - Pointer to **FILE** structure.

Return Value

fwrite returns the number of full items actually written, which may be less than *count* if an error occurs. Also, if an error occurs, the file-position indicator cannot be determined.

```
#include "stdafx.h"
#include "stdio.h"
#include "string.h"

using namespace System;
using namespace std;


void main(void)
{
   // File pointer
   FILE* fptr = nullptr;

   // General text to be stored to file
   char buffer[] = "First Name, Last Name \nAddress \nCity, State, Zip \n";

   // Create the file.
   fptr = fopen("./BinaryFile.txt", "wb");
   if (!fptr)
   {
      printf("Unable create or open file. \n");
      return;
   }

   fwrite (buffer, strlen(buffer), 1, fptr);

   fclose (fptr);
   fptr = nullptr;

   return;
}
```

Read Data from Binary File

size_t fread(void **buffer*, **size_t** *size*, **size_t** *count*, **FILE** **stream* **);**

Parameters
buffer - Storage location for data.
size - Item size in bytes.
count - Maximum number of items to be read.
stream - Pointer to **FILE** structure.

Return Value
fread returns the number of full items actually read, which may be less than *count* if an error occurs or if the end of the file is encountered before reaching *count*.

Use the **feof** or **ferror** function to distinguish a read error from an end-of-file condition. If *size* or *count* is 0, **fread** returns 0 and the buffer contents are unchanged.

```
#include "stdafx.h"
#include "stdio.h"
#include "string.h"

using namespace System;
using namespace std;


void main(void)
{
   // File pointer
   FILE* fptr = nullptr;

   // Open the file
   fptr = fopen("./BinaryFile.txt", "rb");
   if (!fptr)
   {
      printf("Unable create or open file. \n");
      return;
   }

   // Read 100 bytes
   fread(buffer, 100, 1, fptr);
   printf("%s", buffer);

   fclose (fptr);
   fptr = nullptr;

   return;
}
```

Time

Algorithm design often requires that we measure the average running times to determine efficiency. We can measure time on a PC within 1 millisecond.

```
#include "stdafx.h"
#include "stdio.h"
#include <string>
#include "time.h"
#include <sys\timeb.h>

using namespace System;
using namespace std;

// Function Prototype
unsigned long CalcTime(_timeb t0, _timeb t1);

void main(void)
{
   // Create empty strings to hold time value.
   char *t0, *t1;
   long t;

   // Create Time Structures
   struct _timeb TimeStructT0;
   struct _timeb TimeStructT1;

   // Get the start time
   _ftime(&TimeStructT0);

   // Execute some function or algorithm

   // Get the stop time
   _ftime(&TimeStructT1);

   // Convert the time to this format 00:00:00
   t0 = ctime(&(TimeStructT0.time));
   t1 = ctime(&(TimeStructT1.time));

   // Calculate the elapsed time in milliseconds.
   t = CalcTime(TimeStructT0, TimeStructT1);

   // Print the start time
   printf ("Start Time = %.19s.%hu \n", t0, TimeStructT0.millitm);

   // Print the stop time
   printf ("Stop Time = %.19s.%hu \n", t1, TimeStructT1.millitm);

   printf ("Total Runing time (ms)= %ld \n", t);

   return;
}
```

```
unsigned long CalcTime(_timeb t0, _timeb t1)
{
unsigned long ms;
double T0, T1;
T0 = ((double) t0.time * 1000.0) + t0.millitm;
T1 = ((double) t1.time * 1000.0) + t1.millitm;
ms = (unsigned long) (T1 - T0);
return ms;
}
```

Chapter 9: Recursion

Factorial

The function Fact() below computes the factorial of a number utilizing a while loop.

```
long Fact (long number)
{
    long res = 1;
    while (number > 1)
    {
        res *= number;
        number--;
    }
    return res;
}
```

Suppose the function is called **result = Fact (4);**

The while loop will be executed until the number is equal 1.

res = 1 * 4	1st Iteration
res = 4 * 3	2nd Iteration
res = 12 * 2	3rd Iteration
res = 24	return res.

We can convert function Fact() with a recursive function considering the following changes.
Every recursive function requires a conditional statement that terminates the recursion.
Every recursive function calls itself until a condition to stop the recursion is met.
Every recursive function includes a body that performs some computation.

```
long Factorial (long number)
{
    long ret = number;
    if (ret > 1)
        ret *= Factorial (--number);
    return ret;
}
```

Suppose function Factorial() is called **res = Factorial (4);**

```
ret = 4 * Factorial (3);
ret = 4 * 3 * Factorial (2);
ret = 4 * 3 * 2 * Factorial (1);
ret = 4 * 3 * 2 * 1;
res = 24;
```

Power Function

```
double Power (double x, long y)
{
  double num = x;
  if (y > 1)
     num *= Power (x, --y);
  return num;
}
```

Suppose function Power() is called **res = Power (2, 4);**

```
num = 2 * Power (2, 3);
num = 2 * Power (2, 3) * Power (2, 2);
num = 2 * Power (2, 3) * Power (2, 2) * Power (2, 1);
num = 2 * Power (2, 3) * Power (2, 2) * 2;
num = 2 * Power (2, 3) * 4;
num = 2 * 8;
num = 16;
res = 16;
```

Chapter 10: Object Oriented Programming

Topic:

- Abstract Data Types
- Keyword class
- Public, Private, Protected Data Members
- Constructors
- Destructor
- Keyword pragma
- Example: CMathFundamental
- Objects
- Memory Leaks
- Templates
- Pointers to Functions

Namespace

A namespace declaration identifies and assigns a unique name to a user-declared namespace. Such namespaces are used to solve the problem of name collision in large programs and libraries.

