

The Computer Science Tutoring Center

C++ Programmers Guide

Programming Course Contents:

Lesson 1	C++ PROGRAMMING COMPONENTS AND VARIABLES
Lesson 2	C++ OPERATORS, CONDITIONS AND BINARY NUMBERS
Lesson 3	C++ ARRAYS, POINTERS AND REFERENCES
Lesson 4	C++ ALLOCATING MEMORY AND STRUCTURES
Lesson 5	C++ FUNCTIONS
Lesson 6	C++ USING FUNCTIONS
Lesson 7	C++ OBJECT ORIENTED PROGRAMMING and CLASSES
Lesson 8	C++ PROGRAMMING STATEMENTS
Lesson 9	C++ INPUT AND OUTPUT STREAM CLASSES
Lesson 10	ENCAPSULATION AND INHERITANCE
Lesson 11	POLYMORPHISM AND VIRTUAL FUNCTIONS
Lesson 12	C++ LANGUAGE IMPLEMENTATION
Lesson 13	C++ TEMPLATES AND ABSTRACT CLASSES
Lesson 14	C++ VIRTUAL CLASSES, EXCEPTIONS AND RUN TIME IDENTIFICATION
Lesson 15	Library Files and DLL'S
Lesson 16	C++ COURSE PROJECT

ABOUT THESE LESSONS

These Lessons are taught at the Computer Science Tutoring Center on a individual basis. We also teach them at local colleges. Students can also take the individual Lessons by E-Mail. E-Mail students pay per Lesson. When you take the Lessons by E-Mail the price of the Lesson includes tutoring and marking. When you buy an E-Book the price of the E-Book does not include tutoring or marking of Lessons. All Lessons are now packaged in a E-Book for self study, or for reference. The last Lesson is a project. You may complete the project and E-mail it in for marking. We charge an additional \$25 charge to mark the project. For more information or to see the course offerings check out our web site:

<http://www.cstutoring.com>

To view our other Programming E-Books check out:

<http://www.csebooks.com>

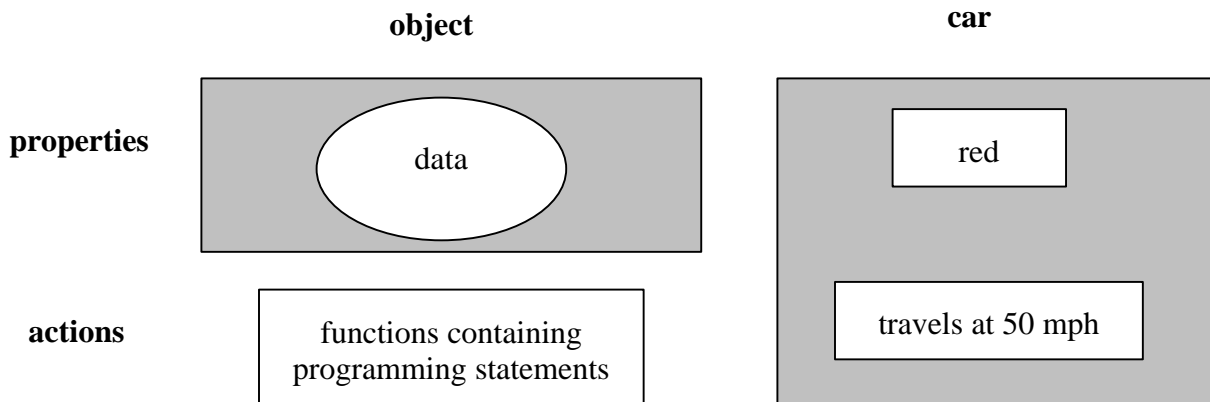
We hope you enjoy these Lessons. The Lessons are designed for easy reading. The Lessons are being updated constantly. An E-Book is the ideal solution, it can always be updated. Our goal is to teach programming to everyone.

C++ PROGRAMMERS GUIDE LESSON 1

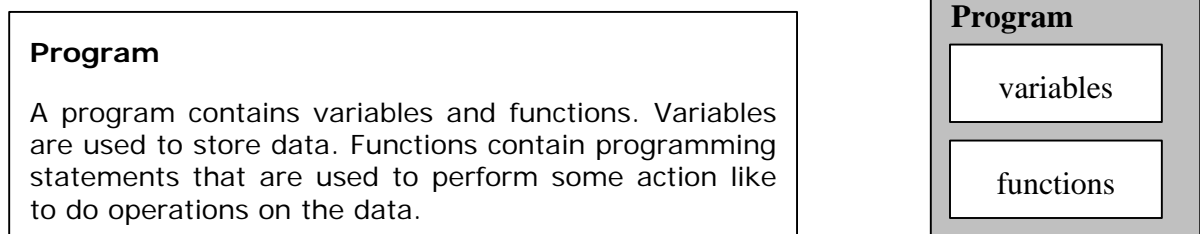
File:	CppGuideL1.doc
Date Started:	July 12, 1998
Last Update:	Mar 21, 2002
Version:	4.0

INTRODUCTION

This manual will introduce the C++ Programming Language techniques and concepts. C++ uses the **Object Oriented Programming** approach. This approach allows a programming language to represent everyday objects like a car. How can it do this? The answer is that an object has **properties**, like the color of the car and **action**, like it travels fast at 50 mph. How can properties and actions be represented by a programming language? **Properties** can be represented by **data** and **action** can be performed by programming statements. A programming language uses **data** to represents **properties** and programming **statements** to perform **actions**. This action can be represented by changing the value of the data in the object



Data is a value stored in the computer memory. Every **data item** is represented in a program as a **variable** that has a **name** for identification like **x**. The **location** where the **data** is stored in the computer memory is represented by the variable name. Programming statements used to perform a certain action are grouped together and are called **functions**. The function also get a name for identification. A function allows the program to **execute** the programming statements represented by the function name. The data and functions make up program.



Variables

A variable is used to represent a place in the computer memory that holds a data value. A variable gets a name like **x** that represents a memory location value.

variable x

Computer
Memory

Programming Statement

A programming statement is instructions containing commands and variable names telling the computer what to do. Each programming statement ends in a semi-colon.

```
cout << "The value of x is: " << x;
```

Function

A function groups together programming statements under a common name so that they can be executed by using the function name. A function may have temporary variables to assist in doing a calculation.

Function

temporary
variables

programming
statements

What's this Object Oriented stuff about anyway?

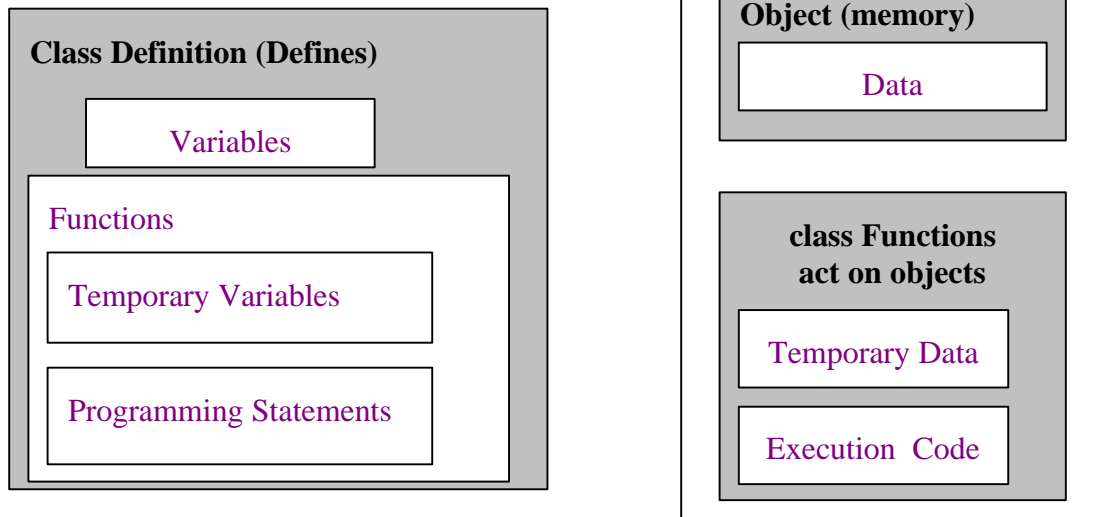
The idea behind object oriented programming is that you can have many **mini programs** each with their own variables and functions. The data part of these mini programs are called objects. The **functions** of the mini program is used to do operations on the data. Objects must be defined before they can be used. A **class** is used to define the objects. When you define a class you need to list all the variables and functions the mini program needs. A **class** is the **definition** for an object that defines the variables and functions to be used by the object. With a common definition you can make many objects all from the same class definition. The functions are associated with the created objects. You only need one set of functions for all the created objects. The Object Oriented Programming approach will enable you to organize your programs so that your programming tasks will be easier to accomplish. You must think that each class is a mini program with its own variables and functions. Objects are created from the data defined in the class. The functions act on the data in the objects. One set of functions operate on many objects.

Class

A class defines the variables an object would need and the associated functions needed to do operations on the data.

Object

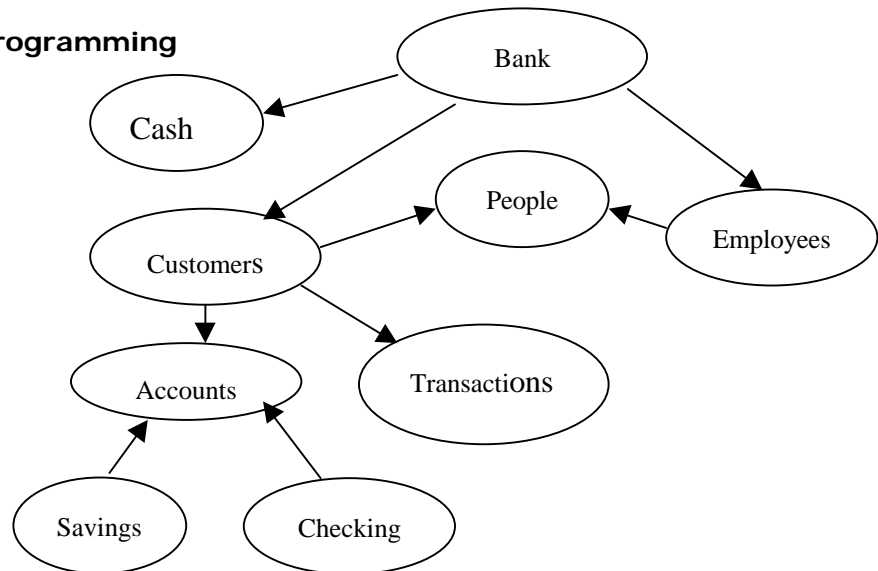
An object is memory for the variables defined in a class. The functions defined in the class are used to do operations on the data in the object.



The functions are associated with the object. What this means the same functions can be used on many objects. Each object will have its own memory area.

Example of Object Oriented programming

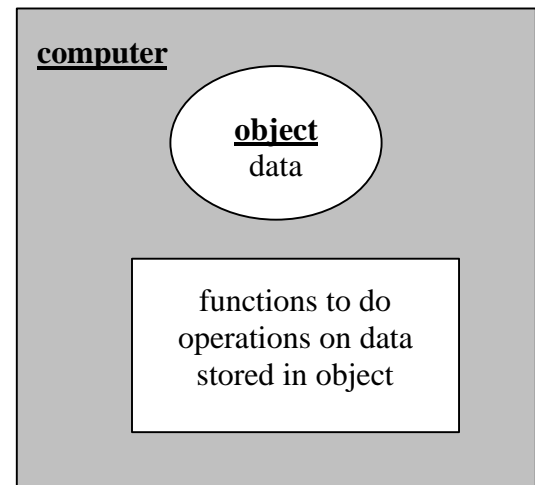
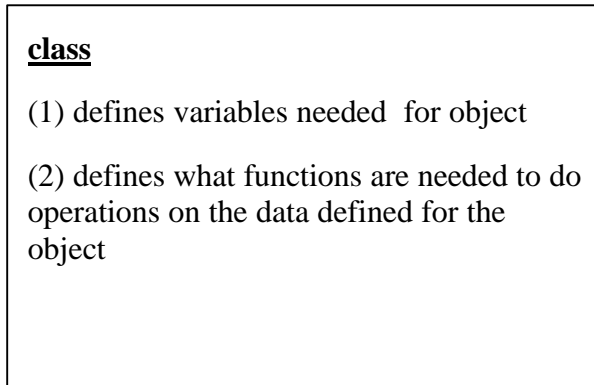
A good example is a banking system. Here we have many objects. A bank has customers, accounts and cash. Accounts may be checking or savings. Customers have accounts and do transactions with the accounts. Banks have Customers and Employees. Customers and Employees are people. Classes may be sub classes of others and classes may use other classes.



Arrows pointing **toward** other classes indicate **sub classes**. This means these classes are related to some common property. Example customers and employees are people. Just like cats and dogs are animals. Arrows pointing **away** means one class **uses another class**. Example a bank has customers and employees.

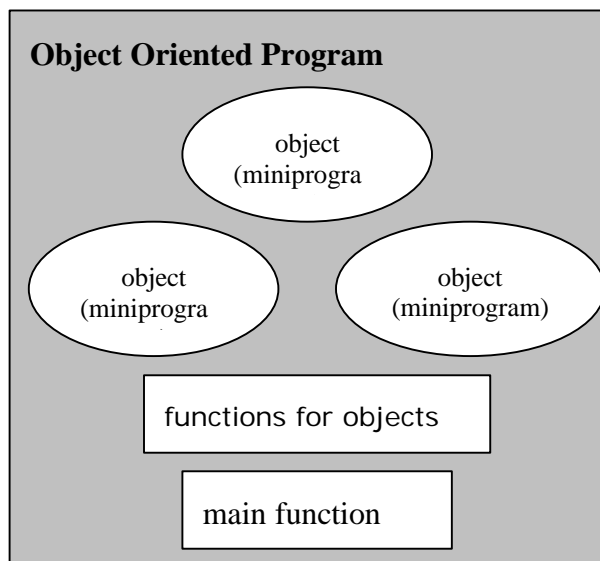
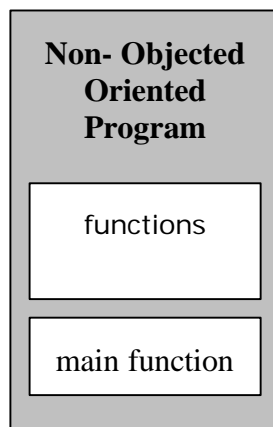
What is the difference between a class and an object ?

A class is the definition that defines what variables an object would need and what functions are needed to do operations on the data stored in the object. The object is memory in the computer for the variables defined in the class.



What is the difference between an object oriented program and a non-object oriented program?

An object oriented program uses objects as subprograms (defined by classes) to do dedicated tasks. Objects are created in computer memory. A non-object oriented programs just uses many functions.



Think that the class is the definition that many objects are made from, just like a recipe is the definition that many cakes are made from. In this situation the recipe is the class and the cake is the object. It is obviously you cannot eat a recipe but definitely you can eat a cake!

How these Lessons Work

This manual teaches by using the **analogy** approach. This approach allows you to understand concepts by comparing the operation to common known principles and concepts. This makes it easier for you to understand and grasp new ideas. We also use the **seeing** approach, the words we use are carefully chosen so that you get a picture of what is happening. Visualization is a very powerful concept to understanding programming. Once you get the picture of what is happening then your programming task is much easier to understand and accomplish. It is very important to visualize and see things as a picture, rather than trying to understand by reading and memorizing textbooks. Pictures are made up of words. This document does not cover all details of C++ programming but acts as a guide so that you can get started right away with C++ programming. The main purpose of this document is to introduce and teach you C++ Object Oriented Programming techniques. The reader is encouraged to have a textbook as a companion to look up important terms for clarification or for more in depth study. Good textbooks are:

Title	Author(s)	Publisher
Object Oriented Programming in C++	Richard Johnsonbaugh Martin Kalin	Prentice Hall
Applying C++	Scott Robert Ladd	M & T Books
Programming with C++	John Hubbard	Schaums Outlines

PURPOSE OF PROGRAMMING LANGUAGES

The purpose of a programming language is to direct a computer to do what you want it to do. In most cases of beginner programs, the computer tells the programmer what to do. Good programmers are in control of the computer operation. A typical computer program lets the user enter data from the keyboard, performs some calculation to solve some problem and finally displays the results on the computer screen. When the user gets tired of typing input data, the user can then store input and output data on a file for future use.

LESSONS AND ASSIGNMENTS

You should understand all the material in lessons before proceeding to the next lessons. Grading is based on the exercises. (P) Pass, (G) Good and (E) Excellent. Excellent is awarded to students with outstanding programs that involves creativity and works of genius. Good is awarded to students that have exceptional working programs. Pass is awarded to students that have minimal working programs. Each lesson has exercises that the student should attempt. If you do not do these exercises then you will not be a good programmer or understand the material in the next lesson. New C++ keywords and concepts are introduced in **bold**, C++ language definitions are in *italics* and programming statements are in **blue** and comments in **green**.

LESSON 1 C++ PROGRAMMING COMPONENTS AND VARIABLES

Before we learn how to write programs we first need to know the individual components that make up a program.

C++ PROGRAM FORMAT AND COMPONENTS

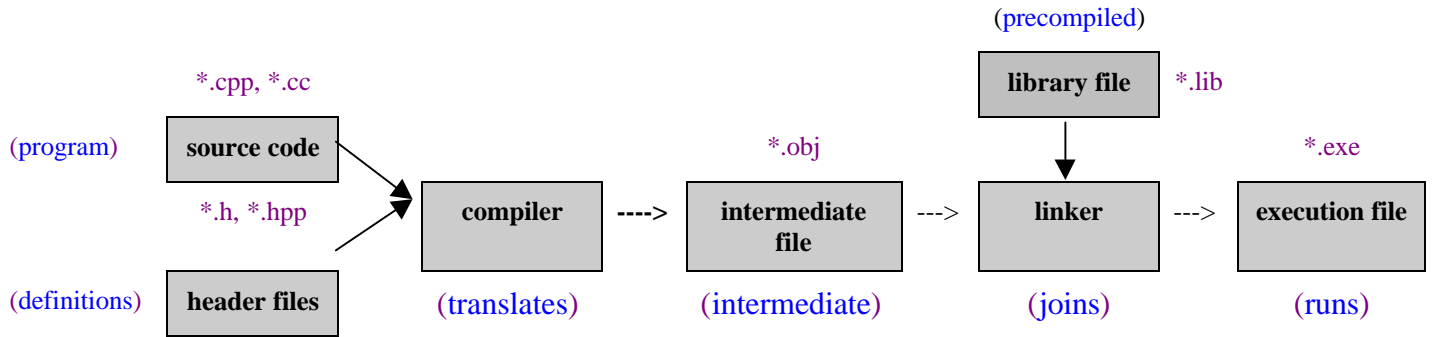
A C++ program is made up of programming components usually arranged in a predefined format. By following a proven format, your program will be more organized and easier to read.

A C++ program usually starts with **include file's**, **define and const statements**, **enumeration's**, **function definitions**, **structure definitions**, **class definitions**, **class implementation**, **function implementation**, **global variables and tables** and finally the **main function**. We will discuss each component in detail.

Comments
#include statements
#defines and const statements
typedef's
Enumeration's
Structure definitions
Function declarations
Class definitions
Class implementations
Global variables
Main function
Function implementations

compiling, linking, executing C++ programs

A C++ program is written as a **text file** made up of words using programming statements. Before you can run your program on your computer it needs to be compiled into an **executable file**. It is this executable file that gets loaded into the computer when you run your program. The program you write is called **source code** and has the extension ***.cpp**. A program may need additional definitions to describe additional information. These additional program definitions may be put into a separate file called a **header file** having the ***.h** or ***.hpp** extension. A header file contains the class and function definition of the source files. The header file provides the compiler with the information it needs to know to compile your program when you refer to external classes and functions. A famous header file is [**<iostream.h>**](#) located on the top of every C++ program. A compiler translates program **source code** and **header files** into an **intermediate code** having the extension ***.obj**. Code that has been previously pre-compiled for functions supplied by the compiler are contained in a library file. All intermediate code files and library files are linked together into an execution file. The linker may link together many intermediate and library files. Intermediate ***.obj** files are not to be confused with Objects. When your program runs on the computer, it is executing the machine code stored in the execution file. The source code files have extension ***.cpp** or ***.cc**, the header files have extensions ***.hpp** or ***.h**. The intermediate files have extension ***.obj** and library files have ***.lib** and the execution files have extension ***.exe** or just the file name with no extension.



Comments

Comments are messages used to explain to the reader what the program statements suppose to do. Comments may be placed anywhere in your program. There are two styles used to write a comment. In the first style a message is proceeded by a `/*` and followed by a `*/`. This style is usually used to explain a section of programming lines. The comment may span multiple lines.

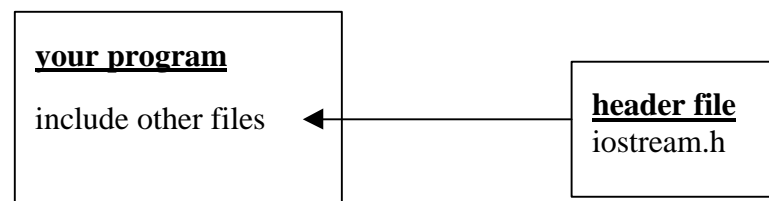
```
/* My First C++ Program */
```

A comment is also introduced by a `//` followed by the message. The end of line is also the end of the comment. This style is used to explain individual lines of code. This method is more preferred. The comment may be only one line.

```
// C++ Course Lesson 1 Program 1
```

#include preprocessor statement

#include statements are used to tell the compiler which header file or source file you want to be included as part of your program. Source files are other C++ programs where header file's contains labels, function, structure and class definitions.



The **compiler** uses the class and function definitions in the header file when it **compiles** your program. An example of a header file is the **input/output** stream class definitions that allows you to get data from a keyboard or send data to the computer screen.

```
#include <header_file_name>
```

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

The `< >` means the file located in **compiler directory**. Include statements are placed at the top of your program.

Your program may need other header files that you wrote. You put your own function and class definitions in the header file. In this case the header file names are enclosed by **quotes** means the compiler will find it in your **working directory**.

```
#include "user_header_file_name"
```

```
#include "mypgm.hpp"
```

Many programmers like to put all their definitions in a header file and all implementation in the source code file. Include files have the extension of **".h"** or **".hpp"** which refers to "header" or "header plus plus" file. Source code files have the extension **".cpp"** or **".cc"**.

Constants

Numbers like (0-9) and letters like ('a'- 'Z') are known as constants. Constants are hardcoded values assigned to variables. There are many different types of constants.

character: 'A' **numeric:** 9 **decimal:** 10.5 **string:** "hello"

Letters are known as characters and are specified by using single quote 'A'. Characters are assigned numeric values for identification from a chart known as an ASCII table. The letter 'A' is assigned the numeric value 65. Numbers like 9 represent a numeric whole number. Decimal numbers like 10.5 represent a whole number and a fraction. What is the difference between '9' and 9 ? Numbers may be grouped together to make larger numbers like 1234. Characters that are grouped together are known as **character strings**: For example: "hello". Character strings are enclosed by double "quotes ". What is the difference between "1234" and 1234 ?

backslash characters

The backslash codes allow you to specify new line, etc. in your character strings like: "hello\n"

code	description	code	description
\b	backspace	\0	end of string terminator
\n	new line	\\	backslash
\r	carriage return	\v	vertical tab
\t	horizontal tab	\a	bell
\"	double quotation mark	\o	octal constant
\'	single quotation mark	\x	hexadecimal constant

representing numbers in a computer

All memory is made up of bits having values 0 and 1. 8 Bits are grouped together into a byte. Each bit has a weight to 2 to the power of N.

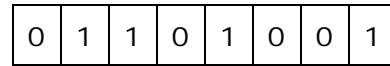
1 byte is 8 bits

0	1	1	0	1	0	0	1
---	---	---	---	---	---	---	---

bits are 1 and 0's

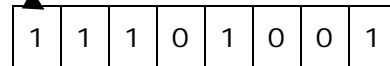
Numbers may use from 1 to many bytes. Groups of memory bytes are used to represent numbers. The smallest to largest number a data type can represent is known as the **range**. Small range numbers use few bytes of memory where large range numbers use many bytes of memory. Numbers may be **unsigned** or **signed**. **Unsigned** meaning positive numbers only where **signed** means negative or positive numbers. The first bit Most Significant Bit (MSB) in a **signed** number is called the sign bit and determines if the number is negative or positive.

Signed **Positive** numbers start with a 0.



sign bit

Signed **Negative** numbers start with a 1



For unsigned numbers it does not matter since the number will always be positive meaning greater or equal to zero. It is important now to understand how signed and unsigned numbers are represented in computer memory. The same binary numbers are used to represent negative or positive numbers. It is just how they are forced to be represented in your program. This means the same binary number can be a positive number like 192 if **unsigned** or a negative number like -64 if **signed**. The number representation are known as **number wheels**.

RANGE OF NUMBERS

The **range** of a number tells us the smallest number and the largest number a memory can represent. Two types of numbers:

(1)	signed	positive and negative numbers	+ -
(2)	unsigned	positive numbers only	+

We now present how to calculate the range of signed and unsigned numbers.

The range of an unsigned number is:

0 to $(2^N \text{ bits}) - 1$

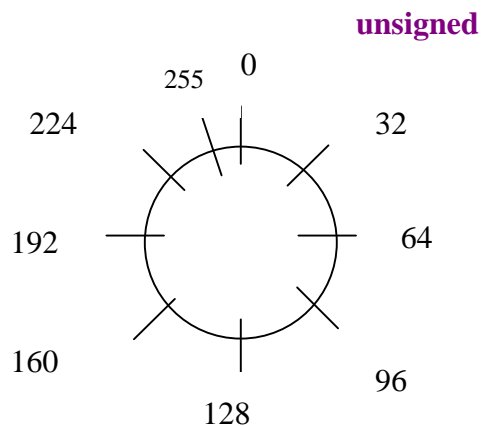
Positive numbers only

Example: if N equals 8 then:

N = 8 bits

0 to $(2^N \text{ bits}) - 1$
0 to $(2^8) - 1$
0 to $(256 - 1) = 255$

The range of an 8 bit **unsigned** number is 0 to 255.



The range of a signed numbers is:

$$-2^{(N-1)} \text{ to } (2^{(N-1)}) - 1$$

Positive and negative

signed

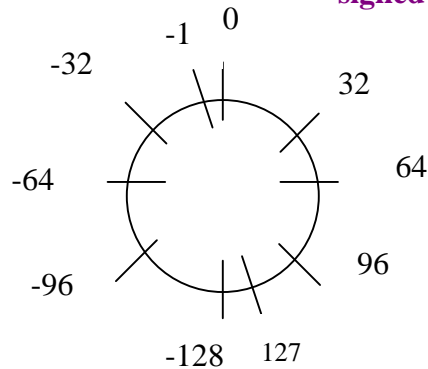
To calculate the range of a signed number for N = 8 then:

N = 8 bits

$$-2^{(N-1)} \text{ to } (2^{(N-1)}) - 1.$$

$$-2^7 \text{ to } (2^7) - 1$$

$$-128 \text{ to } 127$$



The range of an 8 bit **signed** number is -128 to 127.

The negative range gets 1 extra number because you are splitting an even number into two parts. The negative numbers are: -128 to -1 and the positive numbers are 0 to 127. Both have 128 numbers.

Data types

Data types are used to specify how data is to be represented in a computer memory. Data types tell the compiler what kind of data, the computer memory is to represent. Why do we need different data types ? We need different data types because we want to minimize the amount of memory space we use and we need to identify what kind of data the memory is to represent. The following chart lists all the C++ data types with range and an example of the data it is to represent.

data type	bytes	bits	example	unsigned range (positive)	signed range (negative / positive)
char	1	8	'A'	0 to 255	-128 to 127
short	2	16	1234	0 to 65,535	-32,768 to 32,767
int(16 bit)	2	16	12345	0 to 65,535	-32,768 to 32,767
int(32 bit)	4	32	12345456	0 to 4294967296	-2147483648 to 2147483647
long	4	32	12345456	0 to 4294967296	-2147483648 to 2147483647
float	4	32	34.56	+/- 1.2e38 to +/- 3.4e38	
double	8	64	2.3456453e12	+/- 2.2e-308 to +/- 1.8e308	

signed and unsigned data types

char, **short**, **int** and **long** data types may be forced to be signed with the **signed** keyword or unsigned with the **unsigned** keyword. The default is supposed to be signed, but if you are not sure then you can force by using the **unsigned** and **signed** keywords.

unsigned int	force int to be unsigned	(positive only)	(default)
signed int	force int to be signed	(positive/negative)	

size of int

The **int** data type may be 16 or 32 bits. The **int** data type is platform dependent, this means the size is depends on the computer and operating system used. You can use the **sizeof** operator to tell you how many bytes a certain data type has.

```
sizeof(int); // number of bytes for data type
```

float and double number representation

Float and **double** data types are used to represent large numbers known as floating point numbers stored in a small memory space. Floating point numbers have a **fraction (mantissa)** part and an **exponent part** and represented in decimal point notation 24.56 or exponent notation 3.456E04. It is the stored exponent that allows the floating point number to represent a large value in a small memory space. The following is a 32 bit floating point format known as a **float**, where the sign, exponent and fractional parts are stored separately. Floats numbers have **low precision**.

1	8	23	32 bit floating point number (float)
sign bit	exponent	fraction (mantissa)	$(-1)^{\text{sign}} * 2^{\text{exponent}-127} * (\text{mantissa})$
-	12	.12365	$1.12365 * 10^{12}$

The following is a 64 bit floating point format known as a **double**, where the sign, exponent and fractional parts are stored separately. Double has greater precision **over float because it stores more bits for the exponent and fraction. Double numbers have high precision** (lots of decimal points)

1	16	47	64 bit floating point number (double)
sign bit	exponent	fraction (mantissa)	$(-1)^{\text{sign}} * 2^{(\text{exponent}-127)} * (\text{mantissa})$
-	12	.12365	$1.12365 * 10^{12}$

VARIABLES

Variables let you store and retrieve data in a computer memory. The **variable name** lets you identify a particular location in the computer memory represented by a memory address. The compiler assigns the memory address automatically for you when it compiles your program. A variable is like a bank account. You can put money in and take money out. The particular place where your money is located is identified by the bank account number.

Variables must be declared before you can use them. Variables are declared with different **data types**. Just as a bank account would have different currencies. Common data types used are **char**, **int**, **float** and **double** etc. Before you can use a variable it must be **declared**. To declare a variable you specify the **data type** and the **variable name**. The data type indicates what kind of data the variable is to represent. Declaring a variable is just like opening a bank account.

declaring variables

In C++ variables may be declared when you need them. When variables are declared, a memory location is reserved for them when the compiler is compiling your program (**compile time**). To declare a variable you specify the data type and variable name. The declaration ends with a semicolon.

data_type variable_name;

int x;

data type
variable name

**Variables represent a value
at a memory location**

When your program runs?? (**run time**), the value of the variable at that memory location is **undefined**. In the computer memory every byte is represented as an address. The addresses for variables are chosen by the compiler and linker and can be any address representing a memory location in the computer memory. Addresses for variables **increment by the data type** size after they are declared.

examples declaring variables

Here are examples declaring variables:

data_type variable_name;

int x; // declare variable x of data type int

int y; // declare variable y of data type int

char ch; // declare variable c of data type char

address	name	value
1000	x	??
1002	y	??
1004	ch	??

Why do the addresses increment by 2 ?

You can also declare many variables of one data type at the same time in a list, each variable name is separated by a comma.

data_type variable_name_list;

int i,j,k; // declare variables i, j, k of data type int

address	name	value
1005	i	??
1007	j	??
1009	k	??

Why is ch at address 1004, i at address 1005 and j at address 1007 ?

Declaring and initializing variables

In C++ variables may be instead declared and initialized at the same time. Declaring and initializing variables is like opening a bank account with an **initial deposit**. A variable is declared and initialized with a constant or another variable name. If a variable is initialized to another variable then that variable is initialized with the value represented by that variable. The assigned value is known as an **expression**. An expression can be a variable or a constant. The expression on the right hand side is evaluated and used to initialize the variable on the left hand side of the initializing operator "=".

data_type variable_name = expression;

`int x = 5;` // declare and initialize variable x to 5

`int y = x;` // declare and initialize variable y to the value of x

address	name	value	
1000	x	5	
1002	y	5	(x)

What is the value of x ? What is the value of y ?

The variable y is assigned the value represented by the variable x.

You can also declaring and initialize many variables of the **same data type** in a name list.

data_type variable_name_list;

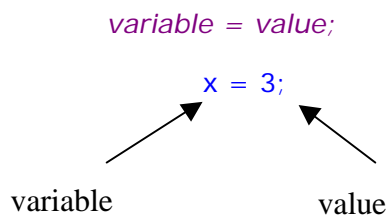
`int i=10,j=20,k=30;` // declare variables i, j, k of data type int

address	name	value
1005	i	10
1007	j	20
1009	k	30

What is the value of i ? What is the value of j ? What is the value of k ?

Assignment statements

Assignment statements allow you to assign a value to a variable that has been **previously declared**. Variables are reusable, you may assign and retrieve values from them at anytime. Assignment statements assign the **value** on the right hand side of the **assignment operator** "=" to the variable on the left hand side.



Once you have your variables you can give them new values.

An value can be a variable or a constant. The assigned value may be a numeric constant or another variable. An assignment statement is like depositing and withdrawing money from a bank account or transferring money from one account to the other. You do not have to re-declare the variable when using the assignment statement. This would be like opening up the same bank account twice. You can only declare a variable only once or you will get compile errors. You may use the assignment statement anywhere in your program. Each variable must be previously declared before you can use the assignment statement.

variable_name = expression;

x = 3; // variable x gets the value of 3

y = x; // variable y gets the value represented by x

ch = 'A'; // variable c is assigned the character 'A'

address	name	value	
1000	x	3	
1002	y	3	(x)
1004	ch	'A'	(65)

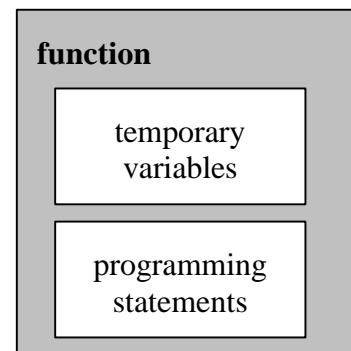
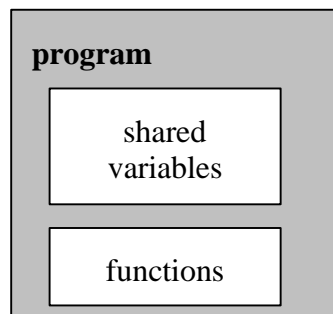
The letter 'A' is represented in computer memory by the numeric value 65. Each letter on the keyboard gets a numeric value. All the numeric values are listed in a chart known as the ASCII chart.

What is the difference between 'x' and x ? One is a constant the letter 'x' and the other is a variable name x representing a value.

When you say **y = x;** **y** gets the **value** represented by **x**.

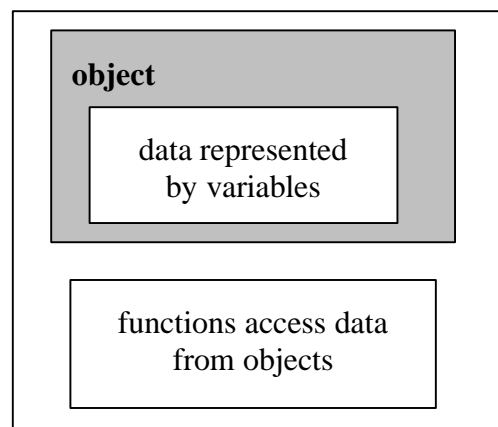
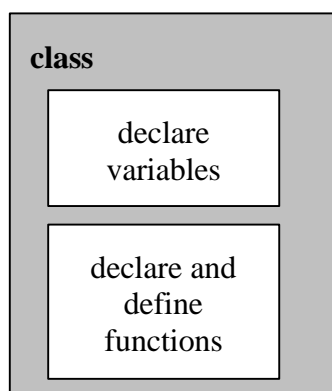
Variables in Programs and Functions

Variables can be declared in a program or a function. Variables are usually declared in a function. Variables declared in a function are only known to that function. They are called temporary or local variables because they only retain their value when the function is being used. Variables declared in a program always retain their value for the lifetime of the program. Variables in a program may be used by all functions in the program.



Variables defined in classes

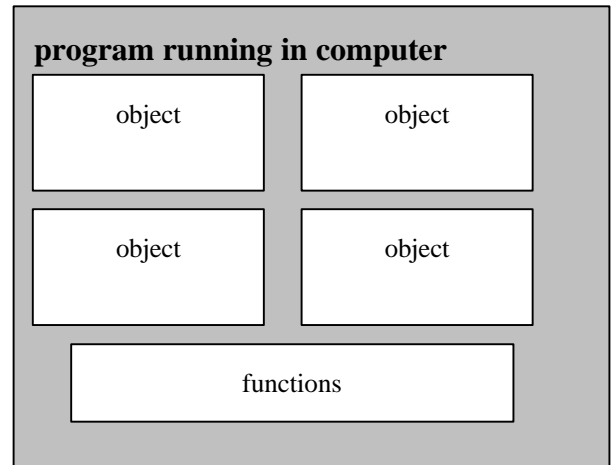
An object is allocated memory for the variables defined in a class. The data lasts for the life time of the object. The functions defined in a class share all the variables defined in the class.



classes and Objects

Classes must be defined before they can be used to create objects. An object is constructed from the class definition. When you define a class you are listing all the variables and functions needed by the object. An object is like a mini-program that has its own variables and functions. An object has a dedicated task to do. The variables used in the object are declared in the class definition. The data represented by variables in an object are permanent and retain their values as long as the object is in memory. The variables declared in a class can be used by all the functions declared in the class. This is one of the major reasons classes were invented, so that a group of functions having a common purpose may share variables.

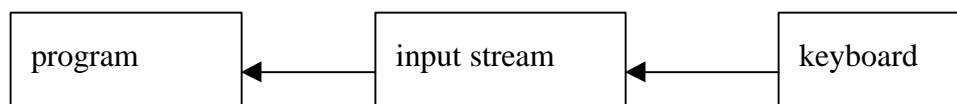
Object Oriented Programming allows programs to be very organized and data to be secure. One program may define many classes. By having many classes, each having their own data variables, makes programming tasks very easy. Each class will have a dedicated purpose. The variables in the class will store the data for the functions declared in the class. The functions of each class will also be used to transfer data to and from other classes. When your program is running, memory is allocated or reserved for the variables defined in your class. The memory is known as an **object**. Objects are created from class definitions.



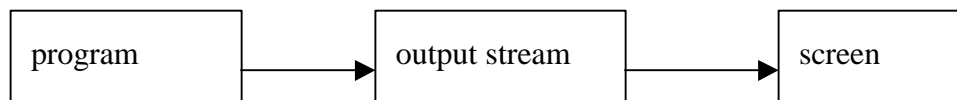
Do not be too worried of understanding classes and functions for now. You just need to be familiar with the terminology. You just need to know something about classes and objects to understand input and output stream classes. Just remember classes are what you type in and objects are allocated memory for the variables defined in your class. Classes act like subprograms with their own dedicated variables and functions.

INPUT/OUTPUT STREAMS

In C++ all input and output is done through streams. Streams allow data to flow. **Input streams** allow your program to receive data from the key board or a file.

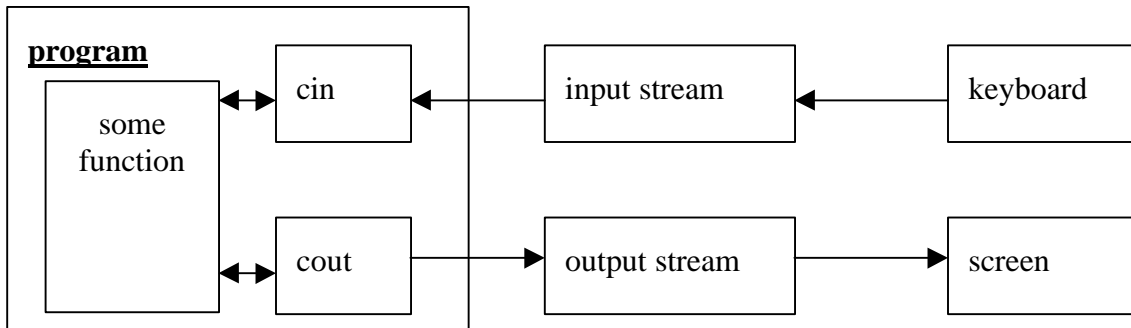


Output streams allow your program to send data to the computer screen or to a data file.



All streams in C++ are grouped into classes. This means all variables and functions needed to process **input** data streams are grouped together in a **input stream** class. At the top of every C++ program you always see `#include<iostream.h>`. This include statement is telling the compiler to include all the information needed to compile your program using the I/O stream classes.

The `<iostream.h>` definition file also automatically creates stream objects **cin** and **cout** from the **istream** class definitions. The **cin** object created from the **istream** class is used to get data from the keyboard and the **cout** object created from the **ostream** class is used to send data to the computer screen.



Each class has a dedicated task. The **cin** object created from the **istream** class is used to get data from the keyboard. The **cout** object created from the **ostream** class is used to send data to the computer screen. They are like little mini programs with dedicated tasks. Objects make your programming life easier. You don't need to use many different functions to read and write data, you just use the objects. A lot of work is done for you.

using cout

To print a message on the screen you use the output stream object **cout** and the **<< output stream operator** and the message you want. The output stream object **cout** is automatically created for you in `<iostream.h>`

```
cout << "Enter a letter: "; // print message to screen
```

Enter a letter:

You can also print out a value of a variable to the screen. To print out the value of a variable you just list the variable with **cout** and the **output stream operator** `<<`.

```
cout << ch << endl; // print value of ch to screen
```

a

endl means starts a new line after printing the message. You can also print a message with a variable value this is called **chaining** `<<`.

```
// print message and value to screen
cout << "the letter you typed is: " << ch << endl;
```

the letter you typed is: a

using cin

To get a number from the keyboard the input stream class object **cin** is used. The input stream object **cin** is automatically created for you in `<iostream.h>`. To read the value of a variable from the keyboard you just list the variable with **cin** and the **input stream operator** `>>`.

```
char ch; // char variable to hold letter from keyboard
cin >> ch; // get a value from the keyboard and put into the variable ch
```

You can get more data from more than one variable at a time by **chaining** >>.

```
int a,b,c,d;
cin>>a>>b>>c>>d; // get many inputs all at once
```

It's easy to remember which way the arrows go when using **cin** and **cout**. For **cout** they point **outward** toward **cout**. For **cin** they point **inward** toward the data variable receiving data.

main function

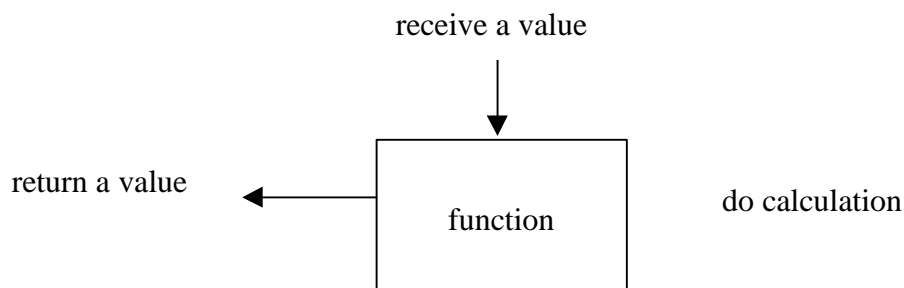
A simple C++ program just only requires a main function and no classes. A **function** is a group of programming statements identified by a function name. The **main** function is the first function to executed in your program. The job of the main function is to create objects from class definitions and execute functions from those objects. We introduce the main functions at this time so that you can run your first C++ program and do the exercises.

<pre>void main() { variables and program statements }</pre>	<pre>#include <iostream.h> // main function void main () { cout << "Enter a letter: " << endl; char ch; // declare char variable ch cin >> ch; // get a character from keyboard cout << "the letter you typed is: " << ch << endl; }</pre>
--	---

program output:

```
Enter a letter: a
the letter you typed is: a
```

Functions usually receive a value, do a calculation and return a value.



The main function starts with the keyword **void**, which indicates that the main function does not return a value. The main function has the name **main**. Function names end with an open and closed round bracket () to distinguish a function name from a variable name. Function statements are introduced by a open curly bracket "{" and the function ends with a closing curly bracket "}". The above C++ program asks the user to type in a character from the keyboard. The **cin** object is used to get the character from the keyboard. The **cout** object is used to print out the character on the computer screen.

LESSON 1 EXERCISE 1

You may type in the above program in your compiler and run it. Call your program file L1ex1.cpp. Change the message and instead ask the person to type in a number, then print out the number.

LESSON 1 EXERCISE 2

Write a C++ program with a main function that declares five variables of different data types each initialize with a value. Declare and initialize variables as you need them. Use **cout** to print the values out to the screen. Ask the user to enter a number for each of the variables. Use **cin** to get numbers from the keyboard for each variable. Print out the new values. Call your program file L1ex2.cpp. Print out the size in bytes of each variable using the **sizeof** operator.

The **sizeof** operator tells you the number of bytes a data type or variable uses.

```
int sz = sizeof(x);

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

EVOLUTION IN PROGRAMMING LANGUAGES

People think in a higher abstract level than computers do. Program needs names to have meaning. People like to work with ideas and representations. Computers like to work with numbers. The job of the compiler is to translate a human ideas into numbers that a computer can work with. C++ has mechanisms that allow you to work in abstract representations, the compiler will convert the abstract representation into a number automatically for you.

Define and const statements

Numeric values like 10 and characters like 'A' are called **constants**. **Define directives** tell the compiler to substitute a numeric or string value for a **label**. A label is a name used to represent a constant. A label is not a variable and does not use any computer memory for storage. A label just is an **identifier** used to represent a constant. Once a label is defined to represent a constant value it cannot be redefined. Define statements are declared at the top of your program outside any functions.

#define label expression

#define MaxSize 10

Label

Constant

Associates a constant with a label

In your program you use the label MaxSize to represent the value 10.

```
int x = MaxSize;
```

Every time the compiler sees MaxSize the numeric value 10 is substituted. You must not put a semi-colon at the end of the **#define** statement because the compiler would substitute **10;** rather than **10**. The define statements are placed at the **top** of a program and cannot be included in the main function or any other functions.

const operator

Constants may also be defined using the **const** operator:

```
const data_type constant_name = constant_value;
```

```
const int MaxSize = 10; // preferred method
```

Read only

When you use **const** you can define constants anywhere in your program. **Const** are known as **read only**. Once they are initialized they **cannot be changed**. **Const** means cannot change. When you declare a constant the data type is optional and can be omitted, the data type is defaulted to integer.

```
const constant_name = constant_value;
```

```
const MaxSize = 10; // integer data type assumed
```

The const method is much preferred over **#define**, because it gives the constant a data type that the compiler can use to enforce for data type checking. A const is considered a read only variable and contain a memory location representing the read only value. const statements may be declared at the top of your program outside any functions or in functions.

purpose of #define and const

Why do we need #define or constant statements? We do not want to put numbers in our programs. Using **#define** or **const** statements is a must. There should be no hardcoded numeric values in your program. The purpose is this, if you change MaxSize to be 12 then you do not need to change all 10's to 12 in your program. Without using a **#define** directive or **const** operator you may inadvertently change some 10's to 12's that don't need to be changed. Your program may then not operate as expected. **Const** statements may be placed anywhere in your program. They maybe included in functions. **Const** statements have the advantage over **#define** statements is that they can be placed anywhere in your program. If you can understand that the compiler is going to substitute a numeric value for a label then you are on your way to becoming a programmer.

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

```
#define MaxSize 10    // define label MaxSize to represent constant 10
void main()
```

```
{
const int Max = 20;
int x = MaxSize;
int y = Max;
cout << "MaxSize: "<< x<< " Max: "<< y<<endl
}
```

Program Output:

```
MaxSize: 10  Max:  20
```

Typedef's

Typedef's (**type definitions**) allow you to define your own **data types**. In C++ common data types are **int**, **char**, **float**, **double** etc. There will be many occasions when it is nice to have your own data type. Why do you need your own data type? When you have your own data type it gives **meaning** to your declared variables. Your own data type must be made up of a known C++ data type. A common user data type people use is the **bool**. Bool means **true** or **false**.

True is a **non zero** value, where false is a **zero** value. (bool is automatically a typedef in most C++ compilers now). With a bool data type then the variables you declared will have **meaning** of **true** or **false** values. **Typedef** lets you associate your own data type for a C++ data type and lets you have your own data types with meaning.

true and **false** can be set with **#define** or **const**.

True is a **non zero** value, where false is a **zero** value.

```
#define true 1
#define false 0
```

```
const int true = 1; // any non zero number
const int false = 0; // zero number
```

Now we can define our own data type called **bool**.

your own data type with meaning

```
typedef data_type user_data_type;
```

```
typedef int bool; //your own data type bool representing true or false
```

C++ data type

user data type

Now you have a **bool** user data type that is the same as the C++ data type **int**. But the advantage is that to you **bool** means **true** or **false** but to the compiler **bool** means **int**. The compiler will substitute your abstract data type that means true or false with its own int data type meaning 1 or 0. You may discover that some compilers have already made a bool data type for you. Typedefs are defined out side any function usually at the top of your program.

using your own user data type

To use your own data type declare a variable with your user data type and assign a value to it:

```
user_data_type variable_name = value;
```

```
bool keyflag = true; // key flag having user type bool
```

```
int keyflag = 1; // compiler substitution
```

The following is a small program using typedef.

```
#include <iostream.h>

// define label Maxsize to represent constant 10
#define True 1
#define False 0
typedef int bool; // define your own user data type to represent true or false

void main()
{
    bool keyflag = true; // key flag having user type bool
    cout << "keyflag: " << keyflag << endl;
}
```

Program Output:

```
keyflag: 1
```

Why does the output say 1 rather than true ?

Type definition is a very important concept in programming. The compiler is always substituting a high level representation to a low level representation. Humans like to use words with meaning that describe what the representation is suppose to do. Computers like words that describe what the computing machine likes to do like crunch numbers. If you can understand the typedef concept then your understanding of programming is almost complete. You may use typedef's anywhere in your C++ program. **Typedefs** are usually defined at the top of your program

LESSON 1 EXERCISE 3

Write a program that uses your own data type like days of week using **typedef**. Ask the user to enter some data for your days of week data type and display the results. You will need 7 definitions or constant values for days of the week. Start with Sunday equals 0. Call your file L1ex3.cpp.

Enumeration's

Enumeration's allow you to assign **sequential values** to a **group** of labels under a common name. When you declare an **enumeration** you are also making your own user data type. For example you may need a days of the week data type and have to assign sequential values to each day of the week. Example assign 0 to Sunday all the way to 6 to Saturday. Enumeration's are like many #define or const statements. The default data type of enumerated labels are **int**.

```
#define Sun    0
#define Mon    1
#define Tue    2
#define Wed    3
#define Thur   4
#define Fri    5
#define Sat    6
```

```
const int Sun = 0;
const int Mon = 1;
const int Tue = 2;
const int Wed = 3;
const int Thur = 4;
const int Fri = 5;
const int Sat = 6;
```

enum lists you do the same thing automatically, but using a list of labels. The first label is automatically assigned to 0 and each following label value is incremented by 1.

```
enum    enum_name    {enumeration_list};
```

0	1	2	3	4	5	6
---	---	---	---	---	---	---

```
enum    DaysOfWeek    {    Sun,    Mon,    Tues,    Wed,    Thurs,    Fri,    Sat    };
```

You now have your own **user data type** representing the days of the week. Enumeration's can be declared anywhere, in classes or functions, so they are used more than typedef's. You do not need to start at zero. You may assign your own enumerated values. The next unassigned one will be the last one's value incremented by one. If **Tues** is assigned 10 then the value of **Wed** would be 11 etc. Enumerations are of data type integer.

0	1	10	11	12	13	14
---	---	----	----	----	----	----

```
enum    DaysOfWeek    {    Sun,    Mon,    Tues=10,    Wed,    Thurs,    Fri,    Sat    };
```

The enum name is optional. You can easily assign 1 to True and 0 to False using an enum.

```
enum {False,True};
```

using enumerations

To use an enumerated data type you declare a variable having a data type of your enumerated data type

```
user_data_type variable_name;
```

```
DaysOfWeek days; // declare variable days of user data type DaysOfWeek
```

You can assign a value to your variable days when it is declared. You declare a variable with a enum data type and assign a value to it. Declare an variable days having data type DaysOfWeek and assign Mon to days.

```
user_data_type variable_name = value;
```

```
DaysOfWeek days = Mon; // assign Mon to variable days
int days = 1; // compiler substitution
```

Now you and the compiler know that weekdays is a DaysOfWeek data type. Again you get to use a data type and labels that make sense and have meaning. Nobody walks around and says today is 2. People walk around and say today is Tuesday. So you should be able to program like this also. Right ? Enumeration lets you do this. This is extremely powerful. Enumeration's may be placed anywhere in your program. The added benefit is that when you trace through your program, the compiler **debugger** will now say "Mon" instead of "1" for values of weekdays. A debugger lets you execute program statements one by one and display the contents of the variables. You may use enumeration's anywhere in your C++ program. Typedefs are usually defined at the top of your program. Using the above enumeration example in a program would be:

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

```
// Declare and define a DaysOfWeek user data type
```

```
enum DaysOfWeek {Sun, Mon, Tues, Wed, Thurs, Fri, Sat};
```

```
void main()
```

```
{
    DaysOfWeek days = Mon; // declare a DaysOfWeek variable called days
    cout << "today is: " << days << endl; // print out value of days
}
```

Why does it say 1 rather than Monday ?

program output:

today is: 1

LESSON 1 EXERCISE 4

Write a program that uses your own data type like days of week using **enum**. Ask the user to enter some data for your days of week data type and display the results. You will need 7 definitions or constant values for days of the week. Start with Sunday equals 0. Call your file L1ex4.cpp.

LESSON 1 QUESTION 1

1. What is constant ?
2. Give examples of constants.
3. What is a label ?
4. What is `#define` or `const` statement used for ?
5. Why would we want to use a `define` or `const` statement in your program ?
6. What is the difference between a `#define` statement and a `const` statement ?
7. What is a data type ?
8. Name 5 data types.
9. Why do we need different data types ?
10. What are variables used for ?
11. Why would you want your own user data type ?
12. How would you make your own user data type ? Give an example.
14. What does an enumeration do ?
15. Give another example of an enumeration.

constant
<code>#define</code>
<code>const</code>
data type
variable
<code>typedef</code>
Enumeration

TypeCasting

C++ is a highly typed language, this means every variable must be assigned to the same data type. It also means functions must also be passed data types it knows about. When you use other peoples functions you must supply them with the data type they know about. Nobody knows about your data types. You need to tell the compiler what they are or you may need to force one data type value to be another data type value. This is what type casting is all about forcing an old data type value to a new data type value. The old data type value does not change. The new data type receives the forced converted value from the old data type value. You type cast by enclosing the forcing data type in round brackets proceeding the variable or value you want to force.

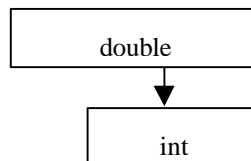
```
new_data_type = (typecast) old_data_type.
```

```
int x = (int) d;
```

A good example of type casting is trying to force a double to be an integer,

```
double d = 10.5;
```

```
int x = (int) d;
```



To be able to force a double data type into a int data type we **type cast**. When we typecast we loose some of the data. In this case we loose the 0.5 and x gets the value 10. int data types cannot represent fractional data.

```
int x = 5;
DaysOfWeek days = (DaysOfWeek)x;
cout << (int)days;
```

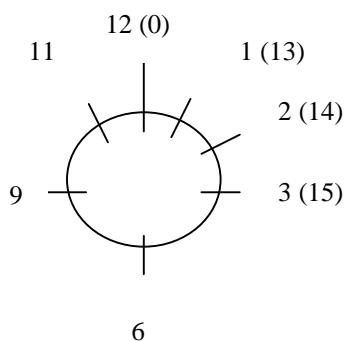
(typecast)
Force one data type value to
represent another data type value

x is an **int** data type initialized to some value. We then force the value of **x** to a **DaysOfWeek** data type assigned to **days**. We next force the **DaysOfWeek** day type to an **int** to print to the screen. **cout** does not know anything about the **DaysOfWeek** data type, but it knows what an **int** is. Although enumeration's have a default of **int** the compiler only thinks **days** is a **DaysOfWeek** data type, the data type it was declared with.

LESSON 1 EXERCISE 5

Write a small program that just includes a main function by answering the following questions. Call your program file L1ex5.cpp. All statements are sequential. Declare variables as you use them. You can use **cout** to print out the values of your variables and **cin** to get values from the keyboard.

1. Make a Colour user data type having three colours: Red, Green and Blue.
2. Declare a Colour data type variable and assign the value Green to it
3. Print out the value of your color variable to the computer screen using **cout**.
4. Declare a variable called **num** and ask the user to type in a number between 0 and 2.
5. Read the number from the keyboard using **cin**
6. Assign the number to your color data type variable.
7. Print out the value of your color variable to the computer screen using **cout**.



To find the modulus of a number: You multiply the remainder (fractional part after division) by the denominator.

For example: $15 \bmod 12$ is $15/12 = 1.25$

To calculate the mod: $.25 * 12 = 3$

Therefore $15 \bmod 12$ is 1.

Which is also $15 - 12 = 3$

Increment and Decrement operators

C++ has increment and decrement operators that are short form operators to add or subtract 1 to or from a variable. They save you time typing. They work in a **prefix** mode and a **postfix** mode. **Prefix** meaning increment the variable **before the assignment**, and **postfix** means increment the variable after the assignment. It is a two step process. In the first step a value is assigned to the variable on the left hand side of the assignment operator and the variable on the right hand side is incremented or decremented **before** the assignment. In the second step a value is assigned to the variable on the left hand side of the assignment operator and the variable on the right hand side is incremented or decremented **after** the assignment.

					step 1	step 2
++	increment after	y++	x = y++;	x gets value of y then y increments by 1	x = y;	y = y + 1;
	increment before	++y	x = ++y;	y increments by 1 and then x gets value of y	y = y + 1;	x = y;
--	decrement after	y--	x = y--;	x gets value of y then y decrements by 1	x = y;	y = y - 1;
	decrement before	--y	x = --y;	y decrements by 1 and then x gets value of y	y = y - 1;	x = y;

LESSON 2 QUESTION 1

Fill in the values for x and y. **Assume all statements continuous**. Each line depends on the previous line result

	x	y
y = 5;	??	5
x = y++;		
x = y--;		
x = ++y;		
x = --y;		

relational operators

Relational operators are used to **compare** two values if they are **greater, greater than** or **equal to, less than** or **less than or equal to** by using a **test condition**. The test condition is enclosed in parenthesis and evaluated to be **true** or **false**:

	(condition)	
(operand	relational operator	operand)
(x	>	5)

Conditions are evaluated as **true** or **false** where:

- true** is a non-zero value
- false** is 0

Evaluate the following for $x = 5$; and $y = 5$;

>	greater than	$(x > 5)$	false
>=	greater than or equal	$(x \geq 5)$	true
<	less than	$(x < y)$	false
<=	less than or equal	$(x \leq y)$	true

You can use **cout** to print out the result of the test. A '1' means true where a '0' means false.

```
cout << (x > 5) << endl;
```

0

Equality operators

Equality operators are used to test if a variable is equal to another variable or constant. Evaluate for $x = 5$:

==	is equal	(x == 5)	true
!=	is not equal	(x != 5)	false
!	not (means opposite)	!(x==5)	false

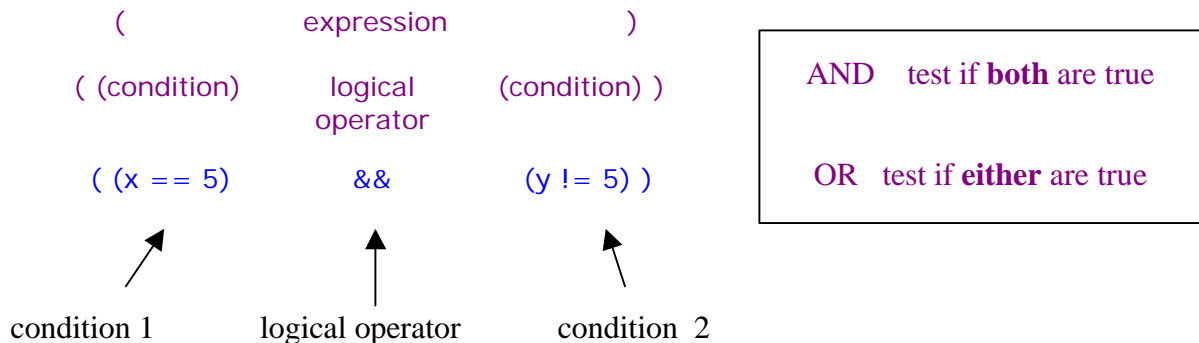
The equality operator is different from the assignment operator "=". Be careful the C++ compiler will accept the assignment operator for a test condition. Instead of testing if a variable is equal to an expression in will assign the evaluated expression to your variable then you will be really in trouble. What is the difference between `x = 5;` and `(x == 5)` ? `x=5` is used to assign 5 to the variable `x` where `(x == 5)` is used to test is the variable `x` is equal the 5.

Logical operators

Logical operators are used to test if two conditions are **both** or **either** true.

AND	(&&)	tests if both conditions are true
OR	()	tests if either condition is true

Parenthesis is used around the test conditions to force the conditions to be evaluated first. The two conditions and logical operator is known as an expression.



The logical **AND &&** operator is used to test if **both** conditions are true and the logical **OR ||** operator is used to test if **either** conditions are true. The logical **&&** and logical **||** are said to be **short circuited** or sometimes called **lazy evaluation**. As soon as it finds one condition not true or false it will not test the other condition. Evaluate the following for `x = 5;` and `y = 5;` Hint: Evaluate each condition as true or false before evaluating the logical operator.

both	&&	logical AND	<code>((x != 5) && (x == y))</code>	test if x is not equal to 5 and x is equal to y	false
either	 	logical OR	<code>((x == 5) (x != y))</code>	test if x is equal to 5 or x is not equal to y	true

LESSON 2 EXERCISE 1

Write a program that asks the user to type in three numbers. Write a logical expression that tests if the **first** number is larger than the **second** number and (&&) the **second** number is smaller than the **third**. Write another logical expression that tests if the **first** number is larger than the **second** number or (||) the **second** number is smaller than the third number. Just use **cout** to print out the results as true (1) or false (0) . No **if** statements are required. Call your program L2Ex1.cpp.

binary numbers and hexadecimal numbers

A **memory cell** in a computer is called a **bit** and can have two possible states **on** or **off** and is represented as a 1 or a 0. Numbers containing 0's and 1's are known as **binary** numbers having a base of 2. Since 0's and 1's are very difficult to work with, so four bits are grouped together to represent the numbers 0 to 15.

4 bit binary number

0	1	0	1
---	---	---	---

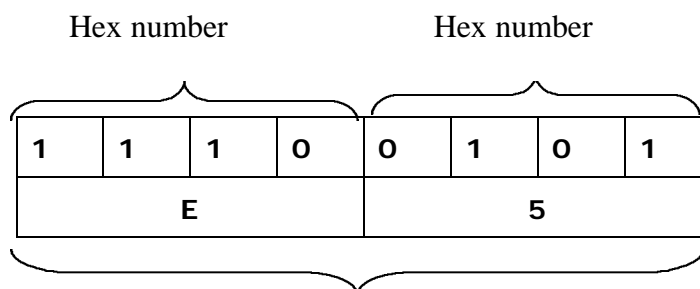
$2^4 = 16$ range is : 0 to 15

The decimal numbers 0 to 15 can be represented in base 16 as 0 to 9, A to F. Base 16 is known as **hexadecimal** numbers. Letters are used because we want one character to represent a number. Base 16 is used because it is more convenient to work with. Hexadecimal numbers are represented by a leading '0x'. 0x05 and 0x0f are examples of hexadecimal numbers. The following chart shows the numbers 0 to 15 in binary, hexadecimal and decimal.

binary	hexadecimal	decimal	binary	hexadecimal	decimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

Hexadecimal numbers use the letters A to F as numbers representing 11 to 15

Computers work with 8 bits rather than with 4 bits. 8 bits are known as a byte of data. 1 byte of data will contain 2 hexadecimal numbers. Hexadecimal numbers in a computer can be 1 to many bytes of data.



1 byte = 8 bits

converting from binary to hexadecimal numbers

It's easy to convert a binary number to a hexadecimal number. All you have to do is group many four binary bits together in groups in multiples of bytes.

1	1	1	0	0	1	0	1	1	0	1	1	1	0	0	1
E				5				B				9			

convert binary: 0111001010110110 to hexadecimal

converting from hexadecimal to binary numbers

If you want to convert hexadecimal numbers to binary just use the chart ! Look up the hexadecimal number and write down the binary equivalent.

E				5				B				9			
1	1	1	0	0	1	0	1	1	0	1	1	1	0	0	1

convert hexadecimal: 72B6 to binary

converting from a binary number to a decimal number

To convert a binary number to decimal you just multiply each bit by its weight and add them up. A weight is the power of 2 of the bit position. Bit positions start from the right most significant bit (RMSB) and has bit position 0. Note: 2 to the power of 0 (2^0) is 1. Example take the binary number 1011

bit position	3	2	1	0	
each bit has a weight:	2^3	2^2	2^1	2^0	
power of two	8	4	2	1	
binary number	1	0	1	1	
multiply bit by power	$1 * 8$	$0 * 4$	$1 * 2$	$1 * 1$	
add up result	8	+ 0	+ 2	+ 1	= 11

Calculate the decimal number for binary number 1101

converting from decimal base 10 to binary base 2

To convert a decimal number to its binary equivalent you continually divide all the quotients by 2 and keep track of the value of each remainder. Each remainder must be multiplied by the base (2). To get the correct binary number you then take the remainders in **reverse**. Example convert decimal 11 to binary 1011. Start at the bottom of the chart. We divide 11 by 2. The quotient is **5** and the remainder is **.5** The remainder is multiplied by 2 Continue with the other quotients. Take the answer from the start of the chart to the bottom: 1011.

	rem	base	ans
0	.5	*	2 = 1
2	.0	*	2 = 0
2	.5	*	2 = 1
5	.5	*	2 = 1

1 0 1 1

$$\begin{array}{r} 2 \overline{) 11} \\ \underline{2} \\ 2 \\ \underline{2} \\ 0 \end{array}$$

Convert 13 to a binary number.

bit wise operators and truth table

A truth table lists all the possible input combinations to calculate the outputs. You know how to add numbers, so we will make a truth table for adding binary numbers.

	0	0	1	1	
+	0	1	0	1	
	0	1	1	1 0	(binary 2)

The bit wise operator truth tables should now be easier to understand. Instead of ADDing you will be ANDing, ORing and XORing. The **bitwise operators** operate on individual bits of variables. You need to understand binary and hexadecimal numbers to use bit wise operators. The following table lists all the bitwise operators and corresponding truth table. 0 means **false** and 1 means **true**.

bit wise operator	purpose	truth table			
& bitwise AND	set bits to zero (mask bits)	0 & 0 = 0	0 & 1 = 0	1 & 0 = 0	1 & 1 = 1
bitwise inclusive OR	set bits to 1's (set bits)	0 0 = 0	0 1 = 1	1 0 = 1	1 1 = 1
^ bitwise exclusive XOR	mask/set bits	0 ^ 0 = 0	0 ^ 1 = 1	1 ^ 0 = 1	1 ^ 1 = 0
~ one's complement	toggle bits	~0 = 1	~1 = 0		

Did you notice everything you AND & a bit with 0 the result is zero. Every time you OR | a bit with 1 the result is 1. When you XOR ^ two bits together that are the same the result is 0. When you XOR ^ two bits that are different the result is 1. ~ is known as tilda

Example using bit wise operators where $x = 5$ (0101) binary.

and	or	xor	complement
<code>x = x & 1;</code>	<code>x = x 1;</code>	<code>x = x ^ 1;</code>	<code>x = ~x;</code>
<div>0101</div> <div>& 0001</div> <hr/> <div>0001</div>	<div>0101</div> <div> 0001</div> <hr/> <div>0101</div>	<div>0101</div> <div>^ 0001</div> <hr/> <div>0100</div>	<div>~ 0101</div> <hr/> <div>1010</div>

LESSON 2 QUESTION 2

1. Fill in the values in the following chart for each bit wise operation. Convert all binary answers to decimal.

	AND	OR	XOR	
	1011	1011	1011	complement
	& 0001	0001	^ 0001	~1011
binary:	<div></div>	<div></div>	<div></div>	<div></div>
decimal:	<div></div>	<div></div>	<div></div>	<div></div>

- How would you use the AND & bitwise operator to test if a number was even or odd ?
Hint: 4 (even) is 0100 and 5 (odd) is 0101 Notice the last bit.
- What do you think the XOR ^ bit wise operator is used for ?
- What would we need the OR | operator for ?
- What other uses would the bitwise AND & operator be used for ?

testing if a number is even or odd

Example using **bit wise** & operator to test if a number is even or odd:

```
int result = (num & 0x01); // x means hexadecimal
```

If the number is odd the result of the test condition is = 0x01 a true condition. Why? If the number is even the result of the test condition is = 0x00 a false condition. Why ?

Another method to test if a number is even or odd is to take the mod 2 of the number.

```
int result = ((num % 2)==0); // x means hexadecimal
```

Try it !

encryption and description using xor

Take any char like 'A' **xor** it with another character like 's' called a **key** and you get a secret character like '%'.
`secret = letter ^ key; // % = 'A' xor 's'`

Take your secret character '%' xor it with your key character 's' and you get your original character back 'A'.
`letter = secret ^ key; //'A' = '%' xor 's'`

Try it !

changing case of letters

The upper case letter 'A' is ASCII code 65 or hex 0x41.

The lower case letter 'a' is ASCII code 97 or hex 0x61.

To change from upper case to lower case or by 0x20.

`lower = upper | 0x20;`

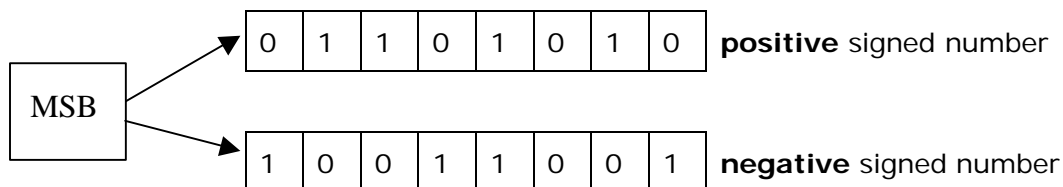
To change from lower case to upper case and by 0xdf.

`upper = lower & 0xdf;`

try it !

representing signed numbers

When the **Most Significant Bit MSB** is 1 then a **signed** number is considered **negative**. When the MSB or most left bit is 0 then a **signed** number is considered **positive**.



The number of bits N in the data type is known as the **resolution**. When N is 4 and we use our signed number **range formula** to determine the smallest (most negative) and largest (most positive) number: (our resolution N is 4 bits)

using range formula: $-2^{(N-1)}$ to $2^{(N-1)}-1$

our range is: -2^{3} to $2^{3}-1 = -8$ to 7

The following chart lists all the numbers for a signed number using 4 bit resolution.

signed representation

-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011	0100	0101	0110	0111
8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7

unsigned representation

How do we calculate binary representation for binary numbers ? To find out what binary number is represented by a negative number we use two's complement on a positive binary number to find its negative binary representation.

converting to/from negative binary numbers

You use 1's complement to complement each bit and then add 1 (2's complement) to convert from a negative binary number to a positive binary number and from a positive number to a negative number.

Convert	to negative		to positive	
number	0101	(5)	1011	(-5)
1's complement	1010	(-6)	0100	(4)
add 1	1		1	
2's complement	1011	(-5)	0101	(5)

Repeat the above for binary number **1101**

SHIFT OPERATORS << >>

Shift operators let you multiply or divide by powers of 2 for char, short, int and long data types. There are two types of shift operators:

(1)	left shift	<<	<code>x = x << N;</code>	<code>// shift x left by N bits multiply by 2ⁿ</code>
(2)	right shift	>>	<code>x = x >> N;</code>	<code>// shift x right by N bits divide by 2ⁿ</code>

To multiply by a power of 2 you shift left N bits. To divide by a power of 2 you shift right N bits.

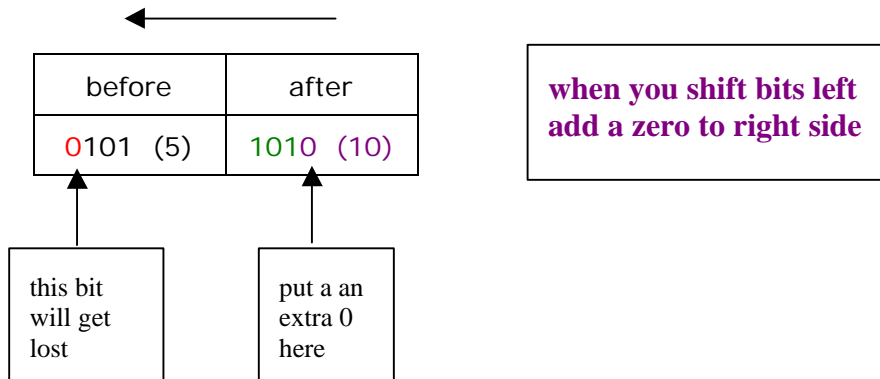
shifting bits left << (multiply by powers of 2) 2^N

variable = variable << N;

x = x << 1; // shift x left by 1 bit multiply x by 2 ($2^1 = 2$)

x = x << 2; // shift x left by 2 bits multiply x by 4 ($2^2 = 4$)

To shift bits **left** you put an extra **0** on the **right hand** side of your binary number. **Red** represents the left most bit that will be lost when we shift left. **Green** represents the shifted bits and **violet** represents the added right bit after we shift left. Example shift 0101 left by 1 bit.



Here is a small program that assigns 5 to x and shifts x left by 1 bit. Don't get the left shift operator << mixed up with the output stream operator <<. They are both use the same symbol!