Programmers can use namespaces to develop new software components and libraries without causing naming conflicts with existing components.

Example:

```
namespace x
{
   int i, j;
}
main ()
{
   x::i = 0;
   y::j = 0;
}
```

Using namespace limits the problem of having function names and variables names duplicated in different header files. The following code causes numerous compiler and linker errors.

Example:

```
// filename:my.h
void ReadFile (char *, char *);
FILE *fptr;
int byte_count;
```

```
// filename:your.h
void ReadFile (char *, char *);
FILE *fptr;
int byte_count;

#include my.h
#include your.h
void main (int argc, char *argv[]) { ... }
```

The compiler will generate an error: error C2086: 'int byte_count' : redefinition
The linker will generate an error: Test error LNK2005: "int cdecl ReadFile (int)“ (?ReadFile@@YAHH@Z) already defined in my.obj

Example:

```
// filename:my.h
namespace My {
    void ReadFile (char *, char *);
    FILE *fptr;
    int byte_count;
}
// filename:your.h
namespace Your {
    void ReadFile (char *, char *);
    FILE *fptr;
    long byte_count;
}
#include my.h
#include your.h
void main (int argc, char *argv[]) {
 ...  My::fptr = fopen ( ...
}
```

The C++ class concept provides the programmer with a tool for creating new data types that can be used as conveniently as the built-in types. With abstract data types we can separate the conceptual transformations that our programs perform on our data from any particular data-structure representation and algorithm implementation.

Definition:

An abstract data type (ADT) is a data type (a set of values and a collection of operations on those values) that is accessed only through an interface. We refer to a program that uses an ADT as a client, and a program that specifies the data type as an implementation.

Structures are abstract data types.

In the C programming language, the structures do not have associated functions; however in the C++ programming language structures do have associated functions.

A class is a user-defined data type.

The key difference between classes and structures has to do with access to information, as specified by the keywords private and public.

A private class member can be referred to only within the class.
A public member can be referred to by any client.
By default, members of classes are private, while members of structures are public

Keyword class

class [*tag* [: *base-list*]]
{
  member-list
} [*declarators*];

tag
Names the class type. The tag becomes a reserved word within the scope of the class.

base-list
Specifies the class or classes from which the class is derived (its base classes).
Each base class's name can be preceded by an access specifier (*public, private, protected*) and the *virtual* keyword.

member-list
Declares members or friends of the class.
Members can include data, functions, nested classes, enums, bit fields, and type names.

Friends can include functions or classes.
Explicit data initialization is not allowed.
A class type cannot contain itself as a nonstatic member.
It can contain a pointer or a reference to itself.

declarators
Declares one or more objects of the class type.

public: [*member-list*]
public *base-class*

When preceding a list of class members, the **public** keyword specifies that those members are accessible from any function. This applies to all members declared up to the next access specifier or the end of the class.

When preceding the name of a base class, the **public** keyword specifies that the public and protected members of the base class are public and protected members, respectively, of the derived class.

private: [*member-list*]
private *base-class*

When preceding a list of class members, the **private** keyword specifies that those members are accessible only from member functions and friends of the class. This applies to all members declared up to the next access specifier or the end of the class.

When preceding the name of a base class, the **private** keyword specifies that the public and protected members of the base class are private members of the derived class.

protected: [*member-list*]
protected *base-class*

The **protected** keyword specifies access to class members in the member-list up to the next access specifier (**public** or **private**) or the end of the class definition.

Class members declared as **protected** can be used only by the following:

- Member functions of the class that originally declared these members.
- Friends of the class that originally declared these members.
- Classes derived with public or protected access from the class that originally declared these members.
- Direct privately derived classes that also have private access to protected members.

When preceding the name of a base class, the **protected** keyword specifies that the public and protected members of the base class are protected members of its derived classes.

Protected members are not as private as **private** members, which are accessible only to members of the class in which they are declared, but they are not as public as **public** members, which are accessible in any function.

Protected members that are also declared as **static** are accessible to any friend or member function of a derived class. Protected members that are not declared as **static** are accessible to friends and member functions in a derived class only through a pointer to, reference to, or object of the derived class.

A constructor is a special initialization function that is called automatically whenever an instance of a class is declared.

The constructor must have the same name as the class itself.
Using a constructor to initialize data members may prevent errors resulting from un-initialized objects.

You cannot specify a return type when declaring a constructor, not even a void.
Consequently, a constructor cannot contain a return statement.
Constructors cannot return values, instead they create objects.

Example:

```
class complex {
public:
        complex () {re = im = 0;}                          // Default Constructor
        complex (double r) {re = r; im =0;}                // Polymorphic Behavior
        complex (double r, double i) {re = r; im = i;} // Polymorphic Behavior
private:
        double re, im;
};
complex m_complexVar();
complex m_complexVar(12.0);
complex m_complexVar(12.0, 4.0);
```

At the time of instantiation objects can be initialized by copying another object.

```
complex m_complexVar(12.0, 4.0);
complex m_Var2 = m_complexVar;
```

This type of an initialization can be troublesome.
For instance:

```
class Table
{
   public:
        Name *p;
        size_t sz;
        Table (size_t s = 15) {p = new Name[sz = s];}          // Constructor
        ~Table() {delete [] p; }                               // Destructor
        Name * lookup (const char *);
        bool insert (Name*);
};

void h()
{
        Table t1;
        Table t2 = t1;  // copy initialization: trouble

        Table t3;
        t3 = t2;          // copy assignment: more trouble
}
```

The Table default constructor is called twice, once for t1 and t3.

The default interpretation of assignment is a member wise copy, so t1, t2, and t3 will at the end of h(), each contain a pointer to the array of names allocated on the free store when t1 was created.

No pointer to the array of names, allocated when t3 was created, remains because it was overwritten by the t3 = t2 assignment.

The destructor is the counterpart of the constructor.
It is a membership function which is called automatically when a class object goes out of scope.

The purpose of the destructor is to perform any cleanup work necessary before an object is destroyed.

The destructor's name is the class name with a tilde (~) as a prefix.

Example:

```
class CreditCardAccount
{
public:
        CreditCardAccount();
      ~ CreditCardAccount();
        ...
}

CreditCardAccount::~CreditCardAccount()
{
   Console::Write(S"Account being destroyed: ");
}
```

A destructor should be used whenever a class creates an object on the free store.
A destructor is implicitly invoked when a class instance goes out of scope.
Parameters cannot be passed to a destructor.
A library call exit() should not appear inside a destructor. This may result in an infinite recursion.

Pragma Directives:
Each implementation of C and C++ supports some features unique to its host machine or operating system.

The **#pragma** directives offer a way for each compiler to offer machine- and operating system-specific features while retaining overall compatibility with the C and C++ languages.

Pragmas are machine- or operating system-specific by definition, and are usually different for every compiler.

#pragma once
Specifies that the file will be included (opened) only once by the compiler in a build.
This can reduce build times as the compiler will not open and read the file after the first

#include of the module.
Microsoft C and C++ compilers recognize 32 pragma directives.
For specifics see *pragma directives (#pragma).*

Example:

```
Namespace MathFundamental {
        class CMathFundamental                      // Class Name
        {
        public:                                     // Public Components
           CMathFundamental (void);                 // Default constructor
          ~CMathFundamental (void);                 // Destructor

           void    Factorial   (int, double *);     // Membership functions (methods)
           void    LogBaseX (double *, int, int);   // Membership functions can return a
                                                    value,
           int     IsPrime    (int);                // constructors and destructors cannot
                                                    return
           void    GetPrimeNumbers (int, int *);    // a value, not even type void.
           void    GetPrimeMultiplier (int, int *);
           double Random (long *);

        protected:                                  // Protected components
           int FactFlag;                            // These membership variables are not
           double temp;                             // accessible from the outside world.
        };
}


using namespace MathFundamental;

namespace MathTrig
{
        #pragma once                                // Compiler Build Directive:
        #include "MathFundamental.h"                // Include the definition of the class
                                                    only once

        class CMathTrig :                           // Class Name
              public CMathFundamental               // Inherited, Base Class
        {
        public:                                 // Public Components
            CMathTrig (void);                   // Constructor
           ~CMathTrig (void);                   // Destructor
            double Sin      (double);           // Membership functions
            double Cosec (double);
         double Cos     (double);
            double Sec     (double);
            double Tan     (double);
            double Cotan (double);
        };
}
```

Creating Objects

Objects can be created and destroyed dynamically using the **new** and **delete** operator.

When an object is dynamically created using the new operator, the following occurs:

- Memory is allocated for the object.
- The constructor is called to initialize the object.
- A pointer to the object is returned.

Example: // Create an object to represent Account number 1234567
```
Account *pA = new Account (1234567);
```

Deleting Objects
Local variables are created on the stack, they will be destroyed automatically when they go out of scope.

Local variables are also called automatic variables.
C++ requires managing the lifetime of dynamically created objects.
We have to destroy the object using the delete operator.

Example: // Delete a dynamically created object.

```
if (!pA) {
   delete pA;
   pA = NULL;
}
```

The delete operator takes a pointer to the object and performs the following:

- Calls the class destructor and cleans up the object state.
- Returns the allocated memory to the operating system.

```
void SomeFunction()
      {
      long num = 1234567;
         Account *pA = new Account (num);
      }
```

When SomeFunction() goes out of scope, long num and pointer pA will be destroyed.
However, the actual object Account is not automatically destroyed.