```
#include <iostream.h>
/* shift number left */
void main()
{
    int x = 5;
    cout << x << endl;
    x = x << 1;
    cout << x << endl;
}
```

program output:

```
5
10
```

Try shifting 0011 left by 1 bit.

signed shift bits right >> (divide by powers of 2) 2^N

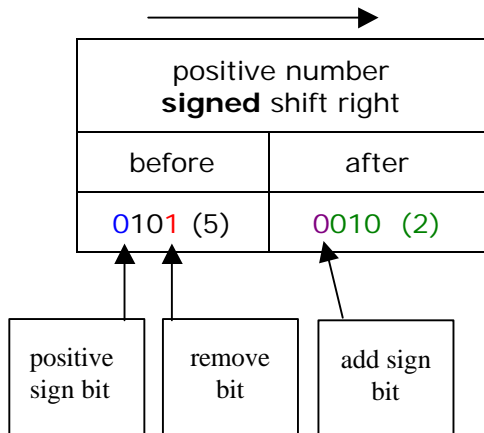
variable = variable >> N;

x = x >> 1; // shift x right by 1 bit

x = x >> 2; // shift x right by 2 bit s

To divide by a power of 2 you shift right N bits. To shift bits right you put an extra bit on the left hand side of your binary number and remove the right most bit. The extra bit on the left is a 0 if the left most bit is 0 and is 1 if the left most bit is 1. This is called signed extension. This means if you divide a negative number by 2 then the answer must still be negative.

shift positive number right



The bit on the left is known as the **sign bit**.

If the number is positive the sign bit is a 0

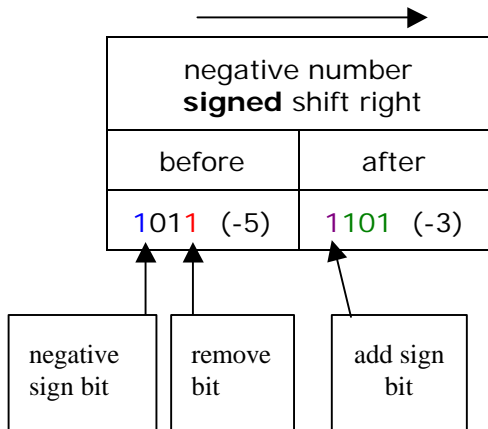
Here is an example program to shift a positive number right.

```
#include <iostream.h>
void main()
{
    int x = 5;
    cout << x << endl;
    x = x >> 1;
    cout << x << endl;
}
```

program output:

5
2

shift negative number right



The bit on the left is known as the **sign bit**.

If the number is negative the sign bit is a 1

Here is an example program to shift a positive number right.

```
#include <iostream.h>
void main()
{
    int x = -5;
    cout << x << endl;
    x = x >> 1;
    cout << x << endl;
}
```

-5
-3

When we shift right the right most bit is lost. We cannot represent fractions with integers data types. When we shift right it appears the values are truncated to a smaller value $5/2 = 2.5 = 2$. (2 is smaller than 3) $-5/2 = -3$. (-3 is smaller than -2).

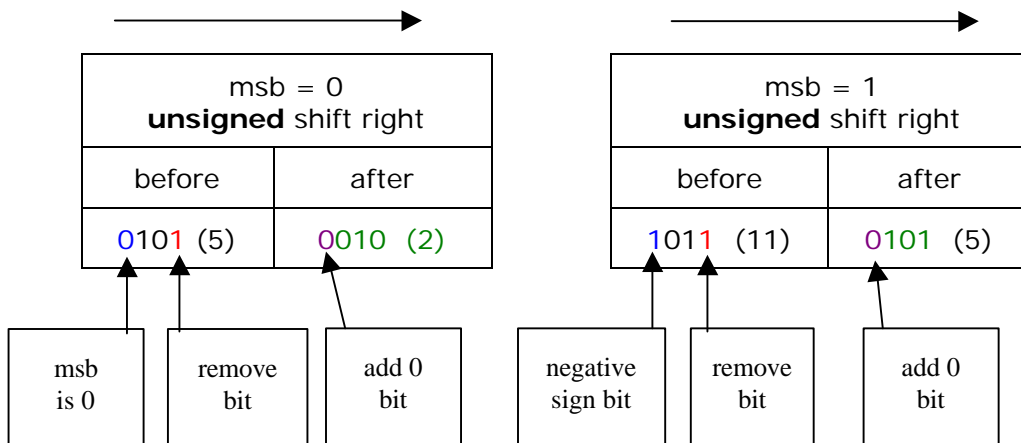
Try shifting 0110 **signed** right by 1 bit

Try shifting 1110 **signed** right by 1 bit

Try shifting 1111 **signed** right by 1 bit

unsigned shift right \gg (divide by powers of 2) 2^N

Unsigned numbers are always positive. The most left bit (MSB) can be 0 or 1. If you shift right an unsigned number then the left most added bit would always be 0. Example shift 0101 left by 1 bit. **Blue** represents the sign bit. **Red** represents the right most bit that will be lost when we shift right. **Green** represents the shifted bits and **violet** represents the signed extended bit after we shift right.



Here's a program that initializes an unsigned int to -5 which is really 65531. When we shift right a 0 is attached to the left most side of x and now the number is 32765.

```
include <iostream.h>

void main()
{
    int x = -5;
    cout << x << endl;
    x = x >> 1;
    cout << x << endl;
}
```

65531 (-5)
32765

Try shifting 1110 unsigned right by 1 bit

LESSON 2 EXERCISE 2

Write a C++ program that just has a main function. In your main function initialize a variable to a positive number. Use all the shift operators to shift by 2 and print out the results. Next initialize the variable to a negative number. Use all the shift operators to shift by 2 and print out the results. Call your C++ file L2ex2.cpp.

LESSON 2 QUESTION 3

1. Fill in the values in the for each shift operation. Convert all answers to decimal.

	signed shift left	unsigned shift left	signed shift right	unsigned shift right
	& 1011 << 1	1011 << 1	1011 >> 1	1011 >> 1
binary:	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
decimal:	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>

2. When we shift right where does the most right binary bit go ?
 3. Why are the left shift and right shift bitwise operators so important ?

shortcut operators

Short cut operators allow you to save typing in adding numbers etc.

+=	x += 5;	x = x + 5;	*=	x *= 5;	x = x * 5;	%=	x %= 5;	x = x % 5;
-=	x -= 5;	x = x - 5;	/=	x /= 5;	x = x / 5;	^=	x ^= 5;	x = x ^ 5;
&=	x &= 5;	x = x & 5;	=	x = 5;	x = x 5;			
<<=	x <<= 2;	x = x << 2;	>>=	x >>= 2	x = x >> 2;			

What is the difference between `x += 5;` and `x = +5;` ?

`x += 5;` means `x = x + 5;`
`x = +5;` means `x = 5;`

Precedence

Precedence tells the compiler what operations are to be performed first for expressions. Round brackets are used to force the compiler to evaluate which operations to be performed first. If round brackets are not used then the compiler must decide for you. The compiler makes it decision by using a **precedence table**. The precedence level tells the compiler which operators are to be performed first. **Associativity** is a term that states in which **order** operations are to be evaluated. from left to right or from right to left. Operators in an expression having the same level precedence are evaluated to the associatively direction.

operator	level	associativity	operator	level	associativity
[] . () (function call)	1	left to right	&	8	left to right
! ~ ++ -- (cast)	2	right to left	^	9	left to right
* / %	3	left to right		10	left to right
+ -	4	left to right	&&	11	left to right
<< >>	5	left to right		12	left to right
< <= > >=	6	left to right	?:	13	left to right
== !=	7	left to right	,	15	left to right
= += -= *= /= %= &= = ^= <<= >>= >>=				14	right to left

Examples forcing and using precedence is as follows.

```
int a=2, b=4, c=6, d=3; // declare and initialize some variables
```

```
x = a + b * c + d; // which operations are done first ? b * c then + a + b
```

```
x = (a + b) * (c + d); // which operations are done first ? a + b then c + d then *
```

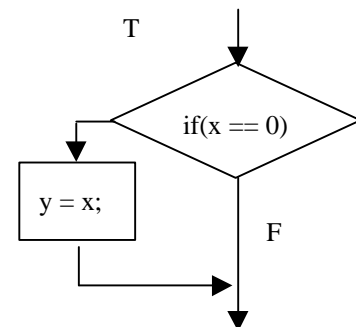
LESSON 2 EXERCISE 3

Design your own mathematical equation using all the arithmetic operators. Assign values to all of your variables, don't use any round brackets. Print out the results. Put round brackets in your mathematical equation and print out the results Call your C++ file L2ex3.cpp.

if statement

The **if statement** is used with conditions to make decisions in a program execution flow. When the **if expression** is **true** then the statements belonging to the **if** statement is executed. When the **if expression** is **false** then program flow is directed to the next program statements. If an **if** statement has more than one statement then the statements must be enclosed by curly brackets. **If** statements uses **comparisons** to direct program execution flow.

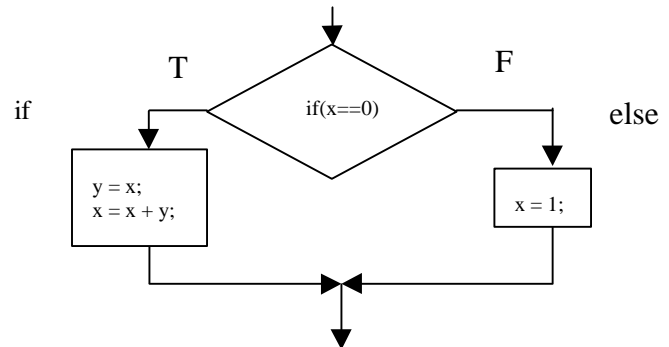
definition	one statement	more than one statements
<i>if(condition)</i> <i>true statement</i>	if (x==0) y = x;	if (x==0) { y = x; x=x+1; }



if - else statement

An **if** statement may have an optional **else** statement. The else statement executes the alternative false condition. Curly brackets are also needed if more than one statement belongs to the **else** statements.

<i>if(condition)</i>	if (x==0)
<i>true statement</i>	{ y = x; x=x+y; }
<i>else</i>	else
<i>false statement</i>	x=1;



Conditional Operator ?:

The conditional operator **?:** evaluates a condition. If the condition is **true** it returns the value of the true expression else returns the value of the false expression. It's like an **if-else** statement but in a condensed form.

(condition) ? true expression : false expression;

`int y = (x == 5) ? x+1 : x-1;`

This means if x is equal to 5 assign x + 1 to y else assign x - 1 to y.

```

if(x == 5)
    y = x + 1;
else
    y = x - 1;
  
```

? if
: else

LESSON 2 EXERCISE 4

Write a small program using the following questions that just includes a main function. Call your program file L2ex2.cpp. All statements are sequential. Declare variables as you use them. You can use **cout** to print out the values of your variables and **cin** to get values from the keyboard. Use **if** and **else** statements to make decisions. Call your C++ file L2ex4.cpp.

1. Make a Color data type having three colors: Red, Green and Blue.
2. Declare a Color data type variable and assign the value Green to it
3. Ask the user to type in a number.
4. Declare an integer variable and use **cin** to get the number from the keyboard.
5. If the number is greater than the color Blue tell them number is too high.
6. If the number is less than the color Red tell them it is too low
7. If the number matched the correct color printout the word "match"
8. Else assign the entered number to your color variable.
9. Print out the color "red" , "green" or "blue" represented by the color variable.

You may have to typecast force an int to be a Color data type or force a Color data type to be an int.

`int x = 1; Color c = (Color)x; x = (int) c;`

C++ PROGRAMMERS GUIDE LESSON 3

File:	CppGuideL3.doc
Date Started:	July 12, 1998
Last Update:	Mar 22, 2002
Version:	4.0

LESSON 3 C++ ARRAYS, POINTERS AND REFERENCES

ARRAYS

An array is like declaring many variables after each other all having the same data type.

```
int a0;
int a1;
int a2;
int a3;
int a4;
```

An array lets you store all variables at once in memory under a common name.

a	int	int	int	int	int
---	-----	-----	-----	-----	-----

Why do we need arrays? We need arrays when we want to access a group of numbers individually that are sequentially stored under a common name. An example is a series of temperatures recorded every hour for a particular day. Each number is the temperature for each hour. We need to access each recorded temperature by a common array name like **temperature**.

temperature	33	37	38	36	35
-------------	----	----	----	----	----

To declare an array we specify the **data type**, the **array name** and **size** of the array in square brackets.

data_type array_name [size_of_array];

int a[5]; // an array of size 5 of integer data type

```

graph LR
    A[array data type] --> B[int]
    C[array name] --> D[a]
    E[array size] --> F[5]
    B --- D
    D --- F
  
```

When you declare an array you get data with sequential memory locations all accessed by a common name. The name represents the starting memory location of the array.

	address	name	array of int data values				
<code>int a[5];</code>	1000	a	int	int	int	int	int

An array containing a single **row** of data is known as a **one-dimensional array**. A row is made up of columns. The above declared array **a** will have 1 row and 5 columns. Each array data value is known as an **element**. Each array column has its own memory location address. The array memory is reserved at compile time when the array is declared. Array elements are uninitialized when they are declared.

1000	1002	1004	1006	1008
??	??	??	??	??

Each element of the array address increments by the data type size

Why do the array element address increment by 2 ?

accessing array elements

Each **element** in an array is identified by an **index**. In C++ arrays indexes start at 0 rather than 1. When you have an array of 5 data types of type integer, the first element has an index of 0 where the last element has an index of 4. Don't forget this, it is easy to get mixed up when accessing array elements. To avoid confusion we will refer to array rows and columns index's starting at zero.

0	1	2	3	4	array index's
??	??	??	??	??	

Each element of an array is accessed by an array index. All indexes start at zero.

Each array element is selected by an index. Index's has a range from 0 to the array size - 1. Why array size - 1 ? An index is enclosed in square brackets that represents an array element. All array indexes start at zero.

<code>a[index]</code>	a[0]	a[1]	a[2]	a[3]	a[4]
<code>a[3]</code>	??	??	??	??	??

writing values to array elements

You use an array index to assign a value to an array element:

`array_name[index] = expression;`

`a[3] = 5; // assign 5 to the array element at index 3`

	0	1	2	3	4
a	??	??	??	5	??

- (1) go to array **a**
- (2) go to index **3**
- (3) assign value **5** to array element at index **3**

You put the values in the array at the specified index.

reading values from array elements

You also use an index to read a value from an array and store in another variable

```
variable_name = array_name[index];
```

```
int x = a[3]; // assign value of the array at index 3 to x;
```

What is the value of x ?

x	5
----------	---

- (1) go to array a
- (2) go to index 3
- (3) get value from array element at index 3
- (4) assign value to variable x

**The array holds the values
the array index tells you
which one you want**

LESSON 3 EXERCISE 1

Write a small program that just includes a main function to do the following called L3ex1.cpp. Do not use loops.

1. Declare a 1 dimensional array of size 5 columns called an of data type int.
2. Assign the value 0 to 4 to each column element.
3. Print out the values of each array element using **cout**.

TWO DIMENSIONAL ARRAYS

Arrays that have more than 1 row are called **two dimensional** arrays and are declared as follows:

```
data_type array_name [number_of_rows] [number_of_columns];
```

```
int b [3][4]; // create a 3 by 4 two dimensional array called b
```

			4 columns			
number of rows	number of columns	3 r o w s	int	int	int	int
			int	int	int	int
			int	int	int	int

accessing array elements

To access an element of a 2 dimensional array you specifying its row and column index. The first dimension is the number of **rows** and the second dimension is the number of **columns**.

array_name[row_index][column_index]

b[1][2]



which row index

which column index

For our 2 dimensional array the row has array indexes 0 to 2 and the each column has array indexes 0 to 3. Each array index allows you to select an array element.

	col 0	col 1	col 2	col 3
row 0	b[0][0]	b[0][1]	b[0][2]	b[0][3]
row 1	b[1][0]	b[1][1]	b[1][2]	b[1][3]
row 2	b[2][0]	b[2][1]	b[2][2]	b[2][3]

assigning values to a 2 dimensional array

To assign a value to a two dimensional array element you must specify the row index and column index.

array_name[row_index][column_index] = expression;

b[1][2] = 5; // assign 5 to element at row index 1 column index 2



row
index

column
index

	col 0	col 1	col 2	col 3
row 0	??	??	??	??
row 1	??	??	5	??
row 2	??	??	??	??

To assign an element of a two dimensional array to a variable again you must specify the row and column indexes.

variable_name = array_name[row_index][column_index];

int x = b[1][2]; // assign to x the element at row index 1 and column index 2



row index

column index

What is the value of x ?

x	5
----------	----------

using variables as array indexes

You may use variables for array index's:

```
int i = 1;
a[i] = 5; // same as a[1] = 5
x = a[i];
int j = 2;
x = b[i][j]; // same as b[1][2]
```

LESSON 3 EXERCISE 2

Write a small program that just includes a main function to do the following called L3ex2.cpp. Do not use loops.

1. Declare a 3 by 3 two dimensional array called **b** of data type int
2. Assign to each array element the value of its row and column.
(example row 1 column 2 would get the value 12)
3. Print out the values the array elements for each row using **cout**.

Initializing arrays

You may set the values of each element of the array only when it is **declared** by using an **initializing list**.

data type array_name[array_size] = { initializing_list };

The Initializing list allows you assign to each array element a value. An initializing list can only contain **constants** each separated by a comma. The initialization list must be enclosed in curly brackets. You must include all the required number of initializers in the initializing list or you will get a compiler error. To initialize a 1 dimensional array of size 5:

```
int a[5] = {10,12,34,21,24};
```

10	12	34	21	24
----	----	----	----	----

To initialize a 2 dimensional array of size 3 by 4, additional curly brackets are needed to group individual rows to avoid compiler warnings

```
int b[3][4] = {{45,46,23,29},{56,23,76,52},{12,45,32,26}};
```

// each row has its own set of brackets.

45	46	23	29
56	23	76	52
12	45	32	26

You can declare and initialize an array elements all to zero:

```
int a[5] = {0}; // initialize all array elements to 0
```

```
int b[2][3] = {0}; // initialize all array elements to 0
```

You need to indicate the array row and column sizes. This may not work with all compilers.

LESSON 3 EXERCISE 3

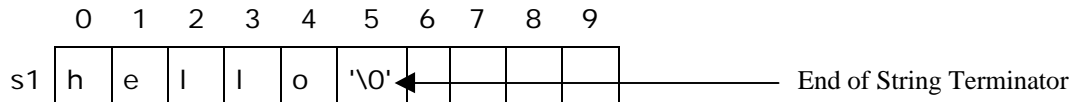
Repeat Exercises 1 and 2 but use an initialization list. Call your program L3ex3.cpp.

character arrays

You may also declare and initialize arrays representing characters. These arrays are known as "character strings".

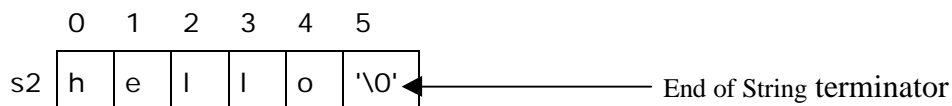
```
char s1[10] = "hello"; // declare and assign message to character string s
```

Character strings are made up of characters terminated by the end of string character '\0'. You need to reserve 1 extra character for the end of string character '\0' when you declare your character string array. If you do not your program will crash. If your message is less than the number of reserved spaces then the extra characters are not used and is wasted memory



If you are not sure of how many characters you need then let the compiler will count for you. You just declare a character string with empty square brackets [] or *. Now there is no wasted memory. The compiler has automatically inserted end of string character '\0'.

```
char s2[ ] = "hello"; // declare and initialize character string s2
```



You can print out the contents of a character string using cout and the character string name.

```
cout << s2 << endl;
```

hello

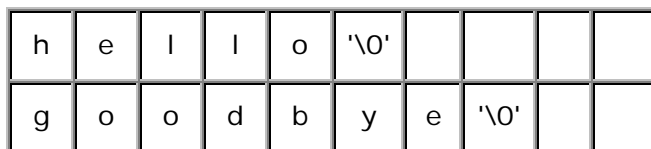
two dimensional character strings

You can also declare and initialized two-dimensional character strings array. Notice we specify the number of rows and columns required. 2 rows and 10 columns.

```
char s3[2][10] = {"hello","goodbye"}; // 2-D character string
```

Only 1 set of curly brackets is needed since each character string is treated as a row. You must always set the number of columns for each row. The row size is optional. For a 2-dimensional character array, the column sizes all must be the same length. You must indicate how many columns you have.

```
char s4[ ][10] = {"hello","goodbye"}; // 2-D character string
```



Why must all columns be the same length ?

To print out the strings of a two dimensional character string you must specify which row you want (which character string you want).

```
cout << s4[1] << endl;
```

goodbye

LESSON 3 EXERCISE 4

Write a program that initializes a two dimensional array of character strings of 5 different words, with each word of 10 characters. Ask the user to type in a number between 1 and 5. Print out the word from the 2 dimensional array corresponding to the word stored in the array. Repeat this 5 time. Use no loops. Call your program L3ex4.cpp.

one dimensional character string array data types

We can make your own user character string data type using **typedef**. You need to set to the maximum number of characters you are going to use for your array data type.

```
typedef char ta[10]; // declare character string data type
```

Now you can declare and initialize a 1 dimensional arrays of character strings with ease.

```
ta a[3] = {"cat","dog","horse"};
```

a	cat	dog	horse
---	-----	-----	-------

This would be the same thing as saying: `char a[3][10] = {"cat","dog","horse"};`

two dimensional character string array data types

We can also make a two dimensional array character string data type.

```
typedef char tb[3][10]; // declare character string data type
```

We can declare and initialize a 2 dimensional array of your user data type of 5 rows of 10 characters quite easily using our string data type.

```
tb b[2] = { {"hello","goodbye","tomorrow"}, {"one","two","three"} };
```

b	hello	goodbye	tomorrow
	one	two	three

This would be the same thing as saying:

```
char b[2][3][10] = { {"hello","goodbye","tomorrow"}, {"one","two","three"} };
```

or using your 1 dimensional array data type:

```
ta b[2][3] = {{"hello","goodbye","tomorrow"}, {"one","two","three"}};
```

Here's a program example using array data types:

```
#include <iostream.h>
typedef char ta[10]; // declare character string data type

void main()
{
    // declare and initialize a 1 dimensional array of character strings
    ta a[3] = {"cat","dog","horse"};

    cout << a[0] << a[1] << a[2] << endl; // print out values

    // declare and initialize a 2 dimensional array of character strings
    ta b[2][3] = {{"hello","goodbye","tomorrow"}, {"one","two","three"}};

    // print out values
    cout << b[0][0] << b[0][1] << b[0][2] << endl;
    cout << b[1][0] << b[1][1] << b[1][2] << endl;
}
```

program output:

```
cat dog horse
hello goodbye tomorrow
one two three
```

LESSON 3 EXERCISE 5

Write a program that initializes a 1 dimensional array of character strings of 5 different words. You need to make a character string array data type. Ask the users to type in a number between 1 and 5. Print out the word from the 1 dimensional array corresponding to the word stored in the array. Repeat this 5 time. Use no loops. Call your program L3ex5.cpp.

Array Index incrementers and decrementers

You can use **incrementing** ++ and **decrementing** -- operators to automatically change your array index when accessing the array elements. **Increment** means add 1 to the array index and **decrement** means subtract 1 from the array index. They work in a **prefix** mode and in a **postfix** mode. **Prefix** means **before** assignment and **postfix** means **after** assignment. It is a two step process. A value is read from the array and the indexed incremented or decremented. or vice versa.

using	description	step 1	step 2
<code>x = a[i++];</code>	increment array index i by 1 after the assignment	<code>x=a[i];</code>	<code>i=i+1;</code>
<code>x = a[++i];</code>	increment array index i before the assignment	<code>i=i+1;</code>	<code>x=a[i];</code>
<code>x = a[i--];</code>	decrement array index i after the assignment	<code>x=a[i];</code>	<code>i=i-1;</code>
<code>x = a[--i];</code>	decrement array index i before the assignment	<code>i=i-1;</code>	<code>x=a[i];</code>

LESSON 3 QUESTION 1

A 1 dimensional array called **a** is pre-initialized with the following values

```
int a[5] = {23,18,35,42,10};
```

and the array index **i** is initialized to 2. `i = 2;`

a	23	18	35	42	10
----------	----	----	----	----	----

Fill in the columns for each value of **x** and **i**, before the statement is executed and after the statement is executed. Consider the statements as part of a program. Each statement depends on the values of the previous statement results and executes continuously. Hint draw the array indexes on top of the array 0 to 4.

statement	i before assignment	x	i after assignment
x = a[i++]	2		
x = a[++i]			
x = a[i--]			
x = a[--i]			

Pointers

A **pointer** is a variable that stores an **address value** rather than a **data value**. You declare a pointer variable by stating the data type and a ***** operator (star) and the pointer variable name. In C++ the star goes with the data type. We call the value of a pointer its **contents**.

```
data_type* variable_name;
```

```
int* p; // declare a pointer variable
```



declare a pointer variable

address	name	contents
2000	p	uninitialized address value

Since the pointer is a variable it occupies memory at an address location and contains an address location as its value (contents). Pointer **p** is located at address 2000. The contents of the pointer is uninitialized meaning a address value has not been assigned to it yet.

Declaring a pointer is no different then declaring a variable. You all know how to declare a variable:

data_type variable_name;*

`int x; // declare a variable`



declare a data variable

address	name	contents
1000	x	?

Declaring a pointer is the same as declaring a variable except the data type gets a * to indicate the variable contains a address value rather than a data value. There is nothing to be afraid of.

`int* p; // declare a pointer variable`



The star means this variable holds an address value rather than a data value

address	name	contents
2000	p	?



store an address to a memory location

The data type indicates what kind of memory the pointer points to. A pointer can represent any data type. When a pointer is declared it is uninitialized its contents does not have an address value. You must assign an address to it or assign the starting address of a memory block to the pointer. If you declare a pointer variable and you do not assign an memory address to it, then you should assign **NULL** to it. NULL is used to represent no known address and has the value of address 0 of no implied data type which is (void*)0. If you do not assign an address to a pointer then the pointer will contain **garbage**. Garbage is what crashes your program and may even lock up (freeze) your computer. Some compilers do not define NULL. If NULL is not defined do not define it use (void*)0 instead.

data_type variable_name = expression;*

`int* p = NULL; //declare pointer with no address assigned`

address	name	contents
2000	p	NULL (0)

pointer analogy

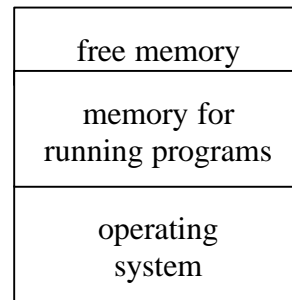
An analogy to a pointer is a mail man who delivers mail. For each letter delivered he knows who to deliver it to, from the address written on the letter. You can consider the letter is a pointer variable with a stated address written on it. The mail man can be considered the executing program, who reads the address on the letter and delivers the letter to the house. The letter is pointing to the house. The house can be considered a memory block containing data contents.

why do we need pointers ?

We need pointers to point to a block of existing memory or point to additional memory allocated in run time. When you allocate memory in run time, the allocated memory does not have a name to represent where it is located in the computer memory. A pointer is used to identify where the allocated memory is by storing an **address** to it. Pointers can be used to point to memory allocated in run time or memory reserved in compile time. Pointers are also be used represent data as a different data types through typecasting.

allocating memory

Memory in a computer is used by the operating system and for running programs. Any memory left over is free memory that your program can use. When you want to use free memory for your computer you have to allocate it. Pointers are used to point to **allocated** memory. Memory used for loaded programs is known as **reserved** memory.



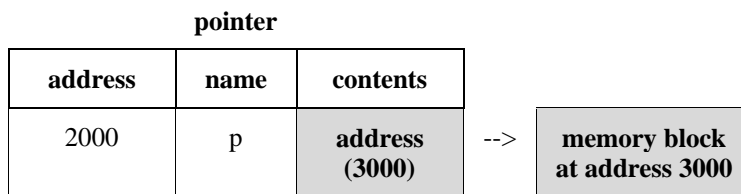
To use a pointer you must assign an address to it. You can assign to the pointer the address of an **reserved** memory represented by a variable or assign the address of **allocated** memory. We will first allocate memory and assign the address of the allocated memory to a pointer. Memory is allocated in C++ by the **new** operator. The **new** operator allocates memory from free memory and returns the starting address of the allocated memory block. We assign the starting address of the allocated memory to a pointer. The **new** operator is used to allocate memory in **run time** when your program is running.

```
data_type* variable_name = new data_type ;
int* p = new int;    // declare pointer, allocate memory
```

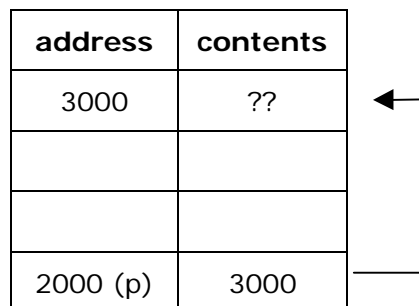


The new operator
allocates memory

In our example a memory block at address 3000 is allocated using the new operator and the starting address 3000 is assigned to the pointer located at address 2000. Although pointers point to memory locations pointers still need an address where they reside.



The starting address of the allocated memory at address 3000 is assigned to p located at address 2000.



assigning memory values pointed to by a pointer

To assign or retrieve a value of a memory pointed to by a pointer we use the * operation on a pointer *p. The * operator means get the value of the memory location pointed to by a pointer. Do not confuse declaring a pointer `int *p` with accessing memory `*p`. The * operator performs many different types of operations. Once you can access memory pointed to by a pointer you can read or write data values to memory locations pointed to by a pointer.

writing to a memory location using a pointer

You use the * star operator to write to the memory location pointed to by p. In this case the * (star) means put the value at the **location** pointed to by the pointer.

```
*p = 5;           // assign the value 5 to the memory location pointed to by p
```

address	name	contents		location 3000
2000	p	3000	--->	5

The value 5 is assigned to the memory location pointed to by p. p is at address 2000 and points to a integer memory at address 3000. The address 3000 gets the value of 5.

p means access the **value of the memory
location pointed to by p*

How do we write to the memory locations pointed to by a pointer ?

step	writing * p = 5;	address	contents
(1)	go to pointer p	3000	5
(2)	read contents of pointer (3000)		
(3)	go to address 3000		
(4)	assign value 5 to that address	2000 (p)	3000

reading from a memory location using a pointer

You can also use the * (star) operator to retrieve a value of a memory location pointed to by a pointer. The * (star) in this case means get the **value of** what the pointer points to.

```
int x = *p;    // assign the value 5 to the memory location pointed to by p
```

address	name	contents		location 3000
2000	p	3000	---->	5

Get the value of the memory location pointed to by p.

What is the value of x ?

address	name	value
1000	x	5

What is p , *p ?

How does the program read from the memory locations pointed to by a pointer ?

step	reading x = * p;
(1)	go to pointer p
(2)	read contents of pointer (3000)
(3)	go to address 3000
(4)	get value at that address (5)
(5)	assign value to the variable x

address	contents
3000	5
2000 (p)	3000
1000 (x)	5



LESSON3 EXERCISE 6

Write a program that declares a pointer called **p** and allocates an int data block. Assign to the data block the value 5. Print out the value of the data block using the pointer. Call your program L3Ex6.cpp.

assigning the address of existing variable to a pointer

You can assign an address of an existing variable to a pointer. When a variable is declared the compiler automatically assigns an address to it at compile time. To find out the **address** of the variable the **&** (ampersand) operator is used.

```
int x;
```

```
p = &x; // assign to p the address of x
```

address	name	contents
2000	p	&x (1000)

--->

address	name	value
1000	x	5

What is p ? What is *p ? What is x ?

p points to x , *p is the value of x which is 5 and x is the variable pointed to by p.

If you change x to 3

```
x = 3; // assign to x the value of 3
```

address	name	contents
2000	p	&x (1000)

--->

address	name	value
1000	x	3

What is p ? What is *p ? p points to x and *p is the value of x which is 3.

*** value of**
& address of

Assign to y the value pointed to by p. Since p points to x then get value from x give to y.

```
int y = *p; // assign to y the value pointed to by p
```

address	name	value
1002	y	3

step	reading x = * p;
(1)	go to pointer p
(2)	read contents of pointer (1000)
(3)	go to address 1000
(4)	get value at that address (3)
(5)	assign value to the variable y

address	contents
2000 (p)	1000
1000 (x)	3
1002 (y)	3



What is the value of y , &p, p , *p, x, &x and &y ?

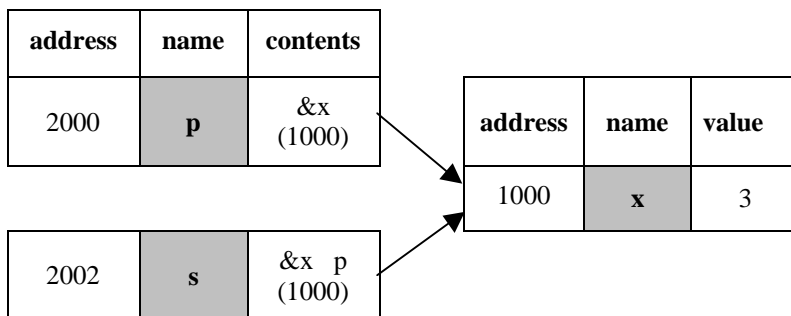
LESSON3 EXERCISE 7

Write a program that declares a pointer called **p** and a int data variable called **x**. Assign to the data block the value 5 using the pointer p. Print out the value of the data variable x using the pointer p. Call your program L3Ex7.cpp.

assigning a pointer to another pointer

You can also assign a pointer to another pointer. In this situation both pointers will point to the same address. Get contents of `p` (1000) and assign to `s`. Now the contents of `s` is `&x`.

```
int* s = p;    // both p and s point to x
```



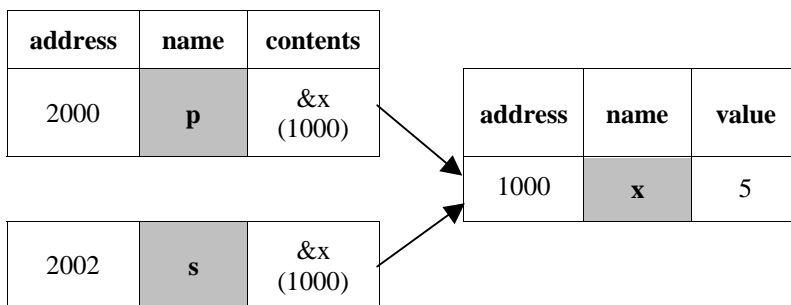
The content of `p` (1000) is assigned to `s`, now `s` contains 1000. Both `p` and `s` point to `x`.

What is `&p`, `p`, `*p`, `&s`, `s`, `*s`, `x`, `y` ?

The following two statements will change the value of `x` to 5. Why ?

```
*s = 5;    // change the value of x to 5
```

```
*p = 5;    // change the value of x to 5
```



What is `&p`, `p`, `*p`, `&s`, `s`, `*s`, `x` and `y` ?

LESSON3 EXERCISE 8

Write a program that declares a pointer called **p** and a int data variable called **x**. Assign to the data block the value 5 using the pointer `p`. Declare another pointer called **s** and assign **p** to it. Print out the value of the data variable using the pointer `s` and pointer `p`. Call your program `L3Ex8.cpp`.

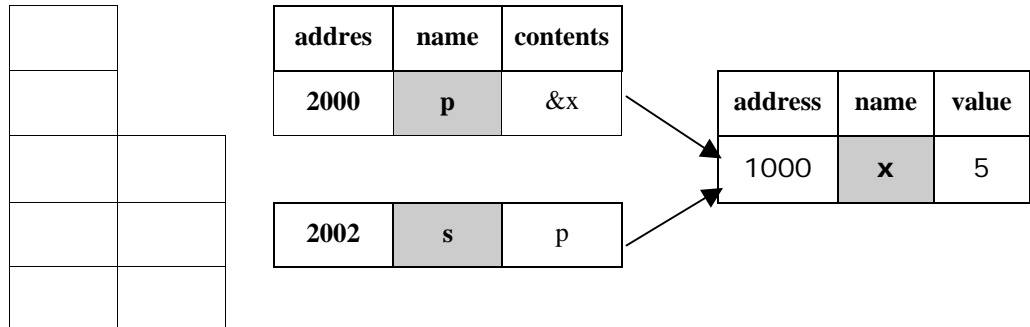
LESSON 3 QUESTION 2

The variable `x` is assigned the value 5. The pointer `p` is assigned the address of `x`.

```
int x = 5;           // declare a variable x and assign the value 5 to it
int* p = &x;         // declare a pointer p and assign the address of x to it
int* s = p;          // declare a pointer s and assign the pointer p to it
```

Using the above answer the following questions.

1. What is `x` ?
2. What is `p` ?
3. What is `*p` ?
4. What is `s` ?
5. What is `*s` ?



writing messages and values of variables and pointers to the screen

You can also print out the contents of pointers using `cout`. A hexadecimal address value is printed to the screen.

```
cout << "the contents of p is: " << p << endl;
```

Where `0x34512344` is an address and `0x` means a **hexadecimal** number

Still having difficulty understanding pointers ?

If you are still having difficulty understanding pointer's than think this. You need a block of memory to hold some data when your program is running. You need some variable to hold the location the starting address of the memory block so your program knows where it is. A pointer variable will store the starting address of the memory block for you. In your program you will have many variables. You need to keep track of which variables store data values and which variables store address values. When you declare your pointer variables you put stars on them to differentiate which variables will hold address values and which variables will hold data values.

```
int x; // declare a variable to hold a data value
```

```
int *p; // declare a variable to hold an address value
```

Now the compiler and you know which variable holds data values and which variables hold address values. When you access the memory block using your pointer you want to read and write data values, you do not want the address values. So you need to devise some mechanism to inform the compiler you want the data at the address stored in the pointer. You use the star symbol again.

```
*p = 5; // assign the value 5 to the memory space pointed to by p
```

```
int y = *p; // assign to y value of the memory space pointed to by p
```

Now everything is working okay. The final thing you want to do is to point to existing memory with a pointer. You need a symbol to inform the compiler you want the address where a variable is stored not the data value it represents. So you use the & symbol to get the address of the variable.

```
int *s = &x; // assign to pointer p the address of the memory represented by x
```

Thats it ! We use the symbols to tell the compiler what we want to do, not to confuse you. Since we do not have a lot of symbols, each symbol serves many purposes.

LESSON 3 EXERCISE 9

Write a small program using the following questions that just includes a main function. Name your file L3ex9.cpp. All statements are sequential. Declare variables as you use them. You can use **cout** to print out the values of your variables. Draw diagrams as you answer each question.

1. Declare an integer variable x and assign the value 5 to it
2. Declare a integer pointer variable called p and assign the address of x to it
3. What is the value of the memory location pointed to by p ?
4. Assign 3 to the memory location pointed to by p .
5. What is the value of the memory location pointed to by p ?
6. What is the value of x ?
7. Declare an integer pointer variable called s and assign p to it.
8. Declare an integer variable y and assign to it the value pointed to by p
9. What is the value of y ?
10. Assign 7 to the memory location pointed to by s
11. What is the value of x ?
12. Assign to the pointer p the address of the variable y
13. What is the value s, *s, p, *p, x, &x , y and &y ?

POINTERS TO POINTERS

A Pointer may also point to another pointer. These pointers are called pointers to pointers. Pointers to pointers are declared with two stars **. A pointer pointing to another pointer. A * for each pointer.

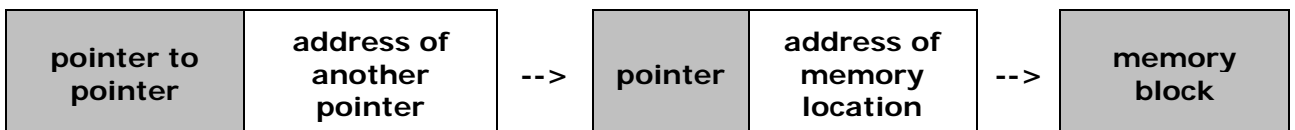
```
data_type ** variable_name;
```

```
int** q; // declare a pointer that points to another pointer
```

q	address of another pointer
---	----------------------------

Pointer to pointer analogy

An analogy to a pointer to a pointer is the mail man reads the address of a letter goes to that address, picks up another letter reads that address and then delivers the letter to that location. What is the difference between a pointer and a **pointer to pointer**? A pointer to pointer points to another pointer. A pointer points to a memory block.

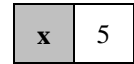


setting up pointer to pointers:

A pointer to pointer can only point to another pointer

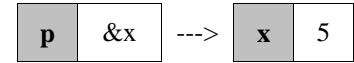
We start with the variable `x` that is assigned the value of 5

```
int x = 5; // declare variable x assigned to 5.
```



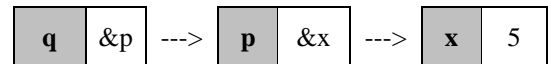
Next we assign the address of `x` to a pointer variable `p`.

```
int* p = &x; // declare pointer p assigned to variable x
```



For the last step you declared a pointer to a pointer `q` and assign the address of `p` to it.

```
int** q = &p; // assign the address of a pointer to q
```



Now `q` points to `p` and `p` points to `x`. What is `q`, `p`, `*p` and `x`?

accessing values from pointer to pointers

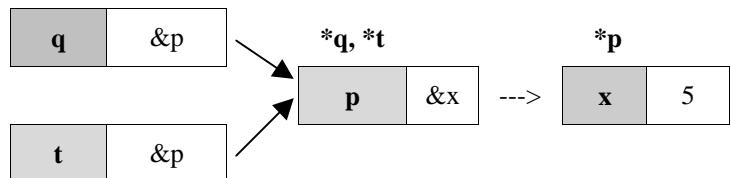
When you access the value of a pointer to pointer, you are getting an address value rather than a data value. You are getting the address of another pointer.

```
int** t = q; // assign to t the contents of q
```

`t` contains the contents of `q` which is the address of `p`. `q` contains `&p`.

`t` must be a pointer to pointer because `p` is a pointer and the contents of `q` is a pointer.

`t` points to `p`



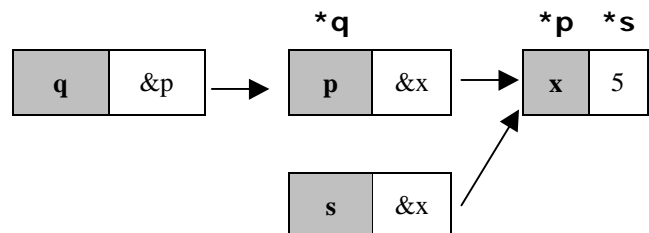
What is the value of `q`, `*q`, `t`, `*t`, `p`, `*p`, `x`?

Next we want to get the contents of what the pointer to pointer is pointing to.

```
int* s = *q; // assign to s the contents of what the pointer to pointer is pointing to
```

Now `s` contains the address of `x` because `*q` is the value of what `q` points to. `q` points to `p` and `p` points to `x`. `*q` is really the contents of `p`. The contents of `p` is `&x`.

`s` points to what `p` points to

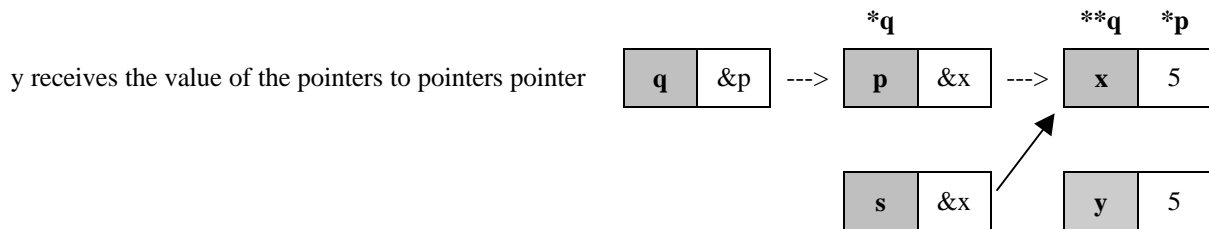


What is the value of `q`, `*q`, `s`, `*s`, `p`, `*p`, `x`?

Finally we want to get the value of what the pointer to pointer's pointer is pointing to.

```
int y = **q; // assign to y the value of what the pointer to pointer's pointer is pointing to.
```

The inner star get you the contents of **p** which is **&x** and the outer star gets you the contents of **x** which is 3. Therefore **y** has the value of 5 also.



What is the value of **q**, ***q**, ****q**, **p**, ***p**, **x**, **y** ?

assigning values to pointers to pointers

We can change the value of what the pointer to pointer 's pointer is pointing to.

```
**q = 3; // assign to y the value of what the pointer to pointer's pointer is pointing to.
```

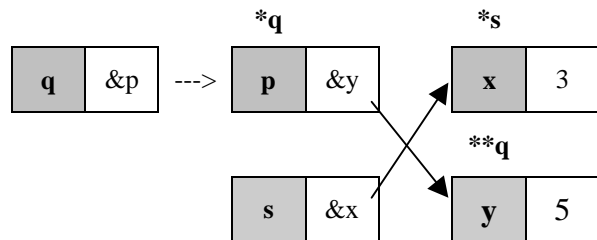
The inner star get you the contents of **p** which is **&x** and the outer star lets you assign a new value to the pointer to pointer's pointer.



You can also change what the pointer to pointer is pointing to.

```
*q = &y; // assign the address of y to the pointer to pointers pointer
```

Now p points to y.

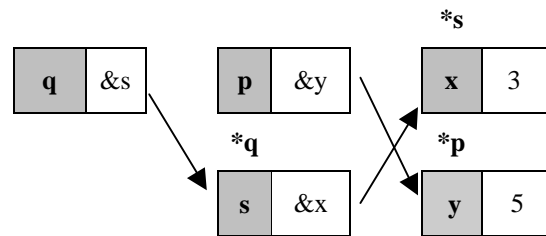


What is **q**, ***q**, **s**, ***s**, **p**, ***p**, **x** and **y** ?

You can also change what the pointer to pointer points to

```
q = &s; // assign the address of s to q
```

q now points to s

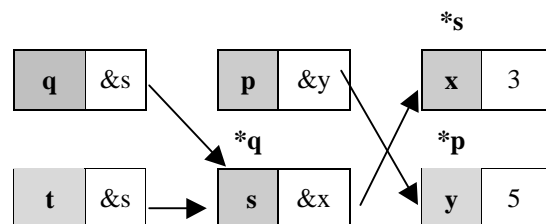


What is **q**, ***q**, **s**, ***s**, **p**, ***p**, **x** and **y** ?

assigning pointer to pointers to other pointers to pointers

You can also assign pointers to pointers to other pointers to pointers. In this case both pointers to pointers will point to the same pointer **s** as continued from above.

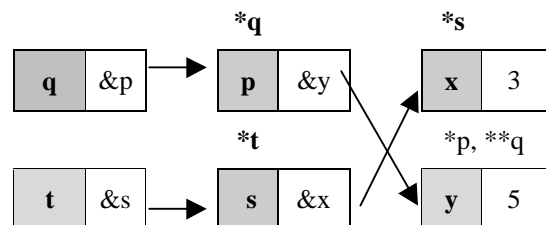
```
int **t = q; // assign the contents of q to t
```



What is the value of **t**, ***t**, ****t**, **q**, ***q**, ****q**, **p**, ***p**, **x** and **y** ?

We change q back to point to p

```
q = &p; // assign the contents of q to t
```

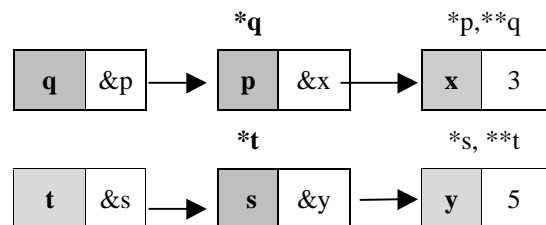


What is the value of **t**, ***t**, ****t**, **q**, ***q**, ****q**, **p**, ***p**, **x** and **y** ?

Lastly we use **q** to have **p** point to **x** and use **t** to have **s** point to **y**

```
*q = &x; // p now points to x
```

```
*t = &y; // s now points to y
```



What is the value of **t**, ***t**, ****t**, **q**, ***q**, ****q**, **p**, ***p**, **x** and **y** ?

LESSON 3 QUESTION 3

The variable **x** is assigned the value 5. The pointer **p** is assigned the address of **x**. The pointer to pointer **q** is assigned the address of the pointer **p**.

```
int x = 5;           // declare a variable x and assign the value 5 to it
int* p = &x;         // declare a pointer p and assign the address of x to it
int** q = &p;        // declare a pointer to pointer, assign the address of p to it
```



Why does q contain &p and not p ?

We want the address of the pointer variable p itself not what p point to .

What is the difference between p and *p; ?

p contains an address of a memory location where *p is the value of the memory location that p points to.

Using the above assignments answer the following questions:

1. What is the value of x ?
2. What is the value of p?
3. What is the value of *p ?
4. What is the value of q ?
5. What is the value of *q ?
6. What is the value of **q ?

writing pointers to pointers to the screen

You can also print out the contents of a pointers to pointer

```
cout << "the contents of q is: " << q << endl;
```

```
the value of q is: 0x34565444
```

Where 0x34512344 is an address and 0x means a hexadecimal number

To print out the address value a pointer to pointer points to

```
cout << "the value of *q is: " << *q << endl;
```

```
the value of *q is 0x3456548
```


LESSON 3 EXERCISE 10

Write a small program using the following questions that just includes a main function that covers the material on pointers. Name your file L3ex6.cpp. All statements are sequential. Declare variables as you use them. You can use **cout** to print out the values of your variables. Draw diagrams as you answer each question. Name your file L3ex10.cpp

1. Declare an variable **x** and assign the value **5** to it
2. Declare a pointer variable called **p** and assign the address of **x** to it
3. Declare a pointer to pointer variable called **q** and assign the address of **p** to it.
4. Declare a variable **y** and assign the value pointed to by **p** to it using **q**
5. What is the value of **y** ?
6. Assign the value **7** to the memory location pointed to by **p** using **q**.
7. Declare a pointer variable called **s** and assign **p** to it using **q**
8. What is the value **q**, ***q**, ****q**, **p**, ***p**, **x**, **y** and **s**?

REFERENCE VARIABLES

Many people find pointers a pain and a nuisance to use. Somebody came up with the great idea to use reference variables instead of those pesky pointers. Reference variables almost do the same job as pointers but you get to use the variable name to do all the work. Reference variables are declared with the "&" operator. The only problem with references is that the memory has to be allocated before hand. You **cannot allocate memory** to a reference variable by using the **new** operator. Reference variables must refer to an existing memory location address. References let you refer by address or by value depending on the context in which it is used. References are different from pointers is that their is no additional memory set aside for the reference. A reference is also referred to as an **alias** to a variable name. Two names for the same memory location. You can only declare a reference to an existing variable. Once you declare a reference the variable can be accessed by two names. A reference is different from a #define label because it refers to a particular variable not all variables. To declare a reference variable:

data_type & reference_name = existing_variable_name;

x	5	<i>//declare x and assign the value 5 to it</i>
r		

// declare reference variable r referenced to x

r refers to x., use r in place of x

just another name for a particular variable

You just use reference variables like ordinary variables.

```

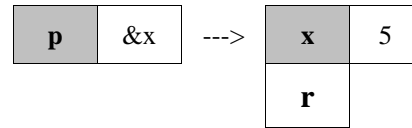
r = 3;           // Assign a value to a variable referred by reference r:

y = r;          // get a value referred to by a reference r

int *p = &r;     // assign to pointer p the address of the variable referred to by the reference
  
```

Why do we use &r rather than r in the above example ? References are referring to actual variables, therefore to get the address of a reference variable you still need to use the & operator. r and x

have the same memory location. What is **r**, **x**, **y**, **p** and ***p** ?



You can also get a reference to a memory area pointed to by a pointer.

```
int& r = *p;           //get a reference to a memory area pointed to by a pointer.
```

In this situation r refers to the memory location pointed to by p.

using reference variables and pointers:

Can you assign a reference variable to a pointer ? ~~`p = r;`~~

**r is a reference
not a pointer**

No a pointer needs an address. r refers to a variable not an address. You still need to use the &.operator to get the **address of** the reference `p = &r;`

**A reference is another name
for a variable**

Can you assign a pointer to a reference variable ? ~~`r = p;`~~

No a reference must refer to a existing variable not a pointer. You must assign the **value of** what the pointer is pointing to. This is called dereferencing. `r = *p;`

**A reference is different from a substitution because
the reference refers top a particular variable only not
all variables of the same name.**

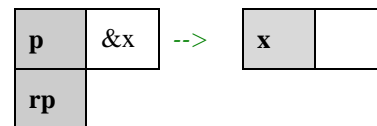
references to pointer variables

That's right we can even have references to pointers. What will they think of next? To declare a reference to a pointer all you need is to declare a reference to an existing pointer.

data_type_pointer & reference_name = existing_variable_pointer;*

```
int *p = &x;           // declare pointer p and assign address of x to it
```

```
int*& rp = p;          // declare a reference to pointer p
```



reference pointer **rp** refers to pointer **p**, you just use reference pointer **rp** just like using **p**

You can use a reference to the pointer just as you would a pointer:

```
rp = &x; *rp = 7; y = *rp;
```

**A reference pointer
refers to a pointer**

When you use reference pointer rp you are actually referring to pointer p

reference to pointer to pointer variable

You've guessed it we can even have references to **pointers to pointers**! To declare a reference to a **pointer to pointer** all you need is to declare a reference to an existing **pointer to pointer**.

*data_type_pointer** & reference_name = existing_variable_name;*

<code>int **q = &p;</code>	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>q</td><td>&p</td></tr></table>	q	&p	<i>// declare pointer to pointer q and assign the address of p to it</i>
q	&p			
<code>int**& rq = q;</code>	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>rq</td><td></td></tr></table>	rq		<i>//declare reference to pointer to pointer q called rq</i>
rq				

You can use the **reference pointer to pointer** just as you would a **pointer to pointer**

```
rq = &p; *rq = &x; y = **rq; p = *rq;
```

reference pointer to pointer **rq** refers to pointer to pointer **q**, you just use **rq** like you use **q**. The power of references is most apparent when using functions.

LESSON 3 QUESTION 4

Answer the following questions it's a good idea to draw a picture for every operation.

```
int x = 3; // declare an integer initialized with 3
int &r = x; // declare an reference i assigned to the address of x
r = 5; // assign 5 to the variable referenced to by r
```

A reference pointer to pointer refers to a pointer to pointer

1. What is the value of x ?

```
x = r; // x gets the value of the memory referenced to by r
int* p = new int; // allocate a data type int and assign the starting address to p
*p = 7; // initialize the first column of p to 7
r = *p; // assign the memory referenced to by r the value of the first column of p
```

2. What is the value of r ?

3. What is the value of x ?

```
p = &r; // assign to p the address of the memory referenced to by r
```

4. What is the value of the memory location pointed to by p ?

```
int y = *p; // assign to y the value pointed to by p
```

5. What is the value of y ?

```
int*& rp = p; // Declare a reference to pointer p called rp
*rp = 9; // assign the value 9 to the memory location refereed by the reference pointer rp
```

6. What is the value of x ?

```
int **q = &p; // Declare a r pointer to pointer q
int**& rq = q; // Declare a reference to pointer to pointer q called rq
**rq = 3; // assign 3 to the memory location pointer referenced to by the pointer to pointers pointer
```

7. What is the value of x ?

LESSON 3 EXERCISE 11

Write a program by answering the following questions that just includes a main function that covers the material on pointers and references. Name your file L3ex11.cpp. All statements are sequential. Declare variables as you use them. You can use **cout** to print out the values of your variables.

1. Declare a variable **x** and assign the value 5 to it
2. Declare a pointer variable called **p**
3. Assign to **p** the address of **x**
4. What is the value pointed to by **p** ?
5. Assign the value 3 to the memory location pointed to by **p**
6. What is the value of the memory location pointed to by **p** ?
7. What is the value of **x** ?
8. Declare a reference variable **r** referenced to **x**
9. Assign the value of 7 to the variable referenced to by **r**
10. What is the value of the memory location pointed to by **p** ?
11. What is the value of **x** ?
12. Assign the value of the memory location pointed to by **p** to the variable referenced by **r**
13. Change the value of the memory location pointed to by **p** to 9
14. What is the value of **x** ? What is the value of the variable referenced by **r** ?
15. Declare a pointer **s** and allocate to an integer memory block.
16. Assign 2 to the memory location pointed to by **s**
17. Assign **p** to **s**
18. What is the value of **x** ?
19. What is the value pointed to by **p** ?
20. Change the value of the variable referred to by **r** to 4
21. Assign to **y** the contents of the memory location pointed to by **s**
22. What is the value of **y** ?
23. What is the value of **x** ?
24. What is the value of the variable referenced to by **r** ?
25. Declare a pointer reference called **rp** reference to the pointer **p**
26. Assign to the pointer reference by **rp** the address of the variable **y**
27. Assign to the memory location pointed to by the pointer referenced by **rp** the value of the memory location pointed to by **s**.
28. Declare a pointer to pointer **q** and assign the pointer referenced to by **rp**, to **q**
29. Declare a pointer to pointer reference called **rq** referred to **q**
30. What is **rq**, ***rq**, ****rq**, **rp**, ***rp** and **r** ?

C++ PROGRAMMERS GUIDE LESSON 4

File:	CppGuideL4.doc
Date Started:	July 12, 1998
Last Update:	Mar 23, 2002
Version:	4.0

LESSON 4 C++ USING POINTERS AND STRUCTURES
const means you cannot change
CONST POINTERS

The **const** keyword means you cannot **modify** a variable once it has been initialized. Pointers may also be declared as constant. What you can modify depends where you put the const keyword. It can mean the pointer cannot be changed once it is initialized:

```
int* const p;
```

cannot modify pointer (const is on p)

or it can mean the value that p points to cannot be changed.

```
const int* p;
```

cannot change what pointer is pointing to (const is on data type)

The following example demonstrates the two different situations:

```
// L4p1.cpp
void main()
{
    int x = 5;

    int* const p1 = &x; // declare p1 as const pointer variable
    *p1 = 10; // change the memory location pointed to by p
    //p1=p1+1; // cannot modify const pointer variable

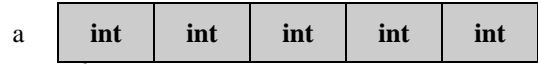
    const int* p2 = &x; // declare what p2 points to as const
    //*p2 = 10; // cannot modify const memory location
    p2=p2+1; // change the contents of p2
}
```

The trick to remember is where the const keyword is on the variable means the variable cannot be changed. What does: `const int * const p = &x;` mean ? In the above program take away the comments and see what happens. Try to fix the program errors without commenting the lines out.

POINTERS TO ARRAYS

You can point to an existing array by using a pointer. The memory is **reserved** for the 1 dimensional array in **compile time**, when the program is being compiled. Do you still remember how to declare an array ? To declare a 1 dimensional array of 5 integer data types:

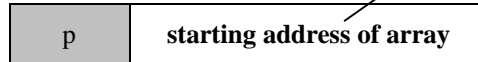
```
int a[5]; // declare a 1 dimensional array of 5 elements
```



You can use a pointer to point to an existing array:

```
pointer _variable = array_name;
```

```
int * p = a;
```



Why do we use **a** and not **&a** ? When would you use an **&** on **a** ? **a** and **&a[0]** mean the same thing. We want the address of the first element in the array. You need to specify the index of the element you want the pointer to point to.

allocating memory for 1 dimensional arrays

We use the **new** operator to allocate memory for arrays in **run time**. Run time is when your program is running and allocates memory from the computer. The starting address of the array is assigned to a pointer variable. Memory for the one dimensional array is allocated in run time, when the program is running.

```
data_type variable_name = new data_type [number_of_items];
```

```
int* p = new int[5]; // memory allocated and starting address assigned to p
```



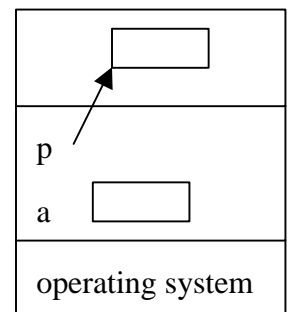
p points to an memory block represent a one dimensional array of 1 row by 5 columns. p must be a pointer in order to assign and allocate memory to it.

difference between compile and run time.

When you declare an array in compile time `int a[5];` the memory is **reserved** in your program and placed in program memory when you run your program. When you declare an array in run time `int* p = new int[5];` the memory is allocated in run time when your program is running and allocated from free memory. The pointer located in program memory points to the allocated memory in free memory.

free memory

program memory



accessing array elements using a pointer

Now that you have allocated memory and assigned the starting address to a pointer:

```
int* p = new int[5];
```

To access individual array elements with a pointer you use an offset. The offset represents which element you want. The compiler calculates the location of the array element from the pointer contents and the offset. The compiler automatically takes account for the data type size when using offsets. A star is used to access the value at that memory location.

```
*(pointer_variable + offset)
```

addresses increase by
the data type size

You can also use array indexes instead of using offsets

The calculated address is $\text{pointer_variable} + \text{offset} * \text{data type size}$.

writing values to an array using a pointer

To assign a value to an array element at a particular element you use an offset from p

```
*(pointer_variable + offset) = expression;
```

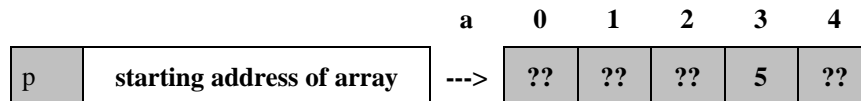
```
*(p+3) = 5; // assign 5 to the address plus offset pointed to by p
```

or use the array index enclosed by square brackets.

```
pointer_variable [index] = expression;
```

```
p[3] = 5; // assign 5 to array index 5
```

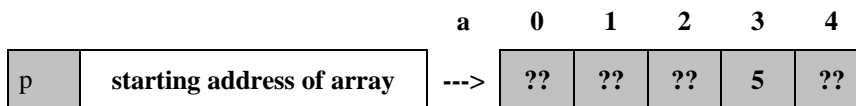
p[3] = *(p+3)



How does the program write to a memory location pointed to by a pointer ?

step	writing * (p + offset) = 5;
(1)	read contents of pointer
(2)	calculate address from pointer contents and offset
(3)	go to calculated address
(4)	assign value to that address

reading values from an array using a pointer



To read a value from an array element you use a pointer offset

```
variable = *(pointer_variable + offset);
```

```
x = *(p+3); // assign to x the value at the offset pointed to by p
```

or an array index

```
variable = pointer_variable [index];
```

```
x = p[3]; // assign to x the value at index 3 of array
```

What is the value of x? How does the program read from a memory location pointed to by a pointer?

step	reading $x = *(p + \text{offset});$
(1)	read contents of pointer
(2)	calculate address from pointer contents and offset
(3)	go to calculated address
(4)	get value from that address
(5)	assign value to a variable

Notice that the brackets are around $*(p+3)$, if it were not for the brackets then

```
x = *p + 3; // get value pointed to by p and add 3
```

Which means take the value at p and add 3. Which is quite different from take the third index of p and give me the value. $p[\text{index}]$ is sort of equivalent to $*(p)$. "sort of" means not all compilers would recognize that these operations are the same. Do you know what $*p$ stands for? $*p = *(p+0)$ or $p[0]$. It should be obvious that $*p$ and $*(p+0)$ is equivalent. The brackets force **precedence**, which states which operation gets done first.

Deallocating memory

Once you are finished with the allocated memory you have to return the allocated memory back to the operating system for other people to use it. Individual memory blocks are deallocated with the delete operator.

```
delete pointer_variable_name;
```

```
delete p; // delete memory location only pointed to by p
```


Memory for arrays are de-allocated using the delete operator and brackets [] to indicate to delete array memory. If you just say `delete p` it will only delete the first memory cell pointed to by p not the whole array.

**delete [] array variable name;*

`delete [] p; // delete whole array memory pointed to by p`

You cannot access memory that has not been allocated with new. You can not deallocate reserved memory.

LESSON 4 EXERCISE 1

Write a small program by answering the following questions that just includes a main function that covers the material on allocating memory for arrays. Call your file L4ex1.cpp. All statements are sequential. Declare variables as you use them. You can use **cout** to print out the values of your variables.

1. Declare and allocate memory for a one dimensional array called **a** of 5 integers.
2. Assign the value 0 to 4 to each column index.
3. Print out the values of your allocated array using **cout** do not use loops.
4. Deallocate memory for the one dimensional array.

CREATING 2 DIMENSIONAL ARRAY'S USING POINTERS TO POINTERS

Pointers to pointers can be used to create 2 dimensional arrays. In this case the first pointer, points to an array of pointers. The second pointer points to a row of columns. There are 2 steps to create a 2 dimensional array using pointers.

step 1

We first allocate an array of integer pointers (`int *`). Each element array of integer pointers will represent a row. Each row will point to another array representing a row of columns.

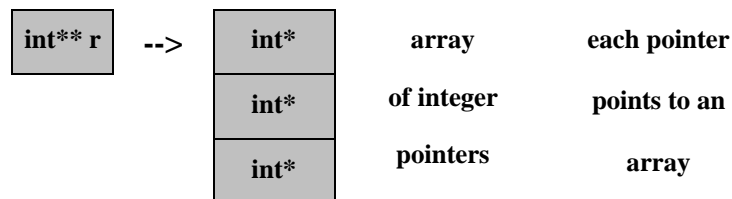
*data_type** variable_name = new data_type * [number of rows] ;*

`int** r = new int*[3] ; // declare an array of row pointers`

└─┘

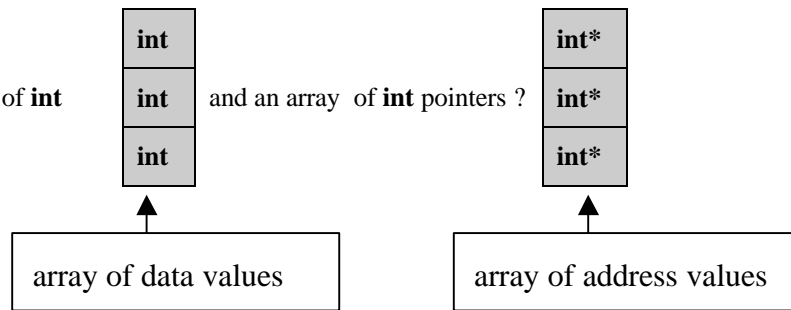
data type of allocated memory

The `int*[3]` means an array of int pointers. The advantage of allocating memory for arrays is that you can have columns of different lengths. **r** points to the starting address of a 1 dimensional array having type integer pointer. One star is used for to point to an array of pointers and the other star is used to point to an array of integers. (pointer to pointer)



array of int's and arrays of int*

What is the difference between an array of **int**



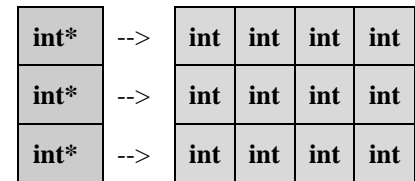
step 2

Next we allocate an array for each row pointer. Each row of columns is an array of integers.

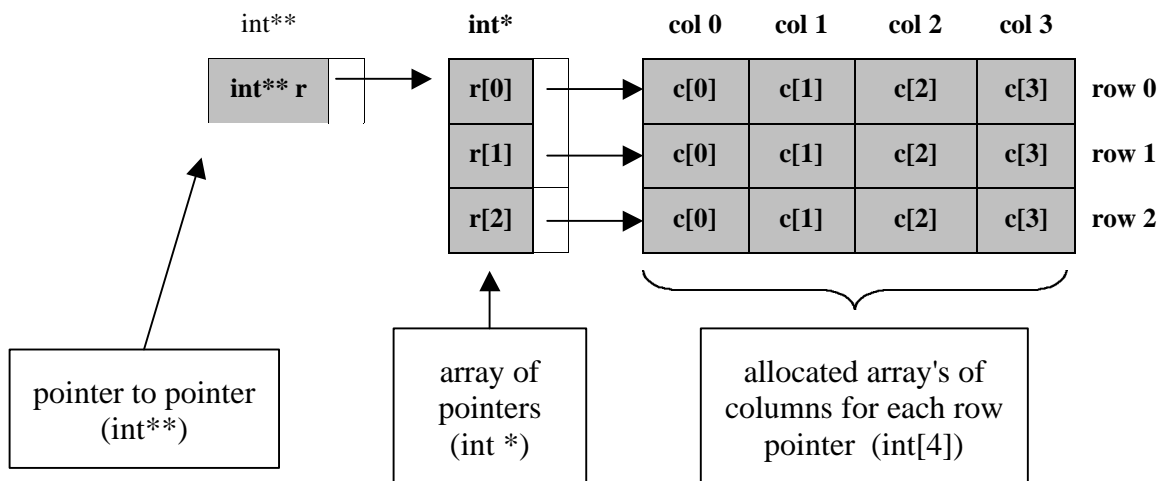
`r[0] = new int[4];` // allocate memory for an array of columns for row index 0

`r[1] = new int[4];` // allocate memory for an array of columns for row index 1

`r[2] = new int[4];` // allocate memory for an array of columns for row index 2



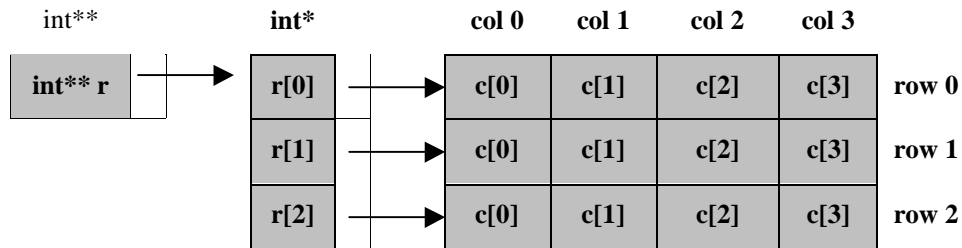
Now we have a 3 * 4 array of two dimensions allocated by using pointers. We first allocate memory for an array of row pointers, then we allocate memory for the columns of each row. Each element of array **r** points to an array of row columns. **r** points to an array of integer pointers.



`r[0]` will hold the starting address of row 0 an array of `int`'s, where `r[1]` will hold the starting address of row 1 an array of `int`'s. What is the difference between `int*[3]` and `int[4]` ?

Accessing individual array elements of a 2 dimensional array

How do I access the individual elements of each row of columns ? Easy ! Each pointer of r point's to a array of row columns! This means all you have to do is access the address of each row that points to the row of columns you want.



You can do everything in one step using the `**` operator on the array of row pointers:

```
x = **r; // get value at row index 0 column 0
```

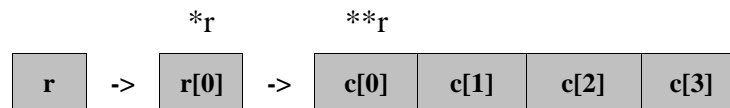


Figure out what the rest represent ?

(Hint : do the inner operations first to get which row of the array)

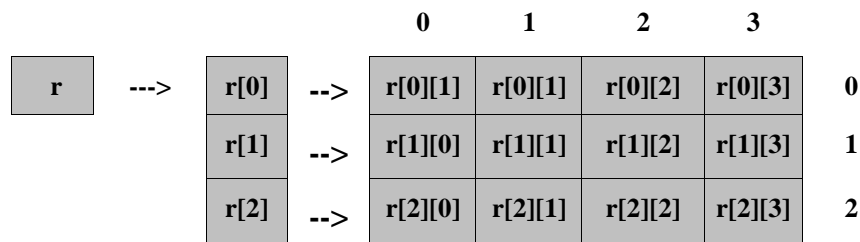
```
x = *(r+1); // value of row index 1 column index 0
x = *(*r+1); // value of row index 0 column index 1
x = *(*r+2)+3; // value of row index 2 column index 3
```

The trick is that the inner star get a pointer to a particular row, the outer star gets you the value at a particular column in the row. The first offset to `r` gets you a particular row and the offset after the first star gets you a particular column. There must be an easier way to access array elements ? Yes there is! An easy method is to use array indexes on `r`:

`x = r[2][3];` // equivalent to `* (* (r + 2) + 3)`

row column which row which column

This works because the compiler sets up a 2 dimensional array as a pointer to pointers You can say that `**r = r[][]`. But not all compilers will do this for you.



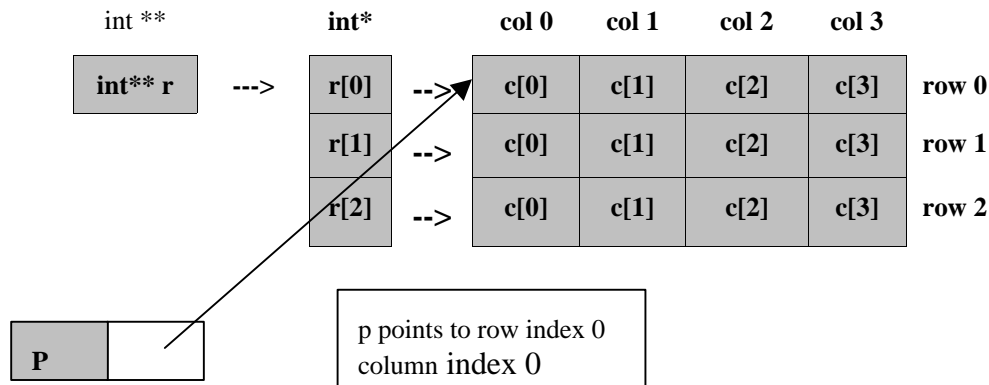
using variables as array index's

You can also use variables for index pointers.

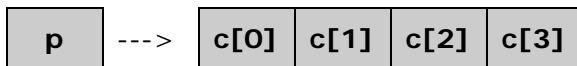
```
i = 2; // row index
j = 3; // column index
x = r[2][j]; // using column index j
x = *(r[i] + 2); // using row index i
x = r[i][j]; // using both row and column index i, j
```

pointing to a row with a pointer

```
int *p = r[0]; // p points to the row index 0
```



You can get a pointer to each row of columns using an array index from the array of row pointers.



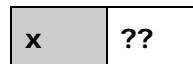
Once you get a pointer to a particular row of column of then it is easy to get the value from which element you want. .You must use an offset from the start of the row of columns.