The memory used by the object can never be freed, unless the program is shut down.
If another function call is made to SomeFuntion(), then additional memory is consumed.
This is what we call a **memory leak**.

Dynamically allocated objects must be destroyed using the delete operator!

```
void SomeFunction()
{
  long num = 1234567;
    Account *pA = new Account (num);
    ...
 if (!pA)
    delete pA;
}
```

There are times when it is convenient to create classes that contain only data.

```
class intClass
{
     int storage[100];
          ...
}

class longClass
{
     long storage[100];
          ...
}
```

We can create a generic class using template

```
template <class genType>
class genClass
{
   genType storage[100];
}

genClass <int> intObject;
genClass <long> longObject;
```

Since we don't want to limit the size of data arrays, we need to dynamically define the size.

```
template <class genType, int size = 128>              // defined default size = 128
class genClass
{
   genType storage[size];
}

genClass <int> intObject;              // The default creates an array of 128 elements
genClass <int, 1024> intObject;        // The array is of size 1024 elements.
```

Templates can also be applied to functions.

```
#include "stdafx.h"

template<class genType>

void swap (genType &var1, genType &var2)
{
      genType tmp = var1;
      var1 = var2;
      var2 = tmp;
}

int main()
{
    int m = 10;
    int n = 5;

    printf ("Before swap: m = %d , n = %d \n", m, n);
    swap (m, n);
    printf ("After swap: m = %d, n = %d \n", m, n);

    return 0;
}
```

Sometimes we need to design a function that takes as an input parameter another function. The C++ language supports passing a function by reference.

Example:

```
double f (double x) {
  return 2 * x;
}

double  sum (double (*f) (double), int n, int m) {
      double result = 0;
        for (int i = n; I <= m; i++) {
          result += f(i);
      }
      return result;
}
```

Function sum() can be called with any built-in or user-defined function double *f (double).

cout << sum (f, 1, 5) <, endl;

Chapter 11: Inheritance

Chapter 12: Linked Lists

A linked list is a linear data structure which is comprised of components (elements) that can be accessed sequentially. The first element in the linked list is called the head and the last element is referred to as the tail. The advantage of a linked list is that the size can be changed dynamically and elements do not have to exist in consecutive memory. The elements do not have to be of homogeneous type; although, it is common practice to design the linked list to be homogenous.

There are three types of linked lists, which are Singly Linked List, Doubly Linked List, and Circular List. Figure 11.1 shows a Singly Linked List. A pointer pNext is implemented to hold the address of the next element in the list. Each element in the linked list can be accessed by traversing the list in sequential order starting from the head. The pointer pNext in the tail of the list is assigned the value 0x0000 (nullptr). Memory address 0x0000 does not exist in the physical memory space and indicates the end of the list.

Figure 11.2 shows a Doubly Linked List and Figure 11.3 shows a Circular Linked List. An additional pointer pPrevious is implemented to provide for traversing the list in reverse. In the Circular Linked List the pointer pNext in the tail element points to the head. Pointer pPrevious of the head element points to the tail of the list. The Circular List can be traversed in both direction if a pointer pPrevious is implemented; otherwise the list can be traversed only in one direction.

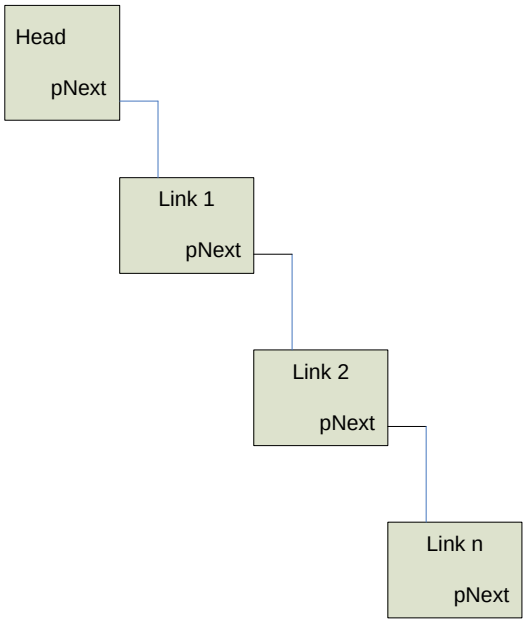


Figure 11.1 – Singly Linked List

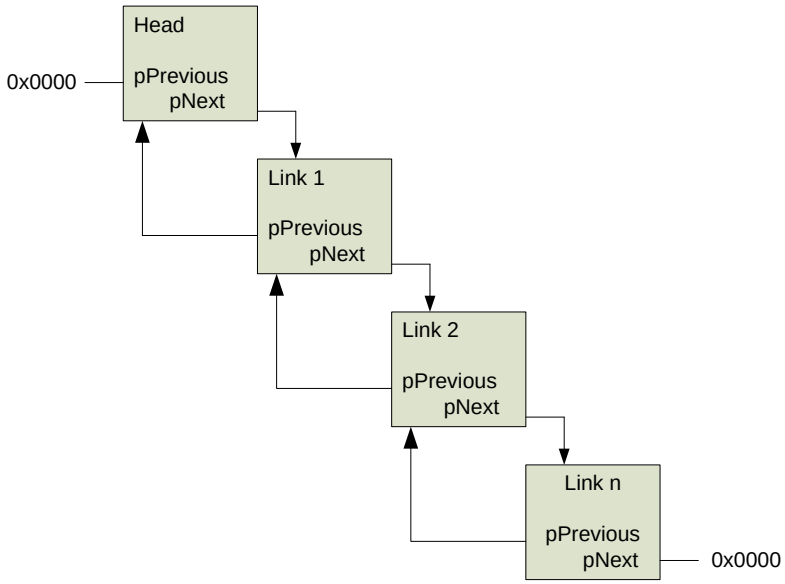


Figure 11.2 – Doubly Linked List

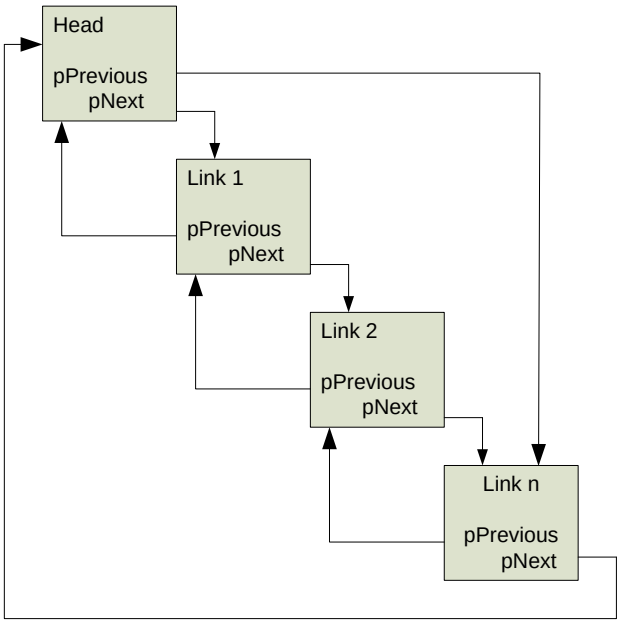


Figure 11.3 – Circular Linked List

When the Linked List is created the individual elements do not have to exist in consecutive memory. Figure 11.4 illustrates how elements may be place anywhere is the program

memory space. Pointer pNext contains the address of the next element. This mechanism allows elements to be created or deleted at any time during program execution. An advantage of the Linked List is that the user data stored in each element does not have to be moved around in memory, which minimizes the opportunity to corrupt the data. Instead a pointer can navigate to the desired element.