```
*p = 5; // assign a value to the element at row index 0 column index 0
```

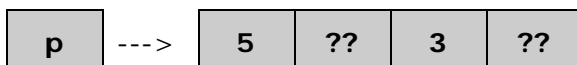


```
x = *p; // x holds the value at array element row index 0 column index
```

What is the value of x ?

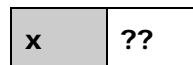


```
*(p+2) = 3; // assign a value to the element at row 0 column 0
```



```
x = *(p+2); // x holds the value at array element row 0 column 1
```

What is the value of x ?



Deallocating memory for two dimensional arrays

For the 2 dimensional array you have to delete all the rows of columns and the array of row pointers. You delete in the **opposite way** you allocate. You first delete each array of row of columns:

```
delete[ ] r[0]; // delete memory allocated for columns of row 0
delete[ ] r[1]; // delete memory allocated for columns of row 0
delete[ ] r[2]; // delete memory allocated for columns of row 0
```

**You deallocate
in the opposite
way you allocate**

Second you delete the row array of integer pointers:

```
delete[ ] r; // delete memory allocated for array of row pointers
```

The square brackets [] after the delete operator indicates you are deleting array memory.

You can not delete reserved memory.

LESSON 4 EXERCISE 2

Write a small program by answering the following questions that just includes a main function that covers the material on allocating memory for 2 dimensional arrays. Call your file L4ex2.cpp. All statements are sequential. Declare variables as you use them. You can use **cout** to print out the values of your variables.

1. Declare a 3 * 3 two dimensional integer array called **b** by using pointers to pointers
2. Assign to each array element the value of its row and column
(example row index 2 column index 3 would get the value 23);
3. Print out the values of your allocated array using **cout** do not use loops.
4. Deallocate memory for the two dimensional array.

USING INCREMENTERS AND DECREMENTERS WITH POINTERS

You can also use incrementers and decrementers with pointers. Incrementers and decrementers increment or decrement the contents of a pointer. The contents of a pointer contain an address to a memory location. Incrementers and decrementer for pointers come in handy when comparing and copying strings. They work in a **prefix** mode and in a **postfix** mode. **Prefix** means **before** assignment and **postfix** means **after** assignment. When the compiler increments the contents of a pointer it increments by the data type size. This means for an integer data type the contents would increment by 2 and for a long data type the contents would increment by 4. Incrementers and decrementers will also increment or decrement the values pointed to by the pointers. In this case the values can only be incremented and decremented by 1.

address Incrementers increment by the data type size

address Decrementers decrement by the data type size

operation	description	before	after
p++;	post increment the contents of p after		p = p+1;
++p;	pre increment the contents of p before	p = p+1;	
p--;	post decrement the contents of p after		p = p-1;
--p;	pre decrement the contents of p before	p = p-1;	
x = *(p++); x = *p++;	get the value of what p points to then increment the contents of p after	x = *p;	p = p+1;
x = *(++p); x = *++p;	increment the contents of p before then get value of what p points to	p=p+1;	x = *p;
x = (*p)++;	increment the value of what p points to after	x = *p;	*p = *p+1;
x = ++(*p);	increment the value of what p points to before	*p = *p+1;	x = *p;
x = *(p--); x = *p--;	take the value of what p points to, then decrement contents of p after	x = *p;	p = p-1;
x = *(--p); x = *--p;	decrement the contents of p before then get value of what p points to	p = p-1;	x = *p;
x = (*p)--;	decrement the value pointed to by p after	x = *p;	*p = *p-1;
x = --(*p)	decrement the value pointed to by p before	*p = *p-1	x = *p;

What is the difference between `x = (*p)++` and `x = *(p++)` ?

`(*p)++;` post increments the value pointed to by the pointer by 1

`*(p++);` post increments the pointer by data type size

LESSON 4 EXERCISE 2

Write a program that demonstrates all the above operations. Call your program L4ex2.cpp.

LESSON 4 QUESTION 1

p points to the starting address of a 1 dimensional array called **a** of 5 integer columns that is preinitialized with the following values:

p	Array starting address is 1000.	----->	a	23	18	35	42	10
---	---------------------------------	--------	---	----	----	----	----	----

Fill in the columns for each value of x and p before the statement is executed and after the statement is executed. The statements are part of a program and execute continuously not separately. x is declared as an integer initialized to zero and p is a pointer of data type integer. The compiler would increment the address by 2 rather than by 1 because an integer is 2 bytes not 1 byte. Consider each statements are part of a program. Each statement depends on the values of the previous statement results and executes **continuously**. Hint write the offsets on the top of the array box and the address on the bottom of the array box for each array element.

statement	p before assignment	x	p after assignment
p++	1000		
++p			
p--			
--p			
x = *(p++)			
x = *p++			
x = *(++p)			
x = *++p			
x = (*p)++			
x = *(p--)			
x = *p--			
x = (--p)			
x = *--p			
x = --(*p)			

USING INCREMENTERS AND DECREMENTERS WITH CHARACTER STRINGS

Incrementers and decrementers are great for copying and comparing character strings.

```
/* lesson 4 program 2 */
void main()
```

```
{
// make 2 character strings
char s1[ ] = "cat";
char s2[ ] = "dog";
```

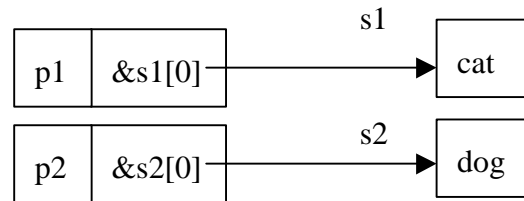
```
char * p1 = s1; // point to start of string s1
char * p2 = s2; // point to start of string s2
```

```
// print out strings
cout << "string s1: " << s1 << " string s2: " << s2 << endl;
```

```
// copy string s2 to string s1
```

```
*p1++ = *p2++;
*p1++ = *p2++;
*p1++ = *p2++;
*p1 = *p2;
```

```
// print out strings
cout << "string s1: " << s1 << " string s2: " << s2 << endl;
}
```

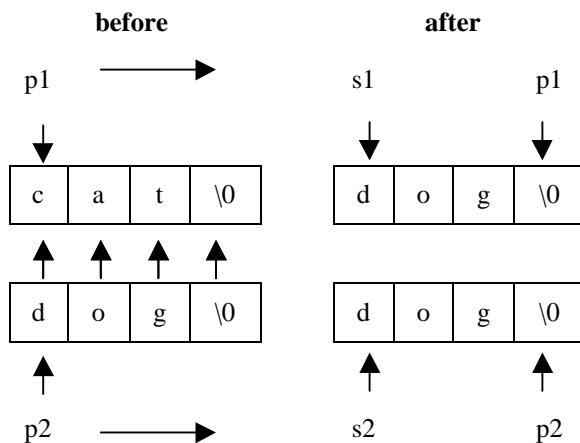


s1: cat s2: dog

What is the value of s1 ? dog What is the value of s2 ? dog

Why do we copy 4 times rather than 3 times ? Copy end of string terminator: '\0'

Every string must end with the character '\0'. If you do not your program will crash.



If we change the following lines. What would the output be ?

From

```
*p1++ = *p2++;
*p1++ = *p2++;
*p1++ = *p2++;
*p1 = *p2;
```

to

```
*++p1 = *++p2;
*++p1 = *++p2;
*++p1 = *++p2;
*p1 = *p2;
```

STRUCTURE DEFINITIONS

Structures allow you to group many **different data types** variables together under one common name. Each data type variable is known as a **member** variable of the structure. Structures may even include other structures and arrays. A Structure must be defined before it can be declared and used. **Typedef** does need to be used with structures. The C++ compiler treats a structure like a data type automatically. A structure must be defined before it can be used. A structure definition gets a name so that the compiler can identify the structure definition you want to use. Once you define a structure you have your own user structure data type.

```
struct structure_definition_name
```

```
{
    data_type variable_name;
    data_type variable_name;
    data_type variable_name;
    data_type variable_name;
};
```

```
struct record
```

```
{
    int x;
    float y;
    char c;
    int a[5];
};
```

Structures are used to store many different data types. Structures are convenient to store information about a particular item. For example a structure may be used to store information about an employee. An employee needs a name, an address, a social insurance number, salary etc. Each **field** of the record is represented by a data type. Why do we add 1 to the char string size ?

struct employee

```
{
    char name[30+1];
    char address[40+1];
    char SIN[9+1];
    float salary;
};
```

name	address	SIN	salary
------	---------	-----	--------

We need structures so that we can group together variables that have something in common.

Structures must be:

- (1) Defined
- (2) Declared
- (3) Used

Declaring structures as a variable

After a structure is defined it must be declared before it can be used.. You declare a structure in the same way you declare a variable. The compiler automatically **reserves** memory for the structure at compile time. Without the structure **definition** the compiler would not know how to make the structure. A structure definition tells how the compiler to make the structure when you declare it. You cannot declare a structure unless you first have a structure definition. To declare a structure you state your structure definition name as your user data type and the name for your structure variable

```
structure_definition_name structure_variable_name;
```

```
record rec; // declare a structure rec of data type structure record
```

rec	x	??
	y	??
	c	??
	a[5]	??

accessing values where structured declared as a variable

The dot "." operator is used to access individual members of a structure when declared as a **variable**

```
record rec; // declare a structure rec of data type structure record
```

```
rec.x = 5; // assign 5 to the x element of structure rec.
```

```
int v = rec.x; // assign the x element of structure rec to i
```

What is the value of v ?

rec	x	5
	y	?
	c	?
	a[5]	

If variable of a structure is an array then its index must be included

```
rec.a[2] = 5; // assign 5 to element index 2 of a belonging to rec
```

```
int v = rec.a[2]; // assign a element index 2 of structure rec to v
```

. dot operator used to access individual members of a structure

EXAMPLE PROGRAM L4P2

The following program demonstrates declaring and using structures as a variable

```
// Lesson 4 program 2
#include <iostream.h>

// define a structure called record
struct record
{
    int x;
    float y;
    char c;
    int a[5];
};
```

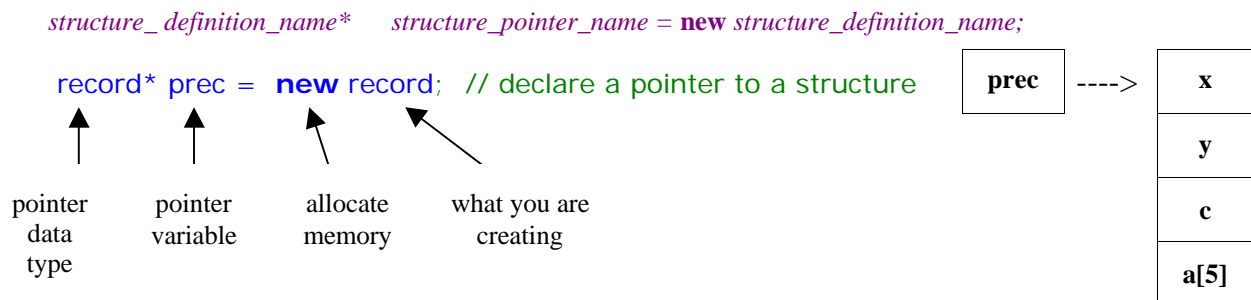
```
// program to demonstrate using structures
void main()
{
    record rec; // declare record as a variable
    rec.x = 5; // assign 5 to member x
    rec.y=10; // assign 10 to member y
    rec.c='A'; // assign letter 'A' to member c
    cout << "x = " << rec.x << endl; // print out value to screen
    cout << "y = " << rec.y << endl; // print out value to screen
    cout << "c = " << rec.c << endl; // print out value to screen
}
```

LESSON 4 EXERCISE 3

Define a structure of your favorite data type for example a structure to represent an employee. In your main function declare the structure as a variable. Initialize each member of the structure from the key board using **cin**. Use **cout** to print out the values to the computer screen Call your program L4ex3.cpp.

Allocating memory for structures

You can also have pointers to structures. You first use the **new** operator to allocate memory for a structure in run time. The start of the address of the structure is assigned to the pointer. To declare a pointer to a structure:



What is the difference between declaring a structure as a variable and as a pointer to a structure ?
What is the difference between reserved memory and allocating Memory ?

Accessing structure elements with a pointer

The arrow "->" operator is used to access individual members of a structure pointed to by a pointer

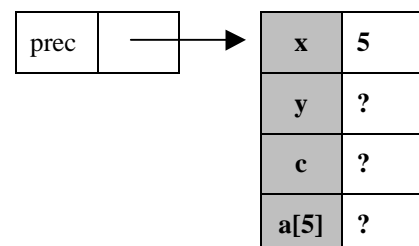
-> arrow operator used to access individual members of a structure pointed to by a pointer

```
record* prec = new record; // allocate memory for a structure
```

```
prec->x = 5; // assign 5 to member x
```

```
int v = prec->x; // retrieve the value of member x
```

What is the value of v ?



If variable of a structure is an array then its index must be included

```
prec->a[2] = 5; // assign 5 to element index 2 of a belonging to rec
```

```
int v = prec->a[2]; // assign a element index 2 of structure rec to v
```

Do you know what `prec->x` means ??

`prec->x` is equivalent to: `(*prec).x`

`p->x = (*p).x`

where `*p` is the value of the memory location pointed to by `p`

EXAMPLE PROGRAM L4P2

The following program demonstrates declaring and using structures as a pointer

```
// Lesson 4 program 2
#include <iostream.h>

// define a structure called record
struct record
{
    int x;
    float y;
    char c;
    int a[5];
};

// program to demonstrate using pointer to structure
void main()
{
    record* prec = new record; // allocate memory for record
    prec->x = 5; // assign 5 to member x
    prec->y = 10; // assign 10 to member y
    prec->c = 'A'; // assign 'A' to c
    cout << "x = " << prec->x << endl; // print out value to screen
    cout << "y = " << prec->y << endl; // print out value to screen
    cout << "c = " << prec->c << endl; // print out value to screen
    delete prec;
}
```

program output:

```
5
10.5
'A'
```

LESSON 4 EXERCISE 3

Define a structure of your favorite data type for example a structure to represent an employee. In your main function declare the structure as a variable. Declare a pointer variable to your structure. Initialize each member of the structure from the key board using **cin**. Set the structure pointer to the address of your structure. Print out the values of the structure by using the pointer to your structure pointer. Use **cout** to print out the values to the computer screen. Call your program L4ex3.cpp.

initializing structures

If you know what values you want defined to your structure you can include them in an initialization list when you declare your structure as a variable.

```
record rec = {5,10,'a',{1,2,3,4,5}};
```

Notice we needed a separate curly brackets for the array located inside the structure. You can initialize all values to zero in the structure with

```
record rec = {0};
```

ARRAYS OF STRUCTURES

reserved structure array

You can also declare arrays of structures:

```
structure_definition variable_name[column_size];
```

```
record arec[4]; // declare an array of 4 records
```

```
// define a structure called record
struct record
{
    int x;
    float y;
    char c;
    int a[5];
};
```

arec

arec[0]	arec[1]	arec[2]	arec[3]
x	x	x	x
y	y	y	y
c	c	c	c
a[5]	a[5]	a[5]	a[5]

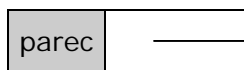
**reserved memory
for arrays of
structure**

allocated structure arrays

You can also declare a structure pointer to point to an allocated array of structures:

```
structure_definition* structure_name = new structure_definition[ size_of_columns];
```

```
record* parec = new record[4 ]; // create an array of records
```



parec[0]	parec[1]	parec[2]	parec[3]
x	x	x	x
y	y	y	y
c	c	c	c
a[5]	a[5]	a[5]	a[5]

**allocated memory
for array
of structures**

The pointer prec points to the first record in the array

```
arec[1].x = *(arec+1).x
parec[1].x = *(parec+1).x
```

accessing elements in an array of structures

When you have an array of structures then the array index must be included to indicate which structure in the array you want to access for writing or reading. The dot operator is used for accessing the elements in each structure of the array. Why the dot operator ? By using the array index you already have a precalculated address location so you just use the dot operator to access the individual members of the structure. You must treat each structure in the array as a separate structure. `arec[1].x` really means `*(arec+1).x` since the `*` is around `*(arec+1)` then we have a value not a pointer so we have to use the dot operator to access the element in that structure.

```
arec[1].x = 5; // assign 5 to the member x value of structure arec index 1
parec[1].x = 5; // assign 5 to the member x value of structure parec index 1
int v = arec[1].x; // assign to v the member x value of structure arec index 1
int v = parec[1].x; // assign to v the member x value of structure parec index 1
```

If the member of the structure you want to access is an array then you must specify it's index too.

```
arec[1].a[2]=5; // assign a element index 2 of structure rec index 1 to v
int v = arec[1].a[2]; // assign a element index 2 of structure rec index 1 to v
```

using variables to specify array indexes

You may use variables having index values to access array structures.

```
int i = 0,j = 1;
int v = arec[i].x;
int v = rec.x[j];
int v = arec[i].a[j];
```

EXAMPLE PROGRAM L4P3

The following program demonstrates declaring and using arrays of structures as a variable and as a pointer.

```
// Lesson 4 program 3
#include <iostream.h>

// define a structure called record
struct record
{
    int x;
    float y;
    char c;
    int a[5];
};
```

```
// program to demonstrate using array of structures
void main()
{
    record arec[10]; // declare record array as a variable
    arec[0].x = 5; // assign 5 to member x of array element 0
    cout << "x = " << arec[0].x << endl; // print out value to screen
    record* parec = new record[10]; // allocate memory for array of reco
    parec[0].x = 5; // assign 5 to member x of array element 0
    cout << "x = " << parec[0].x << endl; // print out value to screen
    delete[] parec;
}
```

5
5

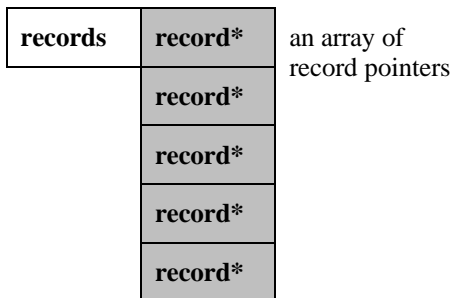
LESSON 4 EXERCISE 4

Make an array of structures like Employee, and initialize one of the structures with values from the keyboard. Declare a pointer to one of your structures in your array. Print out the values of the structure to the screen using the pointer. Call your program L4ex4.cpp.

array of pointer structures

Sometimes you need an array of pointers to structures. Do you know what an array of structure pointers is ? An array of structures pointers is simply an array of pointers to structures. Each pointer will contain the address to a memory block representing an structure. You can declare an array of pointer to structures as a variable or as a pointer. To declare an array of pointer to structures as a variable you declare a variable with square brackets to indicate an array and use a structure data type pointer indicated by a star *.

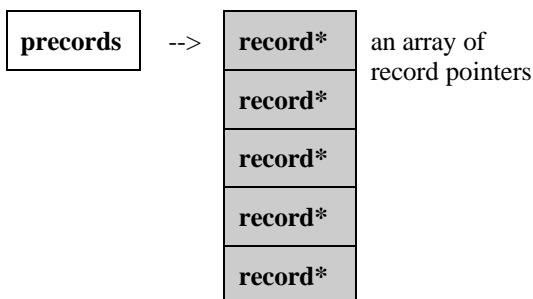
```
record* records[5]; // declare an array of structure pointers
```



reserving memory for an array of
structure pointers in compile time

To declare an array of allocated structures pointers you need a pointer to pointer. Notice we create an array of pointers to structures. `record*[5]`.

```
record** precords = new record*[5]; // declare and allocate a pointer to an array of structure pointer
```

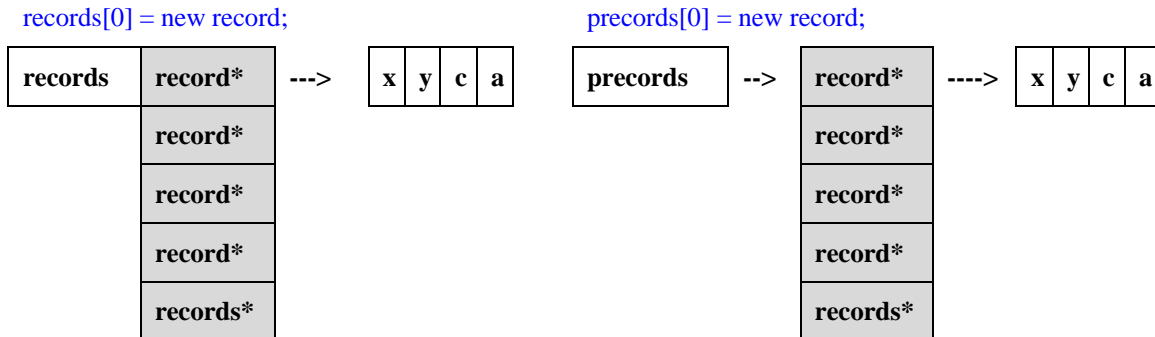


allocating memory for an array of
structure pointers in run time

What is the difference between an array of structures and an array of structure pointers ?

allocating structures

Before you can use your array of pointers to structures you need to allocate some structures. You allocate a structure and assign the starting address of the structure to one of the elements in your array of pointers to structures.



Accessing individual pointers in an array of structure pointers

To access an individual element of a structure pointer to you first need to get a pointer to the structure

```
record* prec = records[0]; // get pointer to structure from array of pointers declared as a variable
record* prec = precords[0]; // get pointer to structure from array of pointers declared as a pointer
```

In either situation you get a pointer to a record. Once you get a pointer to the structure you want to access its individual members. We use the arrow `->` operator because we have a pointer to a record. The contents of the pointer stores the address of where the structure is located in memory. Remember `prec->x` means `*(prec).x`

```
prec->x = 5; // write to x member
int v = prec->x; // access x member pointed to by prec
```

We use the `--->` arrow operator because the contents of the array are pointers

You can access the structure members directly.

```
records[0]->x = 5; // access x member of structure pointed to array of record pointer
precords[0]->x=5; // access x member of structure pointed to array of record pointers
int v = records[0]->x; // access x member of structure pointed to array of record pointers
int v = precords[0]->x; // access x member of structure pointed to array of record pointers
```

When accessing members of the structure why do we use the arrow operator and not the dot operator ?

EXAMPLE PROGRAM L4p4

The following program demonstrates using an array of structure pointers declared as a variable and as a pointer.

```
#include <iostream.h>

// define a structure called record
struct record

{
    int x;
    float y;
    char c;
    int a[5];
};

// program to demonstrate using an array of structure pointers
void main()

{
    record* records[5];
    records[0] = new record; // allocate a structure to one of the elements of the array
    records[0]->x=5; // assign a value to one of the members of the structure
    cout << records[0]->x << endl; // print out value to the screen
    record** precords; // declare an array of record pointers as a pointer
    precords = new record*[5]; // declare an array of record pointers as a pointer
    precords[0] = new record; // allocate a structure to one of the elements of the array
    precords[0]->x = 5; // assign a value to one of the members of the structure
    cout << precords[0]->x << endl; // print out value to the screen
    delete[] precords[0];
    delete precords;
}
```

LESSON 4 EXERCISE 5

Make an array of structure pointers, allocate memory for a structure. Declare a pointer to your allocated structure using the array of structure pointers. Initialize your allocated structure with values from the keyboard using the pointer. Print out the values of your allocated structure to the screen using the pointer. Call your program L4ex5.cpp

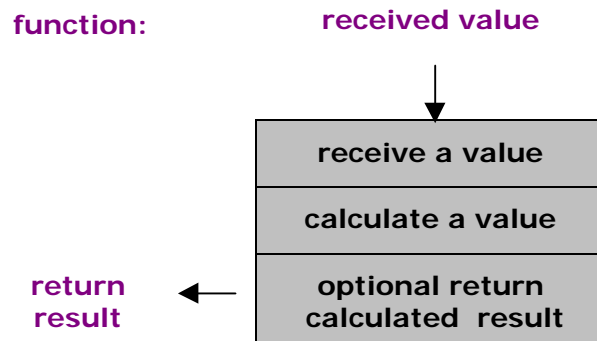
C++ PROGRAMMERS GUIDE LESSON 5

File:	CppGuideL5.doc
Date Started:	July 12, 1998
Last Update:	Mar 23, 2002
Version:	4.0

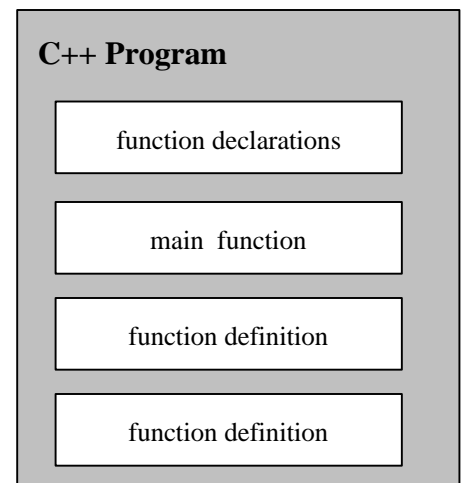
LESSON 5 C++ FUNCTIONS

FUNCTIONS

Functions are made up of variables and programming statements. Variables in a function are temporary and are used for intermediate calculations. Programming statements in a function execute sequentially, one at a time, one by one until the end of the function is reached. Functions receive values from other functions, calculate values and may or may not return a calculated value. Functions are needed to avoid repetition in a program. Functions allow an easy way out to accomplish difficult tasks. For example, a "square" function can be used repeatedly over and over again to calculate the square of a number. Without functions, there would be lots of repetitious typing.



Functions need to be **declared** and **defined** before they can be used. When you declare a function you tell the compiler its name, what values it receives and what values it returns. Function declarations are places at the top of a program. When you define a function you list the function name, what values it receives and what values it returns and then you write all the programming statements. All function definitions are usually placed after the main function. The main function is used to call other functions. The main function is the first function to be executed when your program runs.



declaring a function

Before a function can be used, it must be declared. Declaring a function is also called a **prototype**. Function prototypes are used to inform the compiler about a function that will be later defined and used in a program. The function **prototype** lists the **return data type**, the **function name** and a **parameter list**. The **parameter list** is used to list all the values a function will receive. Each receive value is stored in a parameter. A parameter has a data type and a parameter name. A function prototype ends with a semicolon to indicate you are declaring a function prototype not defining one.

```
return_data_type function_name (parameter_list);
```

`int square (int x);` // declare function called square

return type function name parameter semi-colon

Once a function is declared, the compiler knows how to use it when it encounters it in your program. The function prototype allows you to use functions before you define them. The **return data type** indicates what kind of data the function will return. The function may return a value or not return a value. When a function does not return a value the keyword **void** is used for the return data type

```
void putValue(int x); // function does not return a value
```

The function **name** identifies the function. A function accepts values through a **parameter list**. A parameter list is a list of **parameters**

```
(parameter1, parameter2 .. parameter n)
```

A parameter has a **parameter data type** and a **parameter name**.

```
parameter_data_type parameter_name
```

`int x`

parameter data type parameter name

```
// function receives a value and returns a value
int square (int x);
```

The parameter **type** indicates what kind of data the parameter is going to represent. Every parameter gets a **name** for identification. The name is optional, but is a good thing to have. The parameter is just used in the function, and is not associated with any other variable outside the function. A parameter acts like a temporary variable inside your function. The only difference is that a parameter is pre-initialized with a received value. If a function does not accept any parameters then the parameter list is empty.

```
int getValue(); // function accepts no parameters
```

void can also be used in the parameter list of a function if it accepts no parameters. This is seldom used, if so only by programmers who are paranoid.

```
int getValue(void); // function accepts no parameters
```

defining a function

Once a function prototype is declared, the function can be defined. A function definition must list the **return data type**, **function name** and a **parameter list** and all the needed variables and programming statements. The **return data type** specifies what kind of data type the function will return. The function **name** is used to identify the function. The **parameter list** contains parameters to receive values from other functions. The parameters act like temporary variables inside a function. A function definition does not terminate with a **semicolon** and has function variables and statements enclosed by curly brackets `{}`. The function definition contains variables and programming statements. Variables inside a function are only known to that function only, and are said to be **local** to the function. Variables inside a function are only temporary meaning the values disappear after the function is called.

```
return_type function_name (parameter_list)
{
    variables and statements
    optional return statement
}
```

```
// calculate square of a number returns a value
int square(int x)
{
    int y = x * x; // calculate square of x
    return y; // return result of calculation
}
```

The square function receives an integer value through its parameter x

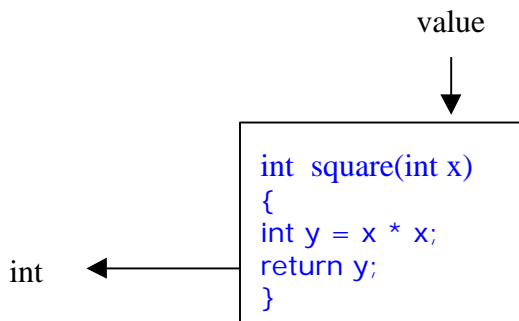
```
int square(int x)
```

it then squares this value.

```
int y = x * x; // calculate square of x
```

It then returns the value.

```
return y; // return result of calculation
```



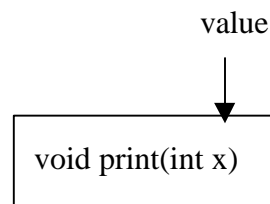
The **return statement** inside a function is used to return a value from the function.

```
return expression;
```

```
return y; // terminate function and return a value
```

Some functions do not return a value. We can define the **putValue** function that prints out a value.

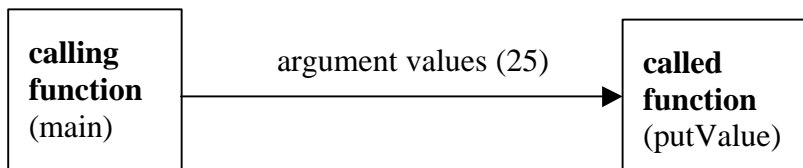
```
// print out value of a number
void putValue(int x)
{
    cout << "the value is: " << x << endl;
}
```



using a function

When one function uses another function it is called a **function call**. For one function to use the services of another function it **calls** that function by its **name** and passes **values** to that function in an **argument list**. The function using another function is known as the **calling** function. The function being used is known as the **called** function. To call a function:

```
function_name ( argument_list);  
putValue(25); // call the function putValue
```



The calling function send values to the called function in an argument list. The called function receives the values through its parameter list. An argument list is a list of values passed to a function, each value is called an argument.

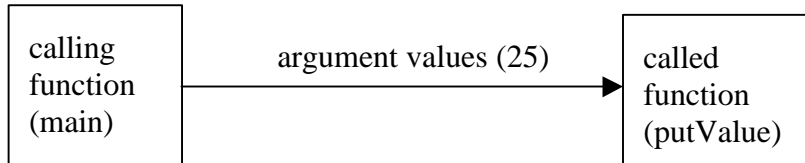
```
(argument 1, argument 2 ..... argument n);
```

An argument value is an expression usually a variable or a constant, but may be an arithmetic expression or another function calls.

```
(expression, constant, variable, function())  
( 5, x, x + y, sum( ) )
```

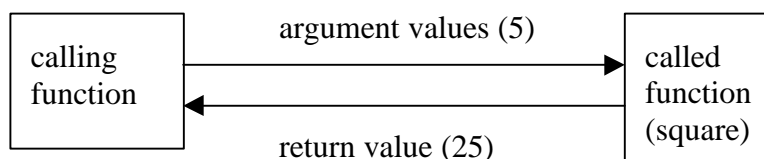
The argument list lets the calling function send values to the called function. The **putValue()** function receives the value 25.

```
putValue(25); // call the function putValue
```

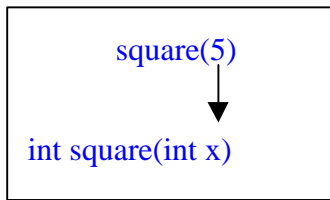


Some functions receive and return a value like the **square** function. When the square function is called, the **called** function receives a value, does a calculation and then returns a value. The return value from the square function is assigned to a variable of the **calling** function by use of an **assignment** statement. The variable **x** receives the value 25 which is the square of 5.

```
variable_name = function_name (argument_list);  
int x = square(5); // variable x gets the square of 5 which is 25
```



When a function receives a value from another function its like an assignment statement



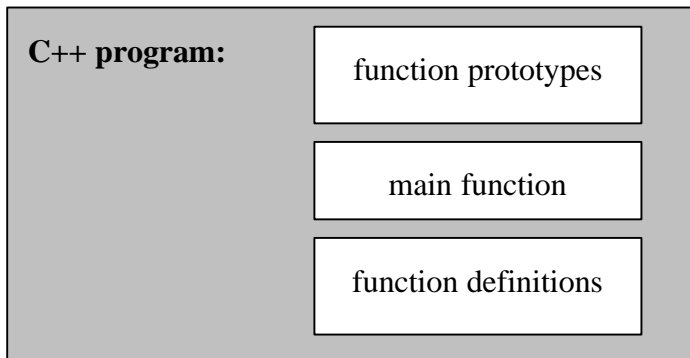
This is the same thing as saying

`x = 5;`

The parameter x can be used inside the function as a temporary variable.

using function prototypes in a program

Prototypes allow you to use functions before you define them. The body of the function is to be defined some where else in your program. Prototypes are declared at the top of a program or in a header file. Prototypes inform the compiler about the functions that are going to be used inside your program. Prototypes enforce data type checking. This means the compiler makes sure if your function requires a int data type as a parameter that it gets an int data type argument. It makes good sense to use prototypes. If you do not use prototypes then you must declare your functions in the order that they are called. This is a nuisance and difficult to do. When you use prototypes, you can use your functions in any order before they are defined.



The functions are declared before they are used and defined.

LESSON 5 PROGRAM 1

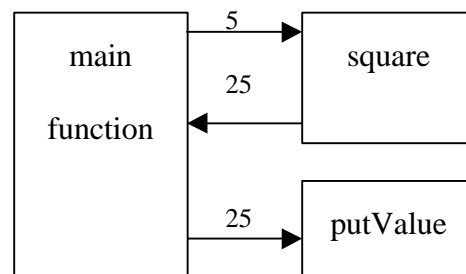
The following program calls the square function, calculates square of a number and prints the results

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

```
// declares function prototypes
int square(int x); // calculate square of x
void putValue(int x); // print out results
```

```
// main function
void main()
```

```
{
// call the square function to calculate the square of 5
int x = square(5); // variable x is assigned the square of 5 which is 25.
putValue(x); // prints out the result
}
```



the value is: 25

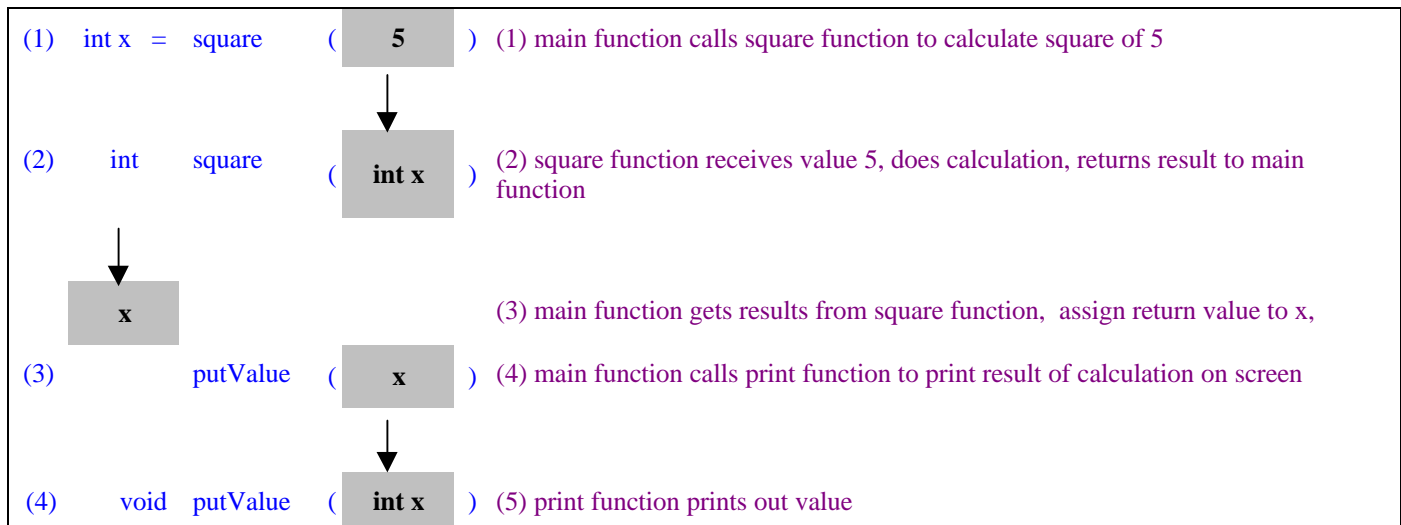
```
// calculate square of a number
int square(int x)
{
    int y = x * x; // calculate square of a number
    return y; // return result of calculation
}

// print out value of a number
void putValue(int x)
{
    cout << "the value is: " << x << endl;
}
```

parameters are like
variables declared
inside a function they
are initialized by the
valued they received

parameter = value
x = 5;

sequence of events



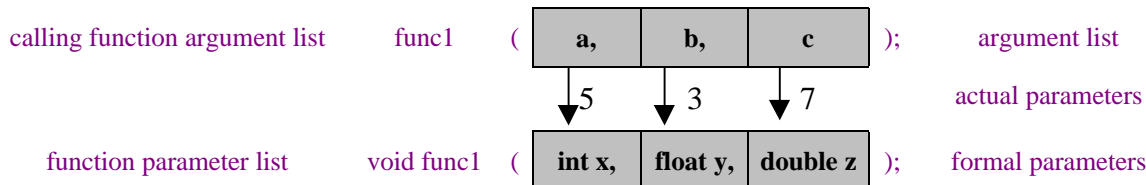
LESSON 5 EXERCISE 1

Write a `int getValue()` function to the above example that prompts the user to enter a number, gets a number using `cin` and returns a number from the keyboard. Call your finished program `L5ex1.cpp`

Passing values to functions

Values are passed to the **called** function from the **calling** function by an **argument** list. The **called** function receives the values through its **parameter list**. There is a sequential one to one correspondence between the **calling** functions argument list and the **called** functions parameter list. The parameter name maybe the same or different but the data types must be identical. The parameters **names** are known as the **formal** parameters and the received values are called the **actual** parameters. The arguments must be of same data type as the parameters. Do you know why ?

The argument values are mapped in order into the parameters.
 assuming: `int a; float b; double c;`



Arguments `a`, `b`, `c` must have data type `int`, `float` and `double` respectively. Arguments are mapped sequentially to their corresponding parameter. Argument `a` is mapped to parameter `x`. The parameters act like declared variables in a function and are initialized from the argument values.

`x = a; y = b; z = c;`

Parameters act like temporary variables and only last as long as the function, then disappear.

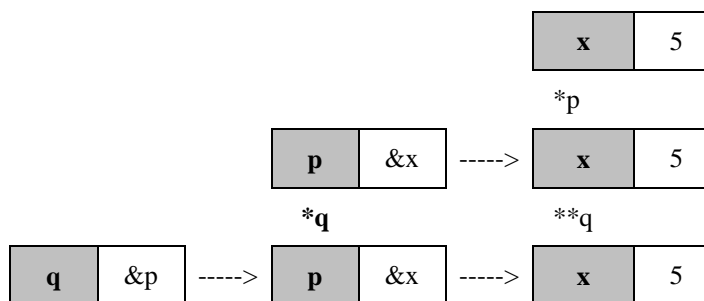
ways to pass parameters to functions

Parameters may receive values and send values back or both receive and send values. Arguments may be passed by **value**, **pointer**, **pointer to pointer**, **reference**, **pointer reference** or **pointer to pointer reference**.

pass types	description
value	passing a value represented by the variable name to the function
pointer	pass an address of a variable to a function (indirect address)
pointer to pointer	pass an address of a pointer to a function (indirect address)
reference	pass a reference of a variable to a function (direct address)
reference pointer	pass a reference of a pointer to a function (direct address)
reference pointer to pointer	pass a reference of a pointer to pointer to a function (direct address)

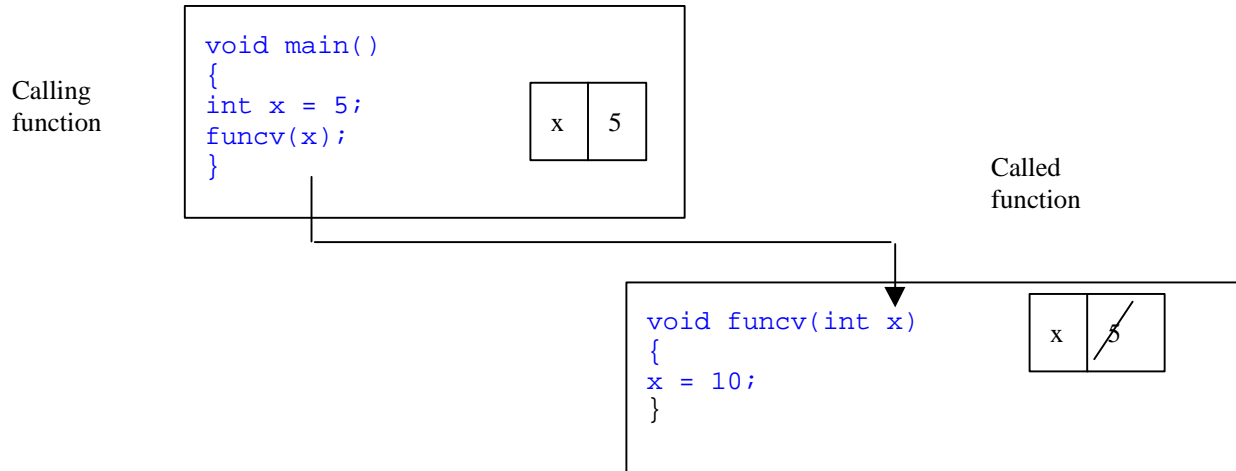
We will first look at pass by value, pointer, and pointer to pointer. The following chart shows the relationships between variables, pointers and pointer to pointers. Every variable `x`, `p` or `q` will have its own address location in memory.

variable	declaring
value	<code>int x = 5;</code>
pointer	<code>int *p = &x;</code>
pointer to pointer	<code>int** q = &p;</code>



pass by value

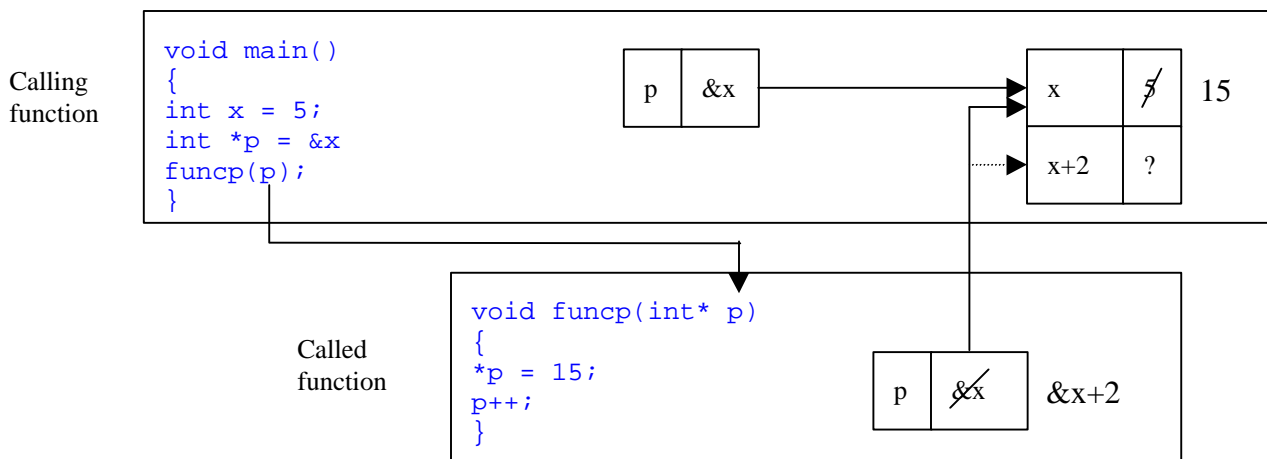
When a variable is passed to a function **by value** the parameter in the called function holds a **copy** of the passed value. The parameter is a temporary memory location to hold the value passed to it. This means you cannot change the value of the variable in the calling function from the called function.



Each function has its own **x**. Each **x** is separate and cannot effect each other.

pass by pointer

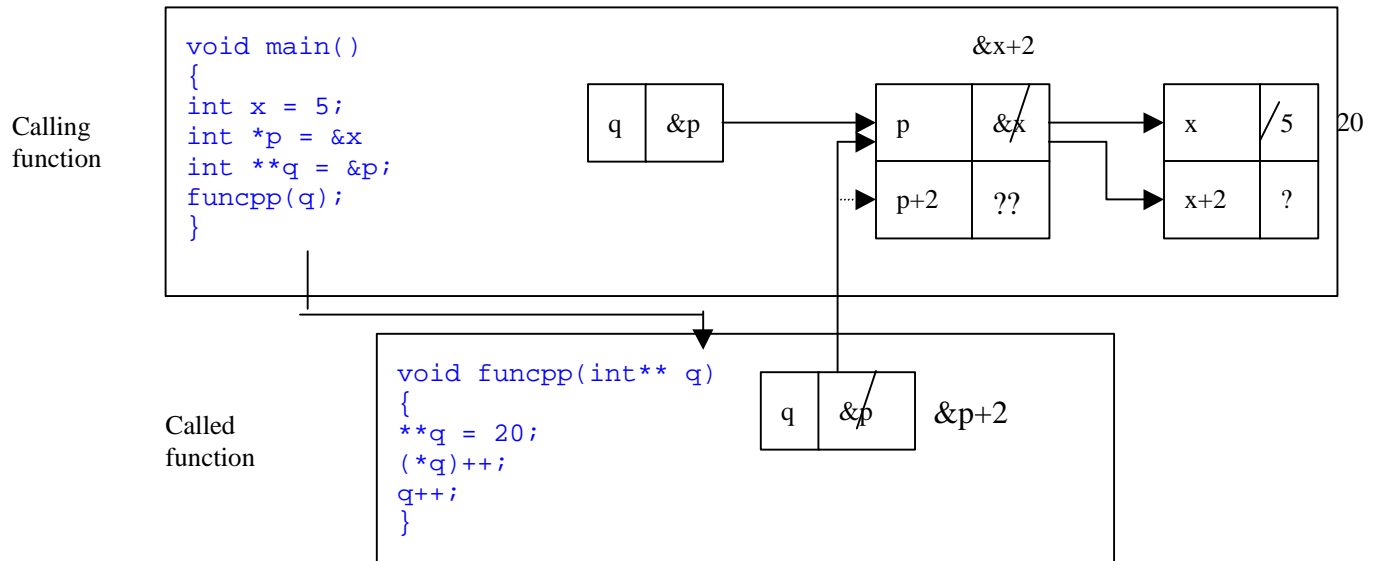
When a variable is passed to a function by pointer, the parameter in the called function stores the contents of the pointer passed to it. This means you can change what the pointer is pointing to in the calling function. You cannot change the contents of the pointer in the calling function from the called function. Passing by pointer is known as **indirect addressing** because a memory location holds the address to be accessed.



Each function has its own pointer **p**. Each pointer **p** is separate and cannot effect each other. Since both pointer point to the same **x** they both can change the value of **x**. You cannot change what the calling function **p** is pointed to inside the called function.

pass by pointer to pointer

When a variable is passed by pointer to pointer the contents of the pointer to pointer is stored in the parameter. You can change what the pointer to pointer is pointing to in the called function. You cannot change the contents of the pointer to pointer in the calling function from the called function. Passing by pointer is also known as **indirect addressing**.



summary:

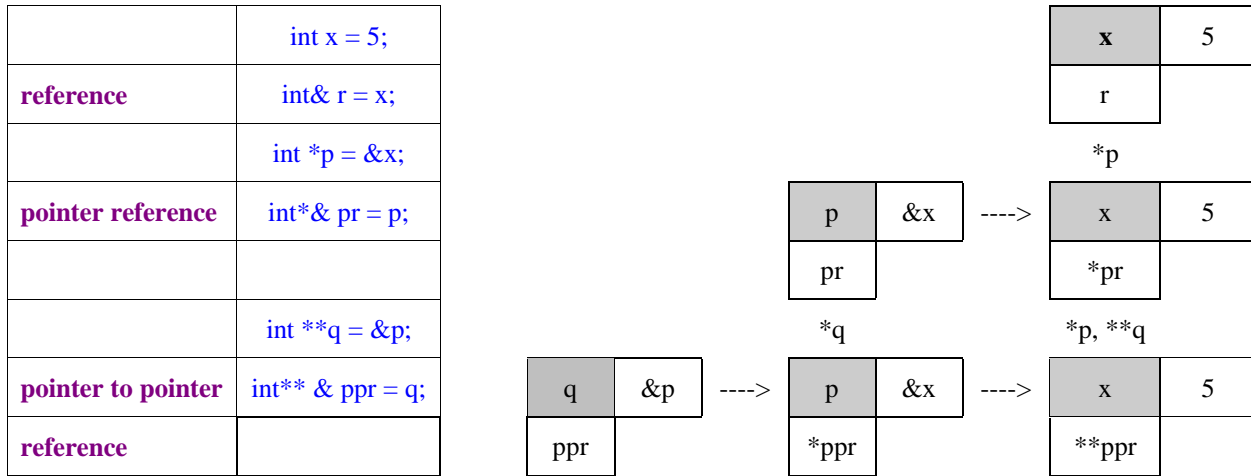
pass by	what can be changed or not changed
value	cannot change value outside function
pointer	can only change the value of what the pointer is pointing to outside function
pointer to pointer	can only change the value of what the pointer to pointer is pointing to and the value of what its pointer is pointing to

LESSON 5 EXERCISE 2

Write a program that demonstrates for pass values by value , pointer and by pointer to pointer for variable x, pointer p and pointer to pointer q. Print out the values of x, p and q before the functions are called and after the functions are called. Trace through the function line by line examining the values. Call your finished program L5ex2.cpp.

pass by reference, pointer reference and pointer to pointer reference

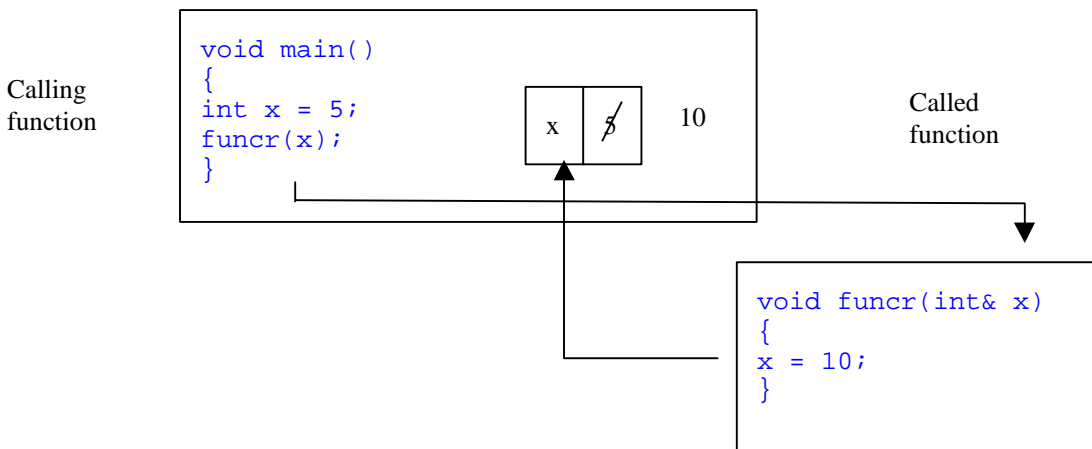
The following chart shows the relationships between references, pointer references and pointer to pointer references. Every variable x, p or q will have its own address location in memory and the references will **refer** to them.



A reference parameter refers to a variable in the calling function. It may refer to a non-pointer, pointer or pointer to pointer variable. When you change the value using a reference parameter in the called function the value in the calling function also changes. This is because the reference parameter refers to that variable. Refer means it is the same variable and you definitely can change it. The power of using references is apparent when using functions. When you pass a variable by reference and change the value of the variable inside the **called** function the value also changes in the **calling** function as well.

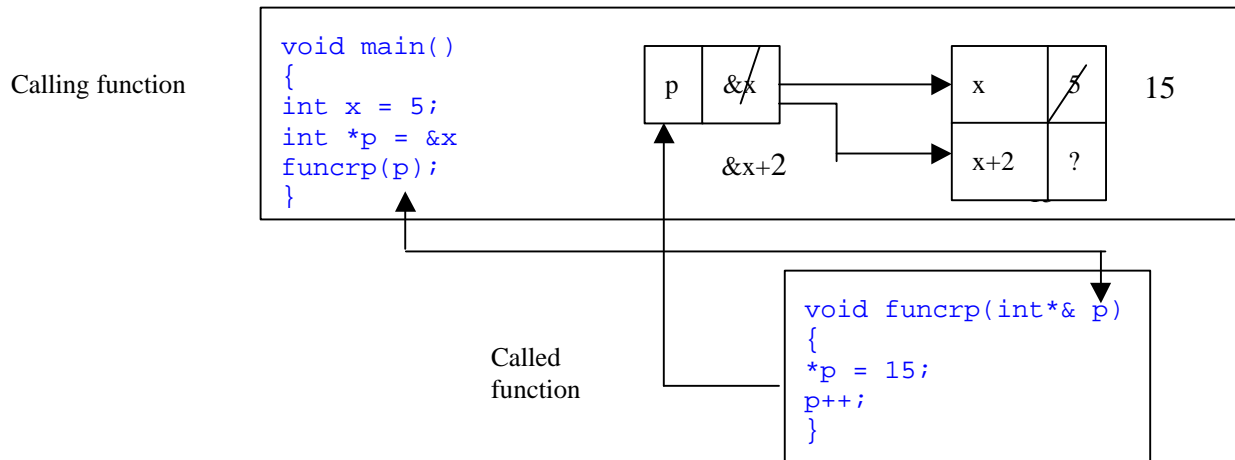
pass by reference

When a variable is passed to a function by reference the parameter refers to the passed variable. You can change the value of the variable of the calling function in the called function. Do not confuse references with a pointer. A pointer is a variable that contains an address value a reference represents another variable. Passing by reference is known as **direct addressing** because the direct address of a variable is passed to the function.



pass by reference pointer

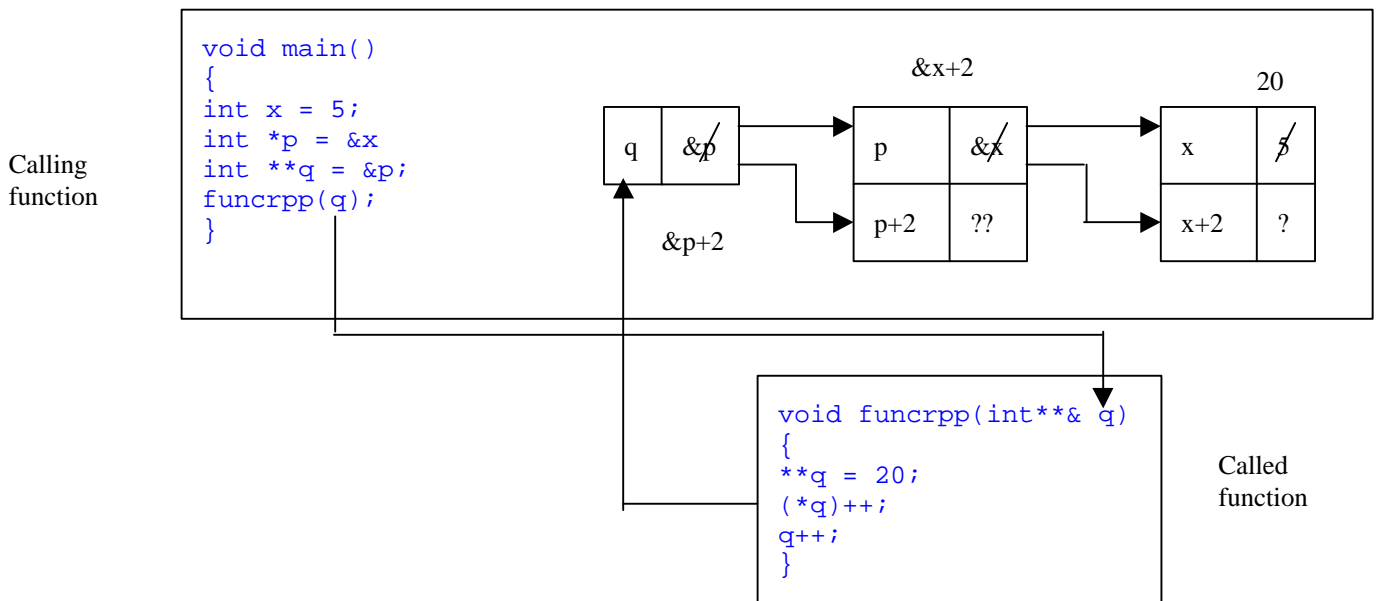
When a variable is passed to a function by reference pointer the parameter refers to the passed pointer variable. You can change the value of the variable of the calling function in the called function. Do not confuse a reference pointer with a pointer to pointer. A pointer to pointer is a variable that contains an address value of a pointer. A reference pointer just represents another pointer variable.



Each function has its own pointer p. Each pointer p is separate and cannot effect each other. Since both pointer point to the same x they both can change the value of x. You cannot change what the calling function p is pointed to inside the called function.

pass by reference pointer to pointer

When a variable is passed to a function by reference pointer to pointer the parameter refers to the passed pointer to pointer variable. You can change the value of the variable of the calling function in the called function. Do not confuse a reference pointer to pointer with a pointer to pointer. A pointer to pointer is a variable that contains an address value of a pointer. A reference pointer just represents another pointer to pointer variable.



LESSON 5 EXERCISE 3

Write a program that demonstrates pass by reference, pointer reference and by pointer to pointer reference. for variable x, pointer p and pointer to pointer q. Print out the values of x, p and q before the functions are called and after the functions are called. Trace through the functions line by line examining the values. Call your finished program L5ex3.cpp.

LESSON 5 EXERCISE 4

Have a function receive a value and equate to a pointer inside a function. Have a function receive a pointer and equate to a reference pointer inside a function. Have a function receive a reference and equate to a pointer inside a function. Have a function receive a pointer to pointer and equate to a reference pointer to pointer inside a function. Have a function receive a reference pointer to pointer and equate to a pointer to pointer inside a function. In all cases print out the value of the received parameter and the temporary variable inside the function. Call your finished program L5ex4.cpp.

returning values from functions

The **return statement** inside a function is used to return a value in the form of a expression from the function. The expression may be a variable or an arithmetic expression.

return expression;

```
return x;           // terminate function and return a
                    value
```

An example using a return statement with an expression.:

```
// calculate square of a number
int square(int x)
{
    return = x * x; // calculate square of a number
}
```



The **return** statement is not necessary if the function does not return a value. If the function does not return a value then you can force the function to terminate before executing a **return** statement. A return statement with no return variable forces the function to terminate.

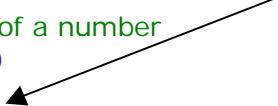
return expression;

```
return;           // terminate function do not return a
                    value
```

To use a return statement without an expression the function must be proceeded with the keyword **void** indicating that the function does not return a value.

Example function using a return statement **without** an expression.

```
// calculate square of a number
void putValue(int x)
{
    if(x < 0) return ; // if number negative return from this function
    else cout << "the value is: " << x << endl; // print out value
}
```



You can also have a return data type of **void***. Using **void*** means return a pointer without any implied data type. In this case you need to return a pointer.

```
void* incp(void* p)
{
    p++; // increment pointer
    return p; // return pointer
}
```

What is the difference between void and void* ?

return by value, pointer, reference

A function can also return by value. pointers and references.

Return by value	simply returns a value to the calling function
Return by pointer	returns a pointer to a memory block or variable.
Return by reference	returns a reference to a variable.

returning a pointer

You must be careful in using return by pointer or return by reference. In both cases you cannot return any variables declared in the function. Variables declared in a function are temporary. They disappear when the function terminates. If you had a pointer or reference to a temporary variable then returned pointer would point to a non-existing value and the reference would refer to non-existing values. To solve this problem for you need to allocated memory inside the function.

The function receives the size of the memory to allocate and returns a pointer to the allocated memory	<code>int* p = pfuncv(5);</code>	<code>int* pfuncv(int size) { return new int[size]; }</code>
--	----------------------------------	--

or return a pointer to existing memory.

The function receives an offset to add to a pointer. A pointer is returned with the new calculated address.	<code>int *a = new int[5]; int* p = pfuncp(a,2);</code>	<code>int* pfuncp(int *a, int i) { a = a + i; return a; }</code>
---	---	--

returning a pointer to pointer

you need to allocate memory for the pointer to pointer inside the function

We allocate memory for an int then assign 10 to it. Next we allocate memory for an int* and assign the int memory to it pointed to by p. Finally we return the int* block pointed to by q.	<code>int **q = ppfunc();</code>	<pre>int** ppfunc() { int *p=new int; *p=10; int**q = new int*; *q = p; return q; }</pre>
--	----------------------------------	---

or return a pointer to existing memory.

The function receives a pointer to pointer that we return.	<code>int **t = ppfuncpp(q);</code>	<pre>int** ppfuncpp(int **q) { return q; }</pre>
--	-------------------------------------	--

return by reference

Return by reference is a little more trickier. you need to return a reference to an existing variable. The only way you can do this is for the function to receive a reference to an existing variable !

The function receives an offset to add to a pointer. A pointer is returned with the new calculated address.	<pre>int x = 5; int& r = rfuncr(x);</pre>	<pre>int& rfuncr(int &x) { x = x + 5; return x; }</pre>
---	---	---

returning a pointer from a reference

The function may receive a reference but you want to return a pointer ! Easy just get the address from the reference.

The function receives a reference but wants to return a pointer. A pointer is returned with the address of the reference	<pre>int x = 5; int* p = pfuncr(x);</pre>	<pre>int* pfuncr(int &x) { x = x + 5; return &x; }</pre>
--	---	--

returning a reference from a pointer

The function may receive a pointer but you want to return a reference ! Easy just get the value to what the pointer points to. Some text books call this **de-referencing**.

the function receives a pointer but wants to return a reference. A reference is returned with the value that the pointer points to.	<pre>int x = 5; int* p = &x; int& r = rfuncp(p);</pre>	<pre>int& rfuncp(int *p) { x = x + 5; return *p; }</pre>
---	--	--

receive and return chart

We can make up a chart that indicates how to return a receive parameter of different pass types.

received:	value	pointer	pointer to pointer	reference	reference pointer	reference pointer to pointer
return:						
value	return x;	return *p;	return **q;	return r;	return *rp;	return **rq;
pointer	-----	return p;	return *q;	return &r;	return rp;	return *rq;
pointer to pointer	-----	-----	return q;	-----	return &rp;	return rq;
reference	-----	return *p;	return **q;	return r;	return *rp;	return **rq;
pointer reference	-----	return p;	return *q;	return &r;	return rp;	return *rq;
pointer to pointer reference	-----	-----	-----	-----	-----	return rq;

LESSON 5 EXERCISE 5

Write a program demonstrating the above chart. Call your finished program L5ex5.

LESSON 5 QUESTION 1

A small program is presented next to demonstrate passing values, pointers and references to functions. You may want to keep a chart to record the values after each function call.

function call	y	x	p	*p	q	*q	**q
start							
funca							
funcb							
funcb							
funcc							
funcd							
funce							
funce							
funcf							
funcg							

```
// l5q1.cpp
// this program demonstrates pass by value pointer and references
```

```
// function prototypes
int funca ( int i ); // pass by value
int funcb ( int* i ); // pass by pointer
int funcc ( int** i ); // pass by pointer to pointer
int funcd ( int** i ); // pass by pointer to pointer
int funce ( int& i ); // pass by reference
int funcf ( int*& i ); // pass by pointer reference
int funcg ( int**& i ); // pass by pointer to pointer reference
```

```
// main function
int main()
```

```
{
    int x = 5; // declare integer variable x and assign 5 to it
    int y; // declare integer variable y
    int* p = &x; // p points to address of x
    int** q = &p; // q points to the address of p
```

```

y = funca ( x ); // pass by value

// what is the value of y ?
// what is the value of x ?
// what is the value of p, *p ?
// what is the value of q, *q, **q ?

y = funcb ( &x ); // pass by address

// what is the value of y ?
// what is the value of x ?
// what is the value of p, *p ?
// what is the value of q, *q, **q ?

y = funcb ( p ); // pass by address

// what is the value of y ?
// what is the value of x ?
// what is the value of p, *p ?
// what is the value of q, *q, **q ?

y = funcb ( &p ); // pass by pointer to pointer

// what is the value of y ?
// what is the value of x ?
// what is the value of p, *p ?
// what is the value of q, *q, **q ?

y = funcd ( q ); // pass by pointer to pointer

// what is the value of f y ?
// what is the value of x ?
// what is the value of p, *p ?
// what is the value of q, *q, **q ?

y = funcf ( x ); // pass by reference

// what is the value of f y ?
// what is the value of x ?
// what is the value of p, *p ?
// what is the value of q, *q, **q ?

y = funcf (*p); // pass by reference

// what is the value of y ?
// what is the value of x ?
// what is the value of p, *p ?
// what is the value of q, *q, **q ?

y = funcf (p); // pass by pointer reference
// what is the value of y ?
// what is the value of x ?
// what is the value of p, *p ?
// what is the value of q, *q, **q ?

```

```

y = funcg (q); // pass by pointer to pointer reference

// what is the value of y ?
// what is the value of x ?
// what is the value of p, *p ?
// what is the value of q, *q, **q ?
return y;

} // end of main

// functions

// pass by value return by value
int funca ( int i )

{
    // what is the value of i ?
    i++; // what is the value of i ?
    return i; // what does the function return ?
}

// pass by pointer return by value
int funcb ( int* i )

{
    // what is the value of i, *i ?
    (*i)++; // what is the value of *i ?
    return *i; // what does the function return ?
}

// pass by pointer to pointer return by value
int funcc ( int** i )

{
    // what is the value of i, *i, **i ?
    int* s = *i; // what is the value of s, *s ?
    (*s)++; // what is the value of *s ?
    return *s; // what does the function return ?
}

// pass by pointer return by value
int funcd ( int** i )

{
    // what is the value of i, *i, **i ?
    int* s = *i; // what is the value of s, *s ?
    s++; // what is the value of s ?
    *i = s; // what is the value of *i ?
    return *s; // what does the function return ?
}

// pass by reference
int funce ( int& i )

{
    // what does i refer to?
    ++i; // what is the value referred to by i ?
    return i; // what does the function return ?
}

```

```
// pass by pointer reference  
int funcf ( int*& i )
```

```
{           // what is does i, *i refer to ?  
++(*i);    // what is the value referred to by i, *i ?  
return *i;  // what does the function return ? Why *i ?  
}
```

```
// pass by pointer to pointer reference  
int funcg ( int**& i )
```

```
{           // what does i, *i, **i refer to ?  
++(**i);   // what is the value referred to by i, *i, **i ?  
return **i; // what does the function return ? Why **i ?  
}
```

C++ PROGRAMMERS GUIDE LESSON 6

File:	CppGuideL6.doc
Date Started:	July 12, 1998
Last Update:	Mar 23, 2002
Version:	4.0

LESSON 6 C++ USING FUNCTIONS

FUNCTIONS WITH STATIC VARIABLES

The variables in a function are temporary, they do not retain their value. This means once the function terminates the declared variable values assigned in the function are gone. When the function is called again the variables will be initialized with new values. If you want to keep the values of the variables in a function you have to make them **static**. A static variable will always retain its value. A static variable must be initialized to a value when it is declared. The static variable can only be initialized once, when it is declared.

```
static data_type variable_name = initialized_value;
```

```
static int count = 0; // static variable initialized to starting value of 0
```

A static variable is like a global variable except it is more elegant to use. You must be very careful using static variables. Each static variable name between functions must be unique. You should not have two functions with the same name static variable name. The compiler may think they are the same variable. The following program uses a function with a static variable.

```
// L6p1.cpp  
#include <iostream.h>
```

```
int add(int x); // prototype
```

```
void main()
```

```
{  
    cout << add(5) << endl;  
    cout << add(10) << endl;  
    cout << add(15) << endl;  
}
```

```
int add(int x)
```

```
{  
    static int count = 0; // static variable initialized to 0  
    count = count + x; // count retains its value  
    return count;  
}
```

Static variables always retain their values even after the function terminates and can only be initialized once when they are declared

5
15
30

Every time the function is called it retains its value as shown from the output:

LESSON 6 EXERCISE 1

Write a program that has a function that declares **static** and a **non-static** variables. Initialize both variables to zero, then increment both variables. Print out the values of the variables inside the function. In the main program call the function 5 times.

OVERLOADING FUNCTIONS

Functions may have the **same name** as long as they have **different parameter** data types. You may need a function to calculate the square of ints, longs, floats and doubles. To solve this problem, you just declare each function with the same name, but each with a different parameter data type. The compiler will figure out which one to call by matching the parameter data types. Unfortunately, function overloading does not work with return data types. Functions that have the same parameter types but different return data types are not considered overloaded functions. The following program demonstrates function overloading.

```
/* L6p2.cpp */
#include <iostream.h>

/* function prototypes */
int square(int x);
long square(long x);
float square(float x);
double square(double x);

/* main function */
void main()
{
    int x = 5;
    long lg = 5;
    float f = 5.0;
    double d = 5.0;
    cout << " the square of 5 is: " << square(x) << endl;
    cout << " the square of 5 is: " << square(lg) << endl;
    cout << " the square of 5 is: " << square(f) << endl;
    cout << " the square of 5 is: " << square(d) << endl;
}

int square(int x)    // overloaded functions
{
    return x * x;
}

long square(long x)
{
    return x * x;
}
```

**Overloading function means
same function name but
different parameter
types and order**

program output:

```
the square of 5 is: 25
the square of 5 is: 25
the square of 5 is: 25.0
the square of 5 is: 25.0
```

```
float square(float x)
{
    return x * x;
}

double square(double x)
{
    return x * x;
}
```

LESSON 6 EXERCISE 2

Type in the above program and try with different values. Inside each function print out the data type as well as the value.

using const modifier for function overloading

The **const** modifier can be used to distinguish two functions with the same name and same parameter types and order. The const modifier can be used on the return type and on a parameter. The following program demonstrates two functions with the same name and parameter list. Each can be distinguished by using the **const** modifier. Unfortunately `char *const s` and `const char *s` are considered the same and cannot be overloaded.

```
// L6p3.cpp
#include <iostream.h>

/* function prototypes */
const char* putMsg(char* s);
char* putMsg(const char* s);

void main()
{
    const char* msg1 = "constant";
    char* msg2 = "not constant";
    cout << putMsg(msg1) << endl;
    cout << putMsg(msg2) << endl;
}

const char* putMsg(char *s)
{
    return s;
}

char* putMsg(const char *s)
{
    return (char *) s;
}
```

Program Output

```
constant
not constant
```

LESSON 6 EXERCISE 3

Type in the above program and run. Try removing the const modifiers and see what happens.

INLINE FUNCTIONS

Inline functions force the compiler to insert code directly, rather than implementing a function call. Inline functions start with the keyword **inline** preceding the return data type.

```
inline int square(int x)
{
    return x * x;
}
```

Inline functions do not call the function but instead generated the code at the line the function call appears

Inline functions are just used to force short functions to have faster execution times by inserting the code directly rather than calling the function. There is a lot of execution overhead when calling functions. The following function call:

```
int x = square(5);
```

would be replaced with:

```
x = x * x;
```

The following program demonstrates an inline function.

```
// L6p4.cpp inline function to calculate the square of a number
#include <iostream.h>

// inline functions
inline int square(int x)
{
    return x * x;
}

/* main function */
void main()
{
    int x = square(5);
    cout << " the square of 5 is: " << x << endl;
}
```

program output:

the square of 5 is: 25

LESSON 6 EXERCISE 4

Type in the above program and try and run. make a separate prototype and function definition.

DEFAULT PARAMATERS

You can set the value of a parameter in a **function prototype** to a default value. If someone calls the function and forgets a parameter than the default parameter value is used instead.

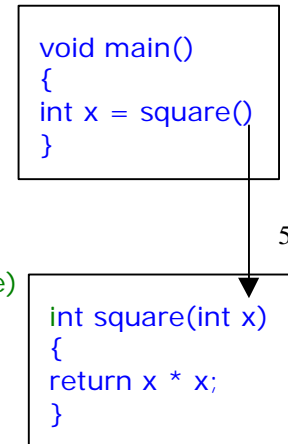
return_data_type function_name (paramter_data_type paramater_name = paramater_value);

```
int square(int x = 5);
```

You can now call the square method without an argument.

```
void main()
{
    int x = square(); // variable x gets square of 5 */
}

// square method (default parameter initialized in the prototype)
int square(int x)
{
    return x * x;
}
```



The compiler has substituted the default value of 5 for you automatically. When you use default parameters, they must start at the beginning of the parameter list and must be sequential with no gaps. In case of more than 1 parameter the first ones must all be default.

```
int many(int x=5, int y = 10, z);
```

LESSON 6 EXERCISE 5

Type in the above program and run. Add an additional default parameter. Add an additional ordinary parameter (not default) as the first parameter. Compile the program. What is wrong ?

POINTERS TO FUNCTIONS

You can even have a pointer to a function. Pointers to functions come in handy when you want to pass many different functions to a function, or want to call functions from an array or structure. To use a function pointer you first have to declare it, and then assign a function to it.

declaring a function pointer

Before you can declare a function pointer you need to understand how a function is declared. A function is declared as an identifier name with round brackets as follows:

function_name ();

```
f (); // declare a function
```

To get a function pointer all you need to do is put a star * on the function name.

**function_name ();*

```
*f (); // declare a function pointer
```

You need to put round brackets around the * and function name because the compiler will get confused.

```
(*function_name) ();
```

```
(*f) (); // declare a function pointer
```

A function may also have a parameter and a return type.

```
return_data_type function_name (paramatert_list );
```

```
double f (double); // declare a function with parameters and return value
```

To declare a function pointer you just include the function name, an argument list and a star to indicate a pointer.

```
return_data_type * function_name (paramater_list );
```

```
double *f(double); // declare a function pointer with parameters and return value
```

Brackets are needed around the star and function name to force precedence that the function name is a pointer.

```
return_data_type (* function_name) (paramatert_list );
```

```
double (*f)(double); // round brackets are needed around the function pointer
```

Declaring a function pointer is the same thing as declaring a pointer variable:

```
int *p ;
```

Except with a function pointer we have a return data type and a parameter list.

assigning a function to a function pointer

Once you declare your function pointer you initialize it with a function name.

```
function_pointer = function name;
```

```
f = square;
```

The starting address of the function is assigned to the function pointer

using a function pointer

Now that you have your function pointer pointing to a function you can call the function by using the function pointer.

```
(*function_pointer)(argument(s));
```

```
(*f)(5);
```

The * star is needed because you want to get the value of the function pointer which is a function ! Again brackets are needed so the compiler does not get confused.

Calling a function pointer is like getting the value from a pointer.

```
int x = *p;
```

Example program using a function pointer.

```
// l6p5.cpp
#include <iostream.h>
int square(int x);

void main()
{
    int (*fp)(int); // declare function pointer
    fp=square; // assign address of function square to function pointer
    int x = (*fp)(5); // call function square using pointer
    cout << "the square is: " << x << endl;
}

int square(int x)
{
    return x * x;
}
```

LESSON 6 EXERCISE 6

Type in the above program and run. Inside the square function make a function pointer that calls itself. That's right, initialize the function pointer to the square function. Before you call the function decrement the value of x. Exit the function when the value is zero. Inside the function print out the value of x.

passing a function pointer to another function

You first need to declare a function with a parameter that receives a function pointer. You must include extra parameters in your function for the function pointer arguments.

```
return_data_type function_name
(return_data_type(*function_pointer name)(argument_list) , passed_parameter_list );
```

```
double calculate(double (*f)(double),double x);
```

The function calculate has a parameter that is a function pointer. Now you can call a function with a function pointer as its arguments. You can use the function name

```
y = calculate(square,5);
```

or you can use a function pointer.

```
f = square;
y = calculate(f,5);
```

example using a function pointer parameter

Here is good example using function pointer parameter. We have a function called **calculate()** and it needs to evaluate many values using different functions, like a straight line, square, cube etc. Instead of writing three or more **calculate()** functions to calculate a line, square or cube we just need one function called **calculate()**. We will then pass a **function pointer** to the function telling which function to use. To declare a function pointer you just include the function name, an argument list and a star to indicate a pointer. Brackets are needed around the star and function name:

return_data_type (function_name) (argument_list)*

*double (*f)(double)*

To include a function pointer in a function declaration the function pointer arguments must be supplied separately in the function prototype after the function pointer parameter:

*return_data_type function_name
(return_data_type(*function_pointer name)(argument _list) , passed_parameter_list);*

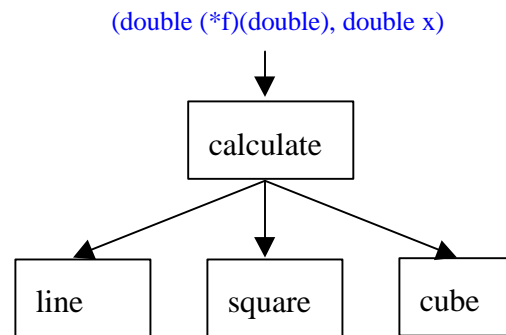
*double calculate(double (*f)(double),double x);*

We will pass function pointers to the calculate the functions to calculate a line, square or cube. Calculate gets a function pointer and a value to calculate. We need to declare function prototypes for our line, square and double functions, before we can use them.

double line(double x); // line function

double square(double x); // square function

double cube(double x); // double function



Now we just call calculate with the function that we want to use. line, square or cube.

double d = calculate(line,5); // call function to calculate line

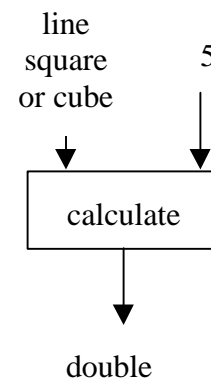
What is the value of d ?

d = calculate(square,5); // call function to calculate square

What is the value of d ?

d = calculate(cube,5); // call function to calculate cube

what is the value of d ?



Defining a function that has a function pointer

Here is the calculate function that has a function pointer as a parameter.

```
// receive function pointer and value
double calculate(double (*f)(double),double x)

{
    double y = (*f)(x); // call function pointer, pass parameter
    return y; // return calculated value
}
```

Notice the function call: `double y = (*f)(x);` where `(*f)` is calling the function and `(x)` is passing the parameter `x` to the function . `y` receives the result of the call to the calculation function.

An example program demonstrating pointers to functions.

```
// L6p6.cpp
#include <iostream.h>

// function prototypes
double calculate(double (*f)(double),double angle);
double line(double x);
double square(double x);
double cube(double x);

// main function
void main()

{
    double y = calculate(line,5);
    cout << "the line of 5 is " << y << endl;
    y = calculate(square,5);
    cout << "the square of 5 is " << y << endl;
    y = calculate(cube,5);
    cout << "the cube of 5 is " << y << endl;
}

// receive function pointer and value
double calculate(double (*f)(double),double x)

{
    double y = (*f)(x); // call function pointer
    return y;
}

// calculate line of x
double line(double x)

{
    return x;
}
```

```
// calculate square of x
double square(double x)
```

```
{
    return x*x;
}
```

```
// calculate cube of x
double cube(double x)
```

```
{
    return x*x*x;
}
```

LESSON 6 EXERCISE 7

In your main function make an array of function pointers. Assign the function line, square and cube to the array elements. Call the function Calculate using a function pointer from the array.

*f	---	line
*f	---	square
*f	---	cube

LESSON 6 EXERCISE 8

In your main function make an array of function pointers. Assign the function line, square and cube to the array elements. Pass the array to the calculate function and an index to one of the functions in the array. Call the appropriate function from the array depending on the index supplied.

*f	---	line
*f	---	square
*f	---	cube

LESSON 6 EXERCISE 9

Make a structure that holds a value and pointers to functions line, square and cube. Call the line, square and cube functions using the function pointers stored in the structure. Don't forget to initialize the structure member variables.

```
struct mypointers
{
    int x;

    double (*line)(double);
    double (*square)(double);
    double (*cube)(double);
};
```

PASSING ARRAYS TO FUNCTIONS

Arrays are passed by

1. array [] `int a[]`
2. pointer `int *p` `int *a`
3. reference `&` `int& a`

Passing 1 dimensional arrays to functions

You may also pass arrays to functions. Can you still remember how to declare an array ? An array is a block of consecutive declared variables all having the same data type.

`int a[5];` // declare a 1 dimensional array of 5 integer elements

a	int	int	int	int	int
---	-----	-----	-----	-----	-----

pass by array

Arrays declared as a variable has the memory **reserved** for the array at compile time.

`int a[5];` // declare 1-d array

You declare an array with an `array[]` parameter

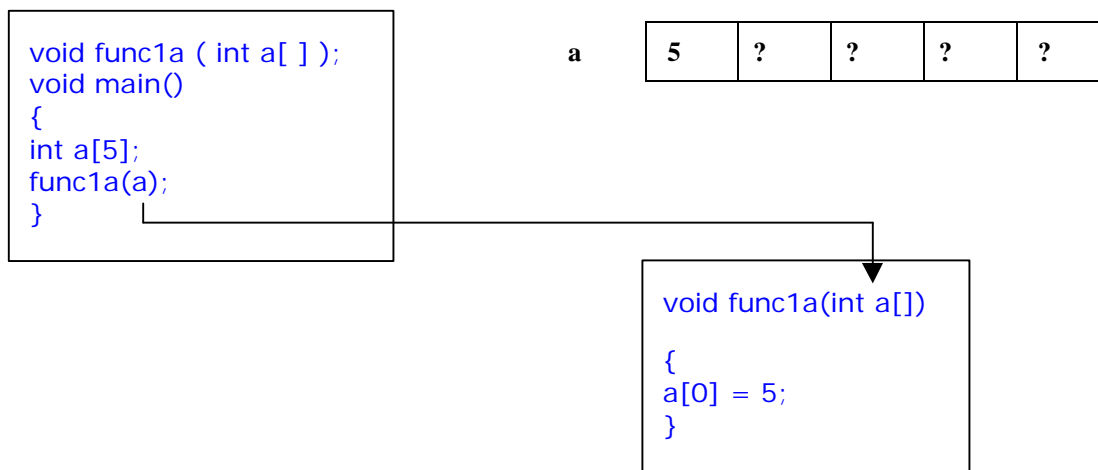
`void func1a (int a[]);` // a parameter of 1-d array of unknown length

The empty square brackets [] indicate an array unknown length. You may also include the array size if you want to.

`void func1a (int a[5]);` // a parameter of 1-d array of unknown length

When an array is passed to a function by array, only the name is used, because it identifies the array.

`func1a(a);` // call function and pass the 1 dimensional array to function by array



LESSON 6 EXERCISE 10

Type in the above program and initialize the array with values when you declare it. Print out values before entering function and inside function and after function is called.

pass by pointer

When you allocate an array at run time, the starting address of the array is assigned to a pointer

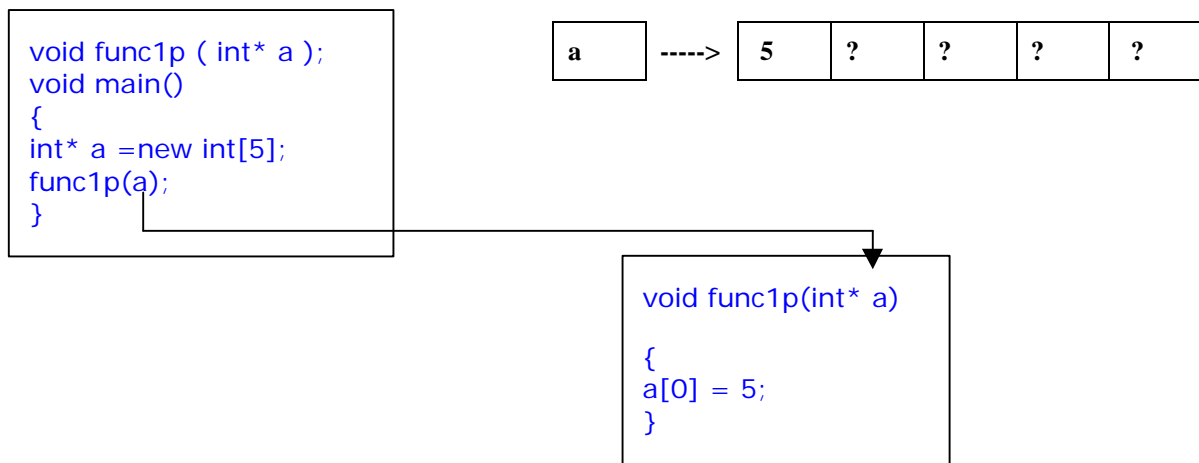
```
int* a =new int[5]; // allocate a 1 dimensional array
```

You then declare an function with an array pointer as a parameter

```
void func1p ( int* a ); // declare a function with a pointer to an array
```

You then call the function passing an array pointer argument.

```
func1p(a); // call function and pass the 1 dimensional array to function by pointer
```



LESSON 6 EXERCISE 11

Type in the above program and initialize the array with values when you declare it. Print out values before entering function and inside function and after function is called.

pass by reference

You may also pass an array by **reference**. Before you can pass an array by reference you need to make your own one dimensional array data type using **typedef**.

```
typedef int ta[5]; // define a 1 dimensional array user data type
```

You can then declare an array using the data type

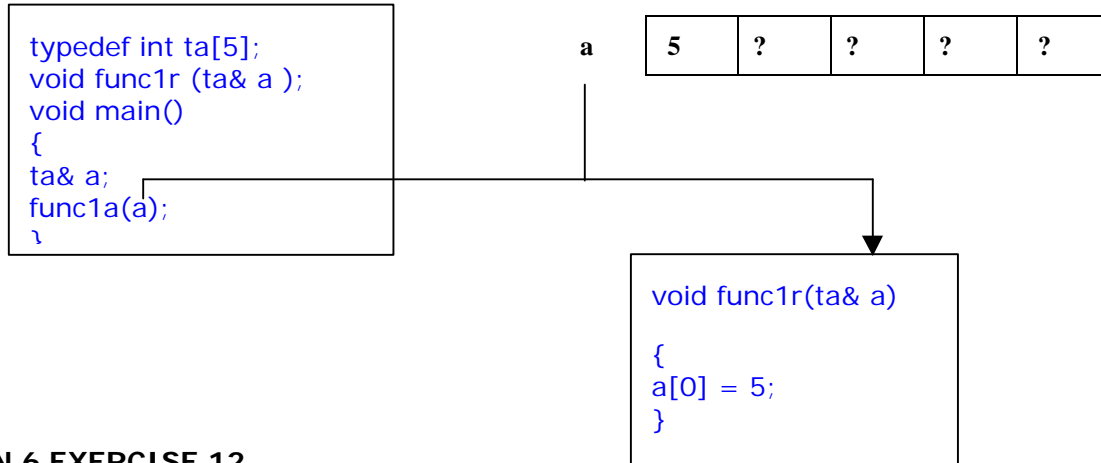
```
ta a; // define a 1 dimensional array ta mean int[5]
```

The 1-d array is then passed to a function having an array reference parameter.

```
void func1r ( ta& a ); // declare a function with pass by reference parameter
```


When an array is passed to a function, only the name is used, because an array name represents an address.

`func1r(a);` // pass the 1 dimensional array to function pass by reference



LESSON 6 EXERCISE 12

Type in the above program and initialize the array with values when you declare it. Print out values before entering function and inside function and after function is called.

example program passing 1 dimensional arrays to functions

The following program illustrates different ways of passing arrays to functions.

```

/* L6p7.cpp */
#include <iostream.h>
typedef int ta[5];           // define a 1 dimensional array user data type

// function prototypes
void func1a ( int a[ ] );    // pass by array
void func1p ( int* a );     // pass by pointer
void func1r ( ta& a );      // pass by reference

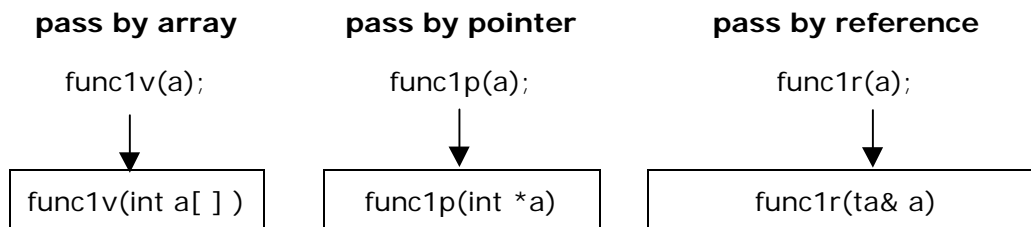
// passing arrays to functions
void main()
{
    int a[5]={1,2,3,4,5};    // declare and initialize a 1 dimensional array of 5 columns
    func1p(a);               // pass the 1 dimensional array to function pass by pointer
    func1a(a);               // pass the 1 dimensional array to function pass by array
    func1r(a);               // pass the 1 dimensional array to function pass by reference
}

// pass a one dimensional array of unknown length
void func1a ( int a[ ] )
{
    cout << a[0] << endl;    // print out 1 element of array
}
    
```

```
// pass a one dimensional array by address of unknown length
void func1p ( int* a )
{
    cout << a[0] << endl; // print out 1 element of array
}

// pass a one dimensional array by reference of unknown length
void func1r ( ta& a )
{
    cout << a[0] << endl; // print out 1 element of array
}
```

The following chart summarizes passing a 1 dimensional array to functions.



LESSON 6 EXERCISE 13

Write a program that has a main function that declares an array of 5 integers pre-initialized to your favorite values. Pass the array and an index to a function called `change(int a[], int index)`. Add to the array at the specified index the array value and the array index. Return the address of the array at the specified index. Print out the array value at the index before and the return value to the screen using **cout**. Call your program L6ex4.cpp. You have to use a pointer to get the return value from the function.

LESSON 6 EXERCISE 14

Repeat previous exercises but use pass by reference and return by reference.

Passing 2 dimensional arrays to functions

Can you still remember what a 2 dimensional array is ?

`int b[3][4];` // 2 dimensional array of 2 rows and 5 columns
 ↗ ↖
 rows columns

b	int	int	int	int	int
	int	int	int	int	int
	int	int	int	int	int

$b[i][j] = *(b + (i * \text{row length}) + j)$

pass 2d array by array

2 dimensional arrays declares as a variable have their memory reserve in compile time

```
int b[3][4]; // declare 2 dimensional array
```

To pass a **declared** 2 dimensional arrays by array to a function you must specify the number of columns or the compiler will report an error.

```
void func2bb ( int b[3][4 ] ); // function prototype array of known column length , but known rows
void func2b ( int b[ ][4 ] ); // function prototype array of known column length , but unknown rows
```

When a 2 dimensional array is passed to a function only the name is used, because an array name identifies the array.

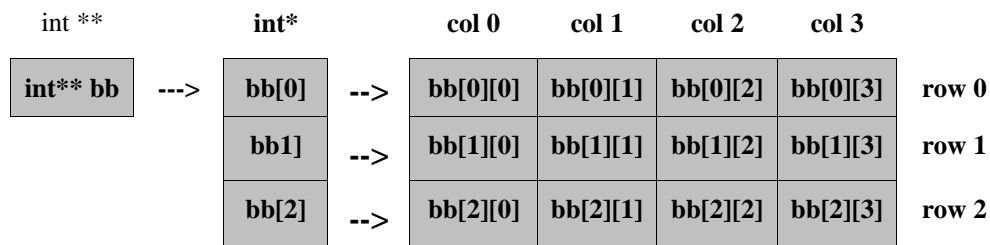
```
func2b(b); // pass a declared 2 dimensional array to function
```

pass 2d array by pointer

If you **allocate** memory for an array

```
int** bb = new int*[3]; // allocate an array of integer pointers

bb[0] = new int[4]; // allocate an array of integers
bb[1] = new int[4]; // allocate an array of integers
bb[2] = new int[4]; // allocate an array of integers
```



A function receives the 2 dimensional allocated array through a pointer to pointer contains the address of the array of pointers.

```
void func2pp ( int** bb); // function prototype to pass an allocated 2 dimensional array
```

When a 2 dimensional allocated array is passed to a function, the pointer to pointer contains the address of the array of pointers.

```
fubc2pp(bb); // pass a two dimensional allocated array to a function
```

pass 2d array by reference

To pass a 2 dimensional reserved array by **reference** you need to make your own 2d **array data type**.

```
typedef int m[3][4]; // define a 2 dimensional array user data type
```

You declare your array using the array data type

```
m b; // declare a 2 dimensional array
```

The 2-d array is then received the function through a reference parameter

```
void func2r ( m& b ); // prototype to pass a two dimensional array reference
```

When a 2 dimensional array is passed to a function only the name is used, because an array name references the array.

```
func2r(b); // pass a declared 2 dimensional array by reference to a function
```

example program passing 2-d arrays to functions

You must be careful in passing reserved and allocated arrays to functions. You cannot pass a reserved 2-d array as a pointer. You cannot pass an allocated array to a function by array[][]. The compiler calculates the rows and columns differently for each. Reserved arrays are passed as reference. The following program demonstrates the passing 2-dimensional arrays to functions

```
/* L6p8.cpp */
#include <iostream.h>

typedef int m[3][4]; // define a 2 dimensional array user data type

// function prototypes
void func2bb ( int b[3 ][4 ] );
void func2b ( int b[ ][4] );
void func2r ( m& b );
void func2pp ( int** b);

void main()
{
    // declare a 2 dimensional array of 2 rows and 5 columns
    int b[3][4]={ { 1,2,3,4},{ 5,6,7,8},{ 9,10,11,12}}; // reserved
    int** pp = new int*[3]; // allocate an array of integer pointers
    pp[0] = new int[4]; // allocate an array of integers
    pp[1] = new int[4]; // allocate an array of integers
    pp[2] = new int[4]; // allocate an array of integers
    pp[0][0]=1; // initialize first value

    func2b(b); // pass the 2 dimensional array to function
    func2bb(b); // pass the 2 dimensional array to function
    func2pp(pp); // pass the 2 dimensional allocated array to function
    func2r(b); // pass the 2 dimensional array data type to function

    delete[] pp[0]; // delete row
    delete[] pp[1]; // delete row
    delete[] pp; // delete array of row pointers
}
```

**You cannot delete
reserved memory**

```
// array of known column length , but unknown rows
void func2bb ( int b[3 ][4 ] )

{
    cout << b[0][0] << endl; // print out 1 element of two dimensional array
}

// array of known column length , but unknown rows
void func2b ( int b[ ][4 ] )

{
    cout << b[0][0] << endl; // print out 1 element of two dimensional array
}

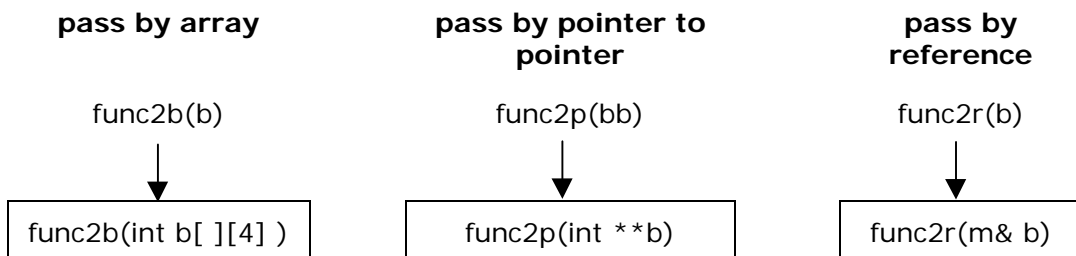
// pass a two dimensional array reference
void func2r ( m& b )

{
    cout << b[0][0] << endl; // print out 1 element of two dimensional array
}

// pass a two dimensional array of unknown rows and unknown columns
void func2pp ( int** pp)

{
    cout << pp[0][0] << endl; // print out 1 element of two dimensional array
}
```

The following chart summarizes passing a 2 dimensional array to functions



LESSON 6 EXERCISE 15

Write a program that has a main function that declares an array of 2 by 3 integers pre-initialized to your favorite values. Pass the array and a row and column index to a function called `change()`. Add to the array at the specified row and column index the array value and the array row and column index. Return the address of the array at the specified row and column index. Print out the value of the array at the index before and the return value to the screen using **cout**. Call your program `L6ex6.cpp`. You have to use a pointer to get the return value from the function.

LESSON 6 EXERCISE 16

Repeat previous exercise but use pass by reference and return by reference.

passing individual values from an array to a function

You specify a particular element in a array to pass to a function.

pass type	prototype	1 dimensional	2 dimensional
pass by value	func(int x)	func(a[4]);	func(b[1][3]);
pass by pointer	func(int *p)	func(&a[4]);	func(&b[1][3]);
pass by reference	func(int& x)	func(a[4]);	func(mb[1][3]);

Why do we use & for pass by pointer ?

passing individual rows of an 2 dimensional array to a function

You must specify a particular row starting address in a array to pass a row to a function. The arrays can be one dimensional or 2 dimensional.

pass type	prototype	2 dimensional
pass by array	func(int a[])	func(b[1]);
pass by pointer	func(int *a)	func(&b[1]);
pass by reference	func(int& ra)	func(mb[1]);

specifies which row of 2 dimensional array

Why do we use & for pass by pointer ?

Returning 1 dimensional arrays to functions

You can return arrays as a pointer or as a reference from a function. An 1-D reserved array is returned from a function as an array pointer. Why ? You cannot return an array by array value. Do you know why ? (because you need to return the address as a pointer or reference)

return type	prototype	using
return by pointer	int* funcap (int a[]);	int* a = funcap(a);
return by reference	v& funcar(int a[]);	int* a = funcar(a);

returning 2 dimensional arrays to functions

You return 2-D arrays to the calling function as a pointer to pointer . A 2-D reserved array is returned as a array pointer.

return type	prototype	using
return by pointer	int** funcbp (int b[][]);	int** b = funcbp(b);
return by reference	m& funcbp(int m&)	int **b = funcbr(mb);

LESSON 6 EXERCISE 17

Declare an array of 5 integers. Ask the uses to enter 5 of their favorite numbers from the keyboard (just use 5 **cin** statements). Write a function called swap that will receive 2 integer pointers. Ask the user to type in 2 numbers where each number is between 0 and 4. Call swap to exchange the integer pointers. When everything is swapped, go through the array and print out all the numbers.

LESSON 6 EXERCISE 18

Declare an array of 5 integer pointers. Ask the uses to enter 5 of their favorite numbers from the keyboard (just use 5 **cin** statements). Allocate memory for each number entered and assign the address of the allocated memory sequentially to each pointer in the array. Write a function called swap that will receive 2 address. Each address will point to the address of an array element. Ask the user to type in 2 numbers For each number get a pointer to the element in the array where the message corresponding to the array index specified by the entered number. Call swap to exchange the integer pointers. Go through the array and print out all the numbers. Remember all you need to do is swap the pointers to swap the numbers.

PASSING STRUCTURES TO FUNCTIONS

You may also pass structures to functions. Can you still remember what a structure is ?

```
struct record
{
    int x;
    long y;
    char c;
};
```

x
y
c

What is the difference between an array and a structure. An array is a collection of the same data types accessed by an index. A structure is a collection of many different data types accessed by a name. Structures are passed by value, pointer or by reference. A structure is automatically a user data type when the structure is defined using the struct keyword and the structure definition name. When you pass a record by value, you cannot change the value of the original record in the called function. When you pass by pointer or reference, you can change the values stored in the original record in the called function.

Structures may be passed by:

- value
- pointer
- reference

pass structure by value

The original structure values do not change because the function receives a copy of the structure.

```

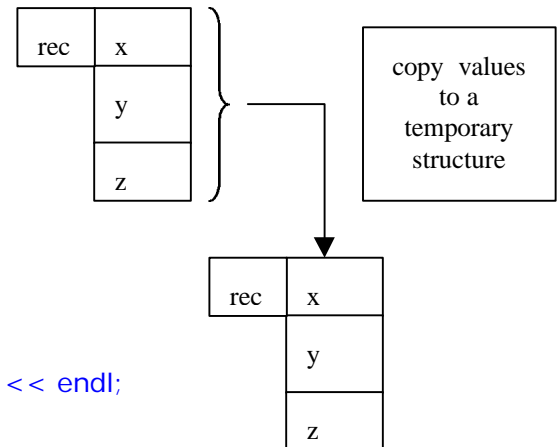
/* L6p9.cpp pass structure by value */
#include <iostream.h>

/* define a structure */
struct record
{
    int x;
    float f;
    char c;
};
/* prototypes */
void funcv(record rec);

/* main function */
void main()
{
    // declare and initialize record as a variable
    record rec = {5,10.5,'A'};
    // pass by value,
    funcv(rec);
}

// pass by value and print out values stored in record
void funcv(record rec)
{
    cout << "pass by value: " << " ";
    cout << rec.x << " " << rec.f << " " << rec.c << " " << endl;
}

```



pass structure by pointer

The original structure values change because the function receives the indirect address of the structure.

```

/* L6p10.cpp pass structure by value */
#include <iostream.h>

/* define a structure */
struct record
{
    int x;
    float f;
    char c;
};

```

The pointer in main and the pointer in the function funcp both point to the same

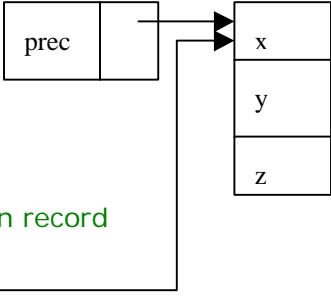

```

void funcp(record* rec);

/* main function */
void main()
{
    // declare and initialize record as a variable
    record* prec = new record;
    prec->x = 5; prec->f=10.5; prec->c='A';
    // pass by value,
    funcp(prec);
}

// pass by value and print out values stored in record
void funcp(record* prec)
{
    cout << "pass by pointer: " << " ";
    cout << prec->x << " " << prec->f << " " << prec->c << " " << endl;
}

```



The diagram illustrates the memory layout for the first example. A variable named 'prec' is shown in a box, with an arrow pointing from it to a record structure. The record structure is a vertical stack of three boxes labeled 'x', 'y', and 'z'.

pass by reference

The original structure values change because the function receives the direct address of the structure.

```

/* L6p11.cpp pass structure by value */
#include <iostream.h>

/* define a structure */
struct record
{
    int x;
    float f;
    char c;
};

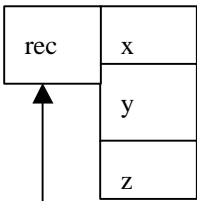
void funcv(record& rec);

/* main function */
void main()
{
    // declare and initialize record as a variable
    record rec = {5,10.5,'A'};

    // pass by value,
    funcv(rec);
}

// pass by value and print out values stored in record
void funcv(record& rec)
{
    cout << "pass by value: " << " ";
    cout << rec.x << " " << rec.f << " " << rec.c << " " << endl;
}

```



The diagram illustrates the memory layout for the second example. A variable named 'rec' is shown in a box, with an arrow pointing from it to a record structure. The record structure is a vertical stack of three boxes labeled 'x', 'y', and 'z'.

The reference
rec in the
function refer
to the same
structure.

LESSON 6 EXERCISE 19

Define a structure and write a function called `getValue()` that receives a structure and initializes the variables using `cin`. In the main function declare and initialize the structure with some default values. Print out the values before and after the function is called. Write functions using pass by value, pass by pointer and pass by reference. Call your program L6ex10.cpp. You will discover something interesting!

returning structures from functions

You can also return structures by value, pointer or reference from a function.

prototype	comment	example
<code>record funcvv(record rec);</code>	pass by value return by value	<code>record rec = funcvv(rec);</code>
<code>record* funcpp(record * prec);</code>	pass by pointer return by pointer	<code>record* prec = funcpp(prec);</code>
<code>record& funcrr (record& rrec);</code>	pass by reference return by reference	<code>record& rec = funcrr(rec);</code>

LESSON 6 EXERCISE 20

Define a structure with 3 members of int, float and char data type. In your main function declare an array of structures as a **variable**. **Declare a pointer** variable to your structure. Pick an index of the array of structures and initialize the structure elements at that array index with your favorite values. Define a function called `update()` to take an array of structures and an index to a structure of the array. The function should return a pointer to the structure identified by the index. Assign the return value from the function to your structure pointer. Print out the values of the structure using the structure pointer.

LESSON 6 EXERCISE 21

Declare an array of structure pointers. Allocate two structures Ask the uses to initialize the structures from the keyboard . Go through the array and print out all the structure contents. Write a function called swap that will receive 2 address Each address will point to the address of an array element. Call swap to exchange the structure pointers. When everything is swapped, go through the array and print out all the structure contents.

PASS AN ARRAY OF STRUCTURES TO A FUNCTION

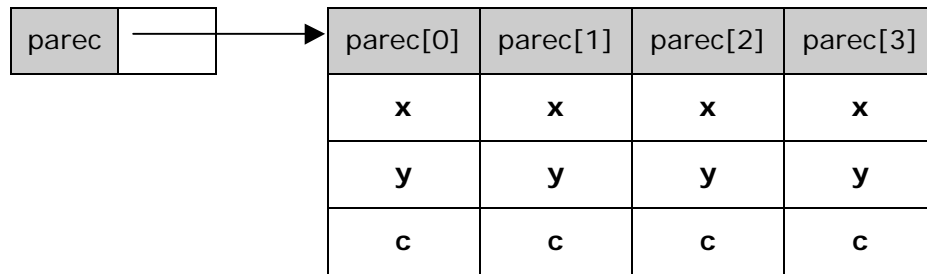
You can pass an array of structures by an array, pointer or by reference. Memory for an array of structures can be reserved at compile time or allocated with a pointer at run time. You declare a reserved array of structures as:

```
record arec[4] = {{5,10.5,'A'}, {7,12.5,'B'}, {11,15.5,'C'}, {17,23.4,'D'}};
```

arec	arec[0]	arec[1]	arec[2]	arec[3]
	x	x	x	x
	y	y	y	y
	c	c	c	c

To allocate an array at run time using a pointer

```
record * parec = new record[4];
```



For reserved and allocated arrays of structures. You access members of the structure with the dot "." operator. We use the dot operator because we are accessing memory contents. `arec[1]` means `*(arec+1)` and `parec[1]` means `*(parec+1)` both point to memory blocks so we just need the dot operator to access memory variables.

```
arec[1].x = 5;          v = arec[1].x;
parec[1].x = 5;        v = parec[1].x;
```

pass an array to a structure by array:

```
/* L6p12.cpp pass structure by array */
#include <iostream.h>

/* define a structure */
struct record
{
    int x;
    float f;
    char c;
};

void func1v(record arec[]);

/* main function */
void main()
{
    // declare and initialize record as a variable
    record arec[4] = {{5,10.5,'A'}, {7,12.5,'B'}, {11,15.5,'C'}, {17,23.4,'D'}};
    // pass by array,
    func1v(arec);
}

// pass by value and print out values stored in record
void func1v(record arec[])
{
    cout << "pass by value: " << " ";
    cout << arec[0].x << " " << arec[0].f << " " << arec[0].c << " " << endl;
}
```

pass an array to a structure by pointer

```

/* L6p13.cpp pass structure by value */
#include <iostream.h>

/* define a structure */
struct record
{
int x;
float f;
char c;
};

void func1p(record* parec);

/* main function */
void main()
{
// declare and initialize record as a variable
record * parec = new record[4];

parec[0].x=5; parec[0].f=10.5;parec[0].c='A';
parec[1].x=7; parec[1].f=12.5;parec[1].c='B';

// pass by pointer
func1p(rec);
}

// pass by pointer and print out values stored in record
void func1p(record* parec)
{
cout << "pass by value: " << " ";
cout << parec[0].x << " " << parec[0].f << " " << parec[0].c << " " << endl;
}

```

pass an array to a structure by reference:

```

/* L6p14.cpp pass structure by value */
#include <iostream.h>

/* define a structure */
struct record
{
int x;
float f;
char c;
};
typedef record ma[4]; // make array data type
void func1r(ma& arec); // pass by reference

```

```

/* main function */
void main()

{
    // declare and initialize record as a variable
    record arec[4] = {{ 5,10.5,'A'},{ 7,12.5,'B'},{ 11,15.5,'C'},{ 17,23.4,'D'}};
    // pass by value,
    func1r(arec);
}
// pass by value and print out values stored in record
void func1r(record arec)
{
    cout << "pass by value: " << " ";
    cout << arec[0].x << " " << arec[0].f << " " << arec[0].c << " " << endl;
}

```

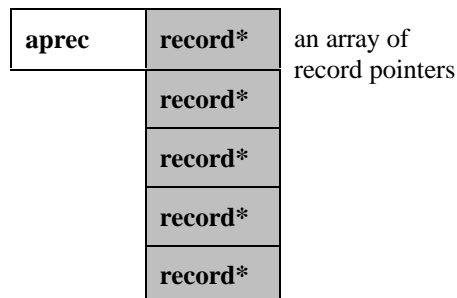
LESSON 6 EXERCISE 22

Define a structure and write a function called `getValue()` that receives a structure and initializes the variables using `cin`. In the main function declare an array of structures and initialize one of the structures with some default values. Print out the values before and after the function is called. Write functions using pass by value, pass by pointer and pass by reference. You will discover something interesting!

PASS AN ARRAY OF STRUCTURES POINTERS TO A FUNCTION

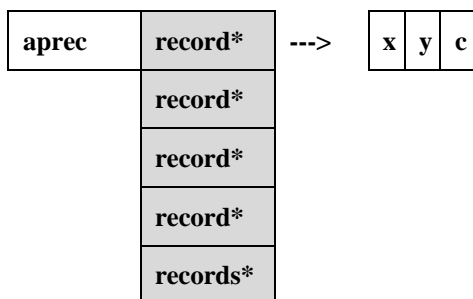
You can pass an array of structures by an array or by pointer or by reference. Memory for an array of structures can be reserved at compile time or allocated with a pointer at run time. you declare a reserved array of structures pointers as

```
record* aprec*[2];
```



Once you get your structure pointers you must allocate memory for structures and assign the starting address to one of your structure pointers in your array of structure pointers.

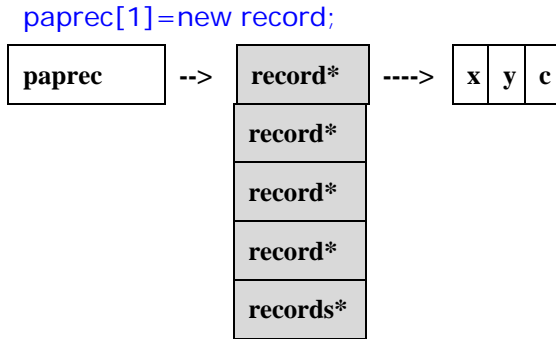
```
aprec[1]=new record;
```



You allocate an array of structure pointers at run time using a pointer to pointer:

```
record** paparec = new record*[2];
```

Once you get your structure pointers you must allocate memory for structures and assign the starting address to one of your structure pointers in your array of structure pointers.



You access individual members of the structure with the arrow "->" operator regardless if it's a reserved or allocated array of structure pointers. You need the arrow because the array contents are pointers. `aprec[1].x` really means `*(*(aprec+1)).x`

```
aprec[1]->x = 5;
paprec[1]->x = 5;
```

pass an array of structure pointers by array

```

/* L6p15.cpp pass structure by array */
#include <iostream.h>

/* define a structure */
struct record
{
    int x;
    float f;
    char c;
};

void func1pv(record* arec[]);

/* main function */
void main()
{
    // declare and initialize record as a variable
    record* aprec[4] ;
    aprec[0]= new record;
    aprec[0]->x=5; aprec[0]->f=10.5; aprec[0]->c='A';
    aprec[1]=new record;
    aprec[1]->x=7; aprec[1]->f=12.5; aprec[1]->c='B';
    // pass by array
    func1pv(aprec);
}
  
```

```

// pass by value and print out values stored in record
void func1pv(record* arec[])
{
    cout << "pass by value: " << " ";
    cout << arec[0]->x << " " << arec[0]->f << " " << arec[0]->c << " " << endl;
}

```

pass an array of structures to a function by pointer

```

/* L6p16.cpp pass structure by value */
#include <iostream.h>

/* define a structure */
struct record
{
    int x;
    float f;
    char c;
};

void func1pap(record** rec);

/* main function */
void main()
{
    // declare and initialize record as a variable
    record ** paprec = new record*[2];

    paprec[0] = new record;
    paprec[0]->x=5; paprec[0]->f=10.5; paprec[0]->c='A';

    paprec[1]=new record;
    paprec[1]->x=7; paprec[1]->f=12.5; paprec[1]->c='B';

    // pass by pointer
    func1pap(paprec);
}

// pass by pointer and print out values stored in record
void func1pap(record** paprec)
{
    cout << "pass by value: " << " ";
    cout << paprec[0]->x << " " << paprec[0]->f << " " << paprec[0]->c << endl;
}

```

pass an array of structures to a function by reference

```

/* L6p17.cpp pass structure by value */
#include <iostream.h>

/* define a structure */
struct record
{
    int x;
    float f;
    char c;
};

void func1ppr(record**& paprec);

/* main function */
void main()
{
    // declare and initialize record as a variable
    record ** paprec = new record*[4];
    paprec[0] = new record;
    paprec[0]->x=5; paprec[0]->f=10.5; paprec[0]->c='A';
    paprec[1]=new record;
    paprec[1]->x=7; paprec[1]->f=12.5; paprec[1]->c='B';

    // pass by pointer
    func1ppr(paprec);
}

// pass by pointer and print out values stored in record
void func1ppr(record**& paprec)
{
    cout << "pass by value: " << " ";
    cout << paprec[0]->x << " " << paprec[0]->f << " " << paprec[0]->c<< endl;
}

```

LESSON 6 EXERCISE 23

Define a structure and write a function called `getValue()` that receives a structure and initializes the variables using **cin**. In the main function declare an array of structure pointers . Allocate and initialize a structure with some default values. Print out the values before and after the function is called. Write functions using pass by value, pass by pointer and pass by reference. You will discover something interesting!

LESSON 6 EXERCISE 24

Write a program that declares an array of structures and an array of structure pointers. Have the array of structure pointers point to the structures in the array of structures. Pick one of the structures in the array pass to a function using the corresponding structure pointer. Change the values of the structure in the function. Print out the structure value before and after calling the function. Call your program L6ex13.cpp

LESSON 6 EXERCISE 25

Pass a 2 dimensional array of records by value, pointer and reference. Call your program L6ex14.cpp

Passing individual members from an array of structures to a function

You must specify the index if the array structure and select the structure member with the dot "." operator to pass an individual element of a structure to a function.

pass method	prototype		structure	array of structure	array of structure pointers
value	<code>funcv(int x);</code>		<code>funcv(rec.x);</code>	<code>funcv(rec[1].x);</code>	<code>funcv(rec[1]->x)</code>
pointer	<code>funcp(int *p);</code>		<code>funcp(&rec.x);</code>	<code>funcp(&rec[1].x);</code>	<code>funcvp(&rec[1]->x)</code>
reference	<code>funcr(int& x);</code>		<code>funcr(rec.x);</code>	<code>funcr(rec[1].x);</code>	<code>funcr(rec[1]->x)</code>

LESSON 6 EXERCISE 26

Write a program that passes individual values to functions demonstrating the above chart.

accessing individual members from an array of structures

You access individual structures in the array with the desired array index enclosed in square brackets. [] .

pass type	prototype		declared array of structure	allocated array of structures	declared array of structure pointers	allocated array of structure pointers
value:	<code>funcv(record rec)</code>		<code>arec[1]</code>	<code>parec[1]</code>	<code>*aprec[1]</code>	<code>*paprec[1]</code>
pointer:	<code>funcv(record* rec)</code>		<code>&arec[1]</code>	<code>&parec[1]</code>	<code>aprec[1]</code>	<code>paprec[1]</code>
reference:	<code>funcr(record& rec)</code>		<code>arec[1]</code>	<code>parec[1]</code>	<code>*aprec[1]</code>	<code>*paprec[1]</code>

Notice that for pass by pointer for arrays of structures you need an address by using the & operator, whereas arrays of structure pointers do not. When you pass an array of structures by value or reference you need to use the star "*" operator to get the value from the structure pointer.

LESSON 6 EXERCISE 27

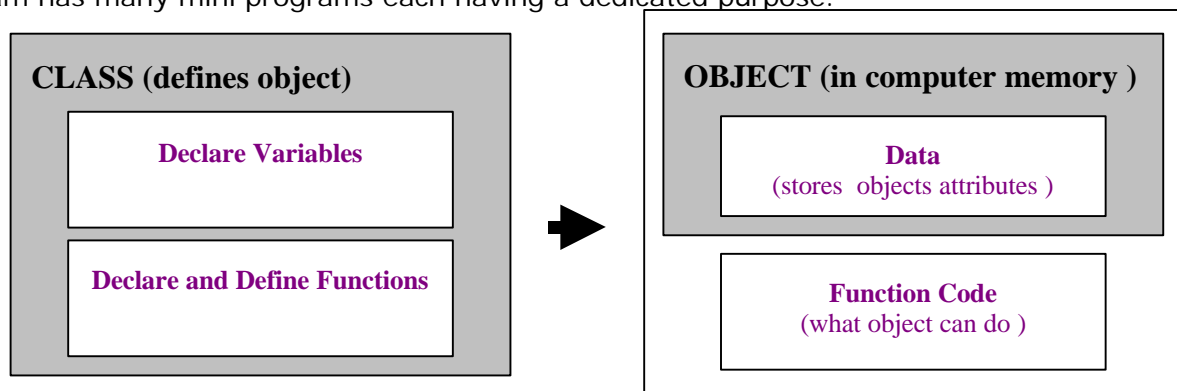
Write a program that passes structures to functions demonstrating the above chart.

C++ PROGRAMMERS GUIDE LESSON 7

File:	CppGuideL7.doc
Date Started:	July 12, 1998
Last Update:	Mar 24, 2002
Version:	4.0

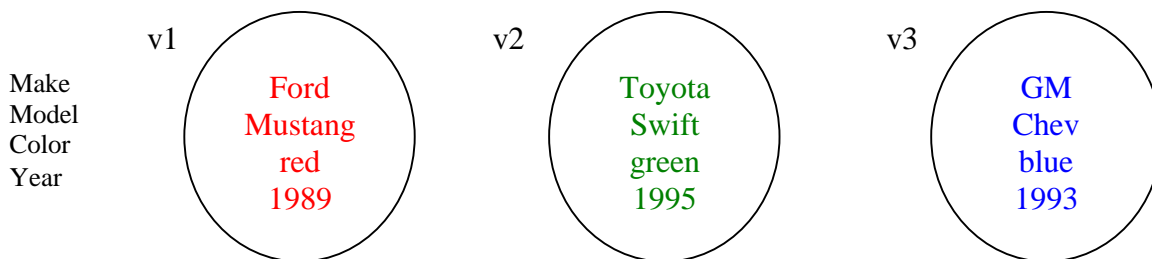
LESSON 7 OBJECT ORIENTED PROGRAMMING AND CLASSES

Object Oriented programming allows a programming language to represent everyday objects. An example of an object is a **vehicle**. Every object has **attributes** and **actions**. The attributes represent the physical properties of an object. The action represents what the object does. For example a **Vehicle** has **attributes** make, model, color and year. A vehicle also has **action** they go fast, slow, stop etc. A programming language uses **data** to represent attributes and programming **statements** to perform **actions**. Data is stored in a computer memory and the **location** where the data is stored is represented by a **variable name**. **Programming statements** are grouped together into a **function**. A function allows the program to **execute** these statements sequentially represented by the function name. A **class** is the **definition** for an object having **data** represented by a list of variables and **code** represented by a collection of functions. Think that a class is a **mini-program** that has its own variables and functions to do operations on the data. The functions are associated with the object. Each function can act on many objects. Think that an object oriented program has many mini programs each having a dedicated purpose.



You need to define an object before you can use it. A **class** is used to define an object. A class definition is like the recipe to bake a cake. You need a recipe before you can bake a cake. A class defines the **variables** and **functions** needed by the object. Just like a cake needs **ingredients** and **instructions** to bake the cake. Variables are used to store and retrieve data. After an object is **defined**, it has to be **created** before it can be used. Just like a cake needs to be made, before it can be eaten. When a class is created in computer memory, it is known as an **object**. When an object is created, it is known as an **instance** of a class. You can make many objects from one class definition. Just like you can make many cakes from 1 recipe. The variables defined in the class become data in the object. The functions in the class do not belong to the object but are said to belong to the class. This is because we only need one function. Each function can access the data of many objects. You need to tell the compiler, which object you want the function to act on. The class is the basic definition that objects are created from.

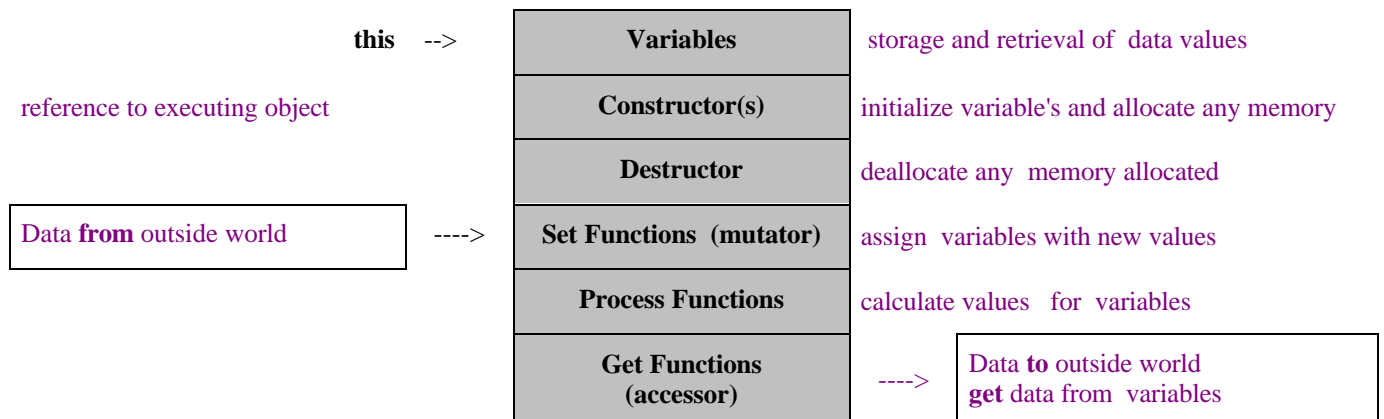
Each instance will have its own properties. Just like each cake can have its own properties. Chocolate cake, vanilla cake etc. This is the highlight of object oriented programming. A class provides a **common definition** that objects are created (**instantiated**) from. Each object created will have distinguishable properties. When you define a class, you list the data variables and functions the class needs. When you create an instance of a class, the object will get distinguishable properties when you initialize its data member variables. The functions are used to store and retrieve data from the object so that other objects can use and give values from this object. For example from the Vehicle class, the Vehicle class may be defined with make, model, color and year. The Vehicle class may be defined with functions to initialize make, model, color and year and to initialize and retrieve these values. Every time you create an object from the Vehicle class definition, you will get a separate Vehicle object. Each Vehicle object would have a different make, model, color and year. Each Vehicle object is distinguishable from other Vehicle objects because each Vehicle object has unique properties and identifying instance name. An object is memory for the variables defined in the class definition. Here are some Vehicle objects instantiated from class Vehicle:



You can make many objects from one class definition, just like you can make many cakes from one recipe. By organizing your program into classes, where each class represents a particular object to do a specific task, your programs will be highly organized and efficient. The object oriented programming approach will let you write complicated programming tasks very easy and fast.

Class Components

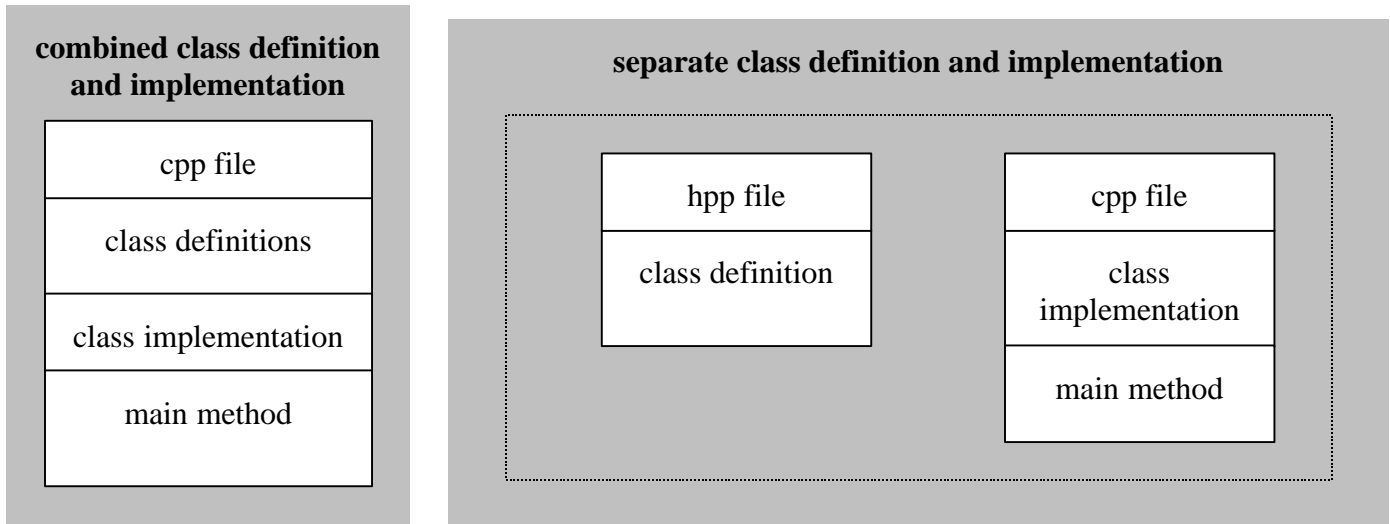
Classes are defined with **variables**, **constructors**, **destructors**, **set functions**, **process functions** and **get functions**.



We will now explain and discuss all the components that make up a class. The **this** pointer is an automatic pointer supplied by the compiler to point to the currently executing object. When you are defining a class you can use the **this** pointer just like a pointer to the variables and methods of the class you are presently defining.

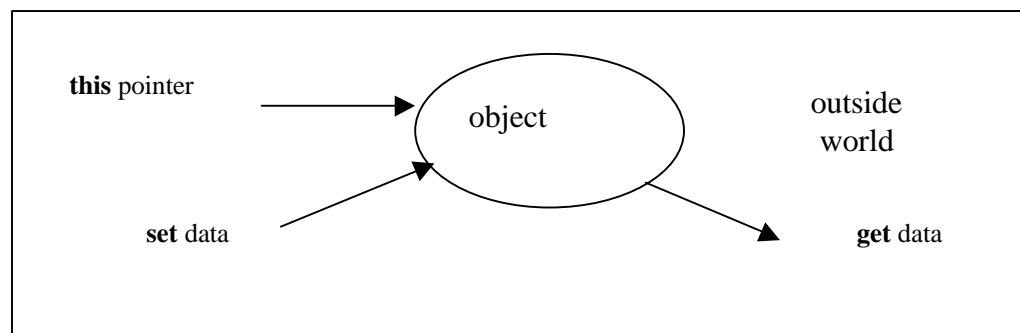
class definition and example

The following is a class definition with example code. Class definitions are either at the top of your program or in a definition header file with the extension ".h" or ".hpp". All C++ programs need a main method to run.



We use the Vehicle class as an example. The Vehicle class holds the color and year of the Vehicle in its variables. The **constructors** have parameters to initialize the variables to user defined values. The Vehicle class has functions to get information and receive information. The Vehicle class also uses another class called the **String class**. The String class is used to avoid using `char *` and `char[]` for character strings. Unfortunately it is not built into C++ language and must be supplied separately. Some compilers will provide their own string class for you. Check your compiler class libraries for one. We have included a watered down String class for you. When your program runs Vehicle objects are created from the Vehicle class definition. An object is reserved or allocated memory for the variables defined in the class. The compiler compiles the functions defined in the class into execution code. Each object can use the function to access and do operations on the data stored in the object. The functions defined in the class are used on all objects instantiated from the class.

Object model



Here is the Vehicle class definition.

```

// vehicle.hpp
#include "string.hpp"
class Vehicle

class class_name
{
    private:
        private variables

    public:
        default constructor
        initializing constructor
        copy constructor

        destructor
        set functions

        process functions

        get functions
};

{
    private:
        String Color; // color of vehicle
        int Year; // year of vehicle
    public:
        Vehicle(); // default constructor
        Vehicle(String& color,int year); // initializing constructor
        Vehicle(Vehicle& v2); // copy constructor
        ~Vehicle(); // destructor
        void setColor(String& color); // set color of vehicle
        void setYear(int year); // set year of vehicle
        bool verifyYear(int current); // check if year is valid
        String& getColor(); // get color of vehicle
        int getYear(); // get year of vehicle
        void print(); // print out info about vehicle
};

```

CLASS IMPLEMENTATION

defining functions outside the class definition

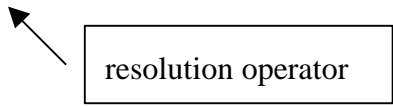
Functions are usually defined outside a class definition and must use the **resolution operator ::** to tell the compiler and programmer that this function belong to this class.

```

return_data type class_name :: function_name (paramater_list)

{
    function statements
}

```



Example the function `getColor()` belongs to the class `Vehicle` and returns a reference to a `String` object. The resolution operator `::` indicates that function `getColor()` belongs to class `Vehicle`. The function `getColor()` must be declared beforehand in the class definition as a prototype for the compiler to recognize the function outside the class.

```

String& Vehicle::getColor()

{
    return Color; // return vehicles color
}

```



defining functions inside the class definition

Functions in a class may be declared and defined right in the class definition. This is handy for short functions that just return a value. For larger functions it is recommended to define the function outside the class. Functions declared inside a class definition are also known as **inline** functions. You do not need to specify the inline keyword it is optional.

```
class class_name
{
    return_data_type  function_name (parameter_list)

    {
        function statements
    }
}
```

The following is an example of function defined inside a class definition:

```
class Vehicle
{
    String& getColor() // declare and define function right in the class

    {
        return Color; // return color of vehicle
    }
}
```

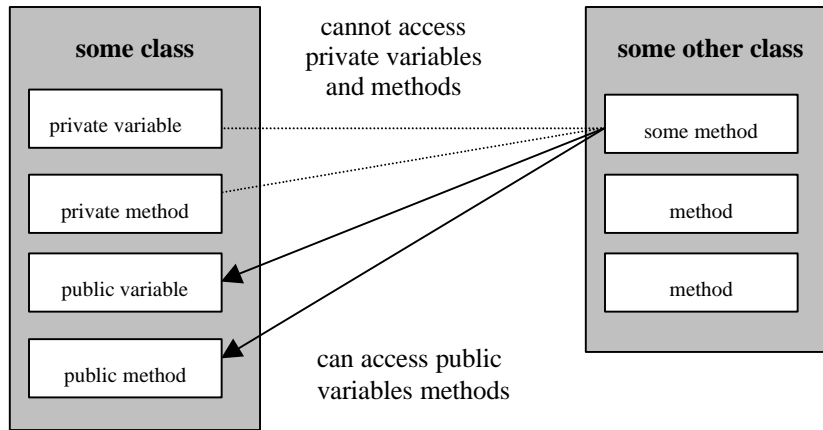
Define function inside class definition

Visibility of variables and functions

A **class** lists all the required variables and functions needed by the object. Variables and functions belonging to a class are called **member** variables and functions. Variables and functions belonging to a class have levels of visibility of **private** or **public**.

private	Only the class can access variables and functions of this class
public	Everybody can access variables and functions of this class

Only the class can access **private** variables and functions. Anyone who wants to access these private variables must use the class functions to access them. Think private variables as being bars of gold in a bank. Think as the security guards as the functions that allow the outside world to access the gold bars. If anybody could just walk into a bank and get gold bars then the bank would soon be broke. The people need the security guards (the class functions) to allow access to the gold bars. In contrast **public variables** allow everybody to access the information. Think as these variables as waste paper baskets on the street. The people as class functions, are free to deposit waste and to pick up waste.



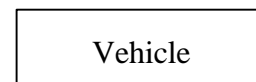
constructors

The beauty of C++ is that it has special functions called **constructors** that automatically initialize object variables. Constructors have the same name as the class but does not have a return type, not even a void. Constructors are optional and are only required if variables have to be initialized and memory has to be allocated. It is good practice to include a constructor in your code even if you do not need them. There are many types of constructors. We can have many different constructors due to constructor overloading. Same name different parameter list.

constructor type	description
default	initialize variables defined in a class to a default value
initializing	initialize variables defined in a class from a user value
conversion	initialize variables defined in a class to a user value of a different data type
copy	initialize variables defined in a class from an existing object

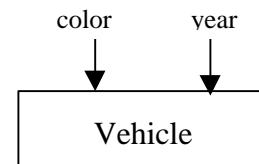
A **default constructor** has no parameter but may initialize all variables to default values when the object is constructed. A default constructor may be always required by the compiler.

```
// default constructor
Vehicle::Vehicle()
{
    Color = ""; // set to default values
    Year = 0;
}
```



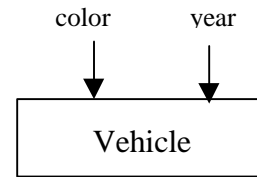
Initializing constructors receive values from the parameter list to initialize the variables of when the object is created.

```
// initializing constructor to initialize color and year
Vehicle::Vehicle(String& color, int year)
{
    Color = color; // initialize color
    Year = year; // initialize year
}
```



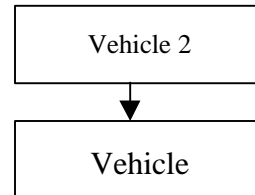
If the data is received as of a different data type then the data must be converted to the data type of the variable belonging to the class. In this situation, the constructor is known as a **conversion constructor**. We convert the year as a string to an int. Our String class has the `toInt()` function of the String class to convert a String object to an integer.

```
// conversion constructor to initialize color and year
Vehicle::Vehicle(String& color, String& year)
{
    Color = color; // initialize color
    Year = year.toInt(); // initialize year
}
```



Copy constructors are used to copy the contents of an existing object and give the values to the object you are creating. The copy constructors receive the object by pass by reference. By passing by reference, this ensures that the object exists. Copy constructors are always a little confusing to understand. Just remember you are still creating an object but you are getting the values from another object.

```
// copy constructors used to copy an existing class
Vehicle::Vehicle(Vehicle& v2)
{
    Color = v2.Color; // copy color
    Year = v2.Year; // copy year
}
```



destructors

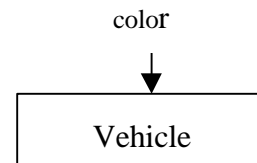
Special functions called **destructors** are automatically called when the object is finished being used. Destructors have the same name as the class but do not return anything not even a void. Destructors have a "~" (tilda) preceding the name to distinguish a destructor from a constructor. The destructor is used to deallocate memory that was allocated with the **new** operator. An example would be like allocated memory for an array. A destructor does not deallocate memory allocated for the object. You would need to this your self. Destructors are only needed if memory has to be deallocated. It is good practice to include a destructors in your code even if you do not need them.

```
// destructor
Vehicle::~~Vehicle()
{
}
```

set, process and get functions

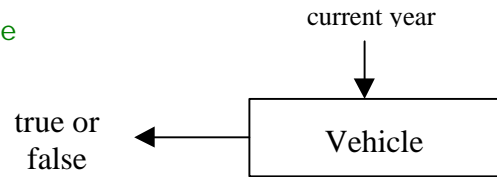
set functions are used to **assign data** values from the outside world **to variables** that were defined in the class. **Set** functions are also called **mutator** functions because they change the contents of the class member variables. The set function **setColor()** receives a colour and assigns it to the class member Color variable.

```
// input function to set color
void Vehicle::setColor(String& color)
{
    Color = color; // set vehicle color
}
```



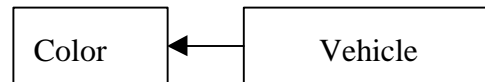
Process functions do calculations on variables. Process functions may private or public. The following function **verifyYear()** checks to see if a Vehicle object has the correct year

```
// process function to verify year return true or false
bool Vehicle::verifyYear(int current)
{
    return(Year < current); // test year
}
```



Get functions **return** data **to** the outside world and must be made public. Get functions are also called **accessor** functions because they access the class member variables. The following **getColor()** function return the color.

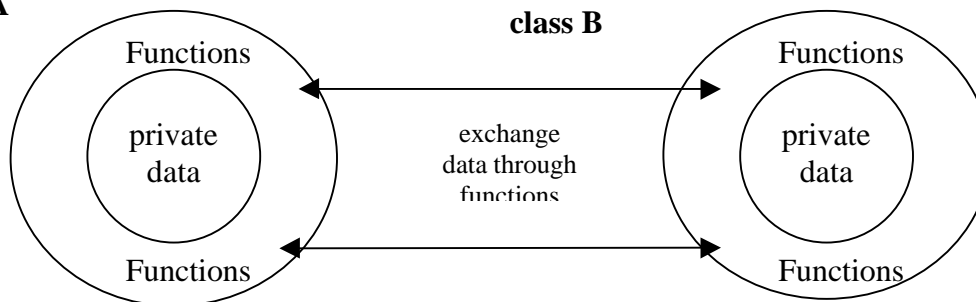
```
// output function to get color
String& Vehicle::getColor()
{
    return Color; // return color of vehicle
}
```



exchanging data between classes

All data in a class should be private. Functions are used to exchange data between classes:

class A



Each class has a dedicated job to do. The variables declared in a class should be private. No other class should be able to access the data directly. The classes use the functions to exchange data and do operations on the data. Data is secure inside a class. When your program runs **objects** are created from the class definitions. In an object, memory is allocated for the variables defined in the class to store data. The functions are used to access the stored data in the object. One function can access many objects,

Class Definition and Implementation example

The class definition is written in a *.h or *.hpp file. The implementation for a class is contained in the *.cpp file. All functions are defined outside the class. The **resolution operator ::** tells which function belongs to which class. The class definition file *.h or *.hpp file and must be included in the *.cpp file using the **#include** statement. This way the compiler knows about the functions and variables defined in the class when you use them. Here is the complete Vehicle class definition and implementation files.

```

// vehicle.hpp

// class Vehicle definition
#include "string.hpp"
class Vehicle

{
private:
String Color; // color of vehicle
int Year; // year of vehicle
public:
Vehicle(); // default constructor
Vehicle(String& color,int year); // initializing constructor
Vehicle(Vehicle& v2); // copy constructor
~Vehicle(); // destructor
void setColor(String& color); // set color of vehicle
void setYear(int year); // set year of vehicle
bool verifyYear(int current); // check if year is valid
String& getColor(); // get color of vehicle
int getYear(); // get year of vehicle
void print(); // print out info about vehicle
};

// vehicle.cpp
// class Vehicle implementation

#include <iostream.h>
#include "vehicle.hpp"
#include "string.hpp"

typedef int bool; // maybe needed for some compilers

// default constructor
Vehicle::Vehicle()

{
Color = ""; // set to default values
Year = 0;
}

// initializing constructor to initialize color and year
Vehicle::Vehicle(String& color, int year)

{
Color = color; // set color
Year = year; // set year
}

```

```

// copy constructors used to copy an existing class
Vehicle::Vehicle(Vehicle& v2)

{
    Color = v2.Color; //copy color
    Year = v2.Year; // copy year
}

// destructor
Vehicle::~~Vehicle()

{
}

// input function to set color
void Vehicle::setColor(String& color)

{
    Color = color; // set vehicle color
}

// input function to set year
void Vehicle::setYear(int year)

{
    Year = year; // set vehicles year
}

// process function to verify year return true or false
bool Vehicle::verifyYear(int current)

{
    return(Year < current); // test if year less than current year
}

// output function to get color
String& Vehicle::getColor()

{
    return Color; // return color of vehicle
}

// output function to get year
int Vehicle::getYear()

{
    return Year; // return vehicle year
}

```

```
// print out vehicle information
void Vehicle::print()
{
    cout << endl << "Vehicle info " << endl;
    cout << "color: " << Color << endl;
    cout << "year: " << Year << endl;
}
```

Declaring objects

When an object is created from a class definition, memory is allocated for the variables declared in the class. This is called **instantiating** the class. When a class is instantiated it is then called an **object**. Objects can be declared as a variable or as a pointer. When declared as a variable memory for the object is reserved on the execution stack. When declared as a pointer memory must be allocated for the object. When an object is created the default constructor is automatically called.

class_name variable_name

Vehicle v1; // a default vehicle class is instantiated having the variable name v1

Classes are declared just as you would an ordinary variable.

int x; // variable x is created having the data type int

You can call other constructors when you declare your object variable by providing an argument list. The compiler knows which constructor to call because it matches your supplied arguments with the parameters you defined in your constructor. This is called constructor overloading.

class_name variable_name(argument list)

Vehicle v2 (String("red"),1989); // a vehicle object instantiated having the name v2

We now have a Vehicle object with color "red" and year 1989. Notice that when we are using **pass by reference** we need to make a String object containing the string "red" to pass to the Vehicle constructor

Vehicle
v(String("red"),1989);

v	"red"
	1989

Accessing variables and functions in an object declared as a variable

Member variables and functions are accessed by the "." dot operator when objects are declared as **variables**.

object_variable_name . function_name
int year = v.getYear();

object_variable_name .variable_name
int year = v.Year

Before we can use variables and functions we must first instantiate a vehicle object:

```
Vehicle v(String("red"),1989); // instantiating a Vehicle class object by declaring variable v
```

Then we call the `getYear()` method of this vehicle object using the vehicle objects variable name, dot operator and function name. The dot operator means we want to call the `getYear()` function **belonging to** the class that instantiated this object. The function will access the data belonging to this object. The function can be used to access many objects all defined from the same class. What is the value of year ?

```
int year = v.getYear(); // access Vehicle year by using function getYear()
```

Creating objects in run time

Objects may also be created in run time, in this situation a pointer to an object is declared pointing to a memory block allocated for the object. The **new** operator is used to allocate memory for the object and the objects address is assigned to the pointer. The pointer `pv1` points to a memory block having data type class `Vehicle`. The constructor is automatically called after the object is created in memory.

```
class_name* pointer_variable_name = new class_name()
```

```
Vehicle* pv1 = new Vehicle(); // allocating memory for a default Vehicle
```

We now have a `Vehicle` object pointed to by the pointer **pv1** created in run time by allocating memory for the object using the **new** operator.

```
Vehicle* pv = new Vehicle();
```



Declaring a pointer to an object is no different then declaring a variable as a pointer.

```
int* p = new int; // assigning and allocating memory for variable
```

Here we allocate memory for an object and initialize to user specified values.

```
class_name* pointer_variable_name = new class_name(argument list)
```

```
Vehicle* pv2 = new Vehicle (String("red"),1989); // allocating for a "red" 1989 Vehicle
```

We now have a `Vehicle` object pointed to by the pointer **pv2** created in run time by allocating memory for the object using the **new** operator. Notice when using pass by reference we need to make a `String` object containing the string "red". to pass to the `Vehicle` constructor

```
Vehicle* pv = new Vehicle(String("red"),1989);
```



Accessing variables and functions in an object pointed to by a pointer

Member variables and functions pointed to by a pointer are accessed by the arrow "->" operator

```
object_variable_name -> function_name
```

```
int year = pv->getYear();
```

```
object_variable_name -> variable_name
```

```
int year = pv->Year;
```

To access variables and functions of an object pointed to by a pointer we first have to allocate memory for the object. After the memory is allocated for the object the constructor is automatically called.

```
// instantiating Vehicle class object pointer variable pv
Vehicle* pv = new Vehicle(String("red"),1989);
```

Next we call the **getYear()** function using the pointer to the object, the arrow operator and the functions name. What is the value of year ?

```
// access class function getYear() using class pointer variable
int year = pv->getYear();
```

Deallocating memory for a class

If a class constructor allocates memory for variables by using the **new** operator then that memory must be deallocated when the class object is no longer needed. The **destructor** is **automatically called** when the class "**loses scope**" meaning the class is no longer needed. This can be when the program ends or when an object created as a variable is no longer needed. When the destructor is called you use the **delete** operator to delete any memory allocated like an array.

```
delete[] a; // delete any allocated memory by user
```

When the program ends its good practice to delete objects that have been allocated by using the **new** operator.