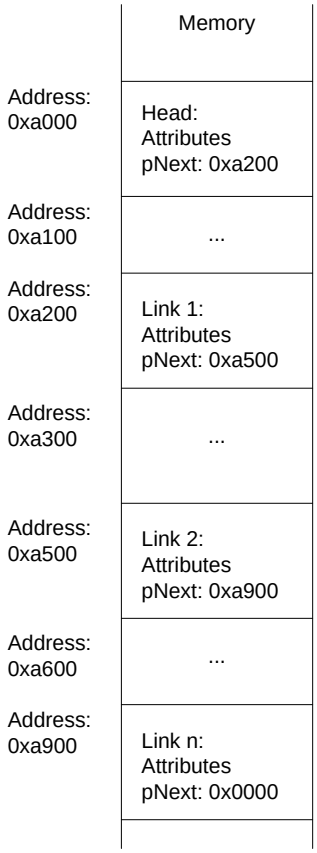


Figure 11.4 – Memory Allocation involving Linked Lists

Example of a Singly Linked List

The following example code implements a Linked List of 10 students.

```
//*********************************************************************
// File: Student.h
//*********************************************************************
public class CStudent
{
public:
   CStudent(void);
   ~CStudent(void);

   int StudentID;
   char FirstName[24];
   char LastName[24];
   char Birthdate[16];
   char Street[24];
   char City[24];
   char State[24];
   int  ZipCode;
   char DegreeProgram[8];

   CStudent *pNext;

   CCourse Course[10];
};


//*********************************************************************
// File: Student.cpp
//*********************************************************************
#include "StdAfx.h"
#include "Student.h"
#include "memory.h"

//*********************************************************************
// Default Constructor
//*********************************************************************
CStudent::CStudent(void)
{
   pNext = nullptr;

   StudentID = 0;
   ZipCode = 0;
   memset(FirstName, 0 , sizeof(char)*24);
   memset(LastName, 0, sizeof(char)*24);
   memset(Birthdate, 0, sizeof(char)*16);
   memset(Street, 0, sizeof(char)*24);
   memset(City, 0, sizeof(char)*24);
   memset(State, 0, sizeof(char)*24);
   memset(DegreeProgram, 0, sizeof(char)*8);
}
```

```
//*********************************************************************
// Destructor
//*********************************************************************
CStudent::~CStudent(void)
{
}

/*********************************************************************
// File: Course.h
//*********************************************************************
public class CCourse
{
public:
   CCourse(void);
   ~CCourse(void);

   char CourseName[24];
   char CourseNumber[16];
   char Grade;
};



/*********************************************************************
// File: Course.cpp
//*********************************************************************
#include "StdAfx.h"
#include "Course.h"
#include "memory.h"

//*********************************************************************
// Default Constructor
//*********************************************************************
CCourse::CCourse(void)
{
   memset(CourseName, 0, 24);
   memset(CourseNumber, 0, 16);
   Grade = 'X';
}

//*********************************************************************
// Destructor
//*********************************************************************
CCourse::~CCourse(void)
{
}



//*********************************************************************
// Function: main()
//*********************************************************************
int main(array<System::String ^> ^args)
{
   //*********************************************************************
   // Create the Linked List and the first student.
   //*********************************************************************
   CStudent *pLinkedList = new CStudent;
```

```
//*****************************************************************
// Create a pointer to the Linked List
//*****************************************************************
CStudent *pL = pLinkedList;
CStudent *pS = pLinkedList;


//*****************************************************************
// Initialize the first student attributes
//*****************************************************************
pL->StudentID = 1;
strcpy_s(pL->FirstName, "Holger");
strcpy_s(pL->LastName, "Findling");
strcpy_s(pL->Birthdate, "09162012");
strcpy_s(pL->Street, "Street Name");
strcpy_s(pL->City, "Oviedo");
strcpy_s(pL->State, "FL");
pL->ZipCode = 32766;
strcpy_s(pL->DegreeProgram, "MSCIS");

//*****************************************************************
// Initialize courses taking by the first student
//*****************************************************************
strcpy_s(pL->Course[0].CourseName, "Algorithm");
strcpy_s(pL->Course[0].CourseNumber, "CSE 5600");
pL->Course[0].Grade = 'A';

//*****************************************************************
// pLinkedList is pointing to the beginning of the linked list
// pL is the last link or student in the Linked List
// pS is the new student added to the Linked List
//*****************************************************************

//*****************************************************************
// Create 9 additional students
// Data may be redundant for simplicity, except for student the id.
//*****************************************************************
for (int i = 2; i < 11; i++)
{
   pS = new CStudent;
   pS->StudentID = i;
   strcpy_s(pS->FirstName, "Holger");
   strcpy_s(pS->LastName, "Findling");
   strcpy_s(pS->Birthdate, "09162012");
   strcpy_s(pS->Street, "Gerber Daisy Ln");
   strcpy_s(pS->City, "Oviedo");
   strcpy_s(pS->State, "FL");
   pS->ZipCode = 32766;
   strcpy_s(pS->DegreeProgram, "MSCIS");

   pS->Course[0].Grade = 'A';
   strcpy_s(pS->Course[0].CourseName, "Advanced Programming");
   strcpy_s(pS->Course[0].CourseNumber, "CIS 5002");

   pS->Course[1].Grade = 'A';
   strcpy_s(pS->Course[1].CourseName, "Algorithm");
   strcpy_s(pS->Course[1].CourseNumber, "CSE 5600");

   pS->Course[2].Grade = 'B';
   strcpy_s(pS->Course[2].CourseName, "Formal Languages");
```

```
      strcpy_s(pS->Course[2].CourseNumber, "CSE 5604");

      //*****************************************************************
      // Assign the address of the new student object to pNext
      //*****************************************************************
      pL->pNext = pS;

     //*****************************************************************
     // Let pL point to the last element in the Linked List
     //*****************************************************************
     pL = pS;
   }

   //*****************************************************************
   // Set pS to the beginning of the Linked List
   //*****************************************************************
   pS = pLinkedList;

   return 0;
}
```

Traversing to the end of the Linked List

Function findLastElement() traverses the Linked List in sequential order until the tail is found. When the conditional statement if (pS->pNext != nullptr) is false, that is the pointer pS->pNext is equal nullptr, then the tail has been found. Function printElement() echoes the student data to the console. Prior to calling the function findLastElement() the pointer pS needs to be set to the Head of the Linked List, pS = pLinkedList.

```
//*****************************************************************
// Function Prototypes
//*****************************************************************
void printElement(CStudent* pS);
CStudent* findLastElement(CStudent *pS);


//*****************************************************************
// Function: findLastElement()
//*****************************************************************
CStudent* findLastElement(CStudent *pS)
{
   while (true) {
      if (pS->pNext != nullptr)
         pS = pS->pNext;
      else
         break;
   }

   return pS;
}
```

```
//*****************************************************************
// Function: printElement()
//*****************************************************************
void printElement(CStudent* pS)
{
   printf("Student ID: %d \n", pS->StudentID);
   printf("First Name: %s \n", pS->FirstName);
   printf("LastName: %s   \n", pS->LastName);
   printf("Birthdate: %s  \n", pS->Birthdate);
   printf("Street: %s  \n", pS->Street);
   printf("City: %s    \n", pS->City);
   printf("State: %s   \n", pS->State);
   printf("ZipCode: %d \n", pS->ZipCode);
   printf("Degree Program: %s \n", pS->DegreeProgram);

   for (int i = 0; i < 10; i++)
   {
      printf("\nCourse Name: %s \n", pS->Course[i].CourseName);
      printf("Course Number: %s \n", pS->Course[i].CourseNumber);
      printf("Grade: %c \n", pS->Course[i].Grade);
   }
   printf("\n**************************\n\n");
   return;
}
```

Inserting an element into the Linked List

Prior to inserting an element into a Linked List, the list must be searched for the position to insert the new student. Function Find() returns the address of the element with the desired student id and function Insert() inserts a new link following that position (address).

The code example below shows that the pointer pS is set to the beginning of the Linked List and is then passed as an argument to function Find(). Find() traverses the list until the student id is found and returns the associated address. If the student id is not found in the list, then pS is set to the nullptr. Function Insert() creates a new student object and inserts the link between pS and pS->pNext.

```
// Find Student ID 5 and insert Student #11
pS = pLinkedList;
pS = Find(pS, 5);
if (pS != nullptr) {
   pS = Insert(pS, 11);
}
```

```
//*****************************************************************
// Function Prototypes
//*****************************************************************
CStudent* Find(CStudent* pS, int id);
CStudent* Insert(CStudent* pS, int NewStudentID);
```

```
//*****************************************************************
// Function: Find()
//*****************************************************************
CStudent* Find(CStudent* pS, int id)
{
   while (true)
   {
      if (pS->StudentID == id)
         break;
      else if (pS->pNext != nullptr)
         pS = pS->pNext;
      else
      {
         pS = nullptr;
         break;
      }
   }

   return pS;
}


//*****************************************************************
// Function: Insert()
//*****************************************************************
CStudent* Insert(CStudent* pS, int NewStudentID)
{
   //*****************************************************************
   // Create the new Student object
   //*****************************************************************
   CStudent* pNewS = new CStudent();
   pNewS->StudentID = NewStudentID;

   //*****************************************************************
   // Insert the new object into the linked list
   //*****************************************************************
   CStudent* pTemp = pS->pNext;
   pS->pNext = pNewS;
   pNewS->pNext = pTemp;

   return pNewS;
}
```

Deleting an element from the Linked List

Deleting a link from the Linked List requires finding the particular link by one of its attributes such as StudentID. Function DeleteLink() performs this operation.

```
// Find Student #11 and delete the link
pS = pLinkedList;
pS = DeleteLink(pS, 11);
```

```
//*********************************************************************
// Function: DeleteLink()
//*********************************************************************
CStudent* DeleteLink(CStudent* pS, int StudentID)
{
   //****************************************************************
   // Find the node (link) before the student to be deleted
   //****************************************************************
   while (true)
   {
      if (pS->pNext->StudentID == StudentID)
         break;
      else if (pS->pNext != nullptr)
         pS = pS->pNext;
      else
      {
         pS = nullptr;
         return pS;
      }
   }

   //****************************************************************
   // Point to the link to be deleted
   //****************************************************************
   CStudent* pTemp = pS->pNext;

   //****************************************************************
   // Break the link and reconnect to the next link
   //****************************************************************
   pS->pNext = pTemp->pNext;

   //****************************************************************
   // Delete the student record
   //****************************************************************
   delete pTemp;
   return pS;
}
```

Chapter 13: Stacks and Queues

Chapter 14: Binary Trees

Many data structures have components arranged in linear form.
A linear data structure has a first and last component.
Arrays, records, lists, stacks, and queues have linear form.

A tree does not have linear form. It is a hierarchical form.

Example of a tree

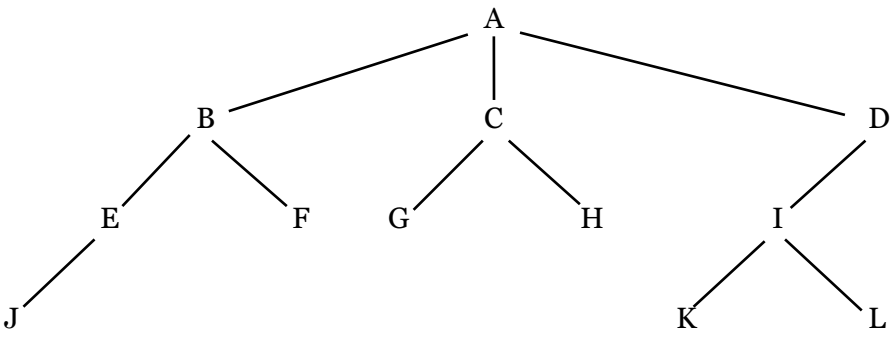


Figure 14.1

The Terminology of Trees

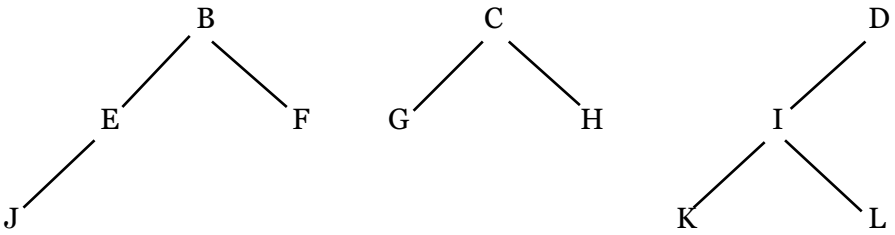
The root of a tree is the single component at the top of the hierarchical tree.
Node A is the root of the tree in figure #1.