```
delete pv; // delete all memory allocated to object pointed to by pv
```

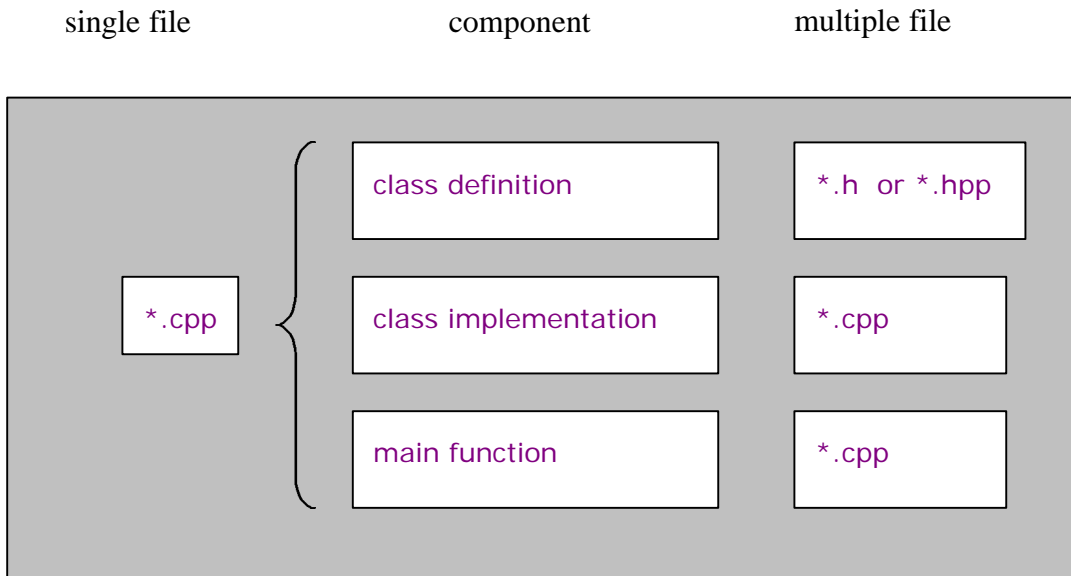
You do not need to delete objects when objects are created as a variable. The reason is that the object memory has been allocated on the execution stack. When the object is no longer needed the stack memory is automatically reclaimed. You cannot delete objects that were created as a variable. If the object was allocated using the **new** operator then you must delete the object. The constructor will be automatically called when you delete the object. If you do not delete the object the constructor will not be called.

Destructor is automatically called
when you delete an object

You cannot delete an object created
as a variable (reserved memory)

C++ PROGRAM FORMAT

A C++ Program usually has the class definitions at the top, then class implementation, then the main function and lastly any other functions. For small programs with few classes its good to put everything in one file. This way you can work with one file to fix up all your bugs. When you work on a large project with many classes you need separate files for class definitions and for class implementations. The main function is the first function to run in your program. Its purpose is to instantiate all the objects in your program and call functions from these objects.



main function to test program for vehicle class

The main function will create all and test all the fun Vehicle objects either declared as a variable or pointed to by a pointer. In C++ arguments to functions are usually passed by reference. All the functions in the Vehicle class use pass by reference. Pass by reference is good because it relieves a lot of overhead by passing the direct address of objects to functions. The only drawback is we need to have already created objects to pass. You will always need to create a String object. Example:

```
Vehicle v1(String("red"),1989); // make a red vehicle
```

We cannot just say: (some compilers let you do this)

```
Vehicle v1("red",1989); // make a red vehicle
```

Some compilers will generate an error other compilers will construct the String object for you and give you a warning message "temporary object created". Most people use pass by reference for fast program execution but in some case pass by value is more convenient to use. Especially if you do not want to change an existing object.

```

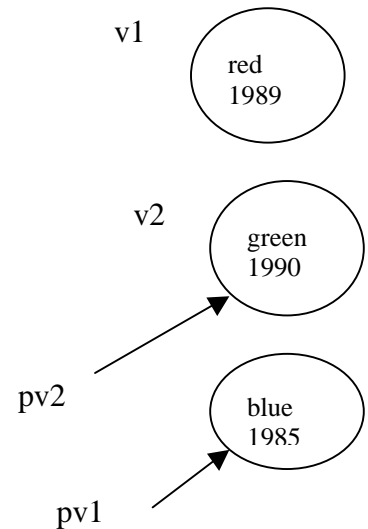
// main program to test class Vehicle

// L7p1.cpp
#include "vehicle.hpp"
#include "string.hpp"

typedef int bool; // maybe needed by some compilers

void main()
{
    Vehicle v1(String("red"),1989); // make a red vehicle
    v1.print(); // print out vehicle info
    Vehicle v2(String("green"),1990); // make a green vehicle
    v2.print(); // print out vehicle info
    Vehicle* pv1 = new Vehicle(); // instantiated a new vehicle
    pv1->print(); // print out vehicle info
    pv1->setColor(String("blue")); // set vehicles color to red
    pv1->setYear(1985); // set year to 1989
    pv1->print(); // print out vehicle info
    Vehicle* pv2 = &v2; // point to vehicle v2
    pv2->print(); // print out vehicle info
    delete pv1;
}

```



program output:

```

Vehicle info
color: red
year: 1989
Vehicle info
color: green
year: 1990
Vehicle info
color:
year: 0
Vehicle info
color: blue
year: 1985
Vehicle info
color: green
year: 1990

```

LESSON 7 EXERCISE 1

Type in the Vehicle class definition, and implementation and main program and run it. You will need to make a project or workspace and add the cpp files to it. Add some more tests in the main program, especially use the **verifyYear** function and **copy constructor** to copy existing Vehicle objects. The definition file should be located in vehicle.hpp and the vehicle implementation code should be in vehicle.cpp. The main program should be typed in L7ex1.cpp. If your compiler does not do have a String class. Use the following provided String class. The String class definition goes into string.hpp and the String class implementation goes into file string.cpp. The String class uses C++ language constructs that will be studied in future lessons. Don't let them intimidate you. You may use " cout "to print out info from the Vehicle objects and data from Strings.

Once your program is running add the **make** and **model** String variables and supporting functions to the Vehicle class. Add and use the **conversion constructor**. Make more vehicle objects to test the new additions in your main function.

Vehicle Object:

make
model
year
colour

Organize your files as follows:

header file	implementation file	execution file (main function)
vehicle.hpp	vehicle.cpp	
string.hpp	string.cpp	
	L7ex1.cpp (main function)	L7ex1.exe

using string class:

To print out a variable name, character string or String separate each variable by a << . The **endl** directive will start a new line at the end of your output.

```
int x = 5;
String s = "hello.";
cout << "x = " << x << " today's message: " << s << endl;
```

```
x = 5 today's message : hello.
```

Here is our String class

```
// string.hpp
#include <iostream.h>

// String class definition
#ifndef __STRING
#define __STRING
```

```

// class for character strings
class String

{
private:
char* Str; // pointer to char string
int Length; // length of character string
public:
String(); //default constructor
String(const char * s); // initializing constructor of a char string
String(String& s); // copy constructor
~String(); // destructor

// assign an existing String
const String& operator=(const String& s);

// convert string to integer
int toInt();

// send string to output stream
friend ostream& operator << (ostream& out, String& s);

//get string from input stream
friend istream& operator >> (istream& in, String& s);
};

#endif

// string class implementation file
// string.cpp
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include "string.hpp"

// default constructor
String::String()

{
Str = new char[1]; // empty character string
*Str = '\0'; // end of string indicator
Length = 1;
}

// initializing constructor
String::String(const char* s)

{
Str = new char[strlen(s)+1]; // make room for a string
strcpy(Str,s); // copy existing character string
Length = strlen(Str); // set length of String
}

```

```

// copy constructor
String::String(String& s)
{
    Str = new char[s.Length+1]; // make room for a string
    strcpy(Str,s.Str); // copy existing character string
    Length = strlen(Str); // set length of String
}

// destructor
String::~String( )
{
    delete[ ] Str; // delete allocated memory for a string
}

// assignment operator
const String& String::operator=(const String& s)
{
    if(this != &s) // don't copy itself
    {
        delete[ ] Str; // delete current string
        Str = new char[s.Length+1]; // allocate memory for new string
        strcpy(Str,s.Str); // copy existing string
        Length = s.Length; // set size
    }

    return *this; // return dereferenced pointer to class with new string
}

// convert string to integer
int String::toInt()
{
    return atoi(Str);
}

// send string to output stream
ostream& operator << (ostream& out, String& s)
{
    out << s.Str; // print out character string
    return out;
}

// read in a string from input stream
istream& operator >>(istream& in, String& s)
{
    char s2[255];
    in.get(s2,sizeof(s2));
    s = String(s2);
    return in;
}

```

LESSON 7 EXERCISE 2

Person Object:

name
age
height
weight

Make a Person class. Each Person will have a name, age, height and weight. Write all the functions for the Person class all constructors, destructors, set, process and get functions. Test everything in a main function. Make sure you use the String class for the person's name. You will have files Person.hpp, Person.cpp and the main function will be in L7ex2.cpp.

Passing objects by value, pointer and reference

Objects may be passed by value, pointer or by reference. This is the most important and dangerous decision to make when using objects. The problem arises when the destructor is called. Pass by reference or pointer is most efficient.

pass object by	operation	destructor	original object
value	A copy of the object is made using the copy constructor. The object is then used in the function.	When the function terminates the passed object destructor is called to destroy the copied object.	The original object contents is not changed
pointer	A pointer (indirect address) to the object is passed to the function. The function uses the object by using the pointer to the object.	The passed object destructor is not called	The original object contents is changed
reference	A reference (direct address) to the object is passed to the function. The function uses the object by using the reference to the object. An actual reference to an object must be passed. You must pass an object variable or the value of a pointer to an object or else you will get a warning message "temporary object created".	The object destructor is not called	The original object contents is changed

LESSON 7 EXERCISE 3

Use the Vehicle object or the Person object. Inside each constructor and destructor put a message like:

```
cout << "Vehicle copy constructor called" << endl;
```

Make a function in your main file that receives objects and prints them out. Pass your object by value, point and reference to objects as a variable, reference and a pointer. You will have 9 possibilities. Call the file with your main and test function L7ex3.cpp.

RETURNING OBJECTS BY VALUE, POINTER AND REFERENCE

Objects may be returned by value, pointer or by reference. This is even more dangerous when using objects. The problem arises when the destructor is called. When you pass by a object by value the object that you are passing may get destroyed by the destructor inside the function. When you return the object by pointer or by reference you may returning a destroyed object. When you use the returned object again it is gone and your program crashes. The alternative is to allocate an object using the new operator in the function and return this object.

```
return new Vehicle(v);
```

Return by reference or pointer is most efficient.

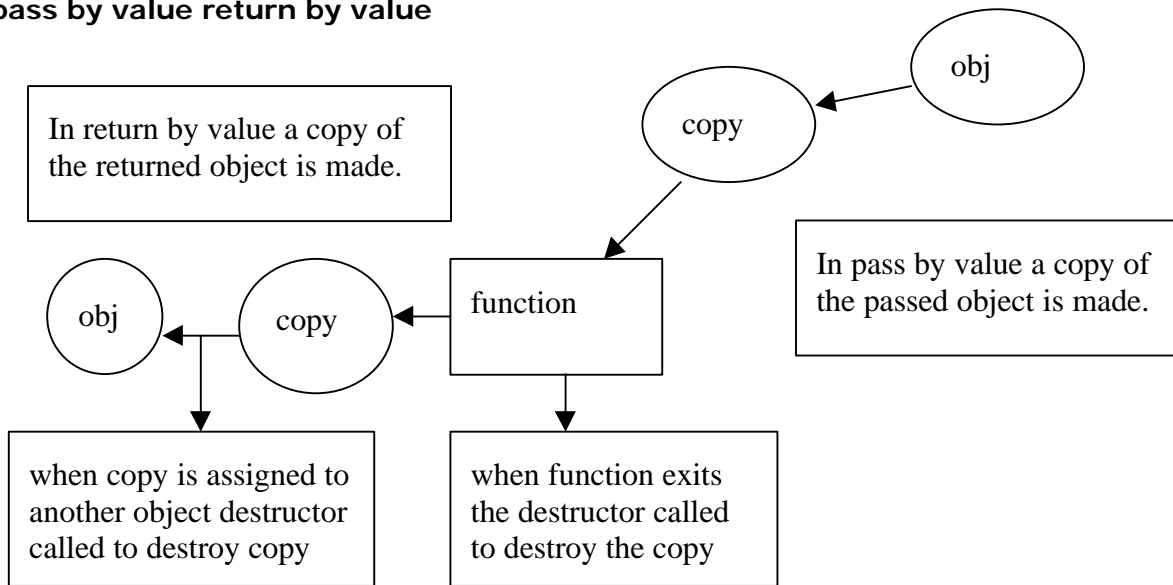
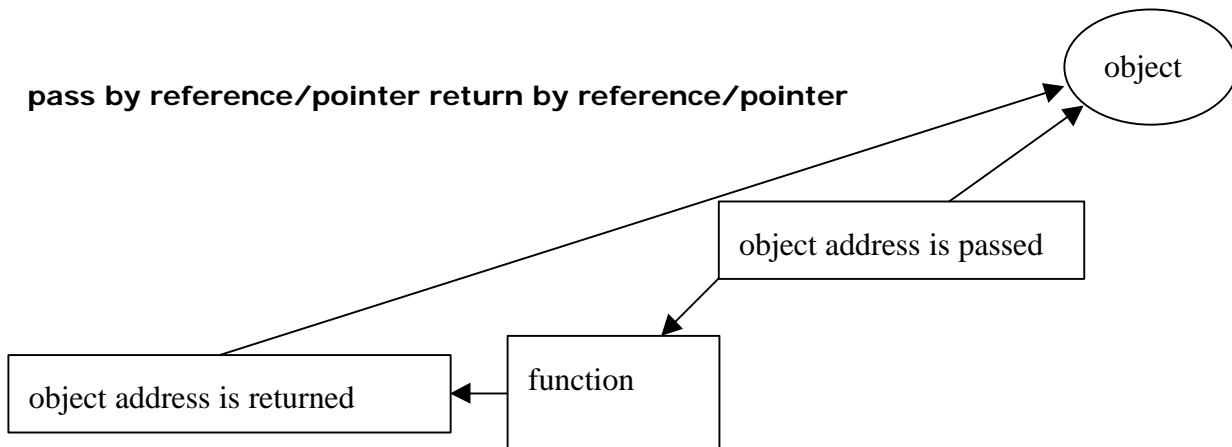
return object by	operation	destructor
value	A copy of the object is made using the copy constructor. The copy is then returned to the receiving object. The receiving object may call the copy constructor	the destructor is called for each copy
pointer	The address of the object located in the function is returned. The object must be allocated or previously passed to the function by reference or by pointer	the object destructor is not called
reference	A reference to a object in the function is returned. The object inside the function cannot be a temporary variable. The object must be allocated or previously passed to the function by reference or by pointer	the object destructor is not called

LESSON 7 EXERCISE 4

Use the Vehicle object or the Person object. Inside each constructor and destructor put a message like:

```
cout << "Vehicle copy constructor called" << endl;
```

Make a function in your main file that receives and returns vehicle objects. Print out the object in the function and after it returns. Return your object by value, pointer and reference to objects as a variable, reference and a pointer. You will have 9 possibilities. Call the file with your main and test function L7ex4.cpp.

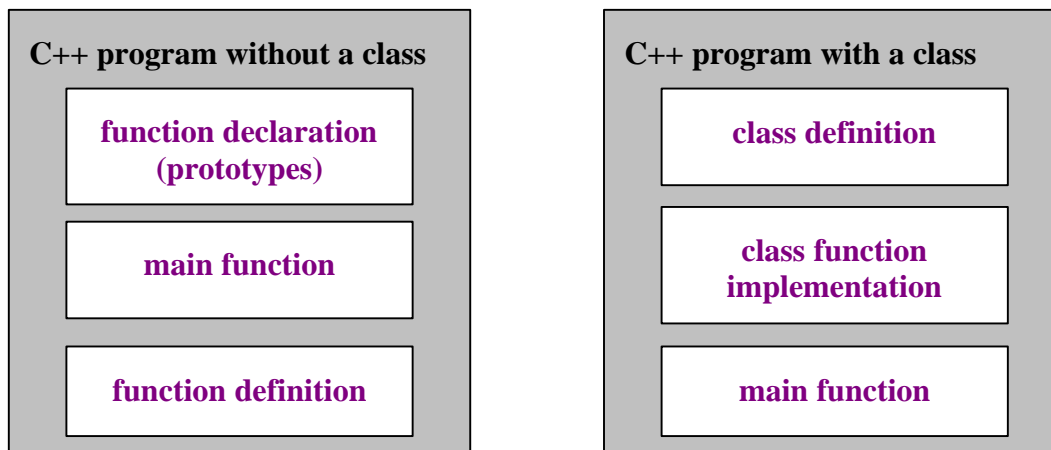
pass by value return by value**pass by reference/pointer return by reference/pointer**

C++ PROGRAMMERS GUIDE LESSON 8

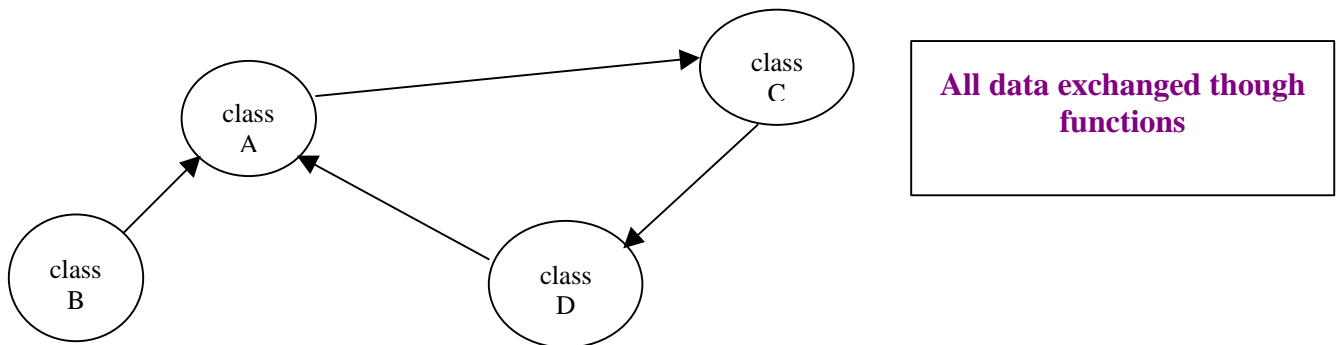
File:	CppGuideL8.doc
Date Started:	July 12, 1998
Last Update:	Mar 26, 2002
Version 4.0:	proof

LESSON 8 C++ PROGRAMMING STATEMENTS

Functions are made up of programming statements. Statements in a function execute sequentially. Functions may be standalone or belong to a class. A class is like a small mini-program that has its own variables and functions. An object oriented program will have many classes each dedicated to a specific task. Programs without classes have many functions instead.

**using classes**

Most C++ programs are written using classes. If you just have to do some small calculation then a C++ program does not need to be written using classes and just a main program is needed. If you use classes each class should have a dedicated purpose. Each class needs supporting functions to implement the required task. You should have many classes each performing a specific task rather than 1 task that does all. It is much better to write programs using object oriented programming, your programming tasks will be much easier to accomplish. By having a dedicated class for each task it is easier to debug. You may also have the opportunity to re-use classes for other projects. Your programming time will be less. A C++ program has many classes all communicating together using functions. Objects are created from classes. An object is just memory allocated or reserved for the variables defined in the class. A class is the definition that objects are created from. You can make many objects all created from the same class. The functions are used to exchange the data between the objects created from the class. What is the difference between a class and an object? A class is the program you type in the computer. The object is the allocated or reserved memory in the computer for the variables defined in the class. The class has defined what the object is.



program examples

The following example program declares and defines function as standalone and as belonging to a class. The function just takes the square of a number:

standalone function program	function belonging to a class program
<pre> #include <iostream.h> // declare square function int square(int x); // main function void main() { // get value from keyboard int x; cout << "enter a number: "; cin >> x; // call square and print results cout<<square(x)<<endl; } // define square int square(int x) { x=x*x; return x; } </pre>	<pre> #include <iostream.h> // define class Test class Test { public: int square(int x); // declare square }; // main function void main() { int x; cout << "enter a number: "; cin >> x; Test t; // make a Test object // call square function cout << t.square(x) << endl; } // define function square belonging to class Test int Test::square(int x) { return x*x; } </pre>

The function declaration prototype tells the compiler all about the function before it is used and defined.

```
int square(int x );
```

By using function prototypes you can start using the function before you define it

```
square(x);
```

Without function prototypes you have to define the function before you use it. The functions are defined later in the program.

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

A class is used to declare variables and functions that have some common purpose.

```
class Test
{
public:
  int square(int x); // declare square
};
```

Functions declared in a class are usually defined outside the class. using the resolution operator :: .

```
int Test::square(int x )
{
  return x*x;
}
```

A class just groups together function prototypes and variables needed by these functions. It's like a mini-program. Objects are created from class definitions.

```
Test t; // make a Test object
```

You can have many objects all created from the same definition. When you use a function you need to indicate which object it wants to use.

```
t.square(x)
```

C++ PROGRAMMING STATEMENTS BY EXAMPLE

C++ programming makes good use of the stream classes to read input from the keyboard and output to the screen.

```
// program to test function statements Lesson8 program 1
#include <iostream.h>
```

Programming statements are now introduced by typical application. There is no specific order in which to use them. You may declare and initialize variables as you need them. The only catch is that if you declare a variable inside a statement block then the variable is not accessible outside the statement block. A statement block is any statements included by an opening curly bracket "{" and a closing curly bracket "}".

```
#define true 1
#define false 0

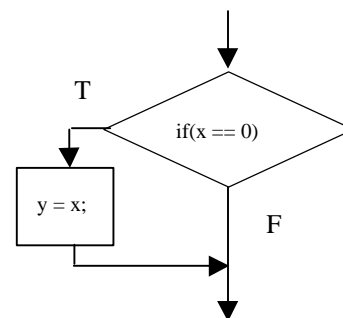
const int Max = 10; // loop size
const int NumRows = 3; // number of rows
const int NumCols = 3; // number of columns

// main function
void main()
{
    int i,j,k;
    int x=5;
    int y=10;
    int a[10],b[10];
```

if statement

The **if statement** is used to test conditions to make a decision in a program flow. An **if** condition is evaluated as **true** or **false**. When the **if condition** is **true** then the statements belonging to the **if** condition is executed. When the **if condition** is **false** then program flow is directed to the next program statement. If an **if statement** has more than one statement then the statements belonging to the **if** expressions must be enclosed by curly brackets. An if statement allows you to make a **choice** in program execution direction.

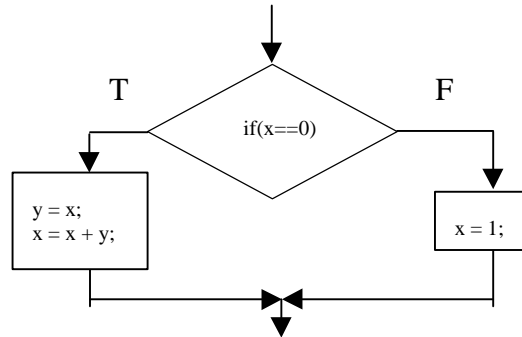
definition	one statement	more than one statement
<i>if(condition)</i> <i>true statement(s)</i>	<pre>if (x==0) y = x;</pre>	<pre>if (x==0) { y = x; x=x+1; }</pre>



if - else statement

An if statement may have an optional **else** statement. The else statement executes the alternative **false** condition. Curly brackets are also needed if more than one statement belongs to the **else** statements.

<i>if(condition)</i>	<code>if (x==0)</code>
<i>true statement(s)</i>	<code>{</code> <code> y = x;</code> <code> x=x+y;</code> <code>}</code>
<i>else</i>	<code>else</code>
<i>false statement(s)</i>	<code> x=1;</code>



nested if-else statements

if - else statements may be nested. In this case the **else** belongs to the balanced **if**. In case of confusion to determine which **else** belongs to which **if** use curly brackets..

```

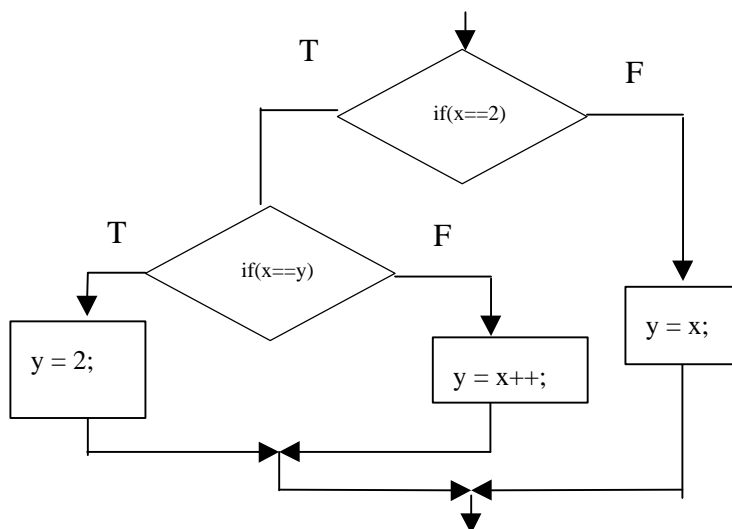
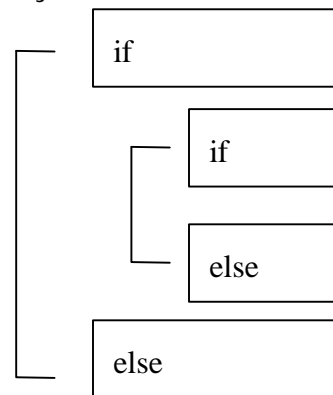
if ( condition )
{
  true statement(s)
  if ( condition )
    true statement(s)
  else
    false statement(s)
}
else
  false statement(s)

```

```

if ( x == 2 )
{
  if(x == y)
    y = 5;
  else
    y = x++;
}
else
  y = x;

```



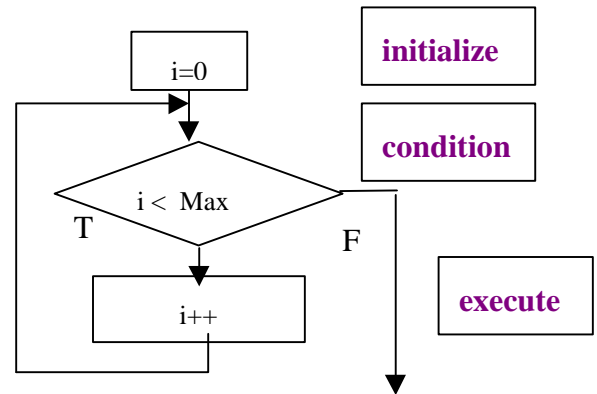
while loop

The while loop tests a **condition** at the top of the loop. If the condition is **true** the statement inside the while loop is executed. If condition is **false** program execution exits the loop. You must be careful and make sure that the test condition is initialized before entering the loop. The **while statement** allows you to read in items when you do not know how many items you have. If the test condition is false before entering the loop the statements never get executed.

```

initialize          i = 0; // initialize counter
while( condition )
{
    loop statement(s)    cout << i << endl;
                        i++; // increment
                        }
                        cout << i << " items" << endl;

```



A **while (true)** means loop for ever.

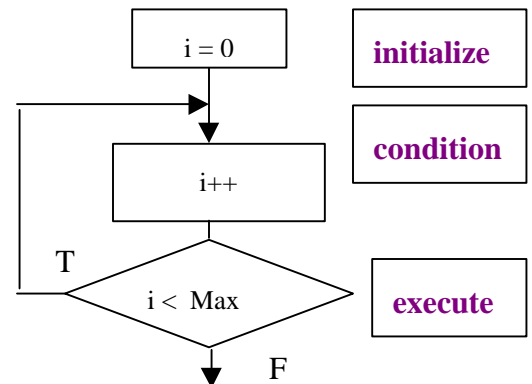
do while loop

A **do while** loop tests the condition at the bottom of the loop. The statements inside the loop are executed at least once. You use the do while loop, when you **do not** know how many items you have and want to execute the do while statements at **least once**.

```

do{                  i = 0; // initialize counter
                    do
loop statement(s)    {
                    cout << i << endl;
}while ( condition );    i++;
                        } while ( i < Max )
                        cout << i << " items" << endl;

```



What is the difference between a **while** loop and a **do while** loop? When would you use a **while** loop.? When would you use a **do while** loop ? A do while loop executes the loop statements at least once, where a while loop may never execute the loop statements. You use a do while loop when you want the loop statements to be executed at least once.

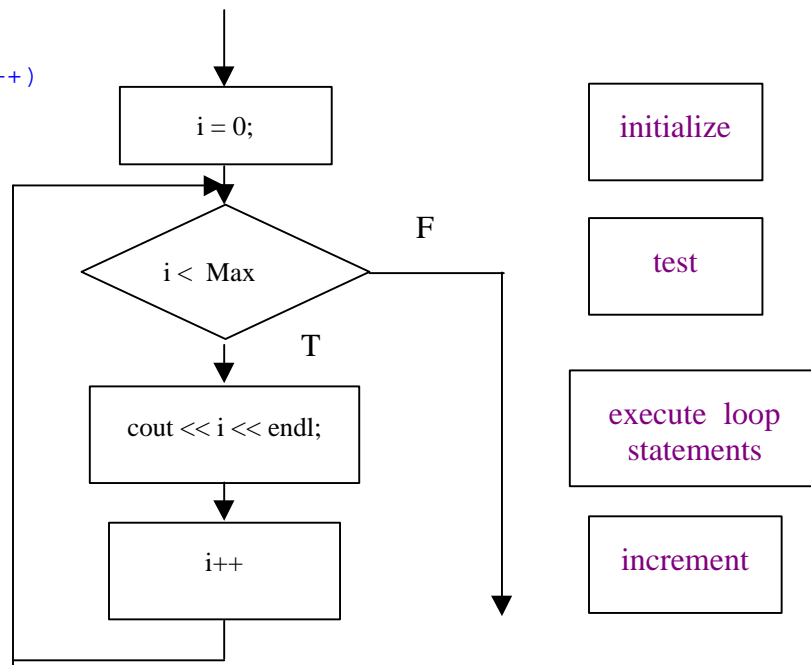
for loop

The **for** loop is used when you know how many items you have. For example to read a known number of items from a file. A for loop statement has a initializer for setting some counter to an initial value. For each iteration of the loop it tests the counter if it is at its limit and increments the counter automatically. In a **for loop** you can declare and initialize the loop counter, test the loop counter, and increment the loop counter.

for (initializing expression; testing condition; incrementing expression)

loop statement(s)

```
for(int i = 0; i < Max; i++)
    cout << i << endl;
```



If the **for** loop has more than one statement then curly brackets must be used to enclose the statements belonging to the for statements.

```
for(int i = 0; i < Max; i++)
{
    x = i;
    cout << i << endl;
}
```

All expressions in the **for** statement are optional. If there were no expressions then there would be an infinite loop.

```
for ( ; ); // loop for ever
```

You can include more than 1 index counter expression in each **for** loop expression, by using the comma operator. Unfortunately you can have only one test expression.

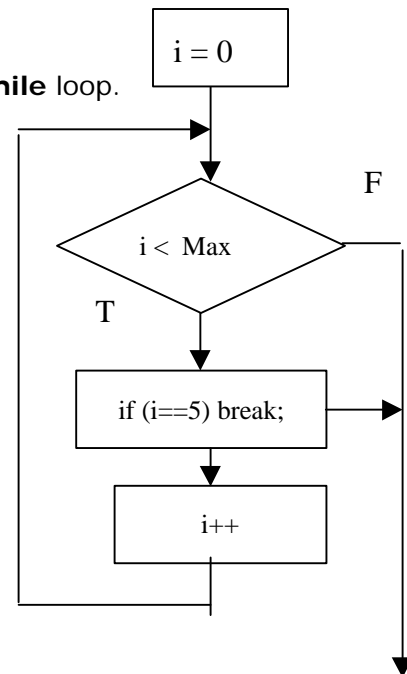
```
for(int i = 0, j = Max-1; i < Max; i++, j--)
    a [ i ] = b [ j ]; // copy reverse of an array
```

break statements

The break statement lets you exit out of a **for**, **while** or **do while** loop.

```
/* loop till break condition encountered */
i = 0; // initialize counter
while(i < Max)
{
    if(i==5)break; //exit loop if test condition true
    i++;
    cout << i << endl;
}

cout << "i got " << i << " items" << endl;
```



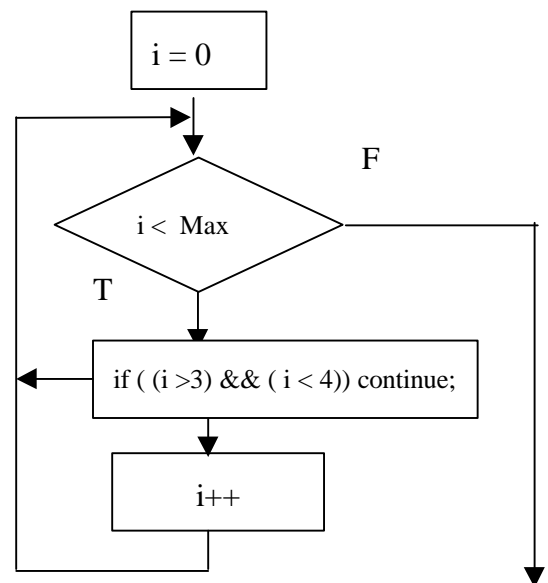
continue statement

The continue statement is used with **for**, **while**, **do while** loop to continue to the next iteration of a loop and skip statements you do not want to evaluate for certain iterations.

```
i = 0;

// loop till i greater than
while( i < Max;)
{
    i++;
    // skip i between 3 and 4
    if ((i >= 3) && (i <= 4)) continue;
    cout >> i; // this line skipped
}
```

What is the difference between a **break** statement and a **continue** statement ? The **break** statement lets you exit the loop early and the **continue** statement lets you skip executing statements in the loop.

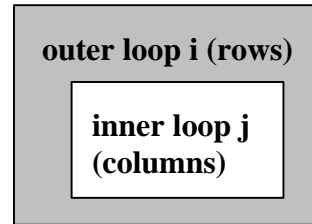


nested for statements

For statements are usually nested. This means a loop inside a loop. An outer loop and a inner lop. Nested **for** statements are handy in printing out the contents of a two-dimensional array. The **outer** loop are the **rows** and the **inner** loop access the **columns**.

```
// outer loop for number of rows
for (int i = 0 ,k=1;i < NumRows; i++,k++)
{
    // inner loop for number of columns
    for(int j =0 ; j < NumCols; j++)
        cout << k << " "; // print out array element

    cout << endl; // start a new line for each row
}
```



The inner loop using the columns index (j) prints out the contents of all the columns of the row pointed to by the outer loop row index (i)

		j				
		0	1	2	i	j
i	0	1	2	3	0	0, 1, 2
	1	4	5	6	1	0, 1, 2
	2	7	8	9	2	0, 1, 2

printing out diagonal of a square array

If you just want to print the diagonal of an array then you just print out the values of the array element when $i == j$, when the row index equals the column index.

```
// outer loop for number of rows
for( i = 0, k=0; i < NumRows; i++)
{
    // inner loop for number of columns
    for(j=0; j < NumCols; j++)
    {
        // test if column equal row, print out diagonal
        if( i == j ) cout << k++ << " ";
        else cout << " ";
    }
    cout << endl;
}
```

		j				
		0	1	2	i	j
i	0	1			0	0, 1, 2
	1		5		1	0, 1, 2
	2			9	2	0, 1, 2

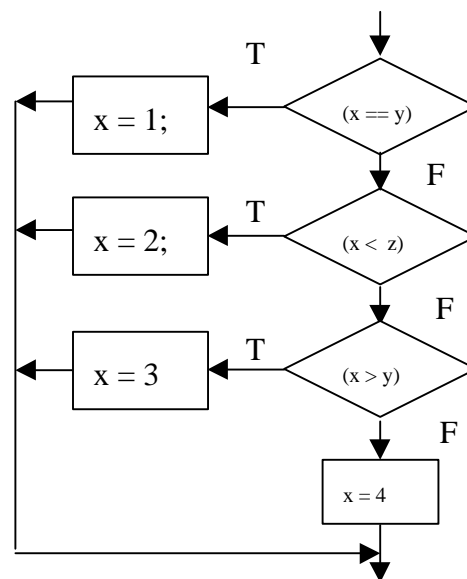
Rewrite the code so that it prints out the reverse diagonal.

		j				
		0	1	2	i	j
i	0			3	0	0, 1, 2
	1		5		1	0, 1, 2
	2	7			2	0, 1, 2

if-else-if

The **if** statement and **else-if** statements can be sequenced to choose a certain condition from **many** listed conditions. We search for a true condition and only execute the statements for that true condition. If the value you are looking for is not found then program execution is directed to the **else** statement.

<i>if (condition)</i>	if(x == y)
<i>true statement</i>	z=1;
<i>else if(condition)</i>	else if(x < z)
<i>true statement</i>	z = 2;
<i>else if (condition)</i>	else if(x > y)
<i>true statement</i>	x = 3;
<i>else</i>	else
<i>default statement;</i>	x = 4;



switch statement

To avoid typing in many else-if statements then the switch statement is used. The **switch expression** must be evaluated to a primitive data type, like char, int, float etc. The **case constant expression** must be a **constant value** and not a **variable name**. The **break statements** stops the program execution from going to the next case. Without the **break statements**, you will get unexpected results. Sometimes it is desirable to omit the **case statements and breaks**. By using this technique you can test for multiple conditions.

<pre> switch(expression) { case constant 1: statements break; case constant 2: statements break; case constant N: statements break; default: statements break; } </pre>	<pre> switch(x) { case 0 : { cout << first item is << x << endl ; break; } case 1: // 1 or 2 case 2: { cout << the items are: << x << endl; break; } case 3: cout << the items are: << x << endl; break; case 4: cout << the items are: << x << endl; break; default: break; } </pre>
---	--

In the switch statement code example why does case 1 not have any statements or break statement ? Because we want to trap for two conditions when $x == 1$ and $x == 2$. Why does case 3 and 4 not have any break statements.? Because we want 2 messages printed out for case 3. Some people put curly brackets `c{ }` in the case statements like cases 1 and 2 some do not like cases 3 and 4. It's up to you. it can be your own preference.

```
} // end main
```

Global Variables

Even with C++ Object Oriented Programming global variables are still needed. Global variables are usually declared before the main program because it's the main program that uses them. Classes should not access global memory directly. Classes should access global memory through its functions and by use of pointers. The main reason Global variables are still used because it is easier for the compiler at compile time to reserve large amounts of memory directly. There may be certain conditions where variables need to be initialized before an object is created. Another approach is to have a class distribute data to other classes. You could call this class a DataManager class.

sharing data between classes

You may need to share data between classes, like a table of values. You need to designate a class to access common data to all the other classes. The other classes can access this data through the class methods. One class distributes data to all other classes. This class could be called **Data Manager**. The other classes can access this class through a static function called **getInstance()** of the **DataManager** class. There should only be one instance of a **DataManager** class. The only way you can do this is to make the constructor private (yes you can have private constructors) and have the **getInstance()** method instantiate the class only once. The other times the **getInstance()** method is called the method will return an reference to the object.

```
// data manager class

#include <iostream.h>

class DataManager
{
private: static DataManager* reference;
// private constructor
private: DataManager()
{}

public: static DataManager* getInstance();
};

DataManager* DataManager::reference=NULL;

DataManager* DataManager::getInstance()
{
if (reference == NULL)
reference = new DataManager(); // create instance of this class
return reference; // return reference to instance of class
}
```

How can another class get a reference to the DataManager class ?

```
// test
void main()
{
DataManager* dm = DataManager::getInstance();
cout << dm << endl;
}
```

CONVENTIONS TO MAKE YOUR PROGRAMMING LIFE EASIER

If you follow certain programming style conventions then when you read your program you will instantly recognize variables and functions as being parameters or member variables.

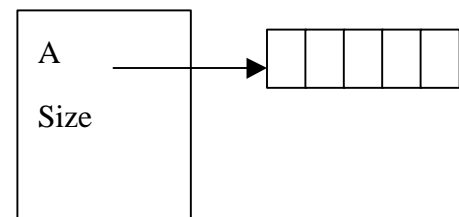
All class names should be capital	<code>class Test</code>
All member variables should be capital	<code>int Item</code>
All member functions names should be lower case	<code>get()</code>
All function and member parameters and should be lower case	<code>int x</code>
Avoid underscores between letters in variable names:	<code>new_word</code>
Use capitalization to distinguish words in a variable name:	<code>newWord</code>
All #defines and const should be capital letters or start with capital letters:	<code>#define MaxDays 7</code> <code>const int MaxDays = 7;</code>
All global variables, should start with a capital	<code>int A[];</code>

*** JUST DO 5 OF THE FOLLOWING EXERCISES, PICK INTERESTING ONES ***

Lesson 8 Exercise 1

Write a class called **Num** that receives numbers from the keyboard, stores them in an array, and reorders (exchanges) them. The Num class will have a function called **get()**, **reorder()** and **print()**. The main function will instantiate a Num object and call the **get()** function to get 5 numbers from the keyboard. The main function will call the function **print()** to display the numbers stored in the Num object. The main function will ask the Num object to start exchanging numbers by calling the function **reorder()**. Exchange the last with the first, the second with the second last etc. The main function will call the **print()** function to display the reordered array. Your constructor should get the size of the array and allocate an array to this size. All variables must be private. If you allocate memory then your destructor should deallocate it. You may put all your code definitions and declarations in one file called L8ex1.cpp.

member	description
<code>int* A;</code>	pointer to array
<code>int Size;</code>	size of array
<code>Num(int size);</code>	initializing constructor
<code>~Num();</code>	destructor
<code>void get();</code>	get array elements from keyboard
<code>void reorder();</code>	exchange array elements
<code>void print();</code>	print out array elements on screen



Lesson 8 Exercise 2

Continue from Exercise 1. Have the reorder function also keep track of the smallest entered number and the largest entered number in the array. You will need Min and Max variables as part of the class definition to keep track of the smallest and largest entered numbers. As before the main function will call the get function to ask the user to enter array values from the keyboard. Use the print() function to print out the min and max values and the contents of the array. Call your program L8ex2.cpp.

Lesson 8 Exercise 3

Include a function called add to add up all the elements in the Num class. As before the main function will call the get function to ask the user to enter values from the keyboard and print out the numbers in the Num object. Use a separate statement in the main function to print out the sum of all the numbers in the Num object. Call your program L8ex3.cpp.

Lesson 8 Exercise 4

Write a class called Many that has three arrays of the same length. The class has a function called get() that the user can enter values for the first two arrays. The Many class also has a function called add() that adds both of the arrays into a the third one. The Many class also has a print() function that prints out the contents of all the arrays. The main function will call the get() function to ask the user to enter array values from the keyboard. The main function will then call the add() function to add the first and second array together, Finally the main function will call the print() function to print out each array and the final result stored in the third array. Call your program L8ex4.cpp.

Lesson 8 Exercise 5

Write a class called Many2D that has three 2 dimensional arrays of the same length. Include a multiply function to multiply the first two arrays and place the result in the third array. The main function will call the get() function to ask the user to enter array values from the keyboard for the first two arrays. . The main function will then call the multiply function to multiply the first and second arrays together, Finally the main function will call the print function to print out each array and the final result stored in the third array. Call your program L8ex5.cpp.

Lesson 8 Exercise 6

Write a program that takes a two dimensional array 5 * 5 and rotates each element by the amount a user enters on the keyboard. If the user enters a positive number 2 all the elements in the array will shift right by 2. If they entered a negative number all the elements in the array will shift left. Print out the array before and after rotation. Call your program L8ex6.cpp.

Lesson 8 Exercise 7

Ask the user to enter in the keyboard how many characters they want on a output line. Ask the user to enter many lines from the keyboard or read text lines from a file. Produce an output on the screen where all the lines are a fixed width where the left and right margins are justified. Hint: use an output buffer to format the lines first before printing. Call your program L8ex7.cpp. Example the above lines would appear as follows or a line width of 28 characters:

Ask the user to enter in the keyboard how many characters they want on a output line. Ask the user to enter many lines from the keyboard or read text lines from a file. Produce an output on the screen where all the lines are a fixed width where the left and right margins are justified.

Lesson 8 exercise 8

Write a program that removes all the comments from a C++ program. file Your program should be able to handle nested comments "comments inside comments" . Report any comments that do not end and nested comments that are unbalanced. Write the results to the screen for viewing and to another file called L8ex8.dat. Call your program L8ex8.cpp. An example input file would be:

```
/* lesson 8 program 2 */
#include <iostream.h>

/* test program /* nested comment */ first comment continued */
void main()
{
    int num;
    srand(time(NULL)); /* randomize(); */
    num = rand() % 100; /* random number between 0 and 99 */
    cout << num << endl; /* print out random number ***/
    num = (rand() % 100) + 1; /* random number between 1 and 100
    cout << num << endl; /* print out random number */
}
```

The result file would be:

```
#include <iostream.h>

void main()
{
    int num;
    srand(time(NULL));
    num = rand() % 100;
    cout << num << endl;
```

error line 10: comment unbalanced

error line 15: file ended but comment started line 12 not finished

LESSON 8 EXERCISE 9

Write a program that prints out the transpose of a matrix. Call your program L8ex9.cpp. For example

1	2	3
4	5	6

1	4
2	5
3	6

LESSON 8 EXERCISE 10

Write a program that print out a triangle. Call your program L8ex10.cpp. The user of your program will specify the width of the base of the triangle.

```

      *
     **
    ***
   ****
  *****

```

LESSON 8 EXERCISE 11

Write a program that print out a Diamond. Call your program L8ex11.cpp. The user of your program will specify the maximum width of the diamond.

```

      *
     **
    ***
   ****
  *****
 *****
  *****
   ****
    ***
     **
      *

```

LESSON 8 EXERCISE 12

Write a program that calculates a magic square. In a magic square the elements of each row and column add up to the same value. No cheating all numbers cannot be 1. When your program runs, the user will enter the size of the square. Also write a function to test if your square is really a magic square. The user will enter the size of the square, the square is always odd. The steps to making a magic square as follows:

- (1) set k to 1 and insert into the middle of the top upper line
- (2) repeat until a multiple of n squares are in place:
add one to k, move left one square and up one square and insert k
- (3) add one to k
- (4) move down one square from the last position and insert k.
- (5) go back to step 2 until all squares are filled

6	1	8
7	5	3
2	9	4

A multiple of n squares will be 3 6 9 for a 3 * 2 square. If you go left and you are at the start of a column you have to wrap around to the last column. If you are the bottom of the row and you have to move down one row then you wrap around to the first row. Call your class and file L8ex12.cpp.

LESSON 8 EXERCISE 13

Write a program that take any number and outputs it into words For example the number 123 will be converted into **one hundred and twenty three**. Call your class and file L8ex13.cpp.

LESSON 8 EXERCISE 14

Write a program that takes a numeric string like \$+-12,564.56 and checks it for validity and correct it if certain elements are missing. Your program should output "INVALID" if the string is invalid and cannot be corrected, else the corrected output string. If a decimal point is missing it will insert the decimal point and two zeros. If there is only one digit after the decimal point it will insert the extra digits. There should be a comma before every three digits, if the comma is missing then your program should insert it. Call your file L8ex14.cpp.

LESSON 8 EXERCISE 15

Write a program that asks the user to type in 10 letters into a character array. Your program should scan the array and look for the letters that repeat. Your program will report the number of the largest group of repeating numbers. The repeating numbers may be anywhere in the array and do not need to be sequential. If there are no letters that repeat then return the largest letter found in the array. Call your file L8ex15.cpp.

LESSON 8 EXERCISE 16

Write a function that receives 3 Strings. A string message, a string to search for and the other string to replace if the string is found. Your function declaration would be

```
const String& replace(const String& s, const String& f, const String& r);
```

For example the input strings are:

```
s: "it is cold today"
f: "is cold"
r: "will be very hot tomorrow"
```

returns: "it will be very hot tomorrow"

Call your file L8ex16.cpp.

LESSON 8 EXERCISE 17

Write a program that asks the user to type in 10 letters into an character array. Your program should scan the array and look for letters that repeat. Your program will report the letters of the largest group of repeating numbers. The repeating letters may be anywhere in the array and do not need to be sequential. If there are no letters that repeat then return the largest letter found in the array. Call your file L8ex17.cpp.

LESSON 8 EXERCISE 18

Write a program that calculates the number of 25, 10, 5 and 1 cent coins needed to make change from a sales transaction. Call your file L8ex18.cpp.

LESSON 8 EXERCISE 19

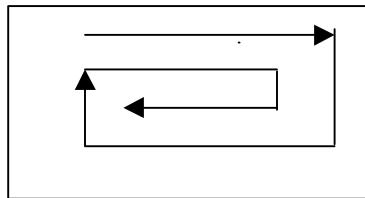
Write a program that has three arrays. One to store names one to store addresses and one to store salaries. Each array stores 10 items. Write a program that sorts the arrays by name, address or salary on demand. This means make a menu that the user selects for which way to sort. You can pre-initialize all arrays with values when you declare them. Hint: go through each array and print out the smallest value, use a fourth array to indicate which item you have used. Call your file L8ex19.cpp

LESSON 8 EXERCISE 20

Write a program that prints out any square array values as a spiral starting from any row or column. Call your file L9ex20.cpp.

1	2	3
4	5	6
7	8	9

1 2 3 6 9 8 7 4 5



LESSON 8 EXERCISE 21

Write a program where the user enters 3 sides of a triangle. From the entered sides determine if the triangle is **equilateral** all sides equal, **isosceles** where only two sides equal and **scalene** no sides are equal. Using can use Pythagorean theorem:

$$a * a = b * b + c * c.$$

Determine if the triangle is a **right triangle** if the largest angle is 90 degrees. An **acute triangle** if the largest angle is less than 90 degrees and an **obtuse triangle** if the largest angle is greater than 90 degrees. Finally determine if all the sides entered make a triangle where the sum all angles must add up to 180 degrees. Call your file L9ex21.cpp.

LESSON 8 EXERCISE 22

Make a checker board. 8 squares by 8 squares whereas each square alternates white and black.. Use the '|' char to bake a black square 8 characters wide by 3 characters high. Use the space character to make a white square.

```

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

```

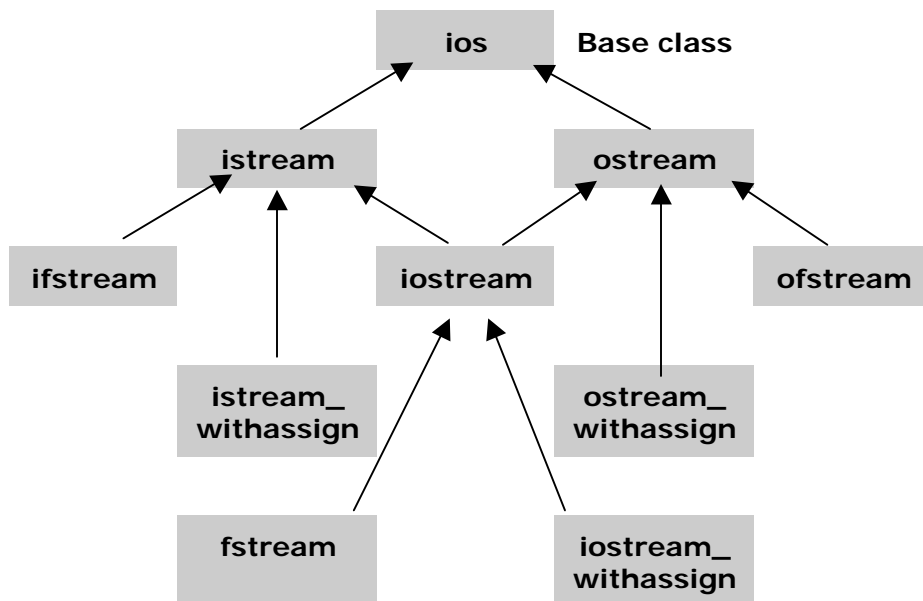
Use nested for loops. No cheating.

C++ PROGRAMMERS GUIDE LESSON 9

File:	CppGuideL9.doc
Date Started:	July 12, 1998
Last Update:	Mar 26, 2002
Version:	4.0

LESSON 9 C++ INPUT and OUTPUT STREAM CLASSES

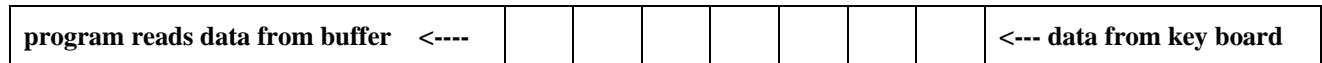
The input and output stream classes lets your program read data from an input device like a keyboard or file and send data to an output device like a terminal screen or output file. Data flows in streams. Data that comes from a device is known as an input stream. Data going to a device is known as an output stream. Devices are keyboards, data files or screens. A data buffer is used to store data from a device or program and send it when a program or device needs it. C++ has classes to manage input and output data streams. You must `#include <iostream.h>` when using the stream classes. The input stream class is **istream** and the output stream class is **ostream**. The **istream** and **ostream** classes are derived from the base class **ios**. The input file stream **ifstream** is derived from **istream** and the output file stream **ofstream** is derived from **ostream**. The **iostream** stream class is used for most I/O operations, **iostream** is derived from **istream** and **ostream**. The file stream class **fstream** is derived from **iostream**.



istream_withassign stream class is derived from **istream** and the **ostream_withassign** derived from **ostream**. The **iostream_withassign** class is derived from **iostream**. These classes enable you to assign one stream to another. For example, you may need to get input from a keyboard or a data file at the same time.

input stream buffer

An input stream buffer receives input from a device like a keyboard or file and stores the characters in a memory buffer until the program is ready to read the data contents. If the buffer was not present, then your program will have to ask the keyboard for data constantly. If the data arrives faster than the program can read it, then data will be missed. If the data arrives very slowly, then the program will be constantly waiting for data. A buffer transferees data from an input device to a program, independent of program operation.



output stream buffer

An output stream buffer sends output data to a device like a computer screen or file. The program sends characters to a memory buffer until the device is ready to read it. If the buffer was not present, the program will have to ask the output screen or file constantly if it ready to receive data. If the program sends data faster than the output device can read data will be missed. If the data arrives very slowly, then the output device will be constantly waiting for data. A buffer transfers data from a program to an output device independent of program operation. The output buffer needs to be flushed to send all the contents of the output data buffer to the output device.



stream objects

The **cin** object is automatically instantiated from the **istream** class when you include [<iostream.h>](#) in your program. **cin** is used to receive data from the keyboard. The **cout** object is automatically instantiated from the **ostream** class when you include [<iostream.h>](#) in your program. **cout** is used to send data to the screen. Make sure you use [#include <iostream.h>](#) on the top of your program before using **cin** and **cout**

ios class iostream.h

The **ios** class is the base for all stream classes and its purpose is to control the input and output stream buffers. **ios** has seven protected member variables that are used for pointers to buffers, output control, formatting and status. [#include <iostream.h>](#)

data type	member variable	default	description	accessor functions
streambuf*	strbuf		points to buffer stream	streambuf* rdbuf()const;
ostream*	tie	cout	points to ostream tied to istream (use to echo input to screen)	ostream* tie()const;
int	width	0	sets width of output field with padding ____5 (5 spaces inserted before number 5)	int width() const;
int	prec	6	sets precision for floats decimal points 5.543456	int precision()const
char	fill	' '	specifies fill character for padding field *****5	char fill()const;
long	flags	0x2001	holds format flags skipws, showpoint, left etc	long flags()const;
char	state	0x08	holds current io state good, eof, fail, bad	int rdstate()const;

strbuf and tie

strbuf points to the buffer stream used to hold the data. **tie** is used to tie an ostream to a istream. tie points to an output stream for cin. that cin can use echo input to the screen. We can get the output stream from cin using the accessor function tie() and use it to write a message to the screen.

```
ostream* out = cin.tie();
*out << "hello" << endl;
```

width, prec and fill

width, prec and fill are used with cout, each has a corresponding functions to set the values.

member	default	description	set function
width	0	sets width of output field with padding ____5	int width(int w) ;
prec	6	sets precision for floats decimal points 5.543456	int precision(int p)
fill	' '	specifies fill character for padding field *****5	char fill(char c)

width holds the width of the output field. The default field width is 0. In the following test program, we use the **width()** member function to print out the value of the width and a string for verification. Next we change the width to 20 and then print out the message again noticing the message is shifted right to accommodate a width of 20 characters. If the message is less than the width then the message is right justified and the preceding characters get padded with the fill character which has a default value of ' '. After printing out the message, the width returns back to its default value of zero. We print out the message again for confirmation.

```
// iostest.cpp
#include <iostream.h>

void main()
{
    // width
    cout << cout.width() << endl;
    cout << "message" << endl;
    cout.width(20);
    cout << "message" << endl;
    cout << "message" << endl;
}
```

```
0
message
message
message
```

fill sets the padding character, we can repeat the above code but this time set the fill character to '*'. We first print out the fill character which is a blank space.

```
// fill
cout << cout.fill() << endl;
cout.fill('*');
cout.width(20);
cout << "message" << endl;
```

```
*****message
```

The fill character must be set back to the default character before proceeding.

```
cout.fill(' '); // set fill character back to space
```

precision sets the number of digits displayed by a floating point number. It does not include the decimal point but includes all the whole number digits and fractions. The floating point numbers are default left justified.

```
// precision
cout << cout.precision() << endl;
cout << 335.146543546576 << endl;
cout.precision(20);
cout << 335.146543546576 << endl;
cout << 335.146543546576 << endl;
```

```
6
335.147
335.146543546576027
```

The precision must be set back to the default value before proceeding

```
cout.precision(6);
}
```

format flags

The format flags let you set the input scanning or output display to the options you want. The format flags are used for both stream objects **cin** or **cout**. An example set flags for left or right text justification,

```
// formatting flags
enum
{
    skipws    = 0x0001, // skip whitespace on input (default)
    left      = 0x0002, // left-adjust output
    right     = 0x0004, // right-adjust output (default)
}
```

```

internal = 0x0008, // padding after sign or base indicator
dec      = 0x0010, // decimal conversion
oct      = 0x0020, // octal conversion
hex      = 0x0040, // hexadecimal conversion
showbase = 0x0080, // use base indicator on output
showpoint = 0x0100, // force decimal point (floating output)
uppercase = 0x0200, // upper-case hex output
showpos  = 0x0400, // add '+' to positive integers
scientific = 0x0800, // use 1.2345E2 floating notation
fixed    = 0x1000, // use decimal point (123.45) floating notation
unitbuf  = 0x2000, // flush all streams after insertion
stdio    = 0x4000 // flush stdout, stderr after insertion
};

```

We can read the value of the format flag bits by using the **flags()** member function. The flag bits are set by using the **ios** member functions **flags()** and **setf()**. The flags bits are unset by using the **unsetf()** member function.

function	description	example using
long flags (long f);	sets flag bits to new setting without retaining all the previous settings	<code>cout.flags(ios::hex);</code>
long setf (long mask);	sets flag bits specified in the mask without clearing others (retain other settings)	<code>cout.setf(ios::hex);</code>
long unsetf (long mask);	clears the flag bits specified in the mask	<code>cin.unsetf(ios::skipws);</code>

You may set individual format flag bits :

```
cout.setf(ios::hex);
```

or set multiple bits by oring bits together :

```
cout.setf(ios::dec | ios::showpoint);
```

You may want to save the original format flag bits when changing bits so that they can be restored later.

```

int oldbits = cout.flags(ios::right);
cout << 5 << endl;
cout.flags(oldbits);

```

Examples of setting individual bits of the format flags using the setf() function:

flag bit	output format flags	example using	input / output
ios::skipws	skip leading white space between words (enter a message with leading spaces they are ignored. Set to ignore leading spaces write space not skipped)	<pre>char s[255]; cin >> s; cout << s; cin.setf(ios::skipws); cin >> s; cout << s << endl;</pre>	hello hello hello hello
ios::left	left justify output	<pre>cout.width(6); cout.setf(ios::left); cout << 5 << endl;</pre>	5
ios::right	right justify output	<pre>cout.width(6); cout.setf(ios::right); cout << 5 << endl;</pre>	5
ios::internal	right justify numeric output , left justify sign or radix pad middle for required width	<pre>cout.width(6); cout.flags(ios::internal); cout << -5 << endl;</pre>	- 5
ios::dec ios::showbase	set output display to decimal	<pre>cout.setf(ios::dec ios::showbase); cout << cout.flags() << endl;</pre>	8337
ios::oct ios::showbase	set output display to octal	<pre>cout.setf(ios::oct ios::showbase); cout << cout.flags() << endl;</pre>	020241
ios::hex ios::showbase	set output display to hex	<pre>cout.setf(ios::hex ios::showbase); cout << cout.flags() << endl;</pre>	0x20c1
ios::showpoint	turn trailing zeros on (zero suppression)	<pre>cout.setf(ios::dec ios::showpoint); cout << 56.0 << endl;</pre>	56.0000
ios::uppercase	uppercase output for hex and scientific notation	<pre>cout.setf(ios::hex ios::uppercase); cout << cout.flags() << endl;</pre>	0X23C1
ios::showpos	show "+" with positive integers	<pre>cout.setf(ios::showpos); cout << 5 << endl;</pre>	+5
ios::scientific	output scientific notation	<pre>cout.setf(ios::scientific); cout << 56.0 << endl;</pre>	+5.600000e+01
ios::fixed	output decimal point notation use with precision to set number of decimal points	<pre>cout.setf(ios::fixed); cout << 56.0 << endl;</pre>	+56.000000

LESSON 9 EXERCISE 1

Write a program that Ask the user to enter 5 decimal numbers and print all decimal numbers in a row where each number only displays 2 decimal point numbers. Call your program L9ex1.cpp. Hint you need to use `width()`, `fixed()` and `precision()`. Example:

entered numbers:	4553.8765 543.876 76545.89 7777.9 76654.988776
displayed numbers:	4553.88 543.88 76545.89 7777.90 76654.99

STREAM MANIPULATORS `iomanip.h`

stream manipulator is used to call functions that set the format flags bits or call functions from the `iostream` classes. stream manipulators add convenience so you can include them with **`cout`** or **`cin`**. For example, **`endl`** is a stream manipulator that calls a function to end a line and flush the output buffer.

```
cout << "hello" << endl;
```

You may need to `#include <iomanip.h>` when using some of the manipulators. The following chart lists all the stream manipulators:

manipulator	stream	action	example
binary	ios	set stream mode to binary	<code>cout << binary << 1234 << endl;</code>
dec	ios	read or write integer base10 (default)	<code>cout << dec << 56 << endl;</code>
endl	ostream	inset carriage return and flush	<code>cout << "hello" << endl;</code>
ends	ostream	insert '0' end of string terminator	<code>cout << 'x' << ends << endl;</code>
flush	ostream	flush output stream	<code>cout << "hello" << flush << endl;</code>
hex	ios	read and write use base 16	<code>cout << hex << 42 << endl;</code>
oct	ios	read or write using base 8	<code>cout << oct << 42 << endl;</code>
resetiosflags(long u)	ios	clear format flags	<code>cout << resetiosflags(0) ;</code>
setbase(int n)	ostream	write integers in base n	<code>cout << setbase(16) ;</code>
setfill(int ch)	ostream	set fill character to ch (default ' ')	<code>cout << setfill('*') ;</code>
setiosflags(long u)	ios	set format flags specified by u	<code>cout << setiosflags(ios::fixed);</code>

setprecision(int n)	ios	set floating point precision to n digits	cout << setprecision(6);
setw(int n)	ios	set field width to n	cout << setw(10);
ws	istream	skip white space	cin >> ws;

Here is two example programs one using ios junctions and set flags the other using stream manipulators.

```
#include <iostream.h>

// using ios and setf
void main()
{
    int x = 5;
    // left justification
    cout.setf(ios::left);
    cout << x << endl;
    // width, fill, right justification
    cout.width(6);
    cout.fill('*');
    cout.setf(ios::right);
    cout << x << endl;
    // precision
    double d =
    335.146543546576027;
    cout << d << endl;
    cout.precision(20);
    cout << d << endl;
}
```

```
#include <iostream.h>
#include <iomanip.h>
// using stream manipulators
void main()
{
    int x = 5;
    // left justification
    cout << setiosflags(ios::left) << x << endl;

    // width, fill, right justification
    cout << setw(6) << setfill('*') << setiosflags(ios::right)
    << x << endl;

    // precision
    double d = 335.146543546576027;
    cout << d << endl;
    cout << setprecision(20) << d << endl;
}
```

Program Outputs:

```
5
*****5
335.147
335.146543546576027
```

```
5
*****5
335.147
335.146543546576027
```


LESSON9 EXERCISE 2

Write a program that ask the user to enter 5 decimal numbers, and print out all decimal numbers in a row where each number only displays 2 decimal point numbers. Call your program L9ex2.cpp. Use all **stream manipulators**. Example:

entered numbers:	4553.8765 543.876 76545.89 7777.9 76654.988776
displayed numbers:	4553.88 543.88 76545.89 7777.90 76654.99

making your own stream manipulator

The ostream class includes the following overloaded insertion operator

```
ostream& operator<<(ostream& (*p)(ostream&))
{
    return (*p) (*this)
}
```

The parameter **p** is a pointer to a function that receives an ostream object. When we use a stream manipulator the overloaded insertion operator is called. **p** points to the stream manipulator function that is to be used. **p** calls the stream manipulator function and passes the cout object (*this) to it.

```
p->endl(*this);
```

An example to use is the **endl** stream manipulator function.

```
cout << "hello" << endl;

ostream& endl(ostream& ostr)
{
    ostr.put("\n");
    ostr.flush();
}
```

To make your own stream manipulator function, simply make a function that receives and returns a ostream& reference. For example we can make a **sp** (space) stream manipulator to insert a blank character when we need it.

```
ostream& sp(ostream& ostr)
{
    return ostr << ' ';
}
```

Now it is much easier to insert spaces between outputs.

```
cout << "56 << sp << "23" << sp << "goodbye" << endl;
```

```
56 23 goodbye
```

io state variables

The state variables hold the current state of the stream. The **state** variable is made up of 4 bit.

```
// stream status bits
enum io_state
{
    goodbit = 0x00, // no bit set: all is ok
    eofbit  = 0x01, // at end of file
    failbit  = 0x02, // last I/O operation failed
    badbit   = 0x04, // invalid operation attempted (conversion error)
    hardfail = 0x80  // unrecoverable error hardware failure
};
```

The state is access by the function **rdstate()** and changed by the **clear()** ios member functions:

variable	function	description	example using
state	<code>int rdstate();</code>	read stream state all state bits	<code>int status = cin.rdstate();</code>
state	<code>void clear(int =0)</code>	set state bits (default zero)	<code>clear();</code>

Here are functions to read each state bits:

bit	function	set	description	example
goodbit	<code>good()</code>	0	everything is ok	<code>if(cin.good()) break;</code>
eof bit	<code>eof()</code>	1	end of file	<code>if(cin.eof())break;</code>
failbit	<code>fail()</code>	1	last operation failed	<code>if(cin.fail())break;</code>
badbit	<code>bad()</code>	1	invalid operation	<code>if(cin.bad())break;</code>

If one of the bits are set your input and output stream is inoperable. You ill need to flush the buffer then clear the stream bits before you can use the stream again.

state operator functions

The following two **operator functions** (functions that have operator names like !) are used to test the state of the stream.

function	description	example
<code>operator void*()const;</code>	conversion operator	<code>while(cin) // returns true if no flags set same as cin.rdstate()</code>
<code>int operator!()const;</code>	calls fail()	<code>while(!cin) // loop while fail bit not set same as cin.fail()</code>

Use Control D (unix) or Control Z (dos) is used to set the eof state bit.

using cin

The **cin** object is instantiated automatically by the compiler from the **istream** class when you `#include <iostream.h>` at the top of your program.. **cin** uses the `>>` overloaded input stream operator to receive data from the keyboard:

```
int x; // declare variable x
cin >> x; // Variable x will receive data from the keyboard.
```

It is easy to remember which way the direction of the operator is. For **cin** `>>` is pointing to the variable that will receive the data. The **cin** object has other functions to assist you in extracting data from the keyboard.

function	description	example
<code>istream& get (char&);</code>	get next character from input stream	<code>cin.get(ch);</code>
<code>istream& get (const char*, int max);</code>	get a message from keyboard including white space, stop when "enter" is pressed or max characters read does not remove delimiter from input buffer	<code>cin.get(s,81);</code>
<code>void getline (const char*, int max);</code>	get a message from keyboard including white space, stop when "enter" is pressed or max characters read removes delimiter from input buffer but does not put in string	<code>char s[30]; cin.getline(s,sizeof(s));</code>
<code>istream& ignore (int n = 1, int delim = EOF);</code>	ignore the remaining characters of a line until the end of line or end of file is reached. removes delimiter from input buffer	<code>cin.ignore(sizeof(s),"\n");</code>
<code>int peek();</code>	looks but does not extract next character	<code>cin.peek();</code>
<code>istream& putback(char ch);</code>	inserts a character into the input stream	<code>cin.putback();</code>

Using >> and cin.getline() together

It is difficult to use >> and **cin.getline()** right after each other, because >> does not extract the '\n' and leaves it in the input stream. If you use **cin.getline()** right after using >> then **getline()** will extract the end of line character (EOL) '\n' and then terminate. No input data will be extracted from the input stream (keyboard). The remedy is to use a **cin.get()** or a **cin.ignore()** to remove the (EOL) '\n' from the input stream before using cin.getline().

improper use	remedy using >> and getline() together
<code>cin >> s;</code>	<code>cin >> s;</code>
	<code>cin.get() or cin.ignore(sizeof(s), '\n')</code>
<code>cin.getline(s, sizeof(s));</code>	<code>cin.getline(s1, sizeof(s));</code>

using state bits to catch errors

You have to use the state bits to catch errors when the user types in the wrong input characters on the key board. You may ask for a **double** data type but the user types in a character. Your program may go into an endless loop. You will need to recover from this situation. The following program demonstrates how to catch and recover from bad entered input data or and end of stream error. Use Control D (unix) or Control Z (dos) is used to set the eof state bit.