Components that have no successors are called leaves.
Nodes J, F, G, H, K, and L are the leaves in the tree in figure #1.

The components of the tree are called nodes.
The nodes of the tree above are A, B , C, ..., L

A structure or class can represent each component of a tree.
In AI we make references to frames. Frames are structures representing nodes.
The data members in the structure are attributes of the tree.
This concept of attributes is important when we are dealing with graphs.
A node is a parent when it has successor nodes. Leaves are children of parents.
Nodes E, C, and I are the parents of the leaves in the tree.

Trees comprise of sub-trees. Some of the sub-trees in figure #1 would be:



Binary Trees

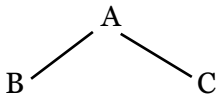
A Binary tree is a special type of a tree. Each parent can have only two children. The order of the nodes is also important. Please note, we would have to select an attribute of the tree that determines the order.

For simplicity the book shows the node numbers as the attribute by which we sort the nodes in the tree.

The two children of a node have special names: the left child and the right child.

Binary Tree 1 consists of three nodes, A, B, C.
Node A is the root node. Node B is the left node, and node C is the right node.

Tree 1



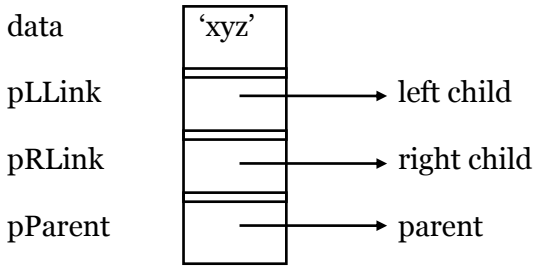
To implement a binary tree we need at least three data members:

1. The data contents of the node
2. The location of the left child of the node (pointer)
3. The location of the right child of the node (pointer)

A fourth data member is highly instrumental when searching trees
4. The location of the parent node (pointer)

```
struct TreeNode {
        char        data;
        NodePtr     pLLink;
        NodePtr     pRLink;
        NodePtr     pParent;
        TreeNode (char, NodePtr, NodePtr, NodePtr);
};
```

Tree node object

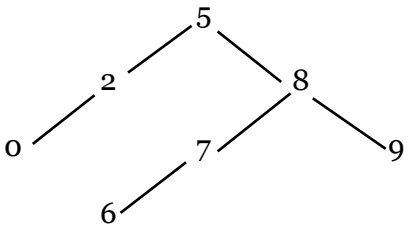


Binary Search Trees

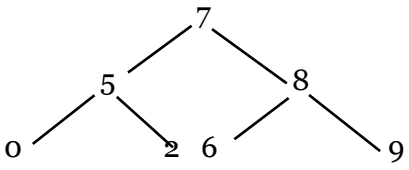
A binary search tree has the following properties:

1. No two nodes contain the same data value.
2. The data values in the tree come from a data type for which the relations greater than and less than are defined.
3. The data value of every node in the tree is
    - Greater than any data value in its left sub-tree.
    - Less than any data value in its right sub-tree.

Tree 2

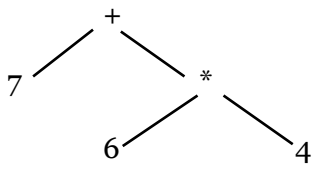


Tree 3



Tree 2 is a binary tree and tree 3 is NOT a binary tree. Node 6 located on the right side of the tree is less than Node 7. All nodes on the right side must be greater than 7.

Binary Expression trees



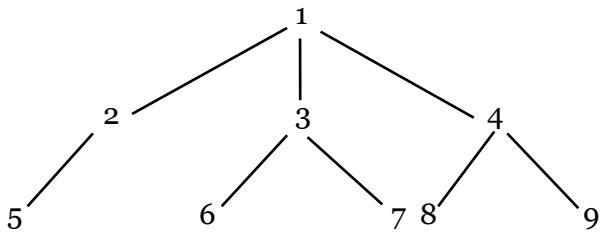
7 + 6 * 4 (infix notation)

+ 7 * 6 4 (prefix notation)

7 6 4 * + (postfix notation)

Vector implementation

Interested in creating an array that keeps track of which nodes are connected.



ARRAY

index	0	1	2	3	4	5	6	7	8	9
node	-1	1	1	1	1	2	3	3	4	4

Depth first search for tree in figure #1

Searching the tree depth first means visiting the nodes in the following order:

A, B, E, J, F, C, G, H, D, I, K, L

Breadth first search for tree in figure #1

Searching the tree breadth first means visiting the nodes in the following order:

A, B, C, D, E, F, G, H, I, J, K, L

Chapter 15: Graphs