```
#include <iostream.h>
// program to catch bad input characters
void main()
{
    double d;
    const int MaxLine = 80;
    cout << " enter number : ";
    cin >> d; // get number from keyboard

    while(!cin.eof()) // loop till correct data entered or end of file key pressed
    {
        if(cin.rdstate()) // check for any state bit set
        {
            cin.clear(); // clear state bits
            cin.ignore(MaxLine, '\n'); // flush input buffer
            cout << endl << "please re-";
        }

        cout << " enter number : ";
        cin >> d; // get number from keyboard
    }
    cin.clear(); // clear state bits
    cin.seekg(0); // set buffer pointer to start of buffer (optional)
    cin >> d; // press any key to continue
}
```

using cout

The **cout** object is instantiated automatically by the compiler from the **ostream** class when you write `#include <iostream.h>` at the top of your program. **cout** uses the `<<` the overloaded operator to send data to the screen. It works for all C++ data types

```
int x = 5;

cout << x; //puts the value of x on the screen
```

It is easy to remember which way the direction of the arrow stream operator is. For `cout <<` is pointing towards `cout` (the screen output). The following are functions belonging to `cout`:

function	description	example	output
<code>ostream& flush();</code>	empties output buffer send to device	<code>cout. flush();</code>	
<code>ostream& put(char ch);</code>	output a single character	<code>cout.put('a'); cout.put(c);</code>	a
<code>ostream& write (const char*, int n);</code>	write string to output stream up to max characters	<code>cout.write(s, sizeof(s));</code>	"hello there"

LESSON9 EXERCISE 3

Write a program that asks them what kind of data they want to type in, then asks them to type in a value. If they type in the wrong data, tell them and let them again re-enter the value. Call your program `L9ex3.cpp`. Example output:

```
Select data type:

(1) number

(2) decimal number

(3) character string

2
Enter your value now: 20.A5
You made a mistake try again !
```

FILE STREAM CLASS `fstream.h`

All file streams classes come from **fstream** class defined in `#include <fstream.h>`. To write to a file you create an **output** file stream using the **ofstream** class. To read a file you create an **input** file stream using the **ifstream** class. The input stream operator `>>` and output stream operator `<<` are also overloaded for file stream classes.

creating and using output file stream functions:

writing to files	example
make a output file stream object	<code>ofstream fout;</code>
make a output file stream object and open a file	<code>ofstream fout("testout.dat");</code>
to open a file for writing: (output)	<code>fout.open("testout.dat");</code>
test if output file is opened:	<code>if(!fout)return;</code>
write to your open output file:	<code>fout << x;</code>
you have to close each file after using them:	<code>fout.close();</code>

creating and using input file stream functions:

reading files	example
make a input file stream object	<code>ifstream fin;</code>
make a input file stream object and open a file	<code>ifstream fin("testin.dat");</code>
to open a file for reading: (input)	<code>fin.open("testin.dat");</code>
test if input file is opened (test ios fail bit)	<code>if(!fin)return;</code>
test if at end of input file (EOF)	<code>if(fin.eof())break;</code>
read data from your input open file:	<code>fin >> x;</code>
you have to close each file after using them:	<code>fin.close();</code>

You can open a console stream with:

```
ofstream out;
out.open("con");
out << "hello" << endl;
```

Hello

Example program opening and reading a file:

```

// opening, writing and reading text files
#include <iostream.h>
#include <fstream.h>

void main()
{
    int x = 5; // declare and initialize an integer

    // create a file output stream and open file for writing
    ofstream fout("test.dat");

    // test if file opened
    if(!fout)
    {
        cout << "file test.dat cannot be opened for writing " << endl;
        return;
    }

    fout << x; // write a character to the file

    fout.close(); // close output file

    // create a file input stream and open file for reading
    ifstream fin("test.dat");

    // test if file opened
    if(!fin)
    {
        cout << "file test.dat cannot be opened for reading " << endl;
        return;
    }

    fin >> x; // read first character from the file

    // loop till end of file encountered
    while(!fin.eof())
    {
        cout << "read " << x << " from file" << endl;
        fin >> x; // read next character from the file
    }

    // close all file
    fin.close();
}

```

read 5 from file

LESSON 9 EXERCISE 4

Write a program that asks the user to enter words from the keyboard. Open up a text file for writing and write the words to the file. Close the file and then reopen the file for reading text. Print out the contents of the file. Make an input file stream object called **fin** and a output file stream object called **fout**. Use **cin** to get information from the keyboard. Use **fin** to get information from the file. Use **fout** to print the information to the data file, don't forget to use **endl**. Use **cout** to print the information to the screen. Call your data file L9ex4.dat. Call your program file L9ex4.cpp

USING FILE SPECIFIERS WITH FSTREAM CLASS fstream.h

File specifiers let you state what options you want when you open your file with. You may use many file specifiers to get many desired options by **oring** them together. (**ios::in** | **ios::out** | **ios::binary**). File specifiers are used with the **fstream** class. You may specify a file to be opened as an input, output or both input and output. You may open a file as a text file (default) or as a binary file. A text file includes only printable characters where as a binary file includes printable and non-printable characters.

specifier	description	example use
ios::in	open file for reading	<code>fstream fin("testout.dat".ios::in);</code>
ios::out:	open file for writing	<code>fstream fout("testout.dat".ios::out);</code>
ios::binary	open file in binary mode	<code>fstream fin("testout.dat", ios::binary);</code>
ios::app	when a file is opened for write the new data is appended to end of file rather than deleting the contents of the opened file first	<code>fstream fout("testout.dat".ios::app);</code>
ios::ate	places file pointer at end of file when opened (file pointer at end of file)	<code>fstream fout("testout.dat",ios::ate);</code>
ios::trunc (default)	causing existing file contents to be deleted when it is opened for writing	<code>fstream fout("testout.dat",ios::trunc);</code>
ios::nocreate	file must exist to open or error reported (handy for input files that don't exist)	<code>fstream fout("testout.dat",ios::nocreate);</code>
ios::noreplace	file already exists or error reported	<code>fstream fout("testout.dat".ios::noreplace);</code>

TEXT FILE STREAM

The `getline()` function can be used with file streams. `Getline()` will get a whole message (record) from a file until the end of line terminator is reached (EOL) '\n' for DOS or "\n\r" for UNIX is reached. **cin** only gets you individual words.

Example to read a record from a file:

```

// write and read a record from a file
#include <iostream.h>
#include <fstream.h>

void main()
{
    const int MaxLine=80; // maximum size of a record
    char line[MaxLine+1]; // make space to hold a message

    fstream fio("L9ex2.dat",ios::out); // open file for writing

    // test if file opened
    if(!fio)
    {
        cout << "file L9ex2.dat cannot be opened for writing " << endl;
        return;
    }

    cout << "enter a message:" << endl; // ask user to enter a message
    cin.getline(line,MaxLine); // get message
    fio << line; // write message to file

    fio.close(); // close file

    fio.open("L9ex2.dat",ios::in); // open file for reading

    // test if file opened
    if(!fio)
    {
        cout << "file L9ex2.dat cannot be opened for reading" << endl;
        return;
    }

    fio.getline(line,MaxLine); // read first record from file

    // loop till end of file
    while(!fio.eof())
    {
        cout << line; // print out record
        fio.getline(line,MaxLine); // read next record from file
    }

    fio.close(); // close file
}

```

```

enter a message:
hello there
enter a message:
goodbye
hello there
goodbye

```

LESSON 9 EXERCISE 5

Write a program that asks the user to enter different data types like **int**, **double**, **char**. Write all the different data types to the file all in one line using a **fstream**. (End the line with **endl**.) Make sure you use spaces between the data types and a **endl**. Read in the line using `getline()` and print to the screen. Call your data file `L9ex5.dat`. Call your program file `L9ex5.cpp`.

SEQUENTIAL BINARY FILES

You use the stream **write()** and **read()** functions to write and read binary data to and from a file. The functions do not look for end of line character to stop reading or writing instead they read or write the number of bytes requested.

binary file function	description	example using
<code>ostream& write (const char* mem, int num)</code>	write binary data to file stream, <i>num</i> is the number of bytes to write	<code>fout.write((char*)record, strlen(record)+1);</code>
<code>istream& read(char* mem, int num);</code>	read binary data from file stream, <i>num</i> is the number of bytes to read	<code>fin.fread((char*)record, MaxLine);</code>

RANDOM FILE ACCESS

Random access allows you to go to a particular file location to get a particular record when you are reading and writing files. This is different from sequential access where you have to read or write each record one by one. We also use the **read()** and **write()** functions in random access. To go to a particular file location, we use the **seekg()** function to specify the file location offset in number of bytes when reading a file. We use the **seekp()** function to specify the file location offset in number of bytes when writing a file. We can specify the offset from the start of the file, from a relative starting location, or from the end of the file:



If you want to know where the file pointer is use the **tellg()** and **tellp()** functions, they both return a long.

How would you determine the length of a file ? Hint: use **seekg()** and **tellg()**.

random access file functions: (**NOTE** they may only work on binary files properly)

functions	description	
isstream& seekp (long pos)	absolute byte position number of bytes from start of file for writing	
istream& seekp (long offset, seek_dir dir);	specify a relative offset from start, current or end of file for writing	
long tellp ();	return the put file pointer position	
isstream& seekg (long pos)	absolute byte position number of bytes from start of file for reading	
istream& seekg (long offset, seek_dir dir);	specify a relative offset from start, current or end of file for reading	
long tellg ();	return the get file pointer position	
<div> <p>Instead of accessing data from the file sequentially you can access data any place you want</p> </div>	seek offset specifiers	description
	ios::beg	specifies offset from beginning of file
	ios::cur	specifies offset from current position of file
	ios::end	specifies offset from the end of file

An Example using a random access to read a record, and write it back to the same location:

```
// write a record and read back
#include <iostream.h>
#include <fstream.h>

void main()
{
    const long RecordSize = 80; // max number of characters in a line
    long recordNum = 1 * RecordSize; // point to record position
    char record[RecordSize]; // declare record

    // initialize record with all printable characters
    for(int=0; i<RecordSize; i++) record[i] = ' ' + i;

    // open file for input, output, binary
    // (notice multiple file specifiers)
    fstream fio("test.dat", ios::in|ios::out|ios::binary);

    // test if file opened
    if(!fio)
    {
        cout << "file test.dat cannot be opened " << endl;
        return;
    }
}
```

```

fio.seekp(recordNum,ios::beg); // seek to record to write
fio.write(record,record_size); // write record to file

for(int i=0;i<RecordSize;i++)record[i]= ' '; // clear record

fio.seekg(RecordSize*-1,ios::cur); // seek to read (go back)
fio.read(record,RecordSize); // read record from file

// print out each character in record
for(int i= 0;i<RecordSize;i++)cout<<record[i];

cout << endl;
fio.close(); // close file
}

```

You need to do a **seekg()** to set a location to read from a file. You need to do a **seekp()** to set a location to write to a file. You do not need to do a **seekg()** or a **seekp()** for continuous reads or for continuous writes from a file.

LESSON 9 EXERCISE 6

Ask the user to type in a sentence of about 10 words. Open a file for read/write binary. Write the words for the sentence to a binary file, the words should all be of the same length. Ask the user to type in a number between 1 and 10. Calculate the file position use **seekg** and read the selected word from the file. Ask the user to type in another number between 1 and 10. Calculate the file position use **seekp** to write the new contents back to the file. Repeat the selection process 5 times., then print out the contents of the entire file. Don't forget to close the file at the end of your program. Call your data file L9ex5.dat. Call your program file L9ex6.cpp.

DELETING FILES

Sometimes you need to delete a file. You can only do this using the **unlink** function.

Syntax

```

#include <io.h>

int unlink(const char *filename);

```

Description

unlink deletes a file specified by filename. Any drive, path, and file name can be used as a filename. Wildcards are not allowed. Read-only files cannot be deleted by this call. To remove read-only files change the read-only attribute of the file. If your file is open, be sure to close it before unlinking it.

Return Value

On success, unlink returns 0.

On error, it returns -1 and sets the global variable **errno** to one of the following values:

EACCES	Permission denied
ENOENT	Path or file name not found

Example Program to delete a file:

```

#include <iostream.h>
#include <fstream.h>
#include <stdio.h>

int main(void)
{
    ofstream fout("test.dat");    // open file for writing

    if(!fout)
    {
        cout << "file could not be opened for writing" << endl;
        return 1;
    }

    fout << "testing" << endl; // write something
    fout.close(); // close file

    ifstream fin("test.dat",ios::nocreate);    // open file for reading

    if(!fin)
    {
        cout << "file could not be opened for reading" << endl;
        return 1;
    }

    char line[81];

    while(!fin.eof())
    {
        fin >> line;    // read something
        cout << line << endl;
    }

    fin.close(); // close file
    unlink("test.dat"); // delete file (it's gone)
    fin.open("test.dat",ios::nocreate); // open file for reading

    if(!fin)
    {
        cout << "file could not be opened for reading" << endl;
        return 1;
    }

    return 0;
}

```

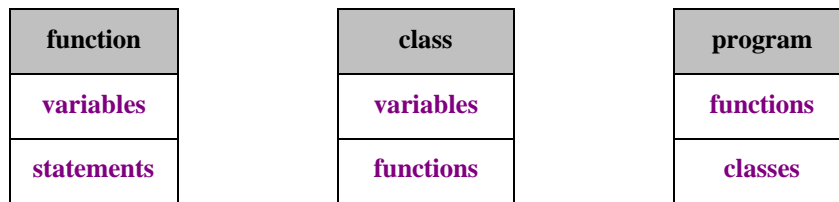
C++ PROGRAMMERS GUIDE LESSON 10

File:	CppGuideL10.doc
Date Started:	July 12, 1998
Last Update:	Mar 26, 2002
Version:	4.0

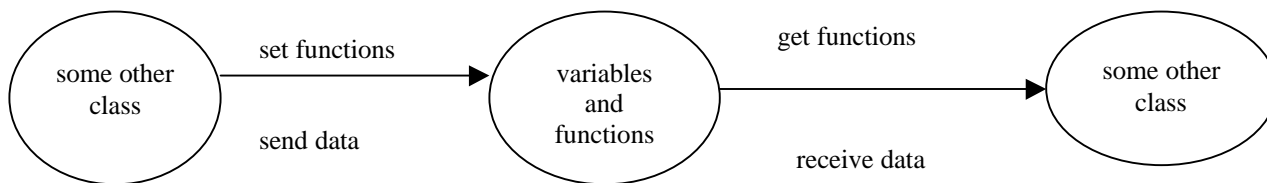
LESSON 10 ENCAPSULATION AND INHERITANCE

CLASSES AND ENCAPSULATION

The **Object Oriented Programming** approach allows a program to be more organized. Functions having common operations can be grouped together in a **class**. A class is like a factory, a car assembly plant, furniture plant etc. Each factory performs a specific operation. Since there are different kinds of factories with certain operations then there must be different kinds of classes with certain operations. Each factory has different **departments** to do specific tasks so that the factory can operate. Classes have **functions** to do specific tasks so that the class can operate.



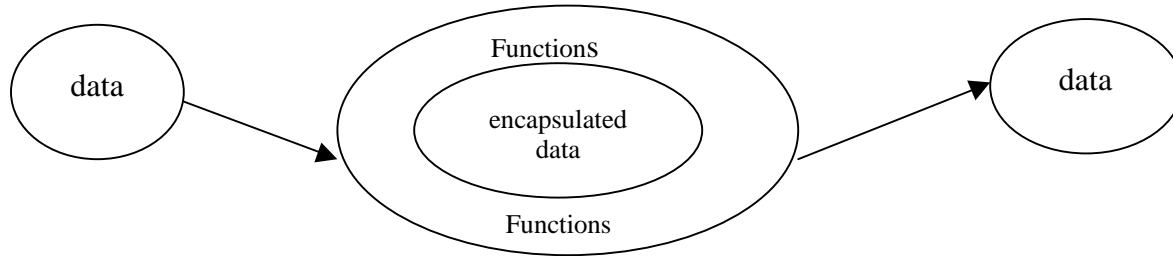
A factory must get some input raw materials, do some manufacturing and then **produce** a product like a car or a washing machine. In a C++ program a class must get data from the keyboard or a file, process the input data by doing a calculation, and then produce an output on the screen or store the output in a file for future use. A class will have **set functions**, **process functions** and **get functions**. The **set** functions receive the data, the **process** functions do calculations on the data and the **get** functions output the data.



All data values will be stored internally in the class in **private variables**. Private meaning no one else can use these variables, only the class. Think that the variables are the inventory parts in the storeroom of a car assembly plant. If they were made **public** then every one would use them and after a couple of days the stock department will be empty. The factory could not make any more cars and the plant would shut down. Private variables make good sense. If they were **public** other classes or functions could use them and maybe corrupt the data. In this case the program will crash and the computer will shut down.

encapsulation

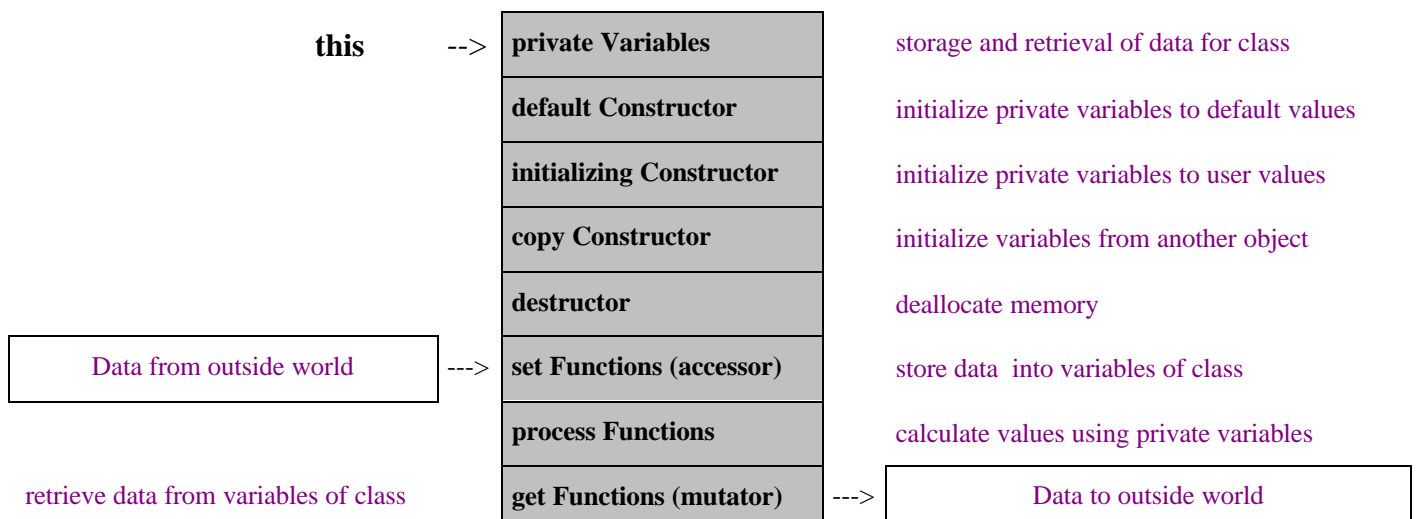
A class has **private** variables the **functions** are used to get, do calculations and put data into these variables. Variables and functions are said to be **encapsulated** in a class. The class **contains** the variables and functions. For encapsulation to work all data must be stored in private variables in the class. Encapsulation is also called **data hiding**. The class will manage all operations on the data. The advantage of **encapsulation** is that the people retrieving this data do not need to know how the data is stored internally, what data base or data structure they are using. They just want to access the data!!! The class will handle the entire overhead in managing the data, this takes the burden away from the programmer. Encapsulation forces you to exchange data through functions.



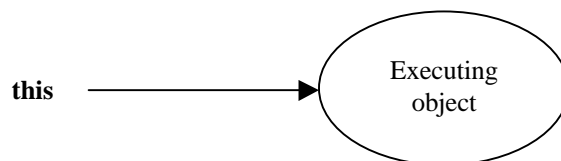
communication between classes

Think that the communication between classes are roads and the data is the raw material being delivered to the factory by delivery trucks. The delivery trucks bring the raw material (which is the data), to the receiving department (which is the set functions). The factory manufactures the products (which is the process functions). When the product is ready for delivery the pickup trucks pick up the data from the shipping department (which is the get functions). Always think of a class as a factory with specific operation to do. If you have to do many different things then you would need many different classes. By having many different classes, each performing a specific operation, your program to be highly organized and efficient. The classes exchange their data through functions.

Class format:



A class is a collection of common things. A class provides the definition that **objects** are made of. A class lists the variables and functions an object needs. An object is allocated or reserved memory for data members of the class definition. The functions are compiled into execution code and stored in the computer. Each compiled function belongs to a certain class. Only those functions belonging to a certain class can act on objects created from the same class. Before a class object can operate the private variables need to be initialized and additional memory may need to be allocated. Initialization and memory allocation is done by **constructors**. Constructors have the same name as the class. Constructors are called only once when a class object is **instantiated**. Instantiate means to create an **object** of this class. When a class is instantiated memory is allocated for the variables of the object. You can have many constructors each having a different purpose. **Default constructors** have no input parameters and is used mainly to initialize variables to default values like 0 and NULL. **Initializing constructors** have 1 or more parameters, and the parameters are used to initialize the private variables to user defined values. **Conversion constructors** convert one data format to another data format. For example time can be represented by a string "8:30 pm" or a number like 2030. If the class stores time as a number, then you will need a conversion constructor to convert time in a string representation to time in a numeric format. **Copy constructors** copy an existing object. When you copy an existing object, the new object will get all the data values of the object that was copied. Copy constructors must allocate memory before receiving the new data values. **Destructors** are called when the class object is no longer needed. The purpose of the destructor is to deallocate memory that was allocated in the constructor. If you did not allocate memory in a constructor then the destructor does not have to de-allocate memory. The destructor will then have no statements and it is then known as a **default constructor**. You should always have at least a default destructor, even if you do not have to de-allocate memory. What is the difference between a class and an object ? A **class** provides the definition that tells the compiler how to construct the object. An **object** contains the memory space for the variables defined in the class definition. A class definition is like the recipe to bake a cake. The cake when baked becomes an object. **Set functions** also called **mutator** functions are used to set data values of the object. **Process functions** do calculations. **Get functions** also called **accessor** functions, send data from one object to other objects. Objects exchange data through functions. When your program is running and objects are being executed a pointers the current object being used is automatically supplied by the compiler. The pointer is known as **this**. You can use it to access methods and variables of the executing object.



Example Person class

A Person class will have a name, address and age instance variables.

Person class{	String Name;	// person's name
	String Address;	// person's address
	int age;	// person's age

The constructor will initialize the Person's name, address and age. The Person class has functions to access instance variables and increment the age.

The class definition file is presented first before the class implementation code file. The definition file is also called a header file. The header file contains the class definition. The class definition contains the variable and function declarations defined for the class. The definition header file has the extension **".hpp"** and implementation file has the extension **".cpp"**. When you have many classes in a programming project, it is good practice to use separate files for the class definition and the class implementation. Your program will be more organized and easier to debug, read and maintain. It is easier to work with many short files than one long file. The Vehicle Plant class also uses the String class for character strings. We have included a String class for you at the end of this lesson. We use the compiler directives:

```
#ifndef __PERSON // which means if not defined 'file_name' and
#define __PERSON // which means define 'file_name'
```

These make sure the class definition header file is included only once when we use the **#ifndef** and **#define** compiler directives. **#ifndef** means if the label is not define include the following code and **#define** means define the label. If **__PERSON** is not defined then define it and include the header file. If it is defined do not include the header file. If the header file is included more than once then multiple definition errors will occur. The header files ends with **#endif** to signify the end of the **#ifndef** block.

class definition file:	person.hpp
class implementation file:	person.cpp

A class to represent a Person is defined as follows: The Person class header file is as follows:

```
// person.hpp
#include "string.hpp" // string class definition file

#ifndef __PERSON // prevent multiple includes
#define __PERSON

// Person class info
class Person

{
    // person info
    private:
    String Name; // name
    String Address; // address
    int Age; // age

    // functions for everyone to use
    public:

    // default constructor
    Person();

    // initializing constructor
    Person(String& name, String& address, int age);

    // copy constructor
    Person::Person(Person& p2);
```

```

// destructor
~Person();

// get Person's name
String& getName();

// get Person's address
String& getAddress();

// print person info
virtual void print();

// get age
int getAge();

// increase Person's age
void incAge();
};

```

```
#endif
```

The Person class implementation file is as follows:

```

// Person.cpp
#include <iostream.h>
#include "person.hpp"

// default constructor,
Person::Person()
{
    Name="";
    Address="";
    Age = 0;
}

```

Rule for calling header files in implementation (.cpp) files:
 Do not call any header file in the cpp file except your class header file and system header files. Always call other program header files in the .hpp file.

```

// initializing constructor, set all variables to user specified values
Person::Person(String& name, String& address, int age)

```

```

{
    Name = name; // initialize person's name
    Address = address; // initialize person's address
    Age = age; // initialize persons age
}

```

```

// copy constructor, copy an existing object
Person::Person(Person& p2)

```

```

{
    Name = p2.Name; // initialize person's name
    Address = p2.Address; // initialize person's address
    Age = p2.Age; // initialize persons age
}

```

```

// default destructor
Person::~Person() {}

// return Person's name
String& Person::getName()

{
    return Name; // return name
}

// return Person's address
String& Person::getAddress()

{
    return Address; // return address
}

// return Person's age
int Person::getAge()

{
    return Age; // return vehicle as type
}

//print out Person's info
void Person::print()

{
    cout << "name: " << Name << endl;
    cout << "address: " << Address << endl;
    cout << "age: " << Age << endl;
}

```

Why do we return a String reference in **getName()** and **getAddress()** ? We use return a String reference in **getName()** and **getAge()** because we are returning a variable declared in a class definition that has a permanent memory location.

```

// increase Person's age
void Person::incAge()
{
    Age = Age + 1; // increment age
}

```

LESSON 10 EXERCISE 1

Type in the Person class. You may want to add functions setName() and setAddress() just in case the person changes their name and moves. Write the **main function** to make Person objects. Print out the objects before and after incrementing the age. Put your main function in a file called L10ex1.cpp.

You should have the following files:

class/function	definition	implementation	execution
Person	person.hpp	person.cpp	
main		L10ex1.cpp	L10ex1.exe
String	string.hpp	string.cpp	

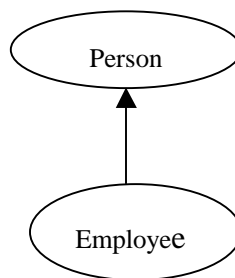
With inheritance you can take any class and add more functionality to it

INHERITANCE

Classes have the ability to use functions and variables from another class. The class using the functions and variables is called the **derived** class. The class providing the functions and variables is called the **base** class. **Derived** classes **inherit** variables and functions from the **base** class. This means the derived class can use variables and functions of the base class. It's like the subsidiary store using the sales clerks from the main office to help out at rush hour. In programming the goal is to avoid repetition, not only is a long repetitions program difficult to follow but it involves a lot of typing. To avoid repetition **inheritance** comes to the rescue. An good example of inheritance is a Employee class and a Person class. The employee class can inherit all the variables of a Person class, since an employee would also need a name, address age etc. The class that is being inherited is known as the **base** class. The **base** class has all the **common** variables. The Person class contains the common variables like name address and age that another class would like to use. The class that inherits the **base** class is known as the **derived** class. The **derived** class has all the **unique** things. The employee has salary that is unique. Inheritance gives base classes extended capabilities.

An Employee class is derived from the Person class.

Inheritance adds functionality to any class and avoids repetition by using existing classes.



base class
(common things)

derived class
(unique things)

The arrow points from the derived class to the base class indicating the derived class can access the base class.. Inheritance is a 1 way street, the derived class can inherit variables and functions from the base class but the base class cannot inherit variables and functions from the derived class. With inheritance instantiation is automatic.

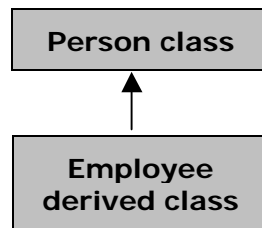
What are the benefits of inheritance ?

- (1) avoid repetition
- (2) smaller code
- (3) use of existing variables and functions
- (4) classes get extended capabilities

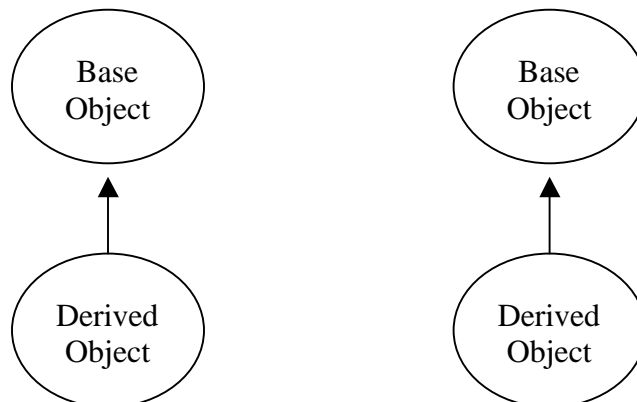
derived classes

The Employee class may inherit the Person class. The Person class will have all the common things like name, address and age. The Employee class will have the unique things like salary. When the Employee class inherits the Person class it can use all the non private variables and functions.

Now an Employee will have name, address, age and salary, Inheritance avoids repetition. The class with the common things is called the **base class**. Our base class is the Person class. The classes that have things that are **unique** and more specialized are called the **derived class**. Our derived class in the Employee class. The derived class uses the functions of the base class for all **common** tasks.



The arrows in the diagram point to the base class. This means the Employee class can use common functions and variables from the Person class. The arrows force the direction that the base class can not use variables and functions of the derived class. In C++ every time you make a derived object a base object is also made, because the derived class is inherited from the base class! The derived object needs to use the variables and methods of the base object. When you create another Employee object you automatically get another Person object.



What gives ??? The popular misconception about **inheritance** is that only one base class is created, but in reality for every derived class created, another base class is also created. You cannot derive a derived object if the base object does not exist. Since the derived class is derived from the base class. A base object always needs to be created when derived classes are instantiated. Inheritance was design to avoid repetition. All it is saying is to create additional objects that use common elements of a base class, and for every derived object created, create a corresponding base object. Inheritance does not mean we only want one copy of a base object and many derived objects. If we had only one copy of the **base** object for all the **derived** object, then each derived object would share and could change the data of the **base** object. This is not what we want. We want a separate copy for each base object, this way data can be independent, to be accessible by the **derived** object. C++ does have mechanisms to create only 1 base object and many derived objects, you will learn about these mechanisms in future lessons. The base class contains the common variables and functions needed by the derived class. The programmer now has to its disposal, the variables and functions of the **derived** class and the variables and functions of the **base** class. Without the base class the programmer would have to type in the same functions and variables to creates a Employee class. Without the base object the derived object could not operate.

base class	contains common variables and functions
derived class	access variables and functions of base class. contains unique variables and functions

writing and using the derived class

To derive a class from a base class, you list the base class name followed by a ":". You may derive a class from more than one base class. This is called **multiple inheritance**. You list the each base classes separated by commas.

class derived_class_name : access_protection_modifier base_class_list.

class Employee : public Person // derive a Employee class from the Person class

The **access protection modifier** sets the **visibility** of the base class variables and functions for the derived class. If the access protection modifier is protected then all functions and variables of the base class will appear protected or private to the derived class. The following chart shows how the base class appears to the derived object for access protection modifiers public, protected and private. Prohibited access applies to a private visibility that cannot be accessed.

derived class access protection modifier	base class visibility		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	prohibited

**How the base class
variables and
functions appear to
the derived class**

You only get a compile error if a derived class object calls a function or variable of the base class outside the derived class. The derived class functions can still access the variables and other functions of the base class with no effect from the visibility modifier. Example:

```
Employee emp;    // create derived object Employee
emp.getName();   // Employee object wants to access getName() method of Person class
```

Employee class

Inheritance allows you to access the **non-private** variables and functions of the base class. The Employee class inherits the Person class. Here is the Employee class definition:

```
// employee.hpp
#include "person.hpp"
#include "string.hpp"

#ifndef __EMPLOYEE // prevent multiple includes
#define __EMPLOYEE

// Employee class is derived from Person class
class Employee: public Person
{
private:
float Salary;

public:
// default constructor
Employee();
// initialize constructor
Employee(String& name, String& address, int age, float salary);
// copy constructor
Employee(Employee& emp2);
// return salary
String getSalary();
// print
void print();
};

#endif
```

derived class constructors

The derived class constructors are declared just like an ordinary constructor.

```
// default constructor
Employee()

// initialize constructor
Employee(String& name, int numVehicles, Inventory* invent);

// copy constructor
Employee(Employee& plant2);
```

implementing derived class constructors

The derived class constructor must call the base class constructor to initialize its variable values and if necessary to allocate memory. Failure to do so will cause your program to crash. You call the base constructor by using the ":" initializing colon operator followed by the base constructors name and argument list. The derived constructor must supply the arguments for the base constructor.

```
derived_class_constructor_definition : call base_class_constructor

Employee::Employee() : Person()
```

The derived class default constructor calls the default base class constructor:

```
// derived class default constructor
Employee::Employee() : Person()// call base class default constructor
{
    Salary = 100; // initialize salary to default value
}
```

The derive class initializing constructor calls the base class initializing constructor and sends it parameters as arguments to the base class initializing constructor. The derived class must call the derived class constructor to initialize the variables of the base class or else they will never get initialized.

```
// initializing constructor
Employee::Employee(String& name, String& address, int age, float salary)
: Person(name, address, age) // call base class initializing constructor
{
    Salary = salary; // initialize salary
}
```

When the derive class copy constructor calls the base class copy constructor it only sends the base class part of itself to the base class. Typecasting is not necessary because we are converting a derived class to a base class. This is known as a **upward** conversion.

```
// copy constructor
Employee::Employee(Employee& emp2): Person(emp2) // call base class copy constructor
{
    Salary = emp2.Salary; // initialize salary from Employee object
}
```



```
// print employee info
void Employee::print()
{
    cout << "name: " << Name << endl;
    cout << "address: " << Address << endl;
    cout << "age: " << Age << endl;
    cout << "salary " << Salary << endl;
}
```

protected variables and functions

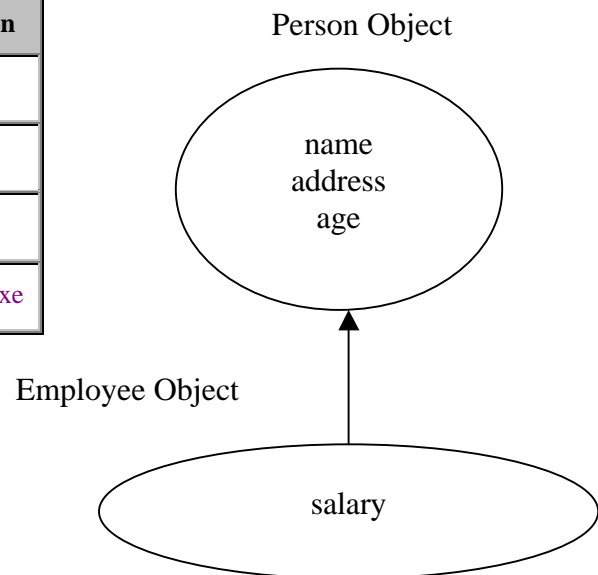
The functions and variables of the base class must be protected or public to be accessed by the derived class. Protected means only the derived classes can use them nobody else. Usually functions of the base class that the derived class uses are protected. In our application we make the variables protected so that the derived class can use them. We make functions of the base class public so that other classes can use. If they were made also made protected then these functions could not be used by other classes

Private	only this class can access variables and functions of this class
Protected	only this class and any derived class can access variables and functions of this class
Public	Everybody can access variables and functions of this class

LESSON 10 EXERCISE 3

Type in the Employee derived class. You will need the Person class and the provided String class, do not use char* or char[]. Write the main function to instantiate employee objects. Use the main function to call functions from the Employee objects and print out the information. Call functions from the base class and derived class. Call your program file with your main function L10ex3.cpp. Put all your work in separate files as follows:

class/function	definition	implementation	execution
Person	Person.hpp	Person.cpp	
Employee	Employee.hpp	Employee.cpp	
String	string.hpp	string.cpp	
main		L10ex3.cpp	L10ex3.exe



derive class is calling functions of the base class

Notice both the Person class and Employee class has a **print** method(). When you instantiate a Person object the **print()** method of the Person class is called. When you instantiate an Employee object the print method of the Employee class is called. The derived class may call the same named functions of the base class by using the class name and resolution operator "::" For example, the print class of the derived class can call overloaded functions of the base class by using the resolution operator "::".

```
Person::print(); // call print function from base class
```

This way the derived class can print out unique information and then call the base class to print out all the common information.

base class calling functions of the derived class

If you create an instance of the base class you **cannot** call the overridden functions of the derived class. Do you know why ? The answer is quite simple. If you create an instance of the base class then there is no derived class !!! There are no derived functions to call.

LESSON 10 EXERCISE 3

Rewrite the print function of the Employee class **print()** function so that it calls the **print()** function the Person class.

THREE WAYS TO INSTANTIATE OBJECTS

Objects can be instantiated three ways as follows:

(1) instantiating a derived class object

You instantiate a Employee just as you would any class.

```
Employee emp1; // as a variable representing an default object
```

```
// as a variable initialized object}
```

```
Employee emp2(String("Tom"),String("121 College St",25,10000);
```

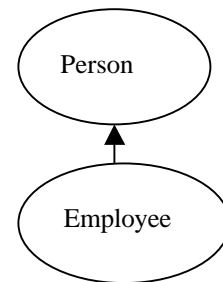
```
Employee emp3(emp2); // as a variable copying an existing object
```

```
Employee* pemp4 = new Employee(); // as a pointer to a default object
```

```
// as a pointer to an initialized object
```

```
Employee* pemp5 = new Employee(String("Tom"),String("121 College St",25,10000);
```

```
Employee* pemp6 = new Employee(*pemp5); // as a pointer copying an existing object
```



When you instantiate a derived class it contains a base class object component. A derived class can automatically access the non-private variables and functions of the base class because the derived class inherits them.

extracting the base object (upward conversion)

You can extract the base class component. This is known as an **upward** conversion.

```
Person p2 = emp2; // extract base class from derived class
```

```
Person* pp5 = emp5; // get pointer to base class
```

Some people typecast, but it is not necessary.

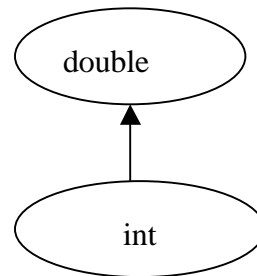
```
Person p2 = (Person) emp2; // extract base class from derived class
```

```
Person* pp5 = (Person*) emp5; // get pointer to base class
```

It is just like getting a double from an int.

```
int x = 5;
```

```
double d = x;
```

**(2) instantiating a base class object**

You can still instantiate a base class. A base class will know nothing about derived class.

```
Person p1; // as a variable representing a default object
```

```
// as a variable initialized object
```

```
Person p2(String("Tom"),String("121 College St",25,10000);
```

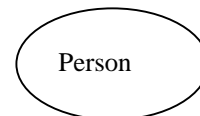
```
Person p3(p2); // as a variable copied object
```

```
Person* pp4 = new Person(); // as a pointer to a default object
```

```
// as a pointer to an initialized object
```

```
Person* pp5 = new Person(String("Tom"),String("121 College St",25,10000);
```

```
Person* pp6 = new Person(*pp5); // as a pointer to a copied object
```



When you instantiate a base class it contains a base class object component, it does not contain a derived class object. You can not access variables and functions of any derived class.

(3) base class represents an instantiated derived class

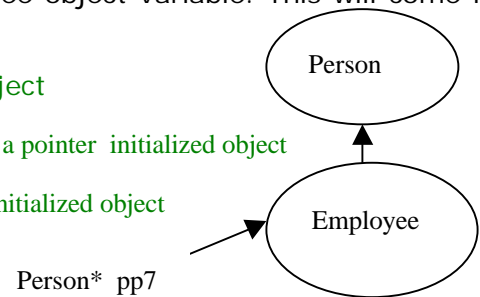
Since the derived class inherits the base class, A base object type can represent an derived object type. The Employee object is now **represented** by a Employee object variable. This will come in handy, you will see soon.

```
Person* pp7 = new Employee(); // as a pointer uninitialized object
```

```
Person* pp8 = new Employee(String("speedy"),10,invent); // as a pointer initialized object
```

```
Person* pp9 = new Employee(*(Employee*)pp8); // as a pointer initialized object
```

Notice in the copy constructor that we type cast the Person pointer to a Employee pointer. This is known as a **downward** conversion.



accessing derived functions from base class pointers (downward conversion)

We need to **typecast** the base class pointer to a derived class pointer if you want to access the functions and variables of the derived class object. This is because the derived object is represented by a base class pointer so the compiler thinks the pointer is pointing to a base object not a derived object..

```
float salary = ((Employee*)pp6)->getSalary(); // get salary
```

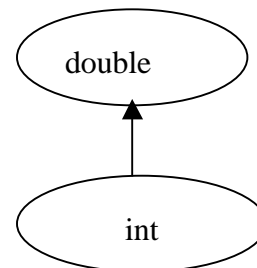
This is known as a **downward** conversion. Notice we typecast only the pointer variable.

It is just like getting an **int** from a **double**.

Notice you need to typecast to extract an **int** from a **double**.

```
double d = x;
```

```
int x = (int)d;
```



derived class overriding functions of the base class

The derived class has the **unique** functions and variables. The base class has all the **common** functions and variables. The derived class may have functions with the same name and same parameters as functions in the base class. If you create an instance of the derived class represented by a base class and you call a function having the same name as the base class then you call the function of the derived class. The derived class function **overrides** the function of the base class.

The only catch here is that the base class function must have the keyword **virtual** or else only the base class function will be called. An example is the `void print()` function of both the derived class Employee and base class Person. **Overriding** functions is a little different from **overloading** functions. Overloading functions are functions in the same class with the **same name** but **different parameters** data types. Overriding functions are in different classes, they have the **same name** and same parameter data types.

For overriding functions to work the functions in the base and derived class must have **identical names** and **identical parameter lists**. Only functions can be overridden not variables. In the case where the base class and derived class each have a variable with the same name. The derived class will always call its own variables and the base class they will always call its own variables. What is the difference between overriding a function and overloading a function ? The following table should let you know.

	function name	class	parameters
overriding	same	different	same
overloading	same	same	different

LESSON 10 EXERCISE 3

Instantiate an employee object using a base class pointer. Call the print method. In the Person class put the keyword virtual on the print function.

```
virtual void print();
```

You only need to put it the class definition not on the implementation.

LESSON 10 EXERCISE 4

Make a Student class that inherits a Person class. What is unique about a Student ? Marks and ID. Write the main function to instantiate Student objects. Use the main function to call functions from the Student objects and print out the information. Call functions from the base class and derived class. Call your program file with your main function L10ex4.cpp.

classes using other classes

There is a big difference between a class using another class then a class inheriting a another class. With inheritance the derived class can use the variables of the base class directly. When a class uses another class it must instantiate an object of the class, use the class object to access the variables and methods.

STRING CLASS

In C++ we want to avoid using char name[32] and char* for character strings. There is a class especially for strings, but unfortunately it is not built in to the C++ language. There are many String class in many books that people use. We have provided a String class to use for this course. We recommend you use a String class because it is good object oriented programming to use a String class. The String class is very easy to use. You can assign character strings or an existing string to a String object.

```
String s1;
```

```
s1 = "Hello"
```

```
String s2 = "Goodbye";
```

```
String s2 = s1;
```

s1	""
s1	"Hello"
s2	"Goodbye"
s2	"Hello"

You can assign and join string objects together:

```
String s3 = s1 + s2;
```

s3	"HelloHello"
----	--------------

The + and = are known as **operator functions**. It allows you to use strings just like using numbers. It is that easy. If you can add numbers than adding strings just means joining strings together.

String class functions:

function	description	example	operator
String()	create an empty String object	String s1;	default constructor
String(char *)	create a String object from a character string	String s1("hello");	char string copy constructor
String (String&)	create a String object from an existing String object	String s2(s1);	copy constructor
=	assign a string or char string to another string (copies existing string to new string)	s1 = s2; s1 = "hello";	assignment operator
<<	print out a string to screen or output stream	cout << s1;	output stream operator
>>	get a string from the keyboard or input stream	cin >> s1;	input stream operator
==	compare if two strings are equal returns < 0 for less , 0 for equal or > 0 for greater	(s1 == s2)	test if two strings equal

example using String class

Example using string class. You will need the string class from previous lessons or use the one with your compiler.

```
// example using string class lesson 8 program 1
#include <iostream.h>
#include "string.hpp"

// example using string class
void main()
{
    String s1 = "hello"; // make string object s1
    String s2 = "goodbye"; // make string object s2
    cout << s1 << " " << s2 << endl; // print out strings

    // test if string s1 equals s2
    if(s1 == s2) cout << "s1 equals s2" << endl;
    else cout << "s2 does not equal s1" << endl;
}
```

program output:

```
hello
goodbye
s2 does not equal s1
```

STRING CLASS

```

// string.hpp
#include <iostream.h>

#ifndef __STRING // prevent multiple includes
#define __STRING

class String
{
private:
char* Str;
int Length;
public:
String(); // default constructor
String(const char* s); // initializing constructor
String(const String& s); // copy constructor
~String();
// assignment operator
const String& operator=(const String& s);
// equality operator
int operator==(const String& s);
int operator==(const char* s);
// friend functions for i/o
friend ostream& operator<<(ostream& out,String& s);
friend istream& operator>>(istream& in,String& s);
};
#endif

// string.cpp
// string class implementation
#include <string.h>
#include "string.hpp";

// default constructor
String::String()
{
Str = new char[1];
*Str = '\0';
}

// initializing constructor
String::String(const char* s)
{
Str = new char[strlen(s)+1];
strcpy(Str,s);
Length = strlen(Str);
}

```

```

// copy constructor
String::String(const String& s)

{
    Str = new char[s.length+1];
    strcpy(Str,s.Str);
    Length = s.Length;
}

// destructor
String::~~String() { delete[ ] Str; }

// assignment operator
const String& String::operator=(const String& s)

{
    if(this != &s)

        {
            delete[ ] Str;
            Str = new char[s.length+1];
            strcpy(Str,s.Str);
            Length = s.Length;
        }

    return *this;
}

// equality to compare to String objects
int String::operator==(const String& s)

{
    return (strcmp(Str,s.Str)==0);
}

// equality to compare a string object with a Character String
int String::operator==(const char* s)

{
    return (strcmp(Str,s)==0);
}

// global functions

// send string to screen
ostream& operator<<(ostream& out, String& s)

{
    out << s.Str;
    return out;
}

```



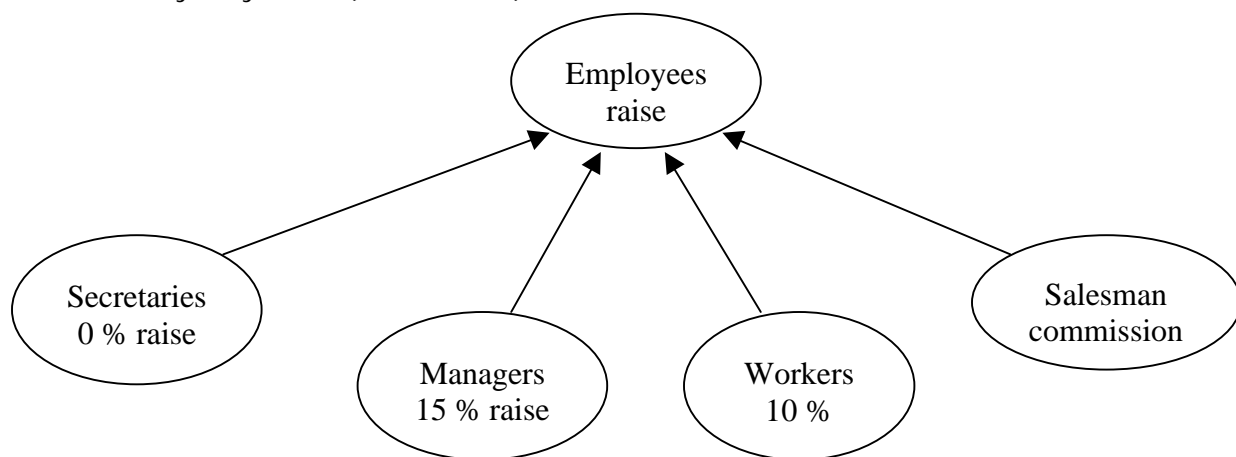
```
// read in a string from keyboard
istream& operator>>(istream& in, String& s)
{
    char s2[255];
    in.getline(s2,255);
    s = String(S2);
    return in;
}
```

C++ PROGRAMMERS GUIDE LESSON 11

File:	CppGuideL11.doc
Date Started:	July 12, 1998
Last Update:	April 3, 2002
Version:	4.0

LESSON 11 POLYMORPHISM AND VIRTUAL FUNCTIONS**Many Derived classes**

We can inherit other classes from our Employee class introduced in the last lesson. We can derive classes Secretaries, Managers, Workers and Salesman classes for your Employee class. Each employee gets a salary. For salesman the yearly salary is a base salary plus a sales commission. Each derived class has a function called **raise()** to boost their yearly salary. Secretaries do not get a raise, Managers get a 15% raise + \$2000 bonus, Workers get a 10% raise and Salesman get a raise based on their yearly sales (commission).



You will need to make the following classes inheriting the Employee class.

class	raise
Secretary	0 % raise
Manager	15 % raise + \$2000 bonus
Worker	10 % raise
Salesman	commission

Our Employee class inherits the Person class. Here is the Person class from last lesson:

```
// person.hpp
#include "string.hpp" // string class definition file

#ifndef __PERSON // prevent multiple includes
#define __PERSON

// class to represent a person
class Person

{
    // person info
private:
    String Name; // name
    String Address; // address
    int age; //age
    // functions for everyone to use
public:
    // default constructor
    Person();
    // initializing constructor
    Person(String& name, String& address, int age);
    // copy constructor
    Person::Person(Person p2)
    // destructor
    ~Person();
    // get Person's name
    String& getName();
    // get Person's address
    String& String getAddress();
    // print person info
    virtual void print();
    // increase Person's age
    void incAge();
};

#endif
```

The Person class implementation file is as follows:

```
// Person.cpp
#include <iostream.h>
#include "person.hpp"

// default constructor,
Person::Person()

{
    Name="";
    Address="";
    Age = 0;
}
```

```
// initializing constructor, set all variables to user specified values
```

```
Person::Person(String& name, String& address, int age)
```

```
{
    Name = name; // initialize person's name
    Address = address; // initialize person's address
    Age = age; // initialize persons age
}
```

```
// copy constructor, copy an existing object
```

```
Person::Person(Person& p2)
```

```
{
    Name = p2.Name; // initialize person's name
    Address = p2.Address; // initialize person's address
    Age = p2.age; // initialize persons age
}
```

```
// default destructor
```

```
Person::~Person() {}
```

```
// return Person's name
```

```
String& Person::getName()
```

```
{
    return Name; // return name
}
```

```
// return Person's address
```

```
String& Person::getAddress()
```

```
{
    return Address; // return address
}
```

```
// return Person's age
```

```
String Person::getAge()
```

```
{
    return Age; // return age
}
```

```
//print out Person's info
```

```
void Person::print()
```

```
{
    cout << "name: " << Name << endl;
    cout << "address: " << Address << endl;
    cout << "age: " << Age << endl;
}
```

```
// increase Person's age
void Person::incAge()

{
    Age = Age + 1; // increment age
}
```

The Employee class is like this:

```
// employee.hpp
#include "person.hpp"
#include "string.hpp"

#ifndef __EMPLOYEE // prevent multiple includes
#define __EMPLOYEE

// Employee class is derived from Person class
class Employee: public Person
{
    private:
        float salary;
    public:
        Employee(); // default constructor

        // initialize constructor
        Employee(String& name, String& address, int age);
        Employee(Employee& emp2); // copy constructor
        String getSalary(); // return salary
        virtual void raise(); // give each employee an raise
        virtual void print(); // print out info about this employee
};

#endif
```

Here is the derived Employee class implementation file:

```
// employee.cpp
#include <iostream.h>
#include "employee.hpp"

// default constructor calls base class default constructor
Employee::Employee():Person ()
{
    Salary = 0;
}

// initializing constructor calls base class initializing constructor
Employee::Employee(String& name, String& address, int age, float salary)
:Person(name, address, age)
{
    Salary = salary;
}
```

```

// copy constructor
Employee(Employee& emp2):Person(emp2)
{
    Salary = emp2.Salary;
}
// return Salary
float Employee::getSalary()
{
    return Salary;
}

//print out car plant info
void Employee::print()
{
    Person::print(); // print vehicle info from base class
    cout << " salary: " << Salary << endl;
}

```

LESSON11 EXERCISE 1

Add the raise function to the Employee class. Since it's the base class of the Worker, Secretary, Manager and Salesman classes the raise function is empty it does not do anything.

```
void Employee::raise(){}

```

Write the Worker, Manager, Secretary and Salesman derived classes. Test each object in a main function. Call your main program L11ex1.cpp. Call the file with your main function L11ex2.cpp. Put all your work in separate files as follows:

class/function	definition	implementation	execution
Person	person.hpp	person.cpp	
Employee	employee.hpp	employee.cpp	
Worker	worker.hpp	worker.cpp	
Secretary	secretary.hpp	secretary.cpp	
Manager	manager.hpp	manager.cpp	
Salesman	salesman.hpp	salesman.cpp	
main		L11ex1.cpp	L11ex1.exe

POLYMORPHISM

We want to have many different derived objects all represented by a base class pointer. When we want to give each derived object a raise we have to keep track of which object is what. In a C program this would be accomplished by many **if-else** statements or **switch-case** constructs. Wouldn't it be nice if someone could do it for you automatically? Yes there is such a person it is C++. The concept is known as **polymorphism** and it is done transparently to the programmer. To implement polymorphism all derived objects are kept in an array of base pointers. In your program you allocate an array of pointers to Employee objects. This is your old friend pointer to pointers.

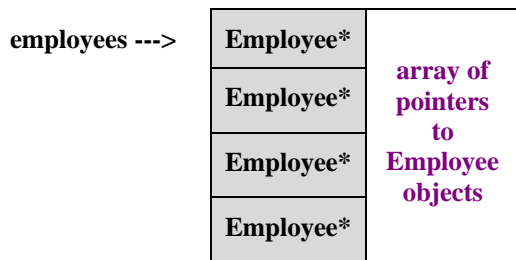
```
Employee** employees new Employee*[MaxEmployees]; // array of Employee pointers
```

Notice the data type is Employee** an array of Employee* objects. You allocate an array of Employee object pointers to the maximum number of employees you will need. You also need to keep a count of how many Employee you have put in the array. Alternatively you could reserve an array of pointers if you knew how many maximum plants you need beforehand.

```
Employee* employees[MaxEmployees];
```

You will need a variable called NumPlants to keep track of how many Employees have been put into the array

```
int NumPlants = 0; // how many plants are in the array
```



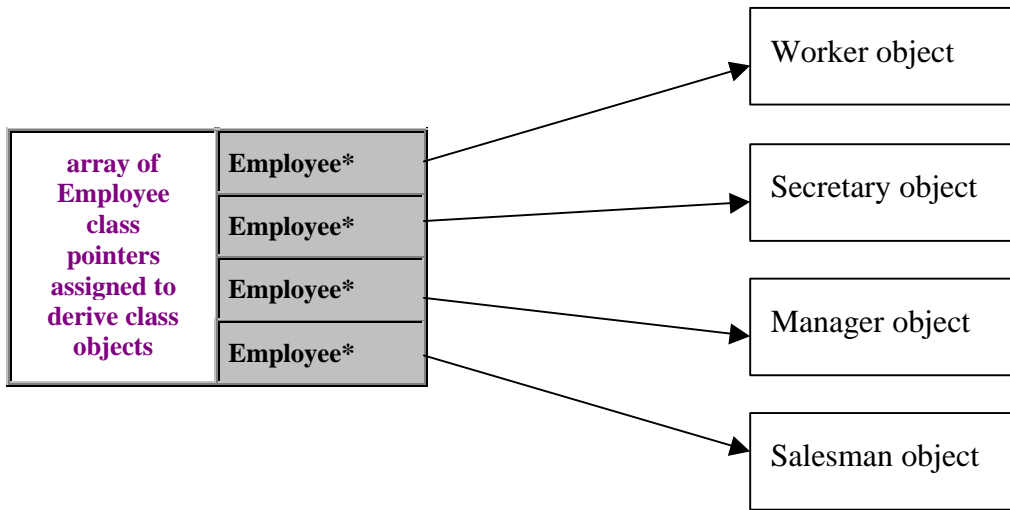
The next thing you have to do is create an objects derived from the employee class.

```
// make Worker object
Employee* pemp = new Worker(String("Tom"), String("121 College Street"), 25, 10000);
```

Notice, when we create a Worker object we assign it to a base pointer. This is what polymorphism is all about, a base pointer representing a derived object. The next thing we do is assign the object pointer to an element in the array of Employee object pointers.

```
employees[NumEmployees++] = pemp; // put in VehPlants table
```

We now have an array of pointers to different types of Employee objects.



The C++ program at **run time** keep tracks of which Employee is a Worker, Secretary, Manager or Salesman not the programmer. You will notice in the Person and Employee class some functions are proceeded by the keyword **virtual**.

```
virtual void raise(); // give each employee an raise
```

```
virtual void print(); // print employee info
```

This **virtual** keyword is used to inform the compiler at **compile time** that the function is a **virtual** function. **Virtual** means when your program is running the function to be called, is from the **derived** class not from the **base** class. The functions of the base class are **overridden** from the functions of the derived class. This is called **dynamic binding**. If the virtual keyword is omitted then only the base class functions will be called not the derived class functions. This is called **static binding**. Static binding is determined at compile time. For polymorphism to work the derived functions must have the same name and same number of parameters with identical data types and order as the base class functions. The functions of the derive class **overrides** the function of the base class. If the derived class **overrides** a **virtual** base class function then the base class function is not called.

The base function may or may not be used. If the base class function does not need to be used then it should be declared as **pure virtual** in the base class definition. To specify a function is pure virtual a '=0' is appended to the end of the function declaration.

```
virtual void raise()=0;
```

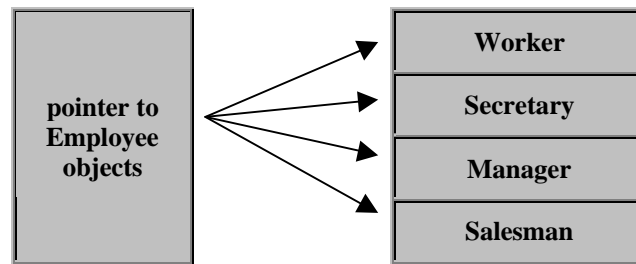
The **disadvantage** of using **pure virtual** functions means the base class can never be instantiated. The **advantage** of using **pure virtual** functions is that you do not need to be bothered about writing code in the base class. **Pure** and **virtual** functions are declared in the base class. All **pure** and **virtual** functions must be implemented by the derived classes. Classes with pure virtual functions are known as **abstract classes**. Classes without pure abstract functions are known as **concrete classes**.

using polymorphism

The important concept of polymorphism is that we have an array of base class pointers to objects derived from the Employee class. The program at run time remember which derived object to apply the function to the Secretary, Manager, Worker and Salesman. When we use the base class pointer, the object it is pointing to transforms itself into a derived class object. Polymorphism means change to. Just as the caterpillar transforms itself into a butterfly or a tadpole into a frog. Okay it's now time to give each employee a raise. All we have to do is the following, is to go through the array and call the raise function to do all the work.

```
for (int i = 0; i < numEmployees; i++)
{
    Employee* emp = employees[i]; // point to vehPlant
    emp->raise(); // call from car, truck or bus object
}
```

The above code gets a pointer to a derived Employee objects in the array and then calls the raise() function. With polymorphism the program knows which employee object to give a raise to. It's like a selector switch.



LESSON 11 EXERCISE 2

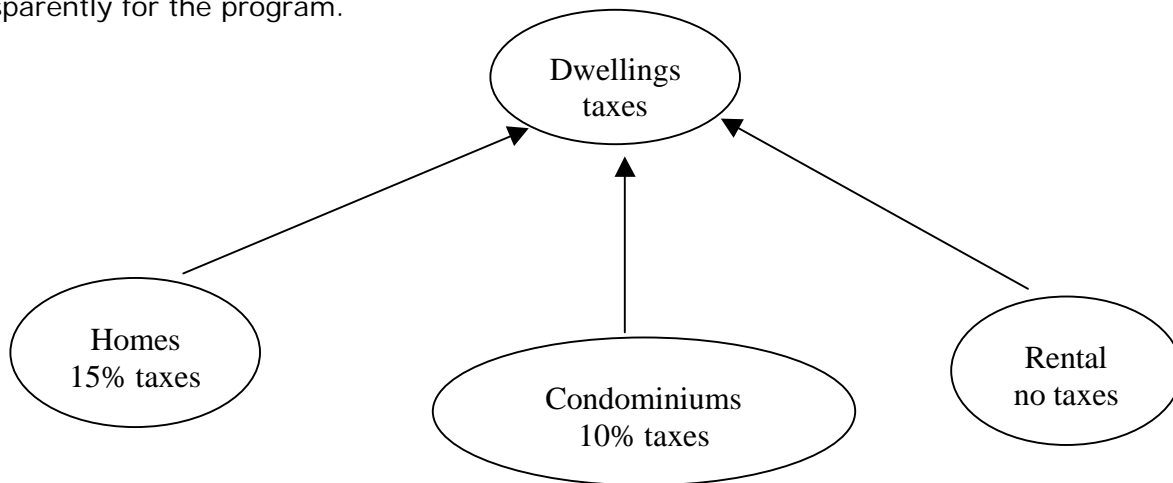
Write a main method that has an array of object pointers. Fill the array with many Employee derived objects. Print out the info of each derived object. Then give each employee a raise. Lastly print out the info about each object. They should now all have a raise. Call the file with your main function L11ex2.cpp. Put all your work in separate files as follows:

class/function	definition	implementation	execution
Person	person.hpp	person.cpp	
Employee	employee.hpp	employee.cpp	
Worker	worker.hpp	worker.cpp	
Secretary	secretary.hpp	secretary.cpp	
Manager	manager.hpp	manager.cpp	
Salesman	salesman.hpp	salesman.cpp	
main		L11ex2.cpp	L11ex2.exe

MORE POLYMORPHISM EXAMPLES

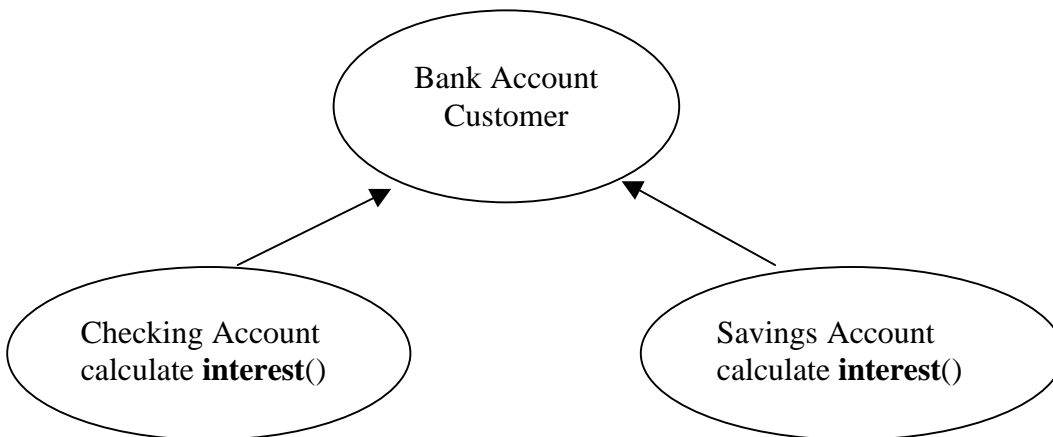
properties and taxes

Another example is taxes. Taxes are calculated from the value of a dwelling, Houses pay the most tax, and levied at 15 %, Condominiums pay a smaller tax, levied at 10 % , Rentals don't pay any tax. Again polymorphism comes to the rescue. It figures out how to calculate taxes from transparently for the program.



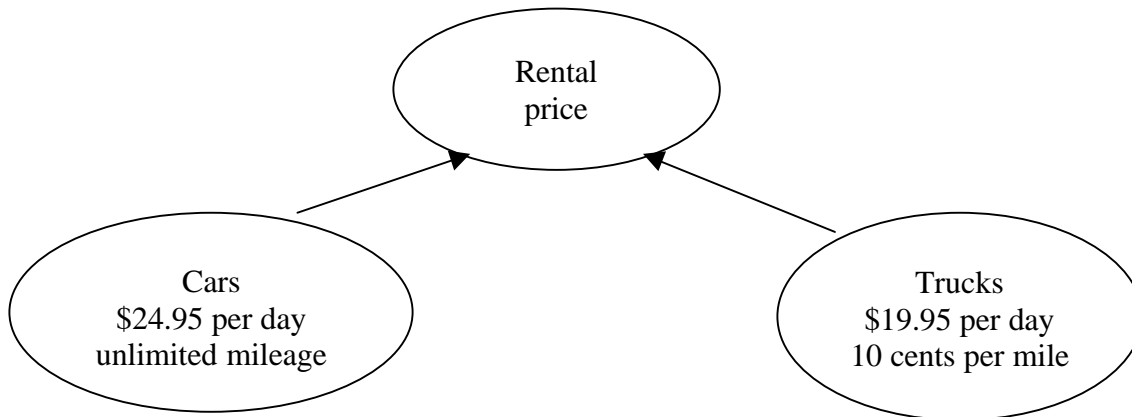
bank account

The classic example is the bank account customer having a savings and a checking account. There will be overridden methods for withdraws each account may have a different overdraft , interest calculation, monthly fees etc. There will be common methods for customer, deposits etc.



rental company

A last example is a rental company, they rent cars and trucks. Cars at a fixed day rate and trucks by a day and mileage rate. They rent cars at \$24.95 per day with unlimited mileage. They rent trucks at \$19.95 per day plus 10 cents per mile. You need to keep track of the days and mileage for each rental. Again polymorphism comes to the rescue. It figures out how to calculate the price of each rental transparently to the programmer.



LESSON 11 EXERCISE 3

Write the code for the rental company. You need a Rental base class and two derived classes RentCar and RentTruck. Keep things simple. The main function will rent some cars and trucks with number of days and mileage and then print out the costs. Call your main program L11Ex3.cpp.

LESSON 11 EXERCISE 4

Write a base class for students called Student containing student id and mark. Write derived classes HistoryStudent, ScienceStudent and ComputerStudent. Each derived class would have an **adjustMark()** function to change the mark. A possible formula could be 10 % of the class average if below 60 or negative if above 80. Have a unique formula for each kind of student. In the main function or Students class keep an array of students. Make some students, calculate the average for each kind of student and adjust the marks. Print out the results before and after the mark adjustments. Call your main program L11Ex4.cpp.

STRING CLASS

Here is the String class needed for your C++ programs. We added more functions than from the previous lesson. Make sure you use this one.

```

// string.hpp
#include <iostream.h>

#ifndef __STRING // prevent multiple includes
#define __STRING

```

```

class String
{
private:
char* Str;
int Length;
public:
String();
String(const char* s);
String(const String& s);
~String();

// assignment operator
const String& operator=(const String& s);
// equality operator
int operator==(const String& s);
int operator==(const char* s);
// friend functions for i/o
friend ostream& operator<<(ostream& out,String& s);
friend istream& operator>>(istream& in,String& s);
};

#endif

// string.cpp  string class implementation
#include <string.h>
#include "string.hpp"

// default constructor
String::String()
{
Str = new char[1];
*Str = '\0';
}

// initializing constructor
String::String(const char* s)
{
Str = new char[strlen(s)+1];
strcpy(Str,s);
Length = strlen(Str);
}

// copy constructor
String::String(const String& s)
{
Str = new char[s.length+1];
strcpy(Str,s.Str);
Length = s.Length;
}

```

```

// destructor
String::~String()
{
    delete[ ] Str;
    Str=NULL;    // for debug
    Length = 0;
}

// assignment operator
const String& String::operator=(const String& s)
{
    if(this != &s)
    {
        delete[ ] Str;
        Str = new char[s.length+1];
        strcpy(Str,s.Str);
        Length = s.Length;
    }

    return *this;
}

// equality String
int String::operator==(const String& s)
{
    return (strcmp(Str,s.Str)==0);
}

// equality char string
int String::operator==(const char* s)
{
    return (strcmp(Str,s)==0);
}

// global functions

// send string to screen
ostream& operator<<(ostream& out, String& s)
{
    out << s.Str;
    return out;
}

// read in a string from keyboard
istream& operator>>(istream& in, String& s)
{
    char s2[255];
    in >> s2;
    s = String(s2);
    return in;
}

```

C++ PROGRAMMERS GUIDE LESSON 12

File:	CppGuideL12.doc
Date Started:	July 12, 1998
Last Update:	April 4, 2002
Version	4.0

LESSON 12 C++ LANGUAGE IMPLEMENTATION

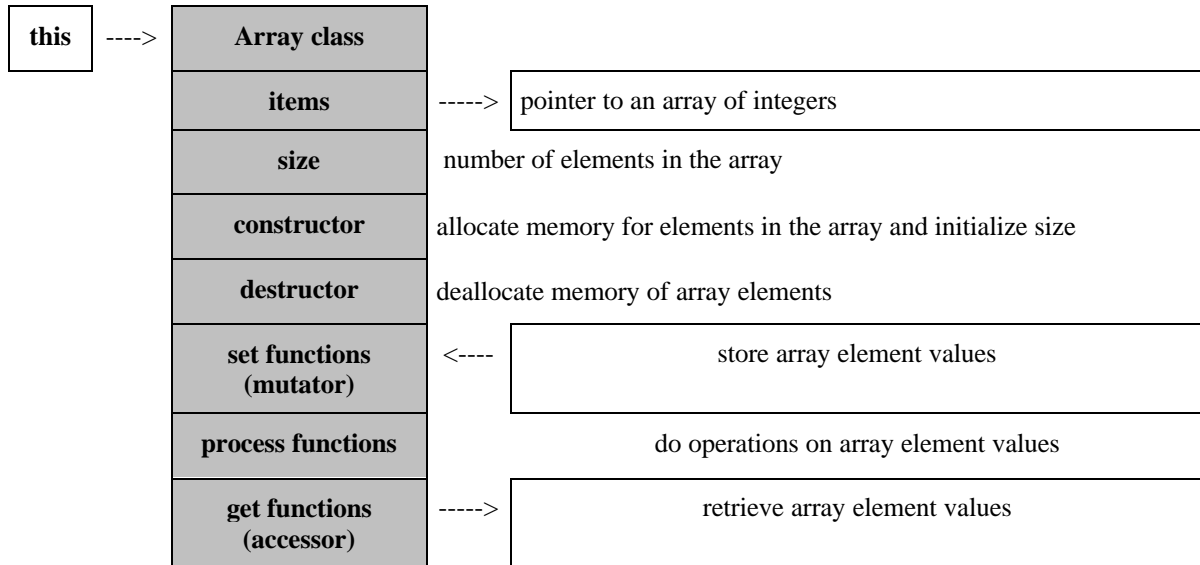
Now we get to real C++ Programming

Every C++ Program will have the following:
#include
#define
class definitions
function prototypes
class function implementation
main function
function definitions

Every C++ class will have the following:
this pointer
variables
constructor
copy constructor
destructor
operator functions
assignment operator function
set functions (mutator methods)
process functions
get functions (accessor methods)
friend functions

ARRAY CLASS

We will now introduce a C++ class called the Array class. The Array class will contain an **array of integer items** and keep track of the size of the array. Notice the Array class starts with a **capital 'A'** and when we talk about an array of integers, we use lower case 'a'.



class definition

A class lists all the variable and function declarations needed by an object group.

<i>class class_name</i> { <i>visibility:</i> <i>variable declarations</i> <i>visibility:</i> <i>function declaration</i> <i>friend functions</i> } 	<pre> class Array // Array class definition { private: int* Items; // pointer to array elements int Size; // holds size of array static int Num; // number of array objects public: Array(); // default constructor Array(int size); // initializing constructor Array(Array& a2); // copy constructor ~Array(); // destructor const Array& operator=(const Array& a); // assignment operator bool operator==(const Array& a); // equality operator const Array operator+(const Array& a); // adding operator int& operator[](int index); // array index bool operator()(int data); // is element of int operator++(); // pre increment after assignment int operator++(int i); // post increment before assignment operator int(); // return current element static int howMany(); // static function how many objects // friend functions friend const Array operator+(int num, Array a2); //add left constant friend const Array operator+(Array a2, int num); //add right constant friend ostream& operator<<(ostream& out, Array& a); //print out friend istream& operator>>(istream& in, Array& a); //keyboard }; </pre>
---	--

Note: Some compilers may give you conflicts (ambiguity) between you may need to drop the `const` modifier to remove the ambiguity.

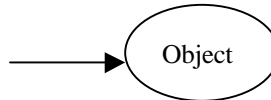
```
const Array operator+(const Array& a); // adding operator
```

and

```
friend const Array operator+(Array a2, int num); // remove const on Array
```

this pointer

this



The **this** pointer is an automatic pointer to the object being acted on. For a function to access the which object instantiated from the same class the compiler provides a pointer called **this**. In your code you can use the **this** pointer to refer to the variables and methods of the class you are defining.

variables

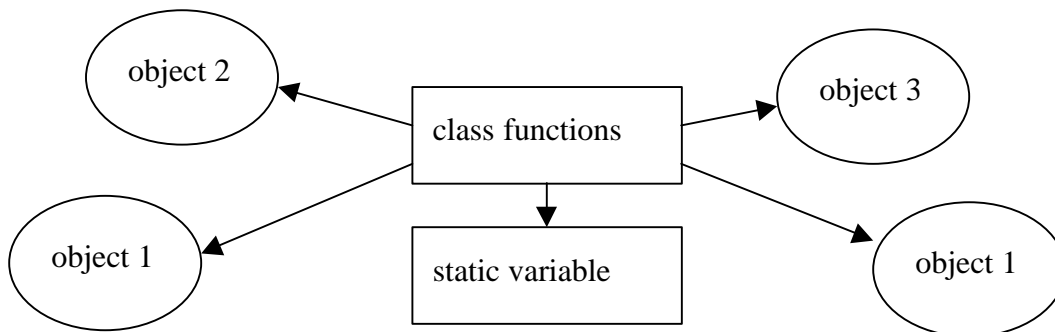
Variables are used to store and retrieve data values belonging to the object. Variables are usually **private** so that they need to be accessed by functions. Variables defined in a class are also known as members.

```
int* Items; // point to array integer items
int Size; // number of elements in array
```

Variables may start with a **Capital** letter to distinguish between variables that are defined in the class to function parameters or local variables. Function parameter or local variables should start with a lower case letter. You may also proceed variable names belonging to a class with a lower case **m** to mean member. example: `mItems`. The choice is yours. Variable visibility may be **public**, **protected** or **private**.

private	Only the class can access variables and functions of this class
protected	only the base class and derived classes from base can access
public	Everybody can access variables and functions of this class

Variables are classified as instance variables or static variables. Instance variables belong to a object. Static variables belong to the class. What this means is for every object created each object will have the same instance variables, but different values. Static variables are not created when objects are created. Static variables are like a global variable to the functions of this class only. Static variables are also called class variables.



To declare a static variable in your class definition you use the **static** modifier:

```
static data_type variable_name;

static int Num;
```

In your implementation file you need to give the static variable its initialized value. The value it starts with. You do not need to specify the keyword static again in the implementation.

```
data_type class_name::variable_name=initialized_value;

int Array::Num=0;
```

Static variables are used to store values that all objects may need. The classic example use for a static variable is to count the number of objects created. In our constructor we will increment the static variable Num everytime an Array object is created. Static variables can be private or public. Private static variables can only be accessed by static functions. Public static variables can be accessed by all functions. We will study static functions very shortly.

constructors

The **constructor** job is to initialize variables with values or to allocate memory for pointer variables defined in the class. Memory is allocated using the **new** operator. Constructors have the same name as the class. The constructor has no return value not even a void. There may be one or many constructors each for a specific situation. This is possible because the constructors can be **overloaded** by different data types in the constructor parameter list. A **default** constructor has no parameter list and is just used to initialize the variables belonging to the class to some default values. Every class should have a **default constructor**. An **initializing constructor** is used to initialize variables of the class to user specified values passed through its parameter list. **Copy constructors** are used to copy an existing object. Copy constructors have a reference parameter to the object that it will copy. Note: In the following discussion we present the definition, the implementation and example using.

default constructor

In default constructors the variables are initialized to default values when the object is created. A default constructor declaration has no parameters:

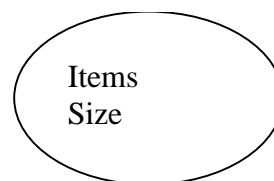
```
// default constructor declaration
Array();
```

When you implement a default constructor you usually initialize variables defined in the class to some default value.

```
// default constructor implementation initialize variables to default values
Array::Array()
```

```
{
  Items = NULL; // no memory for item
  Size = 0; // set array size to zero
  Num++;
}
```

Array object



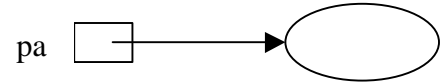
When you create an object using a default constructor as a variable then you just list the variable name to represent the reserved object.

```
// using default constructors
Array a; // reserved Array object
```



When you create an object using default constructed using a pointer then the pointer variable points to the allocated object.

```
Array* pa = new Array(); // allocated Array object
```



initializing constructor

Initializing constructors initialize variables of the class to user defined values obtained from the constructor parameters.

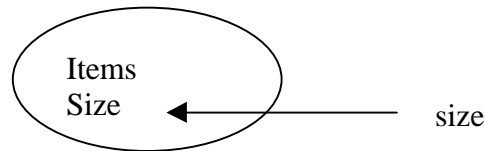
```
// Initializing constructor declaration
Array(int size);
```

Array object

When you implement an initializing constructor, you use the parameters to initialize the variables defined in the class.

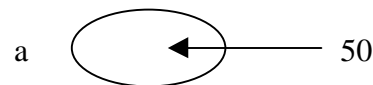
```
// Initializing constructor implementation
// initialize variables to user values
Array::Array(int size)
```

```
{
    Items = new int[size]; // allocate memory for integer array
    Size = size; // set array size
    Num++;
}
```



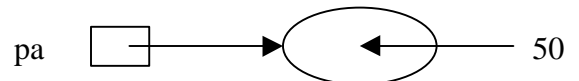
When you create a reserved object you include the values you want to initialize the object with as arguments to the constructor. represented by the variable name.

```
// using initializing constructor
Array a(50); // reserve array object set array size
```



When you create an allocated object you include the values you want to initialize the object with as arguments to the constructor.

```
Array* pa = new Array(50); // allocate array object
```



colon initialization

If the member variables are **const** or another **object** that needs to be initialized you must use **colon initialization**. Colon initialization can also be used for other variables as well. In the constructor header you list and initialize the member variables in round brackets separated by commas.

```
Array::Array(int size): Size(size), Items(new int[size])
{}
```

Initialize size Initialize Items

Here is an example using colon initialization a object. We have another class called Test using our Array class. We can use colon initialization to set the size of that Array object.

```
// test colon initialization for Array objects
class Test
{
private:
Array a; // declare an Array object
public:
Test(int size); // initializing constructor
};

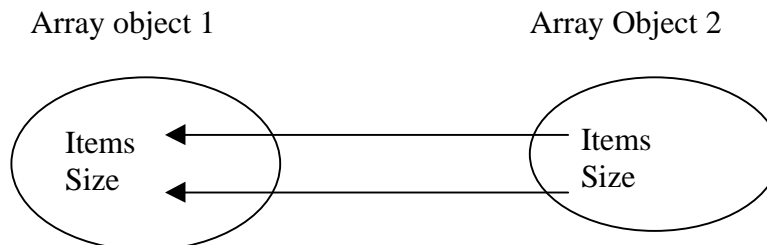
// initializing constructor implementation
// use colon initialization to set size of Array object
Test::Test(int size ): a(size)
{}
```

Make a Test object that will set the size of the Array object. The Array object initializing constructor is called to set the size of the array to 50.

```
Test t(50); // make a test object to initialize Array size to 50
```

copy constructor

The purpose of the copy constructor is to copy an existing object. This is really cloning.



A copy constructor has a **const** reference parameter to an Array class it will copy. The parameter must be a **reference** because a copy is being made of a known existing object.

```
// copy constructor definition
Array(const Array& a2);
```

const means the copy constructor will not change the original contents of the object to be copied. A copy constructor is needed to make a copy of an existing Array object. When an Array object type is being passed as a parameter in a function, a copy of it will be made. If you do not declare a copy constructor then the compiler will generate one for you. In most cases, the compiler will not know how to make the copy constructor properly. In our array copy constructor we need to allocate memory for the array items. The **const** keyword here means not to change the contents of a2

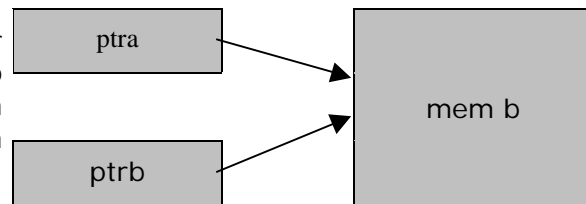
```
// initialize variables with an existing object values
Array::Array(const Array & a2)
{
    items = new int[a2.size]; // allocate memory for a2
    Size = a2.size; // set to size of a2

    // copy items one by one
    for (i=0; i<Size; i++) Items[i] = a2.Items[i];
}
```

The following example demonstrates this problem. If you are allocating memory in your constructor the compiler will not use the **new** operator to allocate memory for the copy constructor's pointer variables. The compiler will just assign the address of the variables of the class object you are copying to your class pointer variables. You will be left with one pointer in each class object pointing to the same memory location. This is very bad, when you change the memory contents in one class object the other class object memory contents will change also.

If you do not make your own copy constructor then you are left with two pointers pointing to the same memory location. Each pointer of each class object points to the same memory location

```
ptrb = ptrb;
```



You must write your own copy constructor that allocates memory based on the size of the array class you are copying.

```
ptrb = new int [b.Size];
```

Now each array class has its own allocated memory.



It's easy to use a copy constructor:

```
// using copy constructor
Array b(50); // make an Array object b of 50 elements
Array a(b); // make an Array object a by copying array b
Array pa = new Array(b); // allocate object and copy an existing
```

destructors

Destructors are used to deallocate memory once the class object is no longer needed. Destructors have the same name as the class but are distinguished from a constructor by being preceded by a ~ (tilda).

```
~Array();// destructor declaration
```

A destructor has no argument list and does no return value not even a void. If you do not include a destructor the compiler will automatically make one for you. You must also delete any memory you have allocated. If you do not include a destructor to delete memory when it is no longer needed, then there will be lots of wasted memory floating around in your computer. You do not need to declare a destructor in classes that do not allocate memory, but it is good practice to do so.

```
// destructor implementation
// deallocate allocated memory
Array::~Array()
{
    delete[ ] Items; // delete array of items
    Items = NULL; size = 0; // indicate memory has been deleted
}
```

The **delete** key word has brackets indicating to delete all the memory allocated for the array elements rather than a single array element memory cell.

Destructors are called when the class **loses scope** or when your program terminates. Losing scope means the class object will never be used again. For allocated objects the constructor is not called. automatically. The destructor is only called when you use the delete operator to deallocate memory used for the object.

```
// calling a destructor
delete a;
```

operator functions

Operator functions use a symbol to represent a function name. The function name is the symbol! Operator functions are very convenient and elegant in C++ programming. With operator functions, you get to use the following operators for your function names. Valid operators are:

+ - / * < > = == [] += -= /= *= () << >>

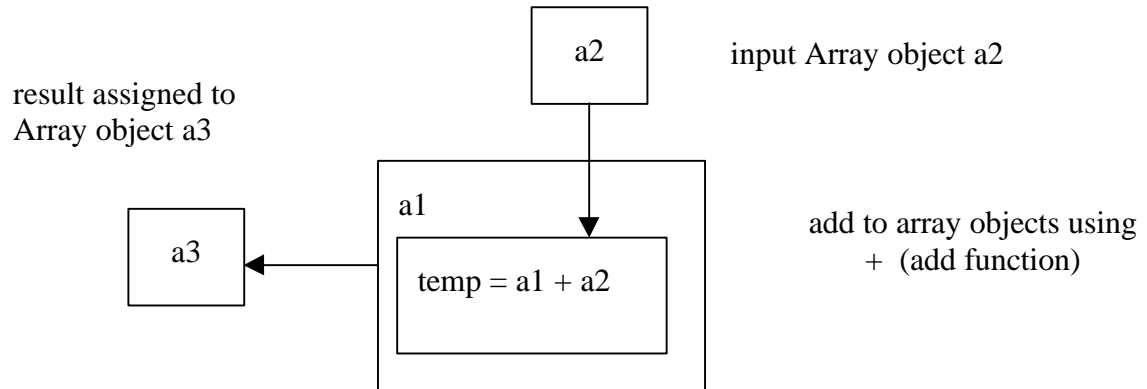
You can use operator functions to add class object array elements together just like using ordinary addition.

```
a3 = a1 + a2; // add two class object array elements together
```

The '+' is the name of the function (add) , and is termed an "operator function". The equivalent operation is:

```
a3 = a1.+(a2); // a3 gets the array element values of a1 + a2.
```

```
a3 = a1.add(a2); // just think that the '+' stands for 'add'
```



Operator functions are declared as follows:

```
const return_value operator operator_function_name (const class_name& paramater_name ) const;
```

```
const Array operator + (const Array& a)const;
```

Notice we have the **const** modifier in 3 places on the return data type a parameter and at the end of the function. The **const** keyword is optional . A function with a **const** modifier is considered different from the same name function that does not have a **const** modifier. This is because the **const** modifier **overloads** the function. Three ways to place **const** modifier on a object

- (1) return object type
- (2) constant parameters
- (3) const class function

placed const modifier	description
(1) return const object type	A const on the return object type means the contents of the returned object cannot be modified.
(2) const object parameters	A const on a object parameter means we don't want the contents of the passed object to be modified
(3) const class function	A const at the end of a class function means we do not want to change the contents of any member variables of the object that called the function. If you want to change the object make a copy of it and change the contents of the copy and return this copy. const class functions do not affect local variables.

passing and returning objects from functions

Objects may be passed by **value**, **pointer** or by **reference**. When passed by **value** a copy is made of the whole object. The objects copy constructor is called to make a duplicate object. When the object is finished being used the destructor is called to destroy the object. There is a lot of overhead involved when passing objects by value. When using **pass by pointer** or **reference** the address of the objects is passed which results in less overhead. Pass by reference is a **direct address** where pass by pointer is an **indirect address**. Direct meaning the actual address is passed indirect means you have to read the contents of the pointer to get the address. Objects may be returned by value, reference or pointer. A duplicate object is made with return by value, again the copy constructor must be called. The copy constructor is not called with return by reference or pointer. Using return by reference or pointer allows assignment to or from the object. As before a return by reference is a direct address and return by pointer is an indirect address.

pass type	description
pass by value	copy of object passed by calling copy constructor
pass by reference	direct address of object passed
pass by pointer	indirect address to object passes

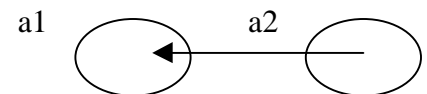
return type	description
return by value	copy of object returned by calling copy constructor
return by reference	direct address of object returned
return by pointer	indirect address to object passed

assignment operator

The assignment operator allows you to assign the class object on the right side to the class object on the left side. When you are assigning then you are basically copying the values of one object to the other object.

```
Array a1(5);
Array a2(5);
```

```
a1 = a2; // assign the values of object a2 to object a1
```



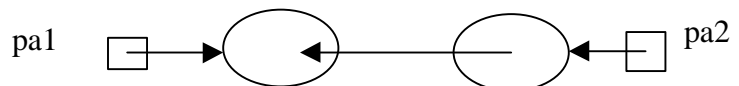
The array object on the right side is copied to the Array object on the left side of the assignment operator.

```
a1 = a2; is just like saying a1.=(a2);
```

In case of Array objects represented by pointers we need to use the star operator.

```
Array* pa1=new Array(5);
Array* pa2 = new Array(5);
```

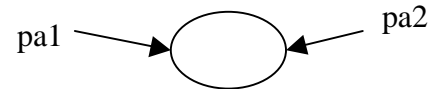
```
*pa1 = *pa2;
```



If we do not use the star * then we just assign the address to the pointer but copying the contents of one object to the other.

If we just use pointers than the addresses are assigned.

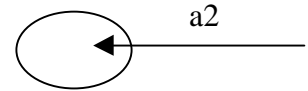
```
pa1 = pa2; // assign the address of object a2 to object a1
```



How would you use references and pointers to copy the object data ?

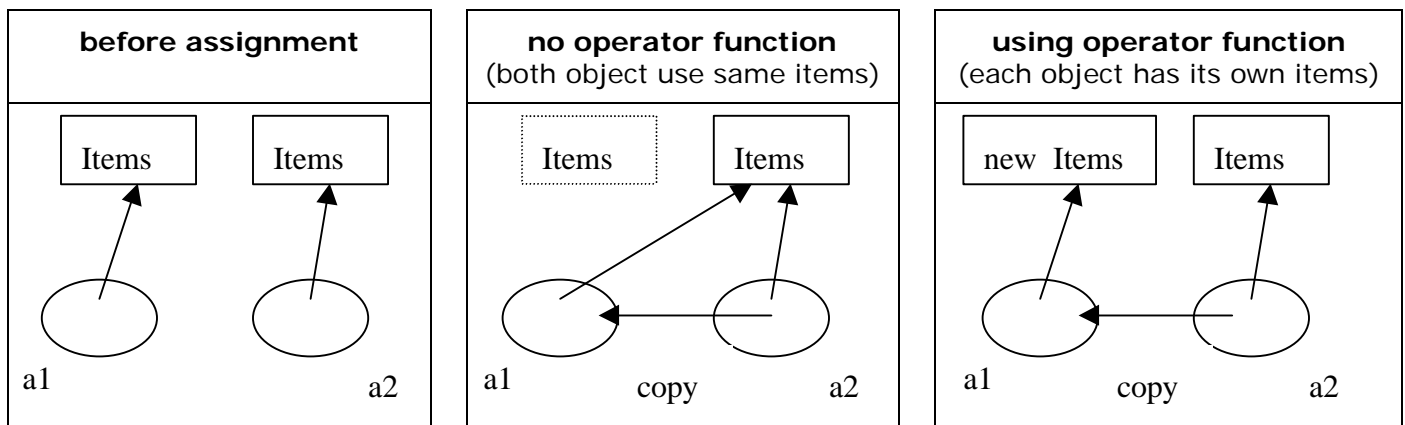
```
Array& ra1=a1;  
Array& ra2=a2;
```

```
ra1 = ra2; // assign the values of object a2 to object a1
```



Why do we need our own assignment operator function ?

In situations where the assigned classes allocate memory for an array then the assigned class must deallocate memory for the array and then allocate new memory for the array and then copy the data from the array. If no assignment operator function existed then the assigned object would point to the copied objects array.



Here is the function definition for the assignment operator =

```
// declare operator = function  
const Array& operator=(const Array& a);
```

Why does assignment operator return by reference ?

The assignment operator returns a **const** reference to the class. The **reference** is used to assign the class object on the left side of the assignment operator to the class object on the right of the assignment operator. It also allows **chaining** of objects.

```
a1 = a2 = a3 = a4; // assign objects to each other right to left.
```

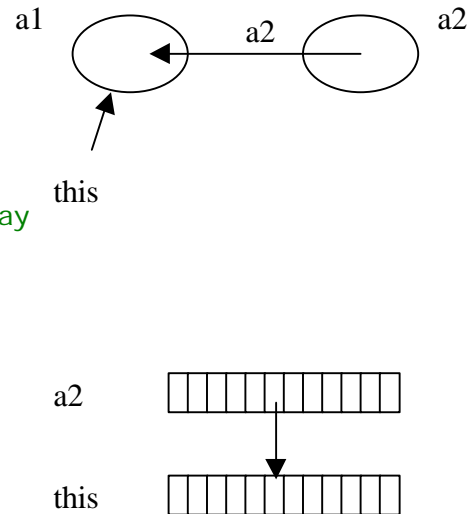
Why is the return data type a const ?

Const insures that the right hand class object member variables will not be altered.

Here is the code for the assignment operator = function. When we say `a1 = a2`. We want to assign the `a2` objects values to array object `a1`. `a1` is the executing object pointed to by the `this` pointer. The operator function first checks to see if the object is itself. It does not copy itself. Next it reallocates memory for the array items to the size of array object `a2`. Then finally copies the values from array object `a2` to array object `a1`. Finally the function returns a reference to itself.

```
// assignment operator = function
const Array& Array::operator=(const Array& a2)
{
    // check if assigned array points to a (itself)
    if (this != &a2)
    {
        // delete all allocate memory in assigned array
        delete[] items;
        // allocate memory for assigned array
        Items = new int[a2.Size];
        // assigned size to assigned array
        Size = a2.Size;
        // copy elements from array a2
        for(int i=0;i<Size;i++)
            Items[i] = a2.Items[i];
    }

    return *this; // returns a reference to the class
}
```



Why do we return `*this` ?

All classes are pointed to by a variable called **this**. The assignment operator returns the **this** variable as a reference. We return ***this** because we need to convert the pointer variable to a reference variable (they call this **dereferenced**). We must return the object that the pointer is pointing to.

equality operator == function

The equality operator function tests if two array objects are equal. The first array is the array object that was called and the second array object is the one passed `a2`.

```
bool operator==(const Array& a2); // equality operator
```

To implement the `equate` operator, you just basically check if both arrays have the same length and all array elements have the same value.

```
// test if two arrays have same data elements and same sizes
bool Array::operator==(const Array& a2)
{
    if(Size != a2.Size) return; // return false if sizes different
    for(int i=0; i<Size; i++)
        if(Items[i] != a2.Items[i]) return FALSE;
    return TRUE;
}
```

You usually use the `equate` operator with an `if` statement to test if two arrays are equal.

```
Array a1(5);
Array a2(5);

// test Equality operator
if(a1 == a2) cout << "a1 equals a2" << endl;
else cout << "a1 not equal a2" << endl;

// test object pointers
Array* pa1 = new Array(a1);
Array* pa2 = new Array(a2);

//test if array data are equal
if(*pa1 == *pa2)cout << "*pa1 == *pa2" << endl;
else cout << "*pa1 != *pa2" << endl;
```

You must be careful in using the `equate` statement with pointers. If you are just testing the pointers it compares the pointer addresses and does not call the equality operator from the `Array` object.

```
//test if array pointers are equal
if(pa1 == pa2)cout << "pa1 == pa2" << endl;
else cout << "pa1 != pa2" << endl;
```

add operator + function

When you use the operator + function like

```
a3 = a1 + a2
```

it is just like calling a function called `add` and would be the same thing as:

```
a3 = a1.+(a2);          (a3 = a1.add(a2);)
```

We define our `add` operator function as follows:

```
// operator + function definition
const Array operator+(const Array& a); // adding operator
```

You will notice we receive a second array by reference and return the addition result by value. We are adding two arrays. The first array is the array object who called the operator function. The `add` operator + function allows you to add the contents of 2 arrays. A new array object is instantiated to hold the intermediate results of adding two array objects. We do this because we do not want the original array to be altered. We return this array by value, Why ? We cannot return a temporary created object in a function using return by reference or pointer. When adding arrays the arrays may be of different length. If array `a2` is longer than array `a1`, then array `a1` needs more storage. New storage must be added and old storage deleted. Here is the code for the operator + function:

```
// add two arrays
// make a new array if one array is large, pad excess elements with zero's
// return by value
const Array Array::operator+(const Array& a2)
```

```
{
    int largest; // keep track of largest size

    // check if executing class Size larger than a2 Size
    if(Size > a2.Size) largest = Size;
    else largest = a2.Size;

    Array t(largest); // allocate temp for largest size
    int data1,data2; // hold values of array elements

    // add two arrays
    for(int i=0;i<largest;i++)
    {
        if(i < Size) data1 = Items[i];
        else data1 = 0; // pad with zeros

        if(i < a2.Size) data2 = a2.Items[i];
        else data2 = 0; // pad with zeros

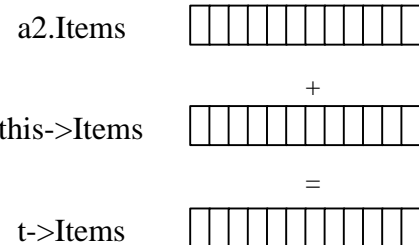
        t.Items[i] = data1 + data2; // add data values
    }

    return t; // return value of what pointer points to
}
```

a3 = a1 + a2

a2

temp = this (a1) + a2



Why is the return type return by value ? We want to return a copy of the temporary Array object containing the result of adding the two Array objects. The temporary Array object will be destroyed after the function finishes executing. In the following example we instantiate two Array objects as variables, pointers and references, and show you how to use operating functions for each

```
// instantiating Array object as variable
Array a1(5);
Array a2(10);
// using operator+ function when Array objects declared as variable
// add array object a2 to array object a
// assign the results to array object a1
a1 = a1 + a2;
// instantiating Array object as pointer
Array* a1 = new Array(5);
Array* a2 = new Array(10);
// using operator + function when Array objects declared as pointers
*a1 = *a1 + *a2;
// instantiating Array objects as reference
Array& a1(5);
Array& a2(10);
// using operator + function when Array objects declared as references
a1 = a1 + a2
```

index operator [] function

The index operator allows you to get the contents of an element in array at a specified index. You can also assign a value to the array element at the specified index. Here is the index function declaration:

```
int& operator[ ](int index);
```

Here is the index function implementation:

```
// index operator function implementation
int& Array::operator[ ] (int index)
{
    return Items[index];
}
```

this->Items



index []

The input parameter is an array index and the return value is an **int** reference. Why ? Because you need both the ability to read array elements and to write array elements. Here is an example using the array index function.

```
// instantiating array as variable
Array a(5);

// using index operator function
a[0] = 1; // assign 1 to the zero index of the array object
int x = a[0]; // get the value of the zero index of the array object
```

What is the value of x ?

```
// instantiating as pointer
Array* pa = new Array(5);

// using index operator with pointer
*pa[0] = 1;
int x = *pa[0];
```

What is the value of x ?

```
// instantiating as reference
Array& a(5);

// using index operator with reference
a[0] = 1;
int x = a[0];
```

What is the value of x ?

test operator function

We use the () operator to test if an element is in the array.

```
bool operator()(int data); // is element of
```

That's right the name of the operator function is (). Here's the code:

```
// test if element is in the array
// return true if found otherwise false
bool Array::operator()(int data)
{
    // return true if array element found
    for(int i=0;i<Size;i++)if(lItems[i]==data)return TRUE;
    return FALSE;
}
```

To use, just put in the array element index you want to test.

```
// using test operator function on Array object
cout << a(1) << endl; // test if number 1 is in the array object

// using test operator function on pointer to Array object
cout << *pa(1) << endl; // test if number 1 is in the array object
```

increment operators ++

There are two types of increments operators:

(1) **pre-increment** increment before assignment

```
a3 = ++a1; // ++ on left side of object variable
```

(2) **post increment** after assignment

```
a3 = a1++; // ++ on right side of object variable
```

(1) pre-increment before assignment

To pre increment a object, it's a two step process:

increment before	increment object	assign object
a3 = ++a1;	a1 = a1 + 1;	a3 = a1;

a3 = ++a1; means a3 = ++.a1()

Here is the function definition

```
Array operator++(); // pre increment before assignment
```

Here is the function implementation

```
// pre increment before assignment
Array Array::operator++()
{
    for(int i=0; i<Size; i++) // add 1 to each item
        Items[i] = Items[i] + 1;
    return *this; // return itself
}
```

Here is using:

```
a3 = ++a1;
```

In the **pre-fix** operation the value of a3 will have the incremented values as a1. a1 will also have the incremented values.

(2) post increment after assignment

To post increment a object, it's a two step process:

increment after	assign object	increment object
a3 = a1++;	a3 = a1;	a1 = a1 + 1;

a3 = a1++; means a = a2.++(0);

Here is the function definition:

```
int operator++(int ); // post increment after assignment
```

Notice the **post-fix** operator ++ function has the **int** dummy parameter to distinguish a **postfix** operation from a **pre-fix** operation. The reasoning is in post fix you are adding 0 to the elements before you do the postfix operation on. (the dummy parameter gets the value 0)

```
a3 = (a1+0)++;
```

Here is the function implementation

```
// post increment after assignment
Array Array:: operator++(int )
{
    Array a(*this); // make copy of object
    for(int i=0; i<Size; i++) // add 1 to each item
        Items[i] = Items[i] + 1;
    return a; // return copy of object
}
```

Here is using:

```
a3 = a1++;
```

In a **post-fix** operation the value of a3 will have the original values of array a1. a1 will have the incremented values.

conversion operator

A conversion operator converts the object type to another type. In our case we are converting our Array object to an int. The conversion operator prototype is:

```
operator data_type()const;

operator int()const;
```

We use the conversion operator to return the size of the array object.

```
Array::operator int()const
{
    return Size;
}
```

The conversion operator is mostly used with cout to print out the value of a object.

```
Array a;

cout << a << endl;
```

In our case we have a conflict with the friend operator function << . In this situation, we need to typecast the conversion operator function by its data type when we call it.

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

calling operator functions of a base class

It is possible to overload operator functions. The problem now is how do you call the base operator function ? The solution is to typecast.

```
a1 = a1 + (BaseArray)a2;
```

Be careful, if its pas by reference or pointer make sure you use the & or * in your typecast.

```
a1 = a1 + ((BaseArray&)a2);
a1 = a1 + ((BaseArray*)pa2);
```

Friend Functions

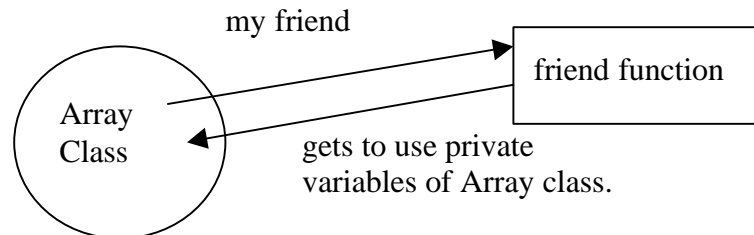
Friend functions are allowed to use the member functions and variables of a class. These are stand-alone functions that don't belong to any class. When a class declares a function to be its friend, then that function gets to use the variables and functions of that class. Friendship is only one way. The class cannot use the variables of the friend function. If this is desired the friend function must also declare the class to be its friend. Friend functions are needed because the operator function is only allowed to have only one parameter. A friend operator function can have two parameters.

The most widely used friend operator functions are the << and >> operators << to get data from a keyboard or print its data to the screen. It's easy for a class declare a function to be it's friend.

```

class array
{
friend const Array operator+(int num, Array a2); // add left constant
friend const Array operator+(Array a2, int num); // add right constant
friend istream& operator >> (istream& in, Array& a); // print
friend ostream& operator<<(ostream& out, const Array& a); // keyboard
};

```



adding constants

Adding constants are a little trickier, you can add constants either on the left hand side like:

```
a1 = 5 + a2;
```

or on the right hand side like:

```
a1 = a2 + 5;
```

We need **friend** functions to handle these situations to distinguish a left hand constant from right hand constant. Also operator functions can have only 1 parameter.

```

// add left hand constant
const Array operator+(int num, const Array& a2)
{
    Array a(a2.Size); // allocate for size of a1
    // add num to array
    for(int i=0; i<a2.Size; i++) a.Items[i] = num + a2.Items[i];
    return a; // return reference to new array object
}

// add right hand constant
const Array operator+(const Array& a2, int num)
{
    Array a(a2.Size); // allocate for size of a2
    // add num to array
    for(int i=0; i<a2.Size; i++) a.Items[i] = num + a2.Items[i];
    return a; // return reference to new array object
}

```

Notice we do not have a **const** modifier on the `Array&` parameter as to distinguish between the `Array Array::operator+(const Array& b);` function. Again we return by value.

operator >> friend function

The overloaded >> operator function allows the user to enter data into the array from the keyboard.

```
// prototype definition
friend istream& operator >> (istream& in, Array& a);

// friend operator function implementation
istream& operator >> (istream& in, Array& a)
{
    cout << "enter " << a.GetSize() << " data elements: " << endl;

    // get all data values for array elements
    for(i=0; i<a.GetSize(); i++)
    {
        cout << (i+1) << ": ";
        in >> a[i]; // use istream class object
    }

    return in;
}
```

Why is there a bracket around (i + 1) ?

using friend >> function:

```
// using friend operator function
Array a(5); // make an array object of 5 elements
cin >> a; // get data from key board for array class object
```

Does the following code use the friend function ? Why not ?

```
cin >> a[3];
```

operator << friend function

The overloaded << operator function writes out the array to the terminal screen.

```
// prototype definition
friend ostream& operator<<(ostream& out, const Array& a);

// friend operator function implementation:
ostream& operator << (ostream& out, const Array& a)
{
    out << "[";
    for(int i=0; i<a.GetSize(); i++)
        out << a.Items[i];
    out << "]" << endl;
    return out;
}
```

printing out array contents: [1 2 3 4 5]

using friend << function:

```
Array a(5); // make an array object of 5 elements
cout << a; // send array object contents to screen
```

Does the following code use the friend function ? Why not ?

```
cout << a[3];
```

STATIC FUNCTIONS

Static functions do not act on any object. The function still belongs to the class it was defined in. The main purpose of static functions is to calculate values from parameters rather than instance variables. A good example is mathematical functions. You don't want to always create a object to use them, you just want to see them. For our array class we can have a static method to return the value of our static variable Num. The static variable is counting the number of array objects created. Here are the step by step instructions

(1) declare static function in class, static functions start with the static keyword.

```
static return_data_type function_name(paramater list);

static int howMany();
```

(2) define the static function out side the class. The static keyword is not needed

```
return_data_type class_name function_name()
{
    programming statements
}
```

```
int Array::howMany()
{
    return Num;
}
```

(3) start using, If you are in a class function then you can call the static variable just buy its name

```
int x = howMany();
```

if you are outside the class you must use the class name and resolution operator:: you access the static function

```
void main()
{
    Array a;

    int x = Array::howMany()
}
```

Static functions can only access static variables. Private static functions cannot access private static variables.

LESSON 12 QUESTIONS 1

1. Why do we need a constructor ?
2. Why do we need a destructor ?
3. Who calls the constructor ?
4. Who calls the destructor ?
5. Do you need to delete memory in a copy constructor ?
6. Why do you need to delete memory and re allocate it in the assignment operator ?
7. What happens if you forget to delete memory ?
8. What happens if you forget to allocate memory ?
9. Do you need to delete memory not allocated with the new operator ?
10. If the answer is no why not ?
11. What is a copy constructor and why do we need a copy constructor ?
12. Why do we need operator functions ?
13. Why do some operator functions return by value ?
14. Why must you instantiate an additional Array object in the + operator function ?
15. Give an example of using an operator function and then write the equivalent code ?
16. What is an assignment operator function and why do we need it ?
17. What is the **this** variable, where is it, why can't you change it ?
18. Why does the assignment operator function return a reference to the **this** variable ?
19. Why is a reference variable returned and not a pointer variable ?
20. What is the difference between the copy constructor and the assignment operator ?
21. When is the copy constructor called ? Give a code example.
22. When is the assignment operator function called ? Give a code example.
23. Why is the const keyword used ?
24. Why does the copy constructor not have a **const** at the end of the constructor declaration.?
25. Why does the assignment operator function not have a **const** at the end of the function declaration.
26. Why does the assignment operator function return a **const** value. ?
27. What happens if you forget the keyword const on the return data type of the assignment operator ?
28. Why does the operator + function return a const class and not a costs class reference ?
29. Why does the index operator return a reference ?
30. What happens if two references or pointer class variables are assigned to each other ?
31. Is the assignment operator called in the question above? If the answer is no why not ?

Array Class Code

Here the complete array class definition:

```
// array.cpp
#include <iostream.h>

//typedef int bool; // not needed for all compilers
//typedef enum {false,true}; // not needed for all compilers

// Array class definition
class Array
{
private:
int* Items; // pointer to array elements
int Size; // holds size of array
```

```

static int Num; // number of objects created
public:
Array(); // default constructor
Array(int size); // initializing constructor
Array(const Array& a2); // copy constructor
~Array(); // destructor
const Array& operator=(const Array& a); // assignment operator
bool operator==(const Array& a); // equality operator
const Array operator+(const Array& a); // adding operator
int& operator[](int index); // array index
bool operator()(int data); // is element of
operator int(); // conversion operator return size of array
Array operator++(); // pre increment before assignment
Array operator++(int i); // post decrement after assignment
static int howMany(); // return number of objects created
friend const Array operator+(int num, Array& a2); // add left constant
friend const Array operator+(Array& a2, int num); // add right constant
friend ostream& operator<<(ostream& out, Array& a); // printout
friend istream& operator>>(istream& in, Array& a); // keyboard
};

int Array::Num=0; // initialize number of objects created

// default constructor
Array::Array()
{
    Items = NULL; // set to default values
    Size = 0;
    Num++; // number of objects created
}

// initializing constructor
// make a new array from indicated size
Array::Array(int size)
{
    Items = new int[size]; // allocate memory for array
    Size = size; // set size to user specified size
    Num++; // number of objects created
}

// copy constructor, copy an existing array object
Array::Array(const Array& a2)
{
    Items = new int[a2.Size]; // allocate memory// copy existing array
    for(int i=0;i<a2.Size;i++)Items[i]=a2.Items[i];
    Size = a2.Size; // set size to a2 size
    Num++; // number of objects created
}

```

```

// destructor
Array::~Array()

{
    delete [] Items; // deallocate memory
    Num--; // number of objects deleted
}

// assignment operator
const Array& Array::operator=(const Array& a2)

{
    // don't assign the object to itself
    if(this != &a2)

        {
            delete [] Items; // delete existing array elements
            Items = new int[a2.Size]; // allocate new memory
            // copy array a2 elements
            for(int i=0;i<a2.Size;i++)Items[i]=a2.Items[i];
            Size = a2.Size; // set array size to a2 size
        } // return reference to array

    return *this; // return reference to array object
}

// add two arrays
// make a new array if one array is larger
// pad excess elements with zero's
const Array Array::operator+(const Array& a2)

{
    int size;

    // find the larger size of a1 and a2
    if(Size > a2.Size)size = Size;
    else size = a2.Size;
    Array a(size); // allocate for size of a1
    int data1,data2; // hold values of array elements for addition
    // add two arrays
    for(int i=0;i<size;i++)

        {
            if(i < Size) data1 = Items[i]; // pad a1 with zeros
            else data1 = 0;
            if(i < a2.Size) data2 = a2.Items[i]; // pad a2 with zeros
            else data2 = 0;
            a.Items[i] = data1 + data2; // add array elements
        }

    return a; // return reference to new array object
}

```

```

// index operator
int& Array::operator[](int index)
{
    return Items[index]; // return reference to array element
}

// test if element is in the array
// return true if found otherwise false
bool Array::operator()(int data)
{
    // return true if array element found
    for(int i=0; i<Size; i++) if(Items[i]==data) return true;
    return false;
}

// test if two arrays have same data elements and same sizes
bool Array::operator==(const Array& a2)
{
    if(Size != a2.Size) return false; // return false if sizes different
    for(int i=0; i<Size; i++)
        if(Items[i] != a2.Items[i]) return false;
    return true;
}

// return size of array
Array::operator int()
{
    return Size;
}

// pre increment before assignment
Array Array::operator++()
{
    for(int i=0; i<Size; i++) // add 1 to each item
        Items[i] = Items[i] + 1;
    return *this; // return array
}

// post increment after assignment
Array Array::operator++(int )
{
    Array a(*this); // make copy
    for(int i=0; i<Size; i++) // add 1 to each item
        Items[i] = Items[i] + 1;
    return a; // return copy
}

```

```

// return number of objects created
int Array::howMany()
{
    return Num;
}

// add left hand constant
const Array operator+(int num, Array& a2)
{
    Array a(a2.Size); // allocate for size of a1
    // add num to array
    for(int i=0;i<a2.Size;i++)a.Items[i] = num + a2.Items[i];
    return a; // return reference to new array object
}

// add right hand constant
const Array operator+(Array& a2, int num)
{
    Array a(a2.Size); // allocate for size of a2
    // add num to array
    for(int i=0;i<a2.Size;i++)a.Items[i] = num + a2.Items[i];
    return a; // return reference to new array object
}

// print out array
ostream& operator<<(ostream& out, Array& a)
{
    out << " [";
    // loop for all array items
    for(int i=0;i<a.Size;i++)
    {
        out << a[i]; // print out array element
        if(i < a.Size-1)out << " ";
    }
    out << "]" << endl;
    return out; // return ostream object
}

// initialize array from keyboard
istream& operator >>(istream& in, Array& a)
{
    cout << "enter " << a.Size << " data elements: " << endl;
    // loop to get all array items
    for(int i=0;i<a.Size;i++)
    {
        cout << (i+1) << ": ";
        cin >> a[i]; // put keyboard value into array element
    }
    return in;
}

```

```

// main method
// the main method is used to test all functions of the array class
void main()

{
    const int SIZE = 10;

    // make and initialize array a1
    Array a1(SIZE);

    for(int i=0;i<SIZE;i++)a1[i]=i; // initialize array element to index

    // test index method
    cout << "the fifth element is: " << a1[5] << endl;

    // test if is Element Of array
    if(a1(4))cout << "4 is element of array" << endl;
    else cout << "4 is not element of array" << endl;

    cout << "array a1" << a1 << endl; // print out array a1

    // make and initialize array a2 of 1/2 size
    Array a2(SIZE/2);
    for(i=0;i<SIZE/2;i++)a2[i]=i;
    cout << "array a2" << a2 << endl;

    // add two arrays and print out results
    a1 = a1 + a2;
    cout << "sum of a1 + a2 is" << a1 << endl;

    // add left constant
    a1 = 5 + a2;
    cout << "sum of 5 and a1 is " << a1 << endl;

    // add right constant
    a1 = a2 + 5;
    cout << "sum of 5 and a1 is " << a1 << endl;

    // post increment
    Array a3 = a1++;
    cout << "a3 = a1++:  a1: " << a1 << "a3: " << a3 << endl;

    // pre increment:
    Array a4 = ++a1;
    cout << "a4 = ++a1:  a1 " << a1 << " a4: " << a4 << endl;

    // test Equality operator
    if(a1 == a2) cout << "a1 equals a2" << endl;
    else cout << "a1 not equal a2" << endl;

    // test object pointers
    Array* pa1 = new Array(a1);
    Array* pa2 = new Array(a2);
    pa2 = pa1; // equate two objects

```



```

//test if array pointers are equal
if(pa1 == pa2)cout << "pa1 == pa2" << endl;
else cout << "pa1 != pa2" << endl;

//test if array data are equal
if(*pa1 == *pa2)cout << "*pa1 == *pa2" << endl;
else cout << "*pa1 != *pa2" << endl;

// print out arrays
cout << "array a1" << *pa1 << endl;
cout << "array a2" << *pa2 << endl;

// test add again using pointers
*pa1 = *pa1 + *pa2;

// print out arrays
cout << "array a1" << *pa1 << endl;
cout << "array a2" << *pa2 << endl;
Array a5(SIZE/2); // make another array
cin >> a5; // test keyboard entry
cout << a5; // print out
a5[0] = a5[4]; // assign values by swapping array elements
// test if 3 is in array 5
if(a5(3))cout << "3 is in array 5" << endl;
else cout << "3 is not in array 5" << endl;
cout << a5; // print out array
cout << "the size of the array is " << int(a5) << endl;

cout << "there are: " << Array::howMany() << " left" << endl;
} /* end main */

```

Program Output:

```

the fifth element is: 5
4 is element of array
array a1 [0 1 2 3 4 5 6 7 8 9]
array a2 [0 1 2 3 4]
sum of a1 + a2 is [0 2 4 6 8 5 6 7 8 9]
sum of 5 and a1 is [0 1 2 3 4]
sum of 5 and a1 is [0 1 2 3 4]
a3 = a1++: a1: [1 2 3 4 5]
a3: [0 1 2 3 4]
a4 = ++a1: a1 [2 3 4 5 6]
a4: [2 3 4 5 6]
a1 not equal a2
pa1 == pa2
*pa1 == *pa2

```

```

array a1 [2 3 4 5 6]
array a2 [2 3 4 5 6]
array a1 [4 6 8 10 12]
array a2 [4 6 8 10 12]
enter 5 data elements:
enter 5 data elements:
1: 1
2: 2
3: 3
4: 4
5: 5
[1 2 3 4 5]
3 is in array 5
[5 2 3 4 5]
the size of the array is 5
there are: 7 left

```

LESSON 12 EXERCISE 1

Type in the Array class and get it working. Use the debugger to trace through the program to see it work. Watch all the different constructors and destructors called. **Write** your own main method to test every function. Call your main method L12ex1.cpp.

LESSON 12 EXERCISE 2

Write the decrementor operator functions to do equations like $a3 = a1 -$; and $a3 = - -a1$; for the Array class.

LESSON 12 EXERCISE 3

Write operator functions to subtract, multiply and to divide arrays For the Array class.

LESSON 12 EXERCISE 4

Change the data type to double for the Array class

LESSON 12 EXERCISE 5

Remove the static modifier on static variable Num and run the program.

C++ PROGRAMMERS GUIDE LESSON 13

File:	CppGuideL13.doc
Date Started:	July 12, 1998
Last Update:	Apr 4, 2002
Version:	4.0

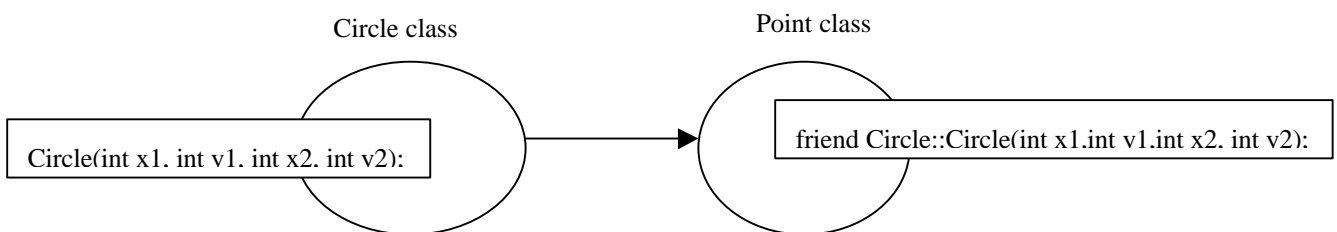
LESSON 13 C++ FRIEND, TEMPLATE, VIRTUAL AND ABSTRACT CLASSES

FRIEND FUNCTIONS OF CLASSES

A class can also make a friend of **functions** in another classes. In the following example we have 2 classes a Circle class and a Point class. The Circle class constructor wants to use the private methods and variables of the Point class to do calculations on Points(like find the distance between two points). The only way to do this is the Point class makes a friend of the Circle class **constructor** by declaring it a **friend** in its class definition.

```
friend class_name :: class_function_name(parameter list);
```

```
friend Circle::Circle(int x1,int y1,int x2, int y2);
```



Here is the complete code:

```
#include <iostream.h>
#include <math.h>
// L13p1.cpp
// this program demonstrates using friend functions of classes
// circle class
class Circle
{
private:
double Radius; // radius of circle
int Height;
int Width;
public:
Circle(int x1, int y1, int x2, int y2); // initializing constructor
friend ostream& operator<< (ostream& out,Circle& c); // print out circle
};
```

The Circle class constructor is a friend of the Point class. The Circle class constructor gets to use the functions and methods of the Point class.

```

// point class
class Point
{
private:
int X,Y; // x,y point
public:
Point(int x, int y); // initializing constructor
private:
double distance(Point p2);
friend Circle::Circle(int x1,int y1,int x2, int y2);
friend ostream& operator<< (ostream& out, Point& pt); // print info
};

// point class implementation
Point::Point(int x, int y)
{
X = x;
Y = y;
}

// calculate distance between 2 points
double Point::distance(Point p2)
{
double l1 = X - p2.X;
double l2 = Y - p2.Y;
return sqrt(l1*l1+l2*l2);
}

// Circle class implementation
Circle::Circle(int x1, int y1, int x2, int y2)
{
Point p1(x1,y1);
cout << "got point 1: " << p1 << endl;
Point p2(x2,y2);
cout << "got point 2: " << p2 << endl;
Radius = p1.distance(p2);
Width = p2.X-p1.X;
Height = p2.Y-p1.Y;
}

// friend functions
// print out info about a point
ostream& operator<<(ostream& out,Point& pt)
{
cout << " x: " << pt.X << " y: " << pt.Y;
return out;
}

```

```
// print out info about a circle
ostream& operator<<(ostream& out,Circle& c)
{
    cout << "circle height: " << c.Height << " width: " << c.Width;
    cout << " radius: " << c.Radius << endl;
    return out;
}
```

program output:

```
// main program to test circle class
void main()
{
    Circle c1(0,0,3,4); // make a circle
    cout << c1; // print out circle info
}
```

```
got point 1: x: 0 y : 0
got point 2: x: 3 y : 4
circle height: 4 width: 3
radius: 5
```

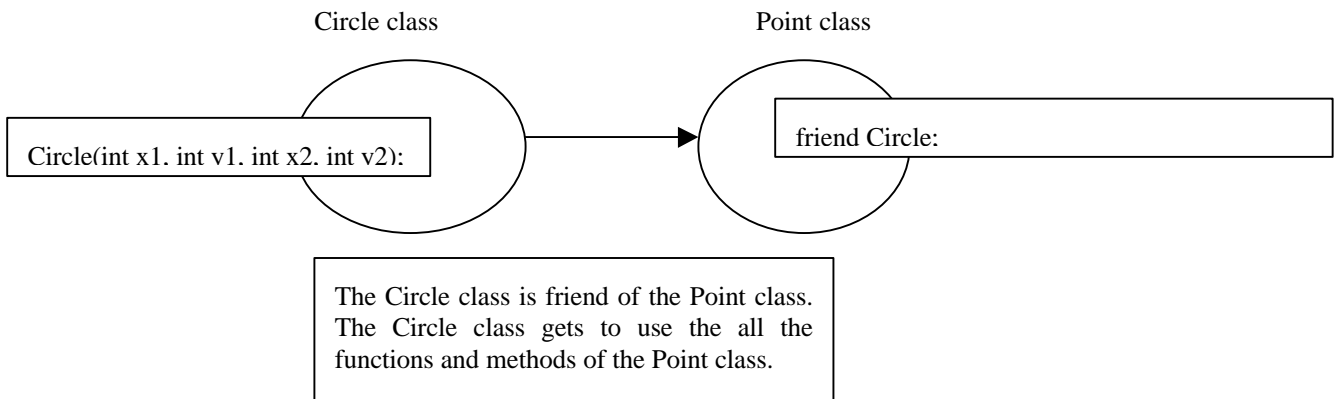
FRIEND CLASSES

Classes may be friends to. A class can make a friend of another class by including the **friend** key word and the name of the class to be its friend inside the class definition.

friend class_name;

friend Circle;

Classes making friends of other classes is more convenient than classes making friends of individual functions in a class. It is basically the same thing except the class makes a friend of private functions and variables in the class. The friend class gets to use **all** the functions and variables. We use the preceding example again. The Circle class wants to use the private methods and variables of the Point class to do calculations on Points (like find the distance between two points). The Point class makes a friend of the Circle class by declaring it to be a friend in its class definition.



Here is the code:

```
// L13p2.cpp
// this program demonstrates using friend classes and friend functions
#include <iostream.h>
#include <math.h>
```

```

class Circle // circle class
{
private:
double Radius;
int Height;
int Width;
public:
Circle(int x1, int y1, int x2, int y2); // initializing constructor
friend ostream& operator<< (ostream& out,Circle& c); // print out circle
};

```

```

class Point // point class
{
private:
int X; // x point
int Y; // y point
public:
Point(int x, int y); // initializing constructor
private:
double distance(Point p2);
friend Circle;
friend ostream& operator<< (ostream& out, Point& pt); // print
};

```

```

// point class implementation
// initialize a point
Point::Point(int x, int y)
{
X = x;
Y = y; // set to user values
}

```

```

// calculate distance between 2 points
double Point::distance(Point p2)
{
double l1 = X - p2.X;
double l2 = Y - p2.Y;
return sqrt(l1*l1+l2*l2);
}

```

```

// Circle class implementation
Circle::Circle(int x1, int y1, int x2, int y2)
{
Point p1(x1,y1);
cout << "got point 1: " << p1 << endl;
Point p2(x2,y2);
cout << "got point 2: " << p2 << endl;
Radius = p1.distance(p2);
Width = p2.X-p1.X;
Height = p2.Y-p1.Y;
}

```

```

// friend functions
// print out info about a point
ostream& operator<<(ostream& out,Point& pt)
{
    cout << " x: " << pt.X << " y: " << pt.Y;
    return out;
}

// print out info about a circle
ostream& operator<<(ostream& out,Circle& c)
{
    cout << "circle height: " << c.Height << " width: " << c.Width;
    cout << " radius: " << c.Radius << endl;
    return out;
}

// main program to test circle class
void main()
{
    Circle c1(0,0,3,4); // make a circle
    cout << c1; // print out circle info
}

```

program output:

```

got point 1: x: 0 y : 0
got point 2: x: 3 y : 4
circle height: 4 width: 3
radius: 5

```

Friendship is only one way. In order for both classes to be friends, each needs to declare each other as a friend in each class.

LESSON13 EXERCISE 1

Have each class make a friend of each other. You may need to forward reference the Point class just ahead of the Circle class.

```
class Point; // forward reference Point class.
```

Have the point class access the private variables of the Circle class. Call your program L13Ex1.cpp.

CLASS TEMPLATES

If we need a Array class that can handle **different** data or object type, then we would have to re-write the Array class with a different data or object type, which would be a lot of work to do. Possible different data type's could be int, float String etc. There must be a better way to do this ? Yes there is and its called **templates!** Template classes let you specify the data type you want. It substitutes your data type for a template data type name. A template class is defined similar to a non-template class. The only difference is the keyword **template** proceeding a template class name and an object type name enclosed in < >. Any function can be a template not just class functions.

defining a template class:

```

template <class_name template_type_name> class class_name
{
private:
template_type_name variable_name;
public:
class_name ( paramater_type parameter_name);
return_data_type function_name (paramater_type paramater_name);
};

// example defining a template class
template <class T> class Array

{
private:
T* Items;
public:
Array (int size);
T& operator[] (int index);
};

```

instantiating a template class object

To instantiate, a template class you must specify the template data type you are going to use.

```

class_name < data_type > class_variable_name ( argument_list );

Array <double> a (10);

```

If you allocate an object with a pointer than you must also specify the template data type twice.

```

class_name < data_type > pointer_variable = new Class_name<data_type>( argument_list );

Array <double> pa = new Array<double>(10);

```

using template class functions

Using template class functions is no different then using non-template class functions. You still call the class function or variable using the object name.

```

class_variable_name . function_name
operator class_variable_name

cin >> a;
a[2] = 1.34563;
cout << a;

```


defining a template class function

A template class function definition is preceded by the key word `template` and the class keyword and substituted data type enclosed by `<>`. You must also specify the template substitution data type name with the class name also enclosed by `<>`.

```
template <class data_type_name>
return_data_type class_name <data_type_name> :: function_name( paramater_list)
{
    variables and statements
}

// index operator
template <class T>
T& Array<T>::operator[](int index)
{
    return Items[index]; // return reference to array element
}
```

ARRAY TEMPLATE CLASS

We can now re-write the basic array class as a template class. Template classes are all written as a *.h or *.hpp file. Template classes **do not need a *.cpp file** because there is no code for a template class. The compiler will generate the code from the template class definition during compile time.

```
/* tarray.hpp */
#include <iostream.h>
//typedef int bool; // not required by all compilers
//enum { false,true}; // not required by all compilers

template<class T> class Array
{
private:
    T* Items; // pointer to array elements
    int Size; // size of array
public:
    Array(); // default constructor
    Array(int size); // initializing constructor
    Array(Array& a2); // copy constructor
    ~Array(); // destructor
    const Array& operator=(const Array& a); // assignment operator
    bool operator==(const Array& a); // equality operator
    const Array operator+(const Array& a); // adding operator
    T& operator[](int index); // array index
    bool operator()(T data); // is element of
    friend ostream& operator<<(ostream& out, Array<T>& a);
    friend istream& operator>>(istream& in, Array<T>& a);
};
```

**No cpp files for template
classes only if they are
included in a main function.**

```

// default constructor
template <class T> Array<T>::Array()

{
    Items = NULL;
    Size = 0;
}

// initializing constructor
template <class T> Array<T>::Array(int size)

{
    Items = new T[size];
    Size = size;
}

// copy constructor
template <class T> Array<T>::Array(Array& a2)

{
    Items = new T[a2.Size];
    // copy existing array
    for(int i=0;i<a2.Size;i++)Items[i]=a2.Items[i];
    Size = a2.Size;
}

// destructor
template <class T> Array<T>::~~Array()

{
    delete [] Items;
    Items = 0; // set to zero for debug
}

// assignment operator
template <class T>const Array<T>& Array<T>::operator=(const Array& a2)

{
    if(this != &a2)

    {
        delete [] Items; // delete existing array elements
        Items = new T[a2.Size]; // allocate new memory

        // copy array a2 elements
        for(int i=0;i<a2.Size;i++)Items[i]=a2.Items[i];

        Size = a2.Size; // set array size
    }

    return *this; // return reference to array object
}

```

```

// add two arrays
// make a new array if one array is larger
// pad excess elements with zero's
template <class T> const Array<T> Array<T>::operator+(const Array& a2)

{
    int size;

    // check if a1 larger than a2
    if(Size > a2.Size) size = Size;
    else size = a2.Size;
    Array<T> ta(size); // allocate for size a2
    T data1,data2; // hold values of array elements

    // add two arrays
    for(int i=0;i<size;i++)

        {
            if(i < Size) data1 = Items[i];
            else data1 = 0; // pad zeros for a1
            if(i < a2.Size) data2 = a2.Items[i];
            else data2 = 0; // pad zeros for a2
            ta.Items[i] = data1 + data2;
        }

    return ta; // return reference to array object
}

// index operator
template <class T> T& Array<T>::operator[](int index)

{
    return Items[index]; // return reference to array element
}

// test if element is in the array
// return true if found otherwise false
template <class T> bool Array<T>::operator()(T data)

{
    for(int i=0;i<Size;i++)if(Items[i]==data)return true;
    return false;
}

// test if two arrays have same data elements
// and same sizes
template <class T> bool Array<T>::operator==(const Array& a2)

{
    for(int i=0; i<Size; i++)
        if(Items[i] != a2.Items[i]) return false;
    return true;
}

```

```

// print out array
template <class T>
ostream& operator<<(ostream& out, Array<T>& a)

{
    cout << " [";
    // print out array elements
    for(int i=0;i<a.Size;i++)

        {
            cout << a[i]; // print out array element
            if(i < a.Size-1)cout << " ";
        }

    cout << "]" << endl;
    return out;
}

// initialize array from keyboard
template <class T>
istream& operator>>(istream& in, Array<T>& a)

{
    cout << "enter " << a.Size << " data elements: " << endl;

    // get data elements from keyboard
    for(int i=0;i<a.Size;i++)

        {
            cout << (i+1) << ": ";
            cin >> a[i]; // put keyboard value into array element
        }

    return in;
}

```

You may need a separate main in a cpp file, or you can include the template class inside the cpp file. All standalone template classes must be a hpp file not a cpp file.

```

/* tarray.cpp */
#include <iostream.h>
#include "tarray.hpp"

// the main function is used to test the array class. Every time you
// instantiate an Array template class you need to specify the desired data // type.
void main()

{
    const int SIZE = 10;
    int i;

```

```

// make and initialize array a1
Array<double> a1(SIZE);
for(i=0;i<SIZE;i++)a1[i]=i * 1.1;

// test get method
cout << "the fifth element is: " << a1[5] << endl;

// test if is Element Of array
if(a1(4))cout << "4 is element of array" << endl;
else cout << "4 is not element of array" << endl;

// print out array a1
cout << "array a1" << a1 << endl;

// make and initialize array a2
Array<double> a2(SIZE/2);
for(i=0;i<SIZE/2;i++)a2[i]= i * 2.2;
cout << "array a2" << a2 << endl;

// add two arrays and print out results
a1 = a1 + a2;
cout << "sum of a1 + a2 is" << a1 << endl;

// test Equality operator
if(a1 == a2)cout << "a1 equals a2" << endl;
else cout << "a1 not equal a2" << endl;

// test object pointers
Array<double>* pa1 = new Array<double>(a1);
Array<double>* pa2 = new Array<double>(a2);
pa2 = pa1; // equate two objects

//test if array pointers are equal
if(pa1 == pa2)cout << "pa1 == pa2" << endl;
else cout << "pa1 != pa2" << endl;

//test if array data are equal
if(*pa1 == *pa2)cout << "*pa1 == *pa2" << endl;
else cout << "*pa1 != *pa2" << endl;

// print out arrays
cout << "array a1" << *pa1 << endl;
cout << "array a2" << *pa2 << endl;

// test add again
*pa1 = *pa1 + *pa2;

// print out arrays
cout << "array a1" << *pa1 << endl;
cout << "array a2" << *pa2 << endl;
Array<double> a3(5); // make another array
cin >> a3; // test keyboard entry
cout << a3; // print out

```

```

a3[0] = a3[4]; // assign values by swapping array elements

// test if 3 is in array 3
if(a3(3))cout << "3 is in array 3" << endl;
else cout << "3 is not in array 3" << endl;

cout << a3; // print out array
} /* end main */

```

program output:

```

the fifth element is: 5.5
4 is not element of array
array a1 [0 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9]
array a2 [0 2.2 4.4 6.6 8.8]
sum of a1 + a2 is [0 3.3 6.6 9.9 13.2 5.5 6.6 7.7 8.8 9.9]
a1 not equal a2
pa1 == pa2
*pa1 == *pa2
array a1 [0 3.3 6.6 9.9 13.2 5.5 6.6 7.7 8.8 9.9]
array a2 [0 3.3 6.6 9.9 13.2 5.5 6.6 7.7 8.8 9.9]
array a1 [0 6.6 13.2 19.8 26.4 11 13.2 15.4 17.6 19.8]
array a2 [0 6.6 13.2 19.8 26.4 11 13.2 15.4 17.6 19.8]
enter 5 data elements:
1: 2: 3: 4: 5: [1.1 2.2 3.3 4.4 5.5]
3.3 is in array 3
[5.5 2.2 3.3 4.4 5.5]

```

LESSON 13 EXERCISE 2

Add to the template class the prefix and postfix operators' ++ and -- to iterate through the array. When you use the [] operator to read or write a value set a current index variable. Use the ++ and -- to go forward or backyard through the array. Use double data types. You may include the array template class in this file or you may put it into its own file called tarray.hpp. Remember template classes do not have corresponding *.cpp files.

LESSON 13 EXERCISE 3

Make an **iterator** template class for the above array class. Every time the template iterator is called it will get the next item. Your iterator class will need mechanisms to reset to start, the iterations from the beginning of the array. You will need a function called **reset()**, and operator functions ++ to retrieve and go forward to next item, and operator function -- to retrieve and go backward from an item. Your iterator must also have a operator () functions to read and write data at the current iterator location. It would be nice if your iterator can do a search and iterate from the search point, you could use the operator ! for this. Make your iterator template class an abstract class. The abstract class definition is as follows. You must derive your own iterator class from the base abstract class. You can call the iterator for the array class ArrayIterator. Your iterator class will take an array class as an argument. Call your file L13ex3.cpp. You may include the array template class in this file or you may put it into its own file called tarrayitr.hpp. Remember template classes do not have corresponding *.cpp files.

```

template <class T>
class Iterator
{
public:
virtual int reset() = 0; // set iterator to start
virtual T& operator =()=0; // read/write current value
virtual int operator!()=0; // return true if item found
virtual T& operator++()=0; // advance to next item
virtual T& operator--()=0; // go back a item
}

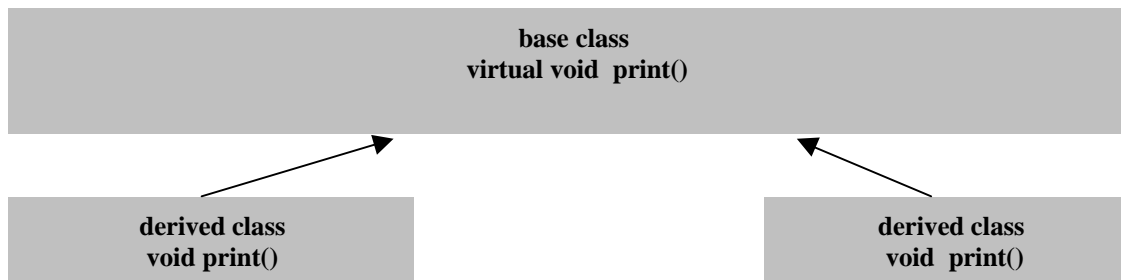
// test driver
main()
{
Array<int>a;
ArrayIterator<int>iter(a);
iter = 7;
int x = iter++;
if(!iter)cout << "cannot find 7" << endl;
else cout << "found 7" << endl;
iter.reset();
}

```

VIRTUAL, PURE VIRTUAL FUNCTIONS AND ABSTRACT CLASSES

Virtual Functions

In previous lessons we studied polymorphism that allows us to call the same name functions from different derived classes stored in an array of base class pointers. C++ kept track of which derived class function to call. For example, we have a base class with a print function and each derived class also has a print. Next we make an array of base class pointers representing derived class objects. When each print function is called from the array of base class pointer then a different message would be printed out for each different derived class..



The base class function print must have the keyword modifier virtual. The base class print function is known as a **virtual** function. Each derived class will also class have a print function. Polymorphism allows us to call the correct derived class function at run time. Which **Virtual** functions to call is determined at run time not at compile time. When the base class function is **virtual**, all derived class functions having the same name and parameter data type will automatically inherit virtual as well. Virtual propagates through all the derived classes. To enable polymorphism C++ the base class makes a table called a VTable (virtual table) that lists all the virtual derived classes, so it knows which function to call for each derived object.

```
// L13p2.cpp
#include <iostream.h>
```

```
// base class
class Base
```

```
{
protected:
int** A;
int Size;
public:
// base constructor
Base(int size)
{
    A = new int* [size];
    Size = size;
}

// base destructor
~Base()
{
    delete A;
    cout << "delete base" << endl;
}

// virtual print function
virtual void print()
{
    cout << "base print" << endl;
}
};
```

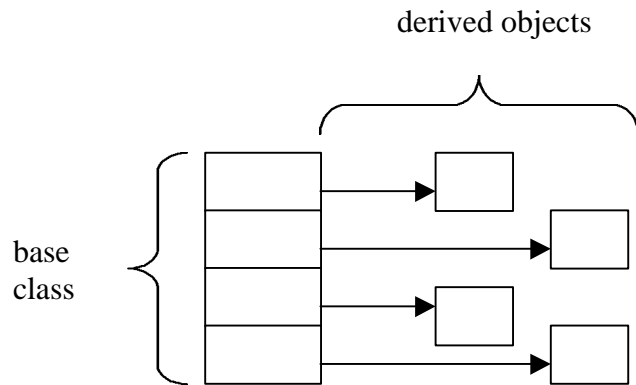
```
// derived class
class Derived: public Base
```

```
{
public:
// derived constructor
Derived(int size): Base(size)
{
    for(int i=0;i<size;i++)
        A[i] = new int[size];
}

// derived destructor
~Derived()
{
    cout << "delete derived" << endl;
    for(int i=0;i<Size;i++)
        delete A[i];
}
```

virtual table

The base class makes a table called a VTable (virtual table) that lists all the derived classes, so it knows which function to call for each derived




```

void print()
{
    cout << "derived print" << endl;
}

};

void main()
{
    Base* d = new Derived(5);
    d->print();
    delete d;
}

```

program output:

```

derived print
delete base

```

In the above example the derived class print is called and the base class destructor.

virtual destructors

Constructors cannot be inherited or overridden but destructors can be. If the derived class is represented by a **derived class** pointer, both the derived class constructor and the base class constructors are called because destructors are inherited. If the derived class is represented by a base class pointer, the derived class destructor is not called only the base class destructor. If the derived class allocates memory its destructor will not be called. The base class destructor must be virtual for the derived class destructor to be called.

```
virtual ~Base();
```

When the base destructor is virtual, both the derived and base destructors are called, the derived class destructor is called first. Here is an example using a derived class that allocates memory as an array.

```

// L13p3.cpp
#include <iostream.h>

// base
class ADT
{
private:
    int size;
public:
    ADT(int size);
    ~ADT();
};

ADT::ADT(int size)
{
    this->size=size;
}

ADT::~~ADT()
{
    cout << "ADT destructor called" << endl;
}

```

```

// derived
class MyArray: public ADT
{
private:
int *pa;
public:
MyArray(int size);
~MyArray();
};

MyArray::MyArray(int size): ADT(size)
{
pa = new int[size];
}

MyArray::~~MyArray()
{
cout << "MyArray destructor called" << endl; \
delete[] pa;
}

// driver
void main()
{
ADT a1(5);
ADT* pa2 = new MyArray(5);
delete pa2;
}

```

The output is as follows: Notice the derived MyArray destructor is not called.

```

ADT destructor called
ADT destructor called

```

If we make the ADT destructor virtual

```
virtual ~ADT();
```

The MyArray destructor is only called for the allocated object pointed to by the pointer. Do you know why ?

```

MyArray destructor called
ADT destructor called
ADT destructor called

```

LESSON13 EXERCISE 4

In the proceeding example add the keyword `virtual` to the base destructor. What did you notice ? Call your program L13Ex4.cpp.

ABSTRACT CLASSES

An **abstract** class is a base class that is required to have a derived class. A class is abstract class if one of its functions is pure virtual. The function is made **pure virtual** by the assignment statement `"= 0"`. Non abstract classes are known as **concrete** classes.

```
virtual void func() = 0;
```

A virtual function cannot be static because they will always need, to be many copies of the derived overridden functions made per class instantiation. The derived class must implement the functions in the abstract class. Why ? If you do not implement the function in the derived class, the print function would have no statements. An abstract class may have pure virtual, virtual and non-virtual functions. You may not instantiate abstract base classes. Why ? There is no code for the pure virtual functions to instantiate. Do abstract classes need constructors ? Abstract classes do not need constructors because they cannot be instantiated. Abstract classes may need constructor's because the derived class may want to initialize member variables of the abstract base class. Why do we need abstract classes ? Abstract classes force that the base class will have a derived class. The abstract class is the definition model, that all derived classes have to follow. Abstract classes are used mostly in a large team oriented programming project, so that all programmers implement all the required functions in each class. An abstract class may represent many derived objects.

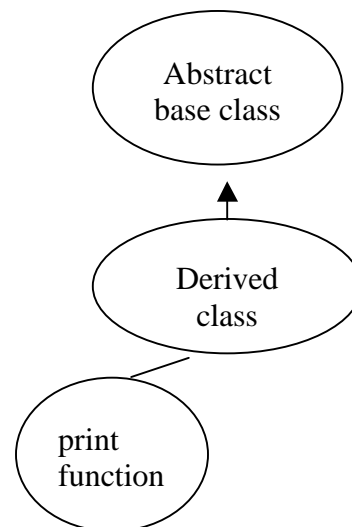
A pure virtual function is a function with no code. The code is to be implemented in the derived class.

example abstract class:

```
// L13p3.cpp
#include <iostream.h>

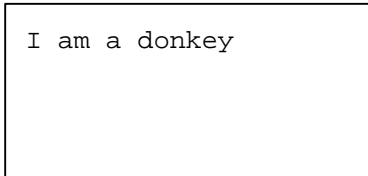
// abstract class
class Base
{
public :
virtual void print()=0; // pure virtual
};

// derived class
class Derived: public Base
{
public:
void print();
};
```



```
// print function of derived class
void Derived::print()
{
    cout << "I am a donkey";
}

// using an abstract class
int main()
{
    Derived d;
    d.print();
    return 0;
}
```



I am a donkey

LESSON13 EXERCISE 5

Type in the above code and try to instantiate the Base class object.



This space intentionally left blank

OVERRIDING OVERLOADED FUNCTIONS

Overloading is when a class has the **same function name** but each function has a (**different parameter list, same class**). **Overriding** is when the derived class function has the same name as the base class function, (**same name, same parameter list, different classes**). If the base class contains overloaded virtual functions that are overridden in the derived class, the compiler will automatic choose the base functions that are not **overridden** but **overloaded**. Overriding only worked for pointer variables since a base pointer has to reference a derived object. For example the following base class has 2 overloaded functions but the derived class just overrides one of them.

```
// overriding overloaded functions
#include <iostream.h>
```

```
// base class
class Base
{
public:
```

```
int square(int x)
```

```
{
    cout << "base int" << endl;
    return x*x;
}
```

```
virtual double square(double x)
```

```
{
    cout << "base double" << endl;
    return x*x;
}
```

```
};
```

```
// derived class
```

```
class Derived: public Base
{
public:
```

```
double square(double x)
```

```
{
    cout << "derived double" << endl;
    return x*x;
}
```

```
};
```

```
void main()
```

```
{
    Base* d = new Derived();
    cout << "the square of 5 is: " << d->square(5) << endl; // call base int
    cout << "the square of 5.5 is: " << d->square(5.5) << endl; // call double square
}
```

Overloading - same name, different parameter list, same class
Overriding - same name, same parameter list, different class

overloaded functions

overriding functions

The derived class square function overrides the base class square function

Program Output:

```
base int
the square of 5 is: 25
derived double
the square of 5.5 is: 30.25
```

LESSON13 EXERCISE 6

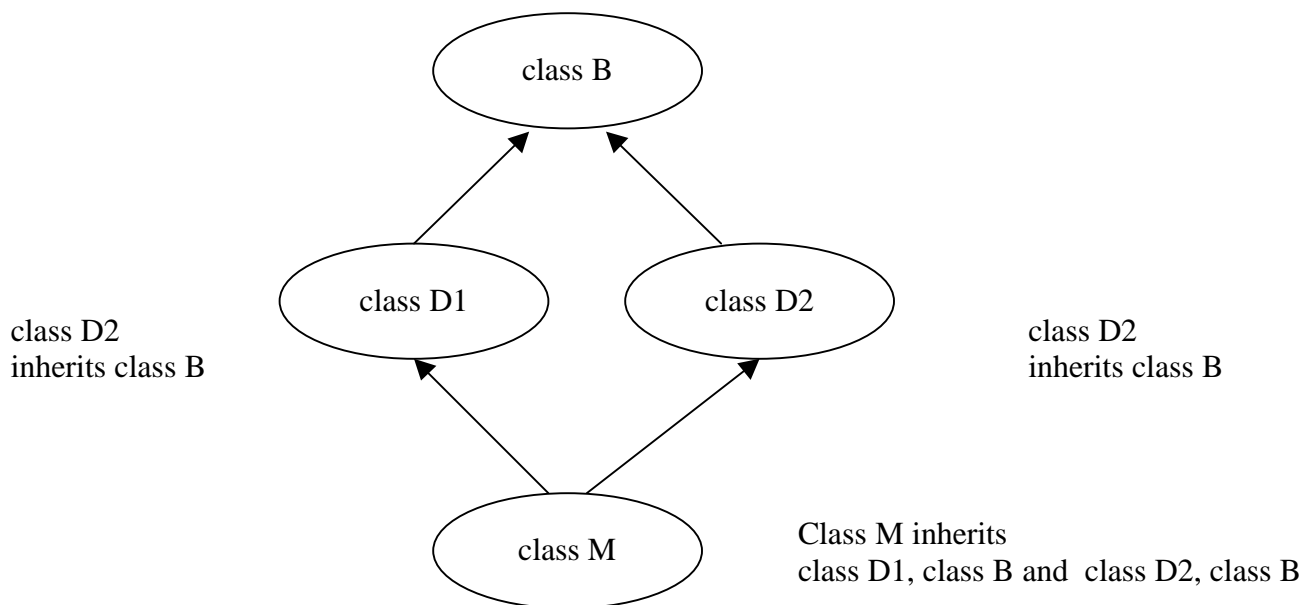
Type in the above program and trace through it.

C++ PROGRAMMERS GUIDE LESSON 14

File:	C++GuideL14.doc
Date Started:	July 12, 1998
Last Update:	Apr 4, 2002
Version:	4.0

LESSON 14 C++ VIRTUAL CLASSES, EXCEPTIONS AND RUNTIME IDENTIFICATION**VIRTUAL CLASSES**

A derived class may inherit the same base class more than once, this is called **multiple inheritance**.



Class D1 and D2 both will inherit class B. Class M will inherit everything its base classes D1 and D2 inherits. Class M will inherit Class D1 and class B and Class D2 and class B. Class B gets inherited twice. The problem can be solved by specifying the keyword **virtual** in the base class initialization list. All derived classes will inherit the virtual keyword.

```

// derived class inherits virtual base class
class D1: virtual public Base
{
public:
D1()
{
    cout << "Derived D1" << endl;
}
};
  
```

Virtual base constructors are invoked before the **non-virtual** constructors. If the virtual base constructors are derived from a non-virtual base constructor the non-virtual base constructors are invoked first. Here is the example program demonstrating virtual base classes.

```
// virtual base classes
// L14p1.cpp
#include <iostream.h>
// base class
class Base
{
public:
Base(){cout << "Base" << endl;};
};

// derived class inherits virtual base class
class D1: virtual public Base
{
public:
D1(){cout << "Derived D1" << endl;};
};

// derived class inherits virtual base class
class D2: virtual public Base
{
public:
D2(){cout << "Derived D2" << endl;};
};

// derived class inherits base classes containing virtual base classes
class M: public D1,public D2
{
public:
M(){cout << "Multiple" << endl;};
};

void main()
{
M m;
}
```

Program Output:

```
Base
Derived D1
Derived D2
Multiple
```

LESSON 14 EXERCISE 1

Run the above example program, with and without the virtual keyword. What is the difference ? Call your program L14Ex1.cpp.

CLASS STATIC VARIABLES

Variables defined in a class may also be declared as **static**. Static variables belong to the class not to an instance of a class (an object). The static variable is shared between all objects and is located in the memory for the class and its functions. This means each object can access the static variable but does not contain one itself. If one object changes a static variable then all other objects will access the updated value because there is only 1 static variable. Static variables are usually public. Static variables are also called class variables. To declare a static variable:

```
static data_type variable_name;

static int num;
```

Some compilers allow you to initialize the static variable when you declare it.

```
static int num = 0;
```

Other compilers require you to initialize the static variable outside the class definition using the data type, class name and the resolution operator.

```
static data_type variable_name = constant;

int MyClass::num = 0;
```

You access static variables in functions belonging to a class by using its variable name.

```
num = num + 1; // access static variable using name
```

Static variables can also be accessed outside a function not belonging to a class by using the class name or object name.

```
MyClass::num++; // access static variable using class name
myobject.num++; // access static variable using object name
```

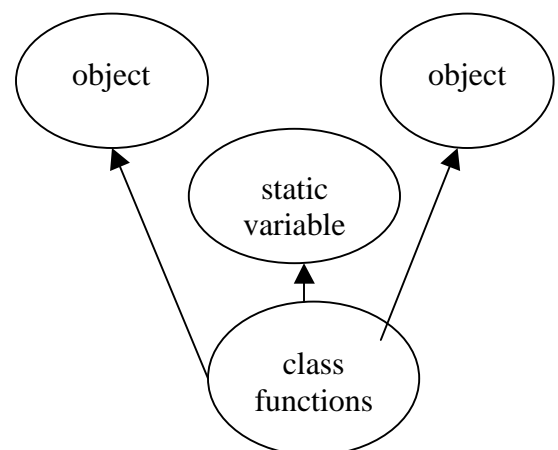
The following program demonstrates use of a static variable and a non-static variable. When you run this program, every time a new object is constructed the static variable is incremented whereas the non-static variable is not.

```
// L14p2.cpp

#include <iostream.h>

// program to demonstrate using static variables

class StaticTest
{
public:
static int num; // declare static variable
int count;     // declare non-static variable
public:
```




```
// default constructor
StaticTest():count(0)
{
count++; // clear non-static variable
num++;
}
```

Many objects but only 1 static variable. The functions accesses objects and static variable.

```
};
```

```
int StaticTest::num=0; // initialize static variable
```

```
void main()
```

```
{
StaticTest st1; // make a StaticTest object
StaticTest st2; // make another StaticTest object
```

```
// print out results
```

```
cout << "st1: num: " << st1.num << " count: " << st1.count << endl;
cout << "st2: num: " << st2.num << " count: " << st2.count << endl;
}
```

program output:

```
st1: num: 2 count: 1
```

```
st2: num: 2 count: 1
```

LESSON 14 EXERCISE 2

Type in the above program, remove the static modifier and re-run the program. Repeat both steps until you understand what a static variable is. Call your program L14Ex2.cpp.

EXCEPTION HANDLING

Exceptions occur when an abnormal error is caused when your program is running. Exceptions need to be caught before they happen. The mechanism to do this is known as a **try catch** block. A catch block must immediately follow a try block. The try block tests code or function that may generate an exception. The catch block will catch the generated exception. The catch block looks for a matching data type object on the execution stack to catch. In our example, the exception data type is **int** and the object is x.. For the try catch mechanism to work the function called in the try block must generate an exception.

```
try
{
call function
}
```

```
catch(exception_data_type object)
{
report error condition
}
```

```
try
{
x = test();
}
```

```
catch(int x)
{
cout << "an error has occurred" << endl;
}
```

What kind of data to catch

A try catch block lets your program continue to run even when an abnormal error occurs.

catching any exception

The (...) can be used to catch any generated exception.

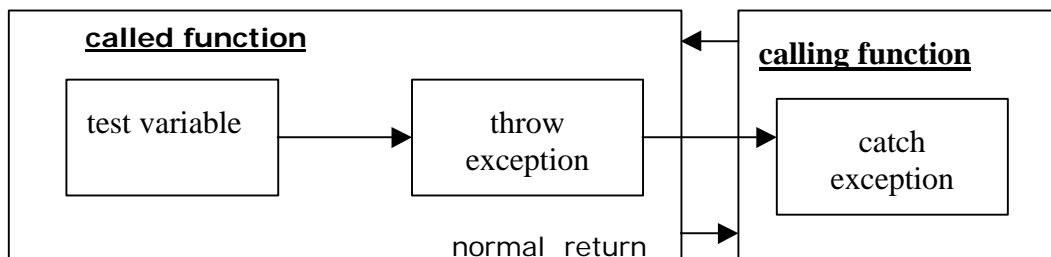
<pre> try { call function } catch(...) { report error condition } </pre>	<pre> try { x = test(); } catch (...) // catch any abnormal error { cout << "some problem has occurred" << endl; } </pre>
---	--

generating exceptions

A function can generate its own exception by use of the **throw** operator.

```
throw(int x);
```

You usually throw the exception in one function the called function and catch the exception in another function the called function. In some function, you first test a variable for some abnormal value, if true then throw an exception. The exception is caught in the calling function. If the variable has no abnormal value then the called function returns normally.



The throw exception is caught by the catch block. To determine which catch block goes with which throw statement the arguments must match the **data type** (not variable name). Here is the example program using try, catch and throw:

```

#include <iostream.h>

// L14P3.cpp
// divide a number
int div(int a,int b)
{
    if(b == 0)throw(b); // generate an exception if divide buy zero possible
    else return a/b;
}

```

```
// main function
void main()
{
    int x=0; // object that will generate exception

    try
    {
        x = div(5,x); // try to divide
    }

    // catch any exceptions from this data type
    catch(int x)
    {
        cout << "error divide by zero " << x << endl;
    }
}
```

error divide by zero

Making an Exception class

When your program needs to handle all kinds of exceptions it is better to make your own exception class to handle each exception. The exception class can handle printing out the error for you.

```
// own exception class
class DivZero
{
    void print()
    {
        cout << error divide by zero << endl;
    }
};
```

When you need to throw an exception you just create an exception object.

```
if(x == 0) throw DivZero(); // generate exception object
```

When you catch the object you can use the object's print function to print out the error message.

```
// catch your own exception
catch(DivZero e)
```

```
{
    e.print(); // print out info about this exception
}
```

exception object

catching multiple exceptions

Sometimes one function may generate many different types of errors. It would be more convenient to have a separate catch block for each error encountered. To handle many exceptions we use an exception class object for each error condition we want to catch.

	<pre>// L14p4.cpp #include <iostream.h> // exception classes // divide by zero class DivZero { public: void print(){cout << "divide by zero"<< endl;} }; // number overflow class Overflow { public: void print(){cout << "input number too large "<< endl;} }; // exception test class class Test</pre>
<p><i>define exception classes</i></p> <p><i>class name</i></p> <pre>{</pre> <p><i>class variables</i></p> <p><i>class constructor</i></p> <p><i>class functions</i></p> <p><i>class exceptions</i></p> <pre>};</pre>	<pre> { private: int op1,op2; public: Test(int op1,int op2); int div(); }; Test::Test(int op1, int op2) { if(op1 > 100) throw Overflow(); if(op2 > 100) throw Overflow(); this->op1=op1; this->op2=op2; } int Test::div() { if(op2 == 0)throw DivZero(); return op1/op2; }</pre>
<p><i>class function definitions that generate exceptions</i></p>	

<pre> main function try { // code to test } catch(class_object e) {} catch(class_object e) {} </pre>	<pre> void main() { try { Test t1(5,10); cout << t1.div()<<endl; Test t2(10,0); cout << t2.div()<<endl; Test t3(200,10); cout << t3.div()<<endl; Test t4(100,200); cout << t4.div()<<endl; } catch(DivZero e){e.print();} catch(Overflow e){e.print();} } </pre>
--	---

EXIT STATEMENT **stdlib.h**

The exit function terminates the calling process. Before termination, all files are closed, any buffered output (waiting to be output) is written, and any registered "exit functions" (posted with atexit) are called. An exit status argument is provided for the calling process as the exit status of the process. Typically a value of 0 is used to indicate a normal exit, and a nonzero value indicates some error. It can be, but is not required, to be set with one of the following:

exit status	value	description
EXIT_FAILURE	1	Abnormal program termination; signal to operating system that program has terminated with an error
EXIT_SUCCESS	0	Normal program termination

The prototype is located in <stdlib.h>

```
void exit(int status);
```

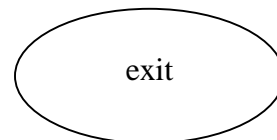
example using exit:

```

// L14p5.cpp
#include <stdlib.h>
#include <iostream.h>

int main()
{
cout << "exiting program"<<endl;
exit(0); // exit program
return 0; // this line is never reached
}

```

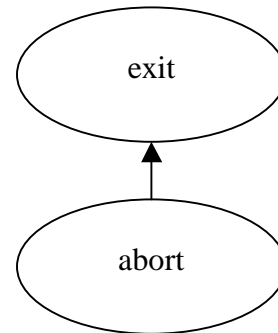


ABORT STATEMENT `stdlib.h`

Abort abnormally terminates a program by causing an abnormal program termination by calling raise (SIGABRT). If there is no signal handler for SIGABRT, then abort writes a termination message (Abnormal program termination) on stderr, then aborts the program by a call to _exit with exit code 3 to the parent process or to the operating system command processor.

```
// L14p6.cpp
#include <iostream.h>
#include <stdlib.h>

int main(void)
{
    cout << "Calling the abort() function" << endl;
    abort(); // abort program
    return 0; // This line s never reached
}
```



ASSERT STATEMENT `assert.h`

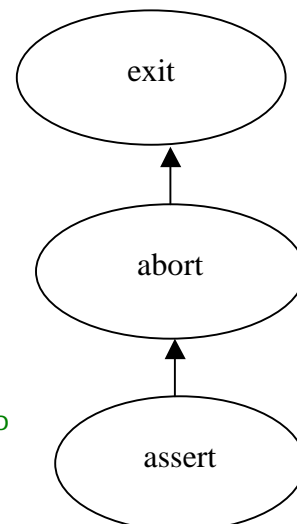
assert is a macro that expands to an if statement that tests a user specified condition. If the test evaluates to zero, then assert aborts the program by calling the abort function. The following a message is sent to stderr:

Assertion failed: test, file filename, line linenum

The filename and linenum listed in the message are the source file name and line number where the assert macro appears. The assert macro is declared in <assert.h> . If you place the #define NDEBUG directive ("no debugging") in the source code before the #include <assert.h> directive, the effect is to comment out the assert statement.

example using assert

```
// L14p7.cpp
#include <assert.h>
#include <iostream.h>
#include <stdlib.h>
int main(void)
{
    int x = 0; // set x to zero
    assert (x > 0); // test if x is zero
    return 0; // this line is not reached if x is zero
}
```



LESSON 14 EXERCISE 3

Write a program that asks the user to type a number and add the sum of the entered numbers continuously in a loop. Use **assert** to stop the program when a non number is entered. Call your program L14Ex3.cpp.

STATIC CAST OPERATOR

The `static_cast` operator is used to convert a pointer to a base class or to a derived class. No run time type checking is done.

```
static_cast<type-id*>(p);
```

DYNAMIC_CAST OPERATOR

The **dynamic_cast** operator is used to perform safe type conversions at run time for classes with virtual functions. The **dynamic_cast** operator checks to see if a pointer is a object type of **type-id**.

```
dynamic_cast<type-id*>(p); // p is a pointer T is a class type
```

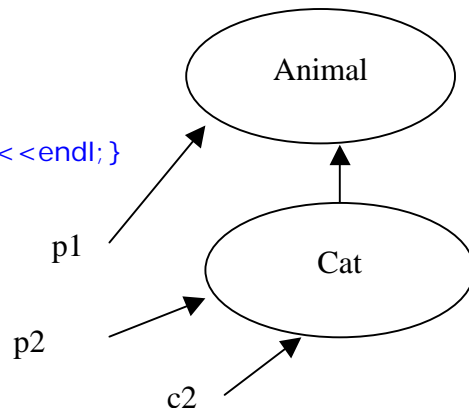
If `p` points to an base or a derived object of **type-id** the value of the expression is `p` and the type is `T*`, otherwise `p` is `NULL`. The following program demonstrates using **static_cast** and **dynamic_cast**. A base pointer can represent a base object or a derived object. You can use the cast operators to convert a base object from a derived object (**upcast**) or convert a base object to its derived object (**downcast**). When you are using VisualC++ the **dynamic_cast** operator can only do upcasts but not downcasts.

```
// L14p8.cpp
#include <iostream.h>

// base class
class Animal
{
public:
    Animal(){}
    virtual print(){cout<<" I am an animal"<<endl;}
};

// derived class
class Cat : public Animal
{
public:
    Cat(){}
    print(){cout<<" I am a cat"<<endl;}
};

void main()
{
    // static downcast
    Animal* p1 = new Animal(); // create an Animal object
    Cat* c1 = dynamic_cast<Cat*>(p1); // type cast to cat
    if(c1==NULL) p1->print();
    else c1->print();
}
```



program output:

```
I am an animal
I am a cat
```

```

// static downcast
Animal* p2 = new Cat(); // create an Cat object
Cat* c2 = dynamic_cast<Cat*>(p2); // type cast to cat
if(c2==NULL) p2->print();
else c2->print();

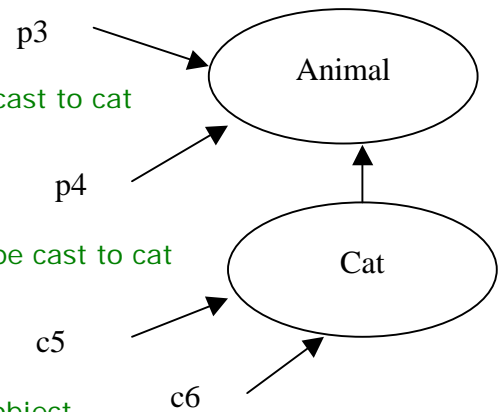
// static upcast
Animal* p3 = static_cast<Animal*>(c2); // type cast to cat
if(p3==NULL) c2->print();
else p3->print();

// dynamic upcast
Animal* p4 = dynamic_cast<Animal*>(c2); // type cast to cat
if(p4==NULL) c2->print();
else p4->print();

// dynamic downcast
Animal* p5 = new Animal(); // create an Animal object
Cat* c5 = dynamic_cast<Cat*>(p5); // type cast to cat
if(c5==NULL) p5->print();
else c5->print();

// dynamic downcast
Animal* p6 = new Cat(); // create an Cat object
Cat* c6 = dynamic_cast<Cat*>(p6); // type cast to cat
if(c6==NULL) p6->print();
else c6->print();
}

```



program output: con't

```

I am a cat
I am a cat
I am an animal
I am a cat

```

CALLING DERIVED CLASS FUNCTIONS FROM THE BASE CLASS

We can use the **dynamic cast** operator to safely call derived function from the base class. When a base class function calls a overridden function it will call the **base class function** if the calling object is a base class object. It will call the **derived class function** if its a derived class object. If the base class wants to call a function from the derived class and this function is only in the derived class then the calling object must be tested. If its a derived object before then the derived class function is called. If you do not test the object then your program could easily crash. This is easily handled using the dynamic cast operator.

```

if(dynamic_cast<DerivedClass*>(this)!= NULL)
    ((DerivedClass*)this)->derivedFunction();

```

Notice we **typecast** this to the derived Class before calling derived function. The following program demonstrates the base class calling **overridden** base class functions or **overridden** derived class functions. We also demonstrate the base class only calling functions belonging to the derived class.

Here is a summary of the possible scenarios:

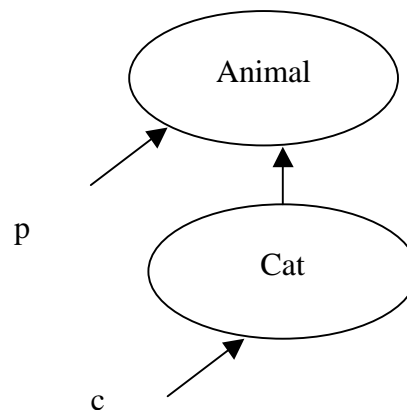
situation		action
1	if base class object	super class calls overridden functions of the base class
2	if derived class object	super class calls overridden functions of the derived class
3	if non-derived method	super class calls functions of derived class provided we typecast

Variables defined in a class cannot be overridden or cannot be made virtual. Functions of the base class will call its own variables. Functions of the derived class will call its own variables. Here is a test program. It does not work with Visual C++.

```
// L14p9.cpp
#include <iostream.h>

// base class
class Animal
{
private:
int age;
public:
Animal(){age=5;}
void baseFunction();
virtual print(){cout<<" I am an animal age: "<<age<<endl;}
};

// derived class
class Cat : public Animal
{
private:
int age;
public:
Cat(){age=10;}
derivedFunction()
{
cout << age << ": ";
print();
}
print(){cout<<" I am a cat age "<<age<<endl;}
};
```



```

// base function belonging to animal class
void Animal:: baseFunction()

{

    cout << age << ": " ;
    print(); // call overridden function
    // test and call derived class function
    if(dynamic_cast<Cat*>(this)!= NULL)
        ((Cat*)this)->derivedFunction();
}

// test driver
void main()

{
    Animal* a = new Animal(); // create an animal object
    a->baseFunction(); // call base function
    Animal* c = new Cat(); // create a cat object
    c->baseFunction(); // call base function
}

```

**We want to call the derived method
from the base function**

program output:

```

5: I am an animal age 5
5: I am a cat age 10
10: I am a cat age 10

```

THE typeid OPERATOR

The **typeid** operator returns a reference to an object that is listed in the library class **Typeinfo**. The class **Typeinfo** describes the run time type of an object. The **typeid** operator checks to see if an object is a specified **object type**.

typeid (typename) or typeid (expression)

If the operand of the typeid operator is the type typename or expression, **typeid** returns a reference to the object that represents the type name. You need to include the header file Typeinfo.h when using the typeid operator.

Example using typeid operator:

```

// L14p10.cpp

#include <typeinfo.h>
#include <iostream.h>

void main()

{
    int x;

    if(typeid(x) == typeid(int))
        cout << "type matches " << endl;
    else cout << "type does not match" << endl;
}

```

program output:

type matches

LESSON 14 EXERCISE 4

Write a program that has a base and derived class. Use the dynamic cast operator to get a pointer to a derived object. Use the type-id operator to test if it really is the derived class. Call your program L14Ex4.cpp.

NAMESPACES

Namespaces are used to distinguish among identical global names. Namespaces allow you to use two libraries that contain identical global names. To use namespaces, you declare your library function names in a namespace.

```
namespace library_name
{
    library function prototypes
}
```

You then use the resolution operator `::` to distinguish which function you want from which library. There is also a **using** statement that forces all function to use a library.

```
using namespace library_name

using namespace lib1;
```

Once the using statement is executed, all functions now are referenced to the library name specified in the using statement. Example using name spaces:

```
// L14p11.cpp
#include <iostream.h>

// declare name space lib1
namespace lib1
{
    void print(){cout << "lib1" << endl;}
}

// declare name space lib2
namespace lib2
{
    void print(){cout << "lib2" << endl;}
}

void main()
{
    lib1::print(); // use print function from library 1
    lib2::print(); // use print function from library 2
    using namespace lib1; // from now on use library 1
    print(); // use print function from library 1
}
```

LESSON 14 EXERCISE 5

Write a program that uses the std library namespace. Call your program L14Ex5.cpp.

```
using namespace std;
```

You must use the header

```
#include <iostream>
```

rather than

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

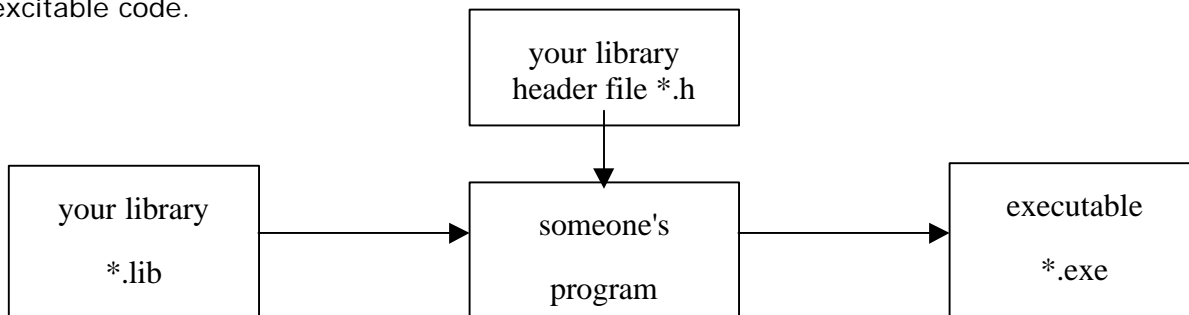
CPP PROGRAMMERS GUIDE LESSON 15

File:	CppGuideL15.doc
Date Started:	Mar 15, 2002
Last Update:	Apr 5, 2002
Version	0.1

LIBRARIES, DLL'S, PROCESSES and THREADS

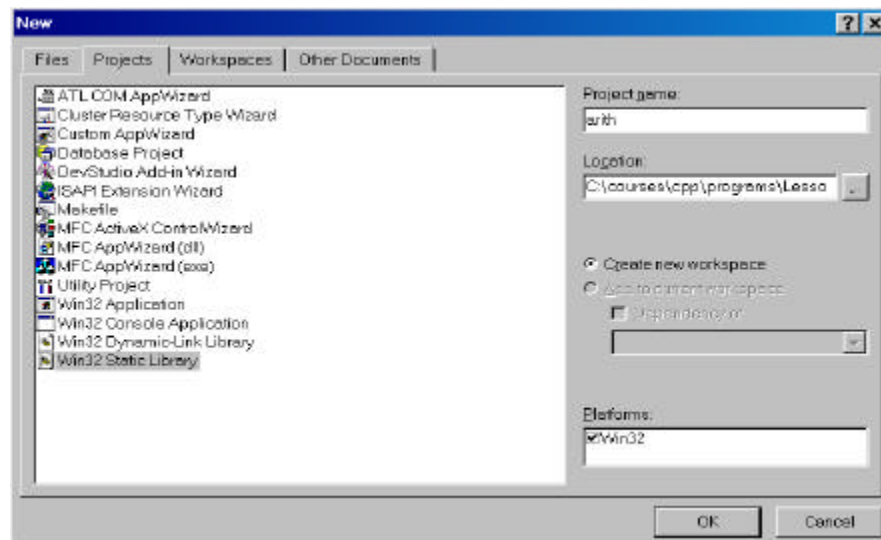
Creating a static Library

A **static** library lets someone else use compiled functions that you wrote. They will by including your library in their project. Your library should have a header ***.h** file. A compiled library file has the ***.lib** extension. Library files are statically linked at compile time and contain no source code only excitable code.



Here are the step by step instructions to make a library file using Visula C++. The library has functions to add, subtract, multiply and divide two double numbers.

step (1) open a new WIN32 Static Library project called: **arith**



You do not need to select MFC support or precompiled headers, select OK then finish.

step (2) create the header file: **arith.h** don't forget to state **extern "C"** because we are externally linking many files

```
// arith.h
extern "C"
double add(double a, double b);
double sub(double a, double b);
double mul(double a, double b);
double divide(double a, double b, int error);
```

step (3) implement arithmetic functions: **arith.cpp**

```
// arith.cpp

#include "arith.h"

double add(double a, double b)
{
    return a + b;
}

double sub(double a, double b)
{
    return a - b;
}

double mul(double a, double b)
{
    return a * b;
}

double divide(double a, double b, int* error)
{
    if(b == 0)
    {
        *error = true;
        return 0;
    }
    else
    {
        *error = false;
        return a/b;
    }
}
```

step (4) compile the library file from the **build** menu

build arith.lib

In the debug sub directory you will find file: arith.lib

step (5) close project: **arith**

step (6) open a WIN32 Console Application called: **arithtest**

Write a program to use the library files:

```
// arithtest.cpp
#include <iostream.h>
#include "arith.h"

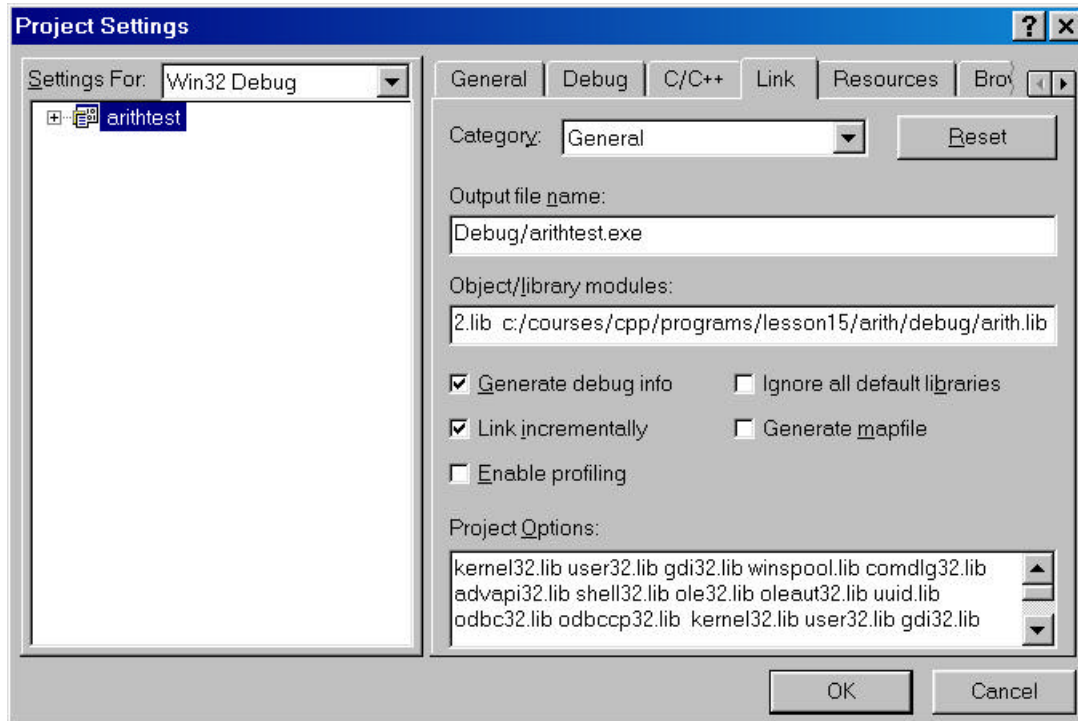
int main()
{
    double a,b;
    cout << "enter a number: ";
    cin >> a;
    cout << "enter a number: ";
    cin >> b;

    double sum = add(a,b);
    double diff = sub(a,b);
    double product = mul(a,b);
    int err=0;
    double quotient = divide(a,b,&err);

    cout << "a : " << a << " b : " << b << endl;
    cout << "The sum of a and b is " << sum << endl;
    cout << "a : " << a << " b : " << b << endl;
    cout << "The difference of a and b is " << diff << endl;
    cout << "a : " << a << " b : " << b << endl;
    cout << "The product of a and b is " << product << endl;
    cout << "a : " << a << " b : " << b << endl;
    if(err==1) cout << "We cannot divide by " << b << endl;
    cout << "The quotient of a and b is " << quotient << endl;
    return 0;
}
```

step (6)

You need to add the compiled library to the project settings, From the project settings menu select **Link** tab. In the **object library modules** textbox, add your library file **arith.lib** at the end. Use your working: path/arith.lib (the link files names are separated by spaces)

**step (7)**

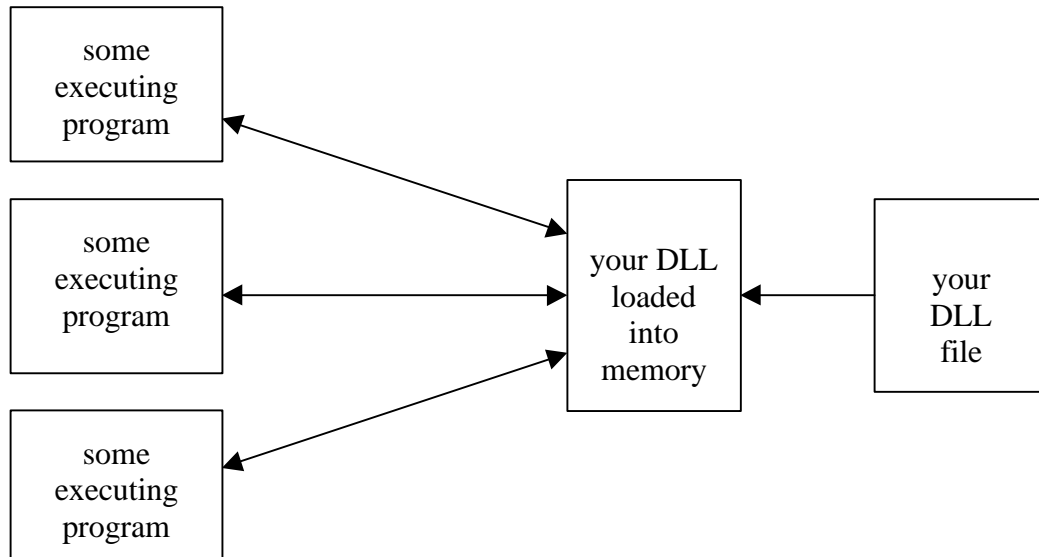
Compile and execute the **arithtest** project Your output should be like this:

```
enter a number: 2
enter a number: 3
a : 2 b : 3
The sum of a and b is 5
a : 2 b : 3
The difference of a and b is -1
a : 2 b : 3
The product of a and b is 6
a : 2 b : 3
The quotient of a and b is 0.666667

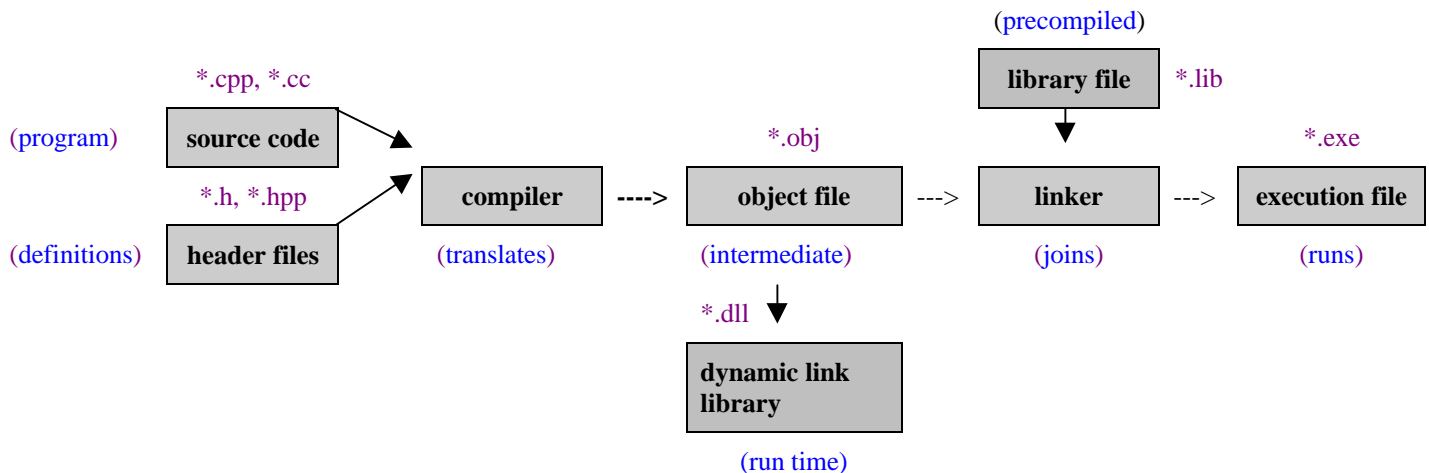
Press any key to continue
```


DYNAMIC LINK LIBRARIES DLL

DLL'S let your executing program execute code loaded from a stored file. The purpose of DLL's is to let many executing programs share the same code modules. DLL's differ from static library files because the code is already precompiled to be executed at run time. When your program need to execute a function located in a DLL file the code is loaded into memory then executed.



DLL's can be created from cpp code or *.obj code. Can you remember the steps needed to build an executable. ?



There are two ways to load DLL's

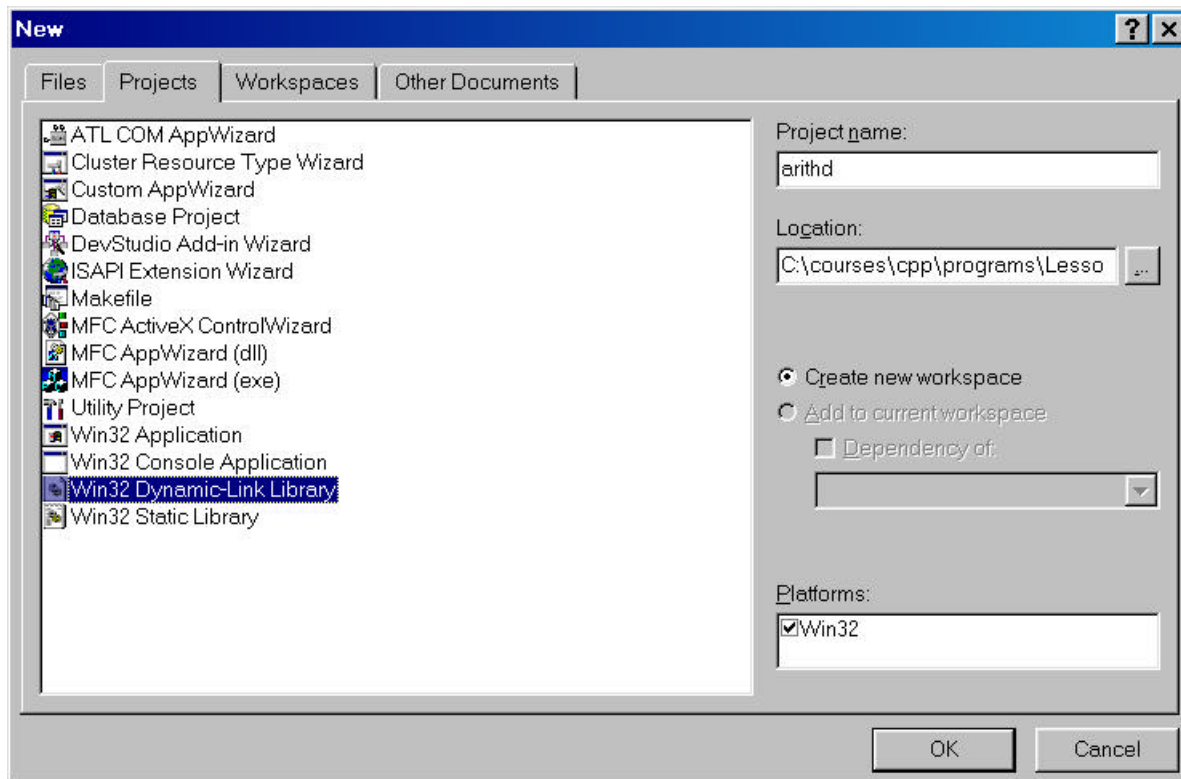
(1) you can tell the linker to **implicitly** use a DLL at run time. This means when your program starts to run it will automatically load the specified DLL.

(2) your program can **explicitly** load a DLL library when it starts to run. Your program will tell the operating system to load a particular DLL and get pointer to the functions loaded in memory.

(1) TELL LINKER TO LOAD A DDL AT RUN TIME

Here are the step by step instructions:

Step (1) create a **WIN32 Dynamic Link Library** project called **arithd**



Select an Empty DLL Project and then press Finish

Step (2) make a definition file *.def

You need to make a **definition file** called **arithd.def** to state the entry points of the functions in your object file. This not an automatic process you need to do it your self. In the definition file you will see **LIBRARY**, **DESCRIPTION** and **EXPORTS** tags. After the **EXPORTS** tag you list the function names followed by a **@** and an **ordinal** number. The ordinal number is the location of the function to execute. Here is the definition file: **arithd.def**

```
LIBRARY arithd
DESCRIPTION "arith dll functions"
EXPORTS
    add @1
    sub @2
    mul @3
    divide @4
```

Step (3)

Add the object file **arith.obj** produced from the static library and the **arithd.def** file to the arithd project by using project menu: **add to project files** (you need to select all files)

step (4)

From the **build** menu select **build dll**. Two files are produced in the debug sub directory

arithd.dll

arithd.lib

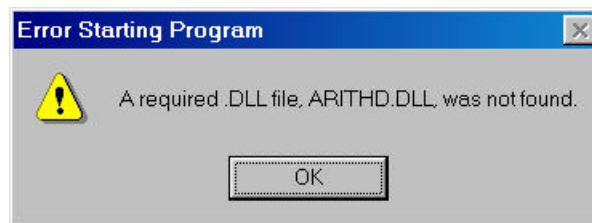
The **arithd.dll** library file is quite different from the static library file **arith.lib**. The **arithd.lib** file contains information to tell the compiler how to locate and load the functions from the DLL when the external functions stored in the DLL are called.

step (6)

close the **arithd** project

step (7)

Make a new **WIN32Console** project called **arithdtest** include files **arith.cpp**, **arith.h** and **arithd.lib**. You need to put the **arithd.dll** file from the **arithd** projects debug subdirectory and put into the debug sub directory of your **arithdtest** project or else you will get the following error message when you try to execute the test program.



(2) YOUR PROGRAM CAN LOAD A DLL LIBRARY WHEN IT STARTS TO RUN

In this situation the executable program will explicitly state to load a dll using the **LoadLibrary()** function located in **windows.h**. Your program calls the **LoadLibrary()** function to explicitly link to a DLL. If successful, the function maps the specified DLL into the address space of the calling program and returns a handle to the DLL. The handle is used with the **GetProcAddress()** function to get the memory address of the loaded function.

```
HINSTANCE harith = ::LoadLibrary("arithd.dll");
```

harith is a handle to our DLL loaded in memory. A handle is used by the operating system to locate your loaded DLL. Once we get our handle to the loaded DLL we can get the address of loaded DLL function s using the **GetProcAddress()** function.

```
add* pAdd = (add*)::GetProcAddress(harith,"add");
```

From the function pointer we can call the function

```
double sum = (*pAdd)(a,b);
```

To just use the function name we make a **typedef** from our function declaration.

```
typedef double add(double,double);
```

If we do not do this then we have to include the complete function declaration;

```
double add(double,double);
```

every time we specify the function name.

Here's the step by step instructions

step (1) Make a **WIN32Console** project called **arithd2test**

step (2) Type in and Add the following cpp file called **arith2test.cpp** to your project

```
// arithd2test.cpp
#include <iostream.h>
#include <windows.h>

typedef double add(double,double);
typedef double sub(double,double);
typedef double mul(double,double);
typedef double divide(double,double,int*);

int main()
{
    double a,b;
    cout << "enter a number: ";
    cin >> a;
    cout << "enter a number: ";
    cin >> b;

    HINSTANCE harith = ::LoadLibrary("arithd.dll");
    add* pAdd = (add*)::GetProcAddress(harith,"add");
    double sum = (*pAdd)(a,b);
    sub* pSub = (sub*)::GetProcAddress(harith,"sub");
    double diff = (*pSub)(a,b);
    mul* pMul = (mul*)::GetProcAddress(harith,"mul");
    double product = (*pMul)(a,b);
    divide* pDivide = (divide*)::GetProcAddress(harith,"divide");
    int err;
    double quotient = (*pDivide)(a,b,&err);
    cout << "a : " << a << " b : " << b << endl;
    cout << "The sum of a and b is " << sum << endl;
    cout << "a : " << a << " b : " << b << endl;
    cout << "The difference of a and b is " << diff << endl;
```

```

cout << "a : " << a << " b : " << b << endl;
cout << "The product of a and b is " << product << endl;
cout << "a : " << a << " b : " << b << endl;
if(err==1) cout << "We cannot divide by " << b << endl;
cout << "The quotient of a and b is " << quotient << endl;
return 0;
}

```

step (3)

Take the **aritdd.dll** from the **arithd** project and put in the debug sub directory of your **arithd2test** project

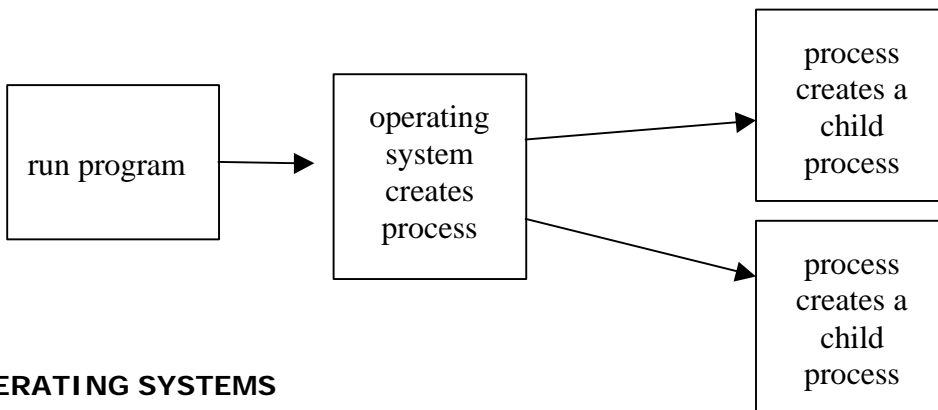
step (4) build and execute

LESSON15 EXERCISE 1

Make a DLL that that has a function calls another DLL.

CREATING PROCESS

Processes are Just instances of executing programs. Processes are also called tasks. A process contains executable code memory for variables and an execution stack. Every time you run a program the operating system creates a new process. Process may run the same program or different programs. A process may also create other process, these processes are called child processes.



OPERATING SYSTEMS

There are predominately two operating systems for PC today is windows or Unix. We use windows for all are programming. The reason is it is easy for us to use and there is an abundance of application software available. Surprising our web server uses the Unix operating system. we will look at process in the Unix operating system first.

Unix processes

There are four main functions dealing with processes.

process function	description
fork()	creates a new process by duplicating the calling process
exec()	calling a new program from an running process
wait()	allows one process to wait until another process finishes
exit()	terminates a process

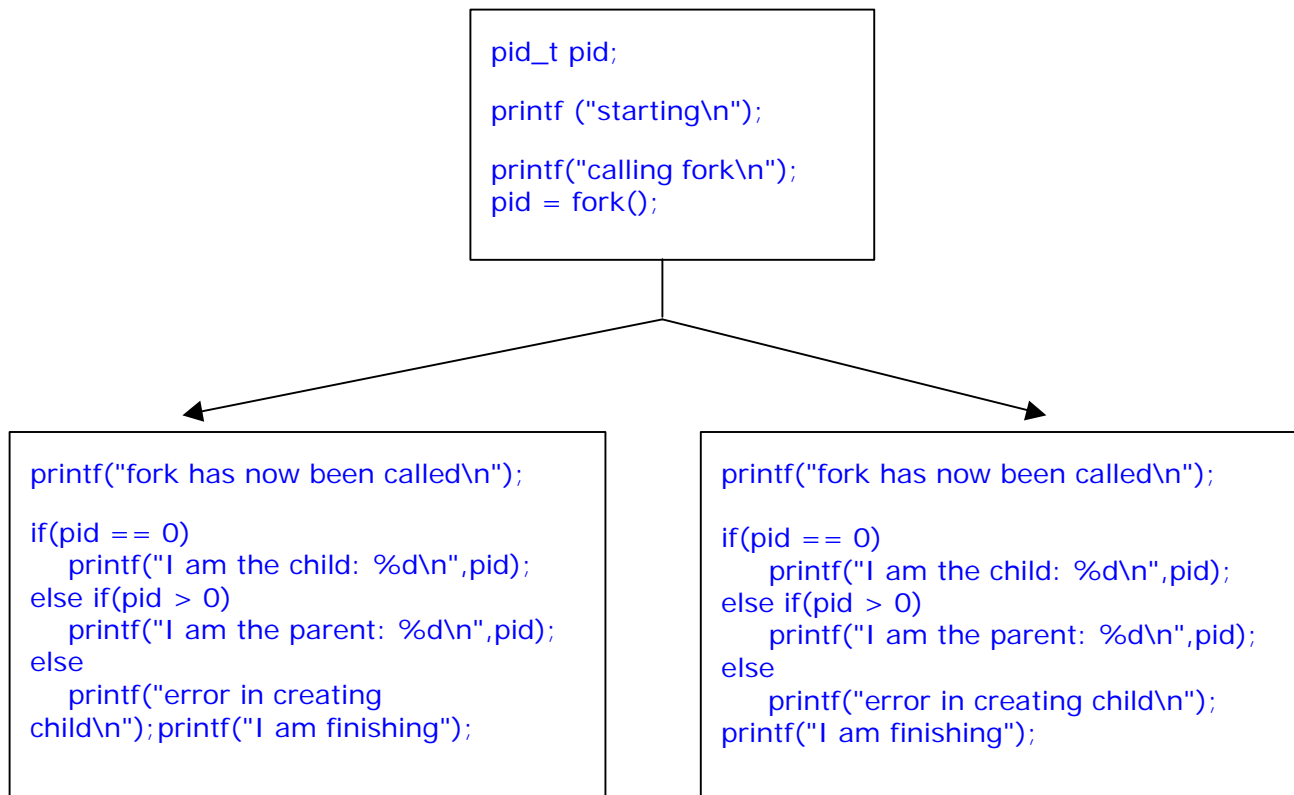
fork()

A call to **fork()** causes a duplication of the presently running process. Now we have two process running. The original one called the **parent** and the new one called the **child**. When you call **fork()** it return a process id for identification. forking may be difficult to understand but let the following code example and diagram help. The following programs started to run then a call to **fork()** is made.

```
#include <unistd.h>

int main()
{
    pid_t pid;
    printf ("starting\n");
    printf("calling fork\n");
    pid = fork();
    printf("fork has now been called\n");
    if(pid == 0)
        printf("I am the child: %d\n",pid);
    else if(pid > 0)
        printf("I am the parent: %d\n",pid);
    else
        printf("error in creating child\n");
    printf("I am finishing");
}
```

```
starting
calling fork
fork has now been called
fork has now been called
I am the child: 0
I am the parent: 80835125
I am finishing
I am finishing
```



LESSON 15 EXERCISE 2

Run the fork program. This program only run under Unix operating system or Cygwin (unix under DOS) download from: <http://www.cygwin.com/>

PROGRAMS CALLING OTHER PROGRAMS `process.h`

One program may execute another program. To do this you use the **exec** group of functions. You can execute **.exe**, **.com** or **.bat** files. Here is a small program that that executes another program called hello. It just print hello to the screen. You can run under windows or unix.

```

// exectest.cpp

#include <iostream.h>
#include <process.h>

int main(int argc, char** argv)
{
    cout << "running" << endl;
    execlp("hello.exe", "hello", NULL); // call program hello
    return 0;
}

```

program output:

running

We have assumed you already have the following hello.cpp program compiled and the executable is in the same directory.

```
// hello.cpp
#include <iostream.h>

int main(int argc, char** argv)
{
    cout << "hello" << endl;
    return 0;
}
```

LESSON 15 EXERCISE 3

Run the above program, it works under Unix or Windows.

argument lists and environment variables

When you call another program it may be located in different directories, you may need command arguments etc. There is an **exec** function to handle all these situations. There are two groups. The **l** group lets you **list** the arguments one by one and then terminate by a NULL.

```
execl("hello.exe","hello",NULL);
```

The **v** group requires you put all passed arguments into an array like:

```
char* args[2];
args[0]="Hello";
args[1]=NULL;
```

You now call **execv** to execute the hello program and pass the command line arguments to it.

```
execv("hello.exe",args);
```

For each group **l** or **v** you can use the **PATH** environment variable to locate programs to execute. **execlp** and **execvp** where **p** means path. The PATH environment variable enables the operating system to search all the listed directories for the program you want to run. Else you could only run programs from the current directory.

```
execlp("hello.exe","hello",NULL);

execvp("hello.exe",args);
```

The new process inherits all the environment variables from the parent process. Environment variables are **name=value** pairs. You first put own environment variables in an array.

```
char* env[] = {"MYENV=my environment",NULL};
```

You may pass your environment variables array to your program using **execpe** or **execve**.

```
execle("hello.exe","hello",NULL,env);

execve("hello.exe",args,env);
```


You may also pass environment variables to the exec functions that use the PATH environment variables.

```
execl("hello.exe", "hello", NULL, env);
```

```
execve("hello.exe", args, env);
```

Here is the summary of the **l** (list) group exec functions:

function	description	using
execl	command line arguments passed individually	execl(prog, prog, arg1...argn, NULL);
execle	use array of pointers for environment settings	execle(prog, prog, arg1...argn, NULL, env);
execlp	use path environment variable	execlp(prog, prog, arg1...argn, NULL);
execspe	use path and environment variables	execl(peprog, prog, arg1...argn, NULL, env);

Here is the summary of the **v** (array arguments) group exec functions:

function	description	
execv	pass command line parameters as an array of pointers	execv(prog, prog, args);
execve	use array of pointers for environment settings	execv(prog, prog, args, env);
execvp	use path environment variable	execv(prog, prog, args);
execvpe	use path and environment variables	execv(prog, prog, args, env);

LESSON 15 EXERCISE 4

Change the **hello.cpp** program to gets a person's name from the command line parameters. Call this program **hello2.cpp**. Change the **exectest.cpp** program to receive a program name (like **hello2**) using the command line parameters. Call this program **exectest2.cpp**. Also have the **exectest2.cpp** program get the persons name to pass to the hello2.cpp program. Execute the hello2.cpp program from the exectest2.cpp program.

```
exectest2 hello2 tom
```

You should now get:

```
running
hello tom
```

LESSON 15 EXERCISE 5

Try all the variations of exec group of functions.

READING AND WRITING ENVIRONMENT VARIABLES

Environment variable are stored by the operating system so that all programs can read and write to them. They are initialized when the computer first boots up. To read an environment variable you use the `getenv` function located in **stdlib.h**.

```
char *getenv( const char *varname );
```

To write to an environment variable (change it) you use the **putenv** function located in **stdlib.h**

```
int _putenv( const char *envstring );
```

The following program passes the PATH environment and prints it out to the screen.

```
#include <iostream.h>
#include <stdlib.h>
void main( void )
{
    // Get the value of the PATH environment variable.
    char* envvar = getenv( "PATH" );
    cout << "Original PATH variable is: " << envvar << endl;
    // set path to current directory
    putenv( "PATH=." );
    // Get new value.
    envvar = getenv( "PATH" );
    cout << "New PATH variable is: " << envvar << endl;
}
```

LESSON 15 EXERCISE 6

Change the hello2.cpp program to print out your own environment variables call the program hello3.cpp. Change the exectest2.cpp program call it exectest3.ccp to call the hello3.cpp program and pass it your own environment variables. You need to put our environment variables in an array.

THREADS

Threads let functions execute simultaneously in one program. Sometimes this is hard to imagine. The operating system runs each function each a little bit of a time so that it appears all functions are running simultaneously. To **start** a thread we use the **_beginthread** function located in **process.h**

```
_unsigned long _beginthread( void( __cdecl *start_address )( void * ), unsigned stack_size, void *arglist );
```

To **terminate** a thread we use the **_endthread** function located in **process.h**

```
void _endthread( void );
```

Both thread functions get the starting address of the function to run, the stack size (set to 0) and a list of arguments that will be passed to the calling function. To enable other functions to run we must stop one thread from running a little but, this is called **sleeping**. Something like what humans do to let other's go to work. To let your function sleep you call the Sleep function located in **windows.h**

```
void Sleep( DWORD dwMilliseconds ); // sleep time in milliseconds
```

If you don't tell your function to sleep then it hogs the operating system and no other threads or programs can run. The following is a small program that demonstrates threads. When you see this program run you will understand threads. We have a function called **Spit** that prints out numbers 1 to 10. It also gets a sleep interval. In our main method, the **_beginthread** function calls the **Spit** function with a sleep interval argument that is passed by address. At the bottom of our main function we sleep for a long time. If we do not the main method will terminate and so will our threads. In each **Spit** function after the **for** loop has finished we terminate the threads using **_endthread**. To avoid compile errors you need to go to projects settings menu select using MFC.

```
// threadtest.cpp
#include <windows.h>
#include <process.h> /* _beginthread, _endthread */
#include <stdlib.h>
#include <iostream.h>

void Spit(void* interval);

void main()
{
    int interval1 = 1000;
    int interval2 = 500;
    _beginthread(Spit, 0, (void*)interval1 );
    _beginthread(Spit, 0, (void*)interval2 );
    Sleep(10000);
}

// this function is called as a thread
// prints out each number after a little sleep
void Spit(void* interval)
{
    for(int i=0;i<10;i++)
    {
        /* Pause between loops. */
        Sleep((int)interval );
        cout << (int) interval + i << " " << endl;
    }

    /* _endthread given to terminate */
    _endthread()
}
```

LESSON 15 EXERCISE 7

Type in the above program and run. Experiment with different sleep intervals. Make 1 very long and make one very short. Make them both the same interval times etc. Watch the effect of each change.

LESSON 15 EXERCISE 8

Make a thread class that can run many threads that you add. Make a **start()** and **stop()** functions. The constructor should get the sleep interval.

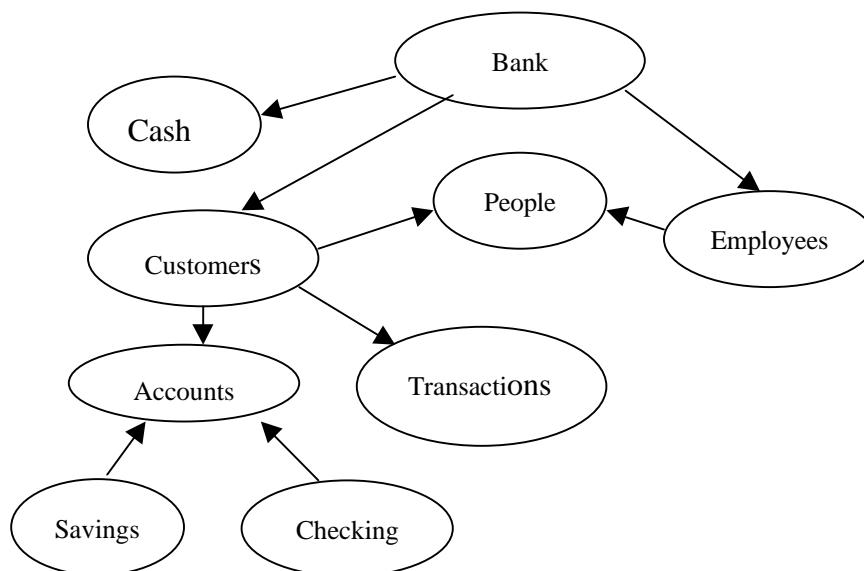
C++ PROGRAMMERS GUIDE PROJECT LESSON 16

File:	CppGuideL16.doc
Date Started:	July 24, 1998
Last Update:	April 5, 2002
Version:	4.0

PROJECTS = WORK = LEARNING = FUN

INTRODUCTION

We will implement a Bank using many classes. A Bank is a very good example of object oriented programming. A bank has customers, employees, accounts, transactions and cash. Customers and Employees are people. Customers have accounts and transactions. Accounts may be Checking or Savings. Transactions identify the customer account operations as deposit, withdraw or transferring amounts between accounts. A bank makes money from charging customers fees for banking services. Banks must also pay interest to customers and salaries to employees. Each component of a bank will be represented by a class. Classes may be sub classes of other classes. Classes may also use other classes. An up ↑ arrows means a class is being derived from a class. A down ↓ arrow means a class will use another class as a variable. The following is our bank class model.



SPECIFICATIONS

Before you can write any program you need to have **specifications**. A specification describes what the program is supposed to do and what components, a program is suppose to have. Specifications usually describe the top level of a program first. A program can be broken down into many sub components. When using object oriented programming each sub component can be a class. Object oriented programming is ideal to break down large complicated programming problems into smaller manageable tasks. We can now write the specification from the real-life components of a real bank. We write our specification top down which means we describe all the top levels first and then proceed down to describe the inner bottom levels.

Bank class

A bank must have Customers, Employees and Cash. Customers and Employees are People where the cash represents what money the bank is making. Banks make money from charging customers fees for banking services. The bank must pay money to Employees and pay interest to Customers. All member variables in the Bank class should be **private**.

member	data type	base class	description
customers	Customer**	Person	represents an array of pointers to Customer objects
maxCustomers	int		maximum number of customers
numCustomers	int		number of customers in array
employees	Employee**	Person	represents an array of pointers to Employee objects
maxEmployees	int		maximum number of employees
numEmployees	int		number of employees in array
cash	float		how much money the bank has

Customers and Employees will inherit the Person class. The Customers will be represented each by an array of pointers to Customer objects. Employees will be represented each by an array of pointers to Employee objects. We use two arrays because we need separate lists of customers and employees. We do not use Polymorphism here because polymorphism is only good at selecting derived objects automatically, but not very good at separating derived objects into distinct groups. Your bank constructor must receive the maximum amount of customers and employees. A bank would have about 1000 customers and about 10 employees. For this exercise 10 customers and 3 employees are enough.

Bank class Operations

A bank needs to perform operations, each implemented by the following functions: Functions used by other classes should be **public**. Functions just used by this class should be **private**.

function prototype	description
Bank(int maxCustomer, int maxEmployee, int initialCash);	initializing maximum number of customers, employees and starting cash
bool addCustomer(Customer *);	add new customer
bool deleteCustomer(Customer *);	remove customer
Customer* searchCustomer (String *lastName);	search for customer by last name
Customer* searchCustomer (long accountNumber);	search for customer by account number
bool makeTransaction();	customer deposits, withdraws or transfers money between accounts.
Customer* update();	pay monthly interest to all accounts, collect all service fees
bool addEmployee(Employee *);	add new employees
bool deleteEmployee(Employee *);	remove employees
Customer* searchEmployee(String *lastName);	search for employee by last name
bool payEmployees();	pay employees
bool addCash(float amount);	add amount to cash

Person class

A Person represents a Customer or Employees name, address, phone and SIN, all represented by String objects. You can use the String class from this course or the one supplied by your compiler. All member variables in the class should be **private**, and only if necessary **protected**.

member	data type	description
firstName	String	First name
middle Initial	String	middle initial
lastName	String	last name
street	String	street number and street name
apt	String	apt or suite number
city	String	city
state	String	state or province
postalCode	String	postal code
phone	String	home phone
SIN	String	social insurance number

Person class operations

For this class you just need a default constructor, a **getName()** function and overloaded operator functions `>>` and `<<` to get and print out the person information. The **getName()** function must combine the first, initial and last name into one string. The operator `>>` function will be used to get customer information from the keyboard or file. The operator `<<` function will be used to print out customer information to the screen or file. All functions are **public**.

function prototype	description
Person();	default constructor
String& getName();	add new customer
friend istream& operator>>(istream& in, Person& p);	get person info
friend ostream& operator<<(ostream& in, Person& p);	print person info

Customer class

The Customer class inherits the Person class. You just need a few more things for the customer class. like a pointer to the bank object, an array of pointers to Accounts objects and an array of pointers to Transactions objects. All member variables in the class should be **private**.

member	data type	description
bank	Bank*	pointer to bank object
business phone	String	work phone number
accounts	Accounts**	an array of pointers to account objects
maxAccounts	int	max number of Accounts
numAccounts	int	number of accounts in account array
transactions	Transaction**	array of pointers to Transaction object
maxTransactions	int	maximum number of transactions
numTransactions	int	number of transactions if Transaction array

Customer class operations

You need an initializing constructor with the maximum number of accounts and maximum number of transactions to allocate memory for the array of accounts and transactions. You will only need about 5 accounts but many transactions. For this exercise you can have about 100 transactions. You will also need functions to add, delete and view accounts. You will also need functions to add and view transactions. You will need an overloaded operator friend function's >> and << to get and print out the Customer information. The Customer class constructor must also call the Persons base class default constructor. Functions to be used by other classes should be **public**. Functions only used by this class should be **private**.

function prototype	description
Customer(int maxAccounts, int maxTansactions);	initializing constructor
bool addAccount(Account* account)	add new customer account
bool viewAccount(long number);	view customer account
bool deleteAccount(Account* account)	delete customer account
bool addTransaction (Transaction* trans)	add transaction to customer
bool viewTransactions(long number);	search for customer account
friend istream& operator>>(istream& in, Customer& p);	get customer info
friend ostream& operator<<(ostream& in, Customer& p);	print customer info

Employee class

The Employee class inherits the Person class. You just need a few more things for the Employee class like a pointer to the bank object, salary and total salary paid so far. . All member variables in the class should be **private**.

member	data type	description
bank	Bank*	pointer to bank object
salary	float	employees yearly salary amount
totalPaid	float	how much salary paid so far

Employee class operations

You need an initializing constructor with the maximum number of employees to allocate memory for the array of Employee objects. You will only need room for about 5 employees. The Employee class must also call the Persons base class default constructor. You need a function called **pay()** to pay the employees. You will need an overloaded operator friend function's >> and << to get and print out the Employee information.

function prototype	description
Employee(int maxEmployees);	initializing constructor
void pay();	pay this employee
friend istream& operator>>(istream& in, Employee& p);	get employee info
friend ostream& operator<<(ostream& in, Employee& p);	print employee info

Account class

The account class contains all the common information about an account like Account number, pointer to Customer, balance and service fees. All member variables in the class should be **private**, and only if necessary **protected**.

member	data type	description
number	long	account number
customer	Customer*	pointer to customer object
balance	float	money in account
serviceFee	float	monthly service fee

Account class operations

You will need an initializing constructor that accepts a pointer to a Customer object, service fee and an initial balance. The account number will be internally generated by the derived classes. You will also need **pure virtual** functions **update()**, **withdraw()**, **deposit()** and **transfer()**. The **update()** function will apply all interest to account balance and pay the bank for service fees. You will need an overloaded operator friend function's **>>** and **<<** to get and print out the Account information. Functions to be used by other classes should be **public**. Functions only used by this class should be **private**.

function prototype	description
Account(Customer* customer, float serviceFee, float balance);	initializing constructor
virtual void deposit(float amount)=0;	deposit funds into account
virtual void withdraw(float amount)=0;	withdraw funds from this account
virtual void transfer(float amount, Account* acct)=0;	funds from this account to another
virtual void update()=0;	update this account, pay all interest, collect all service fees
friend istream& operator>>(istream& in, Account acc);	get account info
friend ostream& operator<<(ostream& in, Account& acc);	print account info

Checking class

The Checking class inherits the Account class. A checking account has an overdraft but do not give interest. There is a base monthly service fee but they also charge 1 dollar for every bank transaction. There is also a static variable to assign new checking account numbers. All member variables in the class should be **private**,

member	data type	description
overdraft	float	Overdraft amount. if no over draft set to 0
transactionFee	float	charge for any transaction
nextNumber	static long	next account number starting from 1000000

Checking class operation

The initializing constructor must also receive the overdraft and transaction fee and call the account base class initializing constructor. It is inside the Checking constructor where the account number is assigned and incremented. The Checking class must also implement the **update()**, **withdraw()** and **deposit()** and **transfer()** functions. The **update()** function will pay the bank for the monthly service fees. For every transaction you must add the transaction fee to the banks cash. You will need an overloaded operator friend function's **>>** and **<<** to get and print out the Checking information. Functions to be used by other classes should be **public**. Functions only used by this class should be **private**.

function prototype	description
Checking(Customer* customer, float serviceFee, float balance);	initializing constructor
void deposit(float amount);	deposit funds into account
void withdraw(float amount);	withdraw funds from this account
void transfer(float amount, Account* acct2);	funds from this account to another
void update();	update this account, pay all interest, collect all service fees

Example code for the deposit function could be like this:

```
void deposit(double amount)
{
    balance = balance+amount
    Transaction trans = new Transaction(Deposit,0,number,balance);
    customer.addTransaction(trans);
    balance = balance - transactionFee;
}
```

Savings class

The Savings class inherits the Account class. The savings account gives interest but no overdraft protection. There is a base monthly service fee. There is also a static variable to assign new checking account numbers. All member variables in this class should be made **private**,

member	data type	description
interest	float	paid monthly interest
nextNumber	static long	next account number starting from 2000000

A Savings class operation

The initializing constructor must also receive the overdraft and interest rate and call the Account base class initializing constructor. It is Inside the Savings constructor where the account number is assigned and incremented. The savings class must also implement the **update()**, **withdraw()**, **deposit()** and **transfer()** functions. The **update()** function will pay the bank for the monthly service fees and pay the customer interest on the balance. You could use a monthly service fee of \$15 and a monthly interest rate of .005 % (6% per annum). You will need an overloaded operator friend function's >> and << to get and print out the Savings information. Functions to be used by other classes should be made **public**. Functions only used by this class should be **private**.

function prototype	description
Savings(Customer* customer, float serviceFee, float balance, float transactionFee);	initializing constructor
void deposit(float amount);	deposit funds into account
void withdraw(float amount);	withdraw funds from this account
void transfer(float amount, Account* acct2);	funds from this account to another
void update();	update this account, pay all interest, collect all service fees

Transaction class

We must keep track of the transactions that each customer makes. A transaction class will have the following members. All member variables in this class should be made **private**,

member	data type	description
type	enum	Withdraw , Deposit, Transfer, ServiceFee, InterestPayment, TransactionFee
fromAccount	String	from account number
toAccount	String	to account number
amount	float	amount to deposit or withdraw

Transaction class operation

You will need an initializing Constructor to initialize transaction type, accounts and amount. All Transaction objects are added to the Customer object the account resides in. In cases of deposit and withdraw the account from/to parameters can be passed codes to identify checkin, cashin, cashout etc. Use an **enum** for the from/to values. You will need an overloaded operator friend function's >> and << to get and print out the Transaction information. . Functions to be used by other classes should be made **public**. Functions only used by this class should be **public**.

function prototype	description
Transaction (TransactionType type, long from, long to, float balance);	initializing constructor
friend istream& operator>>(istream& in, Transaction& t);	get accountinfo
friend ostream& operator<<(ostream& in, Transaction& t);	print account info

USER INTERFACE

You will need a menu for the bank employee for bank operations like this:

<p>Welcome to Super Bank</p> <p>=====</p> <p>(1) Add new Customer</p> <p>(2) View Customer Accounts</p> <p>(3) Make a Transaction</p> <p>(4) Pay Interest</p> <p>(5) Add Employee</p> <p>(6) View Employees</p> <p>(7) Pay Employee</p> <p>(8) View Bank Info</p> <p>(9) Go home for the day</p>
--

Each menu selection is described as follows:

(1) Add /Delete/View new Customer	add or delete customer
(2) View Customer Accounts	view all customer accounts
(3) Make a Transaction	customers make a withdraw, deposit or transfer
(4) Update Accounts	Pay monthly interest to all account. apply service charge to all accounts

(5) Add/delete/View Employee	add a new employee or delete an existing employee
(6) View Employees	view all employees
(7) Pay Employee	pay employees their weekly salary
(8) View Bank Info	view all bank information

Each menu selection will call sub menu operations as follows:

Add/Delete/View customer

(1)	ask for customer account number or N for new and D for delete
(2)	if new customer get all customer information and the account they want to open
(3)	if existing customer show information and transactions

Make transaction

(1)	Get customer account by account number or by name. If no account tell this person they are at the wrong bank
(2)	Ask if Deposit Withdraw or Transfer. Inform customer if they cannot withdraw amount
(3)	ask for amount, this must be a positive number

Update customers

(1)	Apply interest and monthly service charge to all customer accounts. The service charges will be added to the bank's cash, the interest payments will come from the bank cash.
-----	---

Add/ Delete/View employee

(1)	ask for employee name or N for new and D for delete
(2)	if new employee get all employee information and salary
(3)	if existing employee show information

Pay employees

(1)	Pay employees their weekly wage. The money must come from the bank's cash
-----	---

class summary

You will have the following classes. Put the **main()** function in the Bank class.

header files:	implementation files:
Bank.hpp	Bank.cpp
Person.hpp	Person.cpp
Customer.hpp	Customer.cpp
Employee.hpp	Employee.cpp
Account.hpp	Account.cpp
Savings.hpp	Savings.cpp
Checking.hpp	Checking.cpp
Transaction.hpp	Transaction.cpp

main function

The main function can include the menu and call functions from the bank class to process the requested bank operation. You may want to put a **run()** function in the Bank class that contains the menu selection and calls the Bank class functions to do the requested operations.

Hints on completing Project:

- (1) Sketch an operational model of all classes showing how all the classes interact.
- (2) Write all the class definitions first top down. Start at the top of the class model and proceed to the bottom class. Write the Base class definitions first.
- (3) Write each class implementation one at a time bottom up. Start at the bottom class of the class model and then finish up at the top. Always write the base classes before writing the derived classes.
- (4) Write your comments so that if someone just reads the comments they know how the program works.
- (5) Use the **new** operator to allocate memory, and the **delete** operator to delete any memory allocated with new.
- (6) Avoid **circular references** when using the **#include** directive. Circular **#include** references arise when one **hpp** file includes another **hpp** file and this **hpp** file includes the one that called it. For example the **bank.hpp** file needs to include **customer.hpp** file. The **customer.hpp** file needs to include **bank.hpp** file. The result is a circular **#include** reference. One way out of this dilemma is to use a **forward reference**.

In the account class just say:

```
class Bank* bank;
```

This will solve all your problems. Now the compiler knows that Bank is a pointer to a Bank object.

Another way to avoid circular reference is to call a static method of the Bank class to get a reference to the bank object.

```
// data manager class
#include <iostream.h>

class Bank
{
private: static Bank* reference;
// private constructor
private: Bank()
{}

public: static Bank* getInstance();
};

Bank* Bank::reference=NULL;

Bank* Bank::getInstance()
{
if (reference == NULL)
reference = new Bank(); // create instance of this class
return reference; // return reference to instance of class
}
```

(7) calling overloaded operator functions

Each class will have the same operator functions so it will be difficult to select the one you want. A solution is to type cast.

```
cin >> customer;
```

If you just want to print out the person part you need to type cast.

```
cin >> (Person&)customer;
```

(Don't forget to pass by reference).

When you have a pointer you need to type cast to the target pass type.

```
cout << *customer;

cout << (Person&)(*customer)
```

We type cast as a reference because we want to pass a reference to the **memory address** the pointer is pointing to.

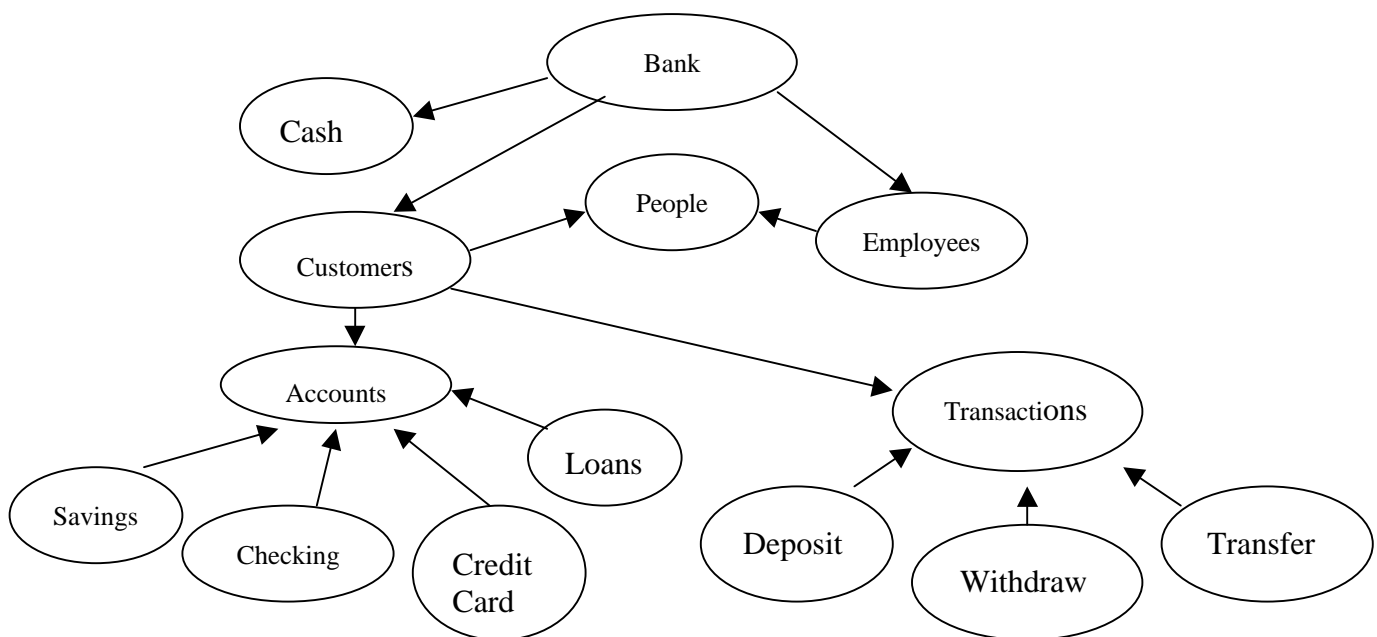
(8) You can clear the screen using: `System("cls");`

Marking Scheme:

CONDITION	RESULT	GRADE
Program crashes	Retry	R
Program works	Pass	P
Program is impressive	Good	G
program is ingenious	Excellent	E

ENHANCEMENTS

- (1) Write all customer accounts, employees and transaction objects to a file. When your program starts up it will read the file and update the arrays used to store our objects. Call your file bank.dat.
- (2) Add a credit card account.
- (3) And a bank loan account. Have different types of bank loans: Personnel, RRSP and mortgage.
- (4) Print out monthly bank statements for each customer account.
- (5) Add Transaction derived classes. Deposit, Withdraw, and Transfer.
- (6) In the bank class make an array of pointers to Account objects. This will make it easier to find an account.
- (7) you may want to types of customers Business Customers and regular customers.



IMPORTANT

You should use all the material in all the lessons to do this assignment. If you do not know how to do something or have to use additional books or references to do the questions or exercises, please let us know immediately. We want to have all the required information in our lessons. By letting us know we can add the required information to the lessons. The lessons are updated on a daily bases. We call our lessons the "living lessons". Please let us keep our lessons alive.

E-Mail all typos, unclear test, and additional information required to:

courses@cstutoring.com

E-Mail all attached files of your completed project to \$25 charge for marking:

students@cstutoring.com

This assignment is copyright (C) 1998-2002 by The Computer Science Tutoring Center "cstutoring"

This document is not to be copied or reproduced in any form. For use of student only