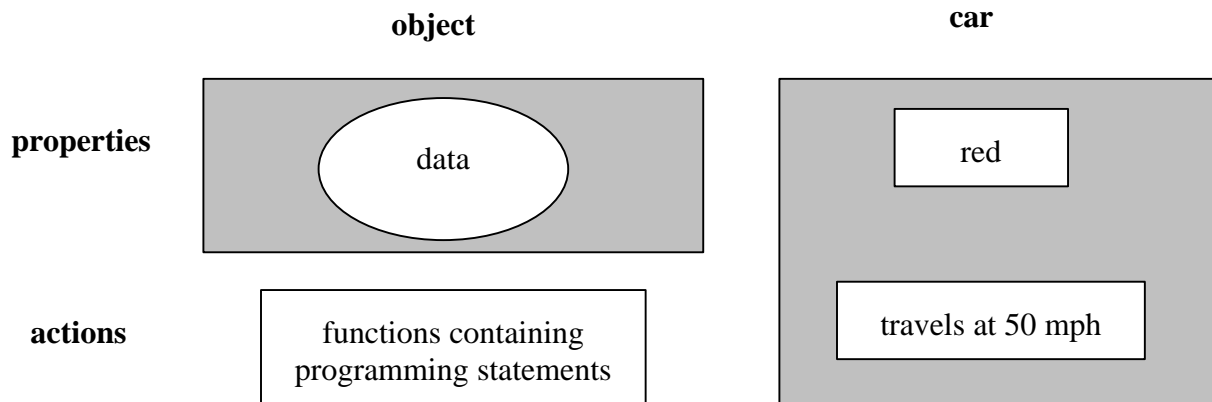


C++ PROGRAMMERS GUIDE LESSON 1

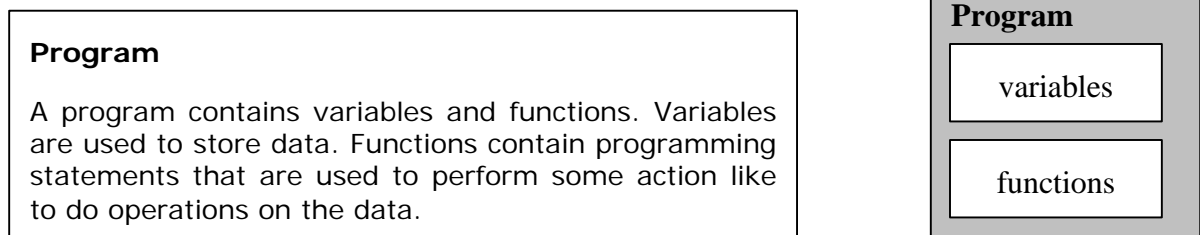
File:	CppGuideL1.doc
Date Started:	July 12, 1998
Last Update:	Mar 21, 2002
Version:	4.0

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 represent **properties** and programming **statements** to perform **actions**. This action can be represented by changing the value of the data in the object

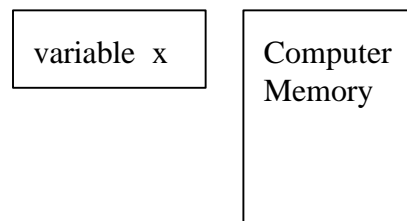


Data is a value stored in the computer memory. Every **data item** is represented in a program as a **variable** that has a **name** for identification like **x**. The **location** where the **data** is stored in the computer memory is represented by the variable name. Programming statements used to 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.



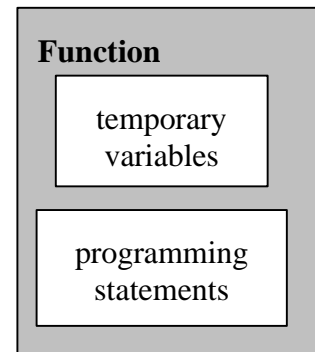
Programming Statement

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.



What's this Object Oriented stuff about anyway?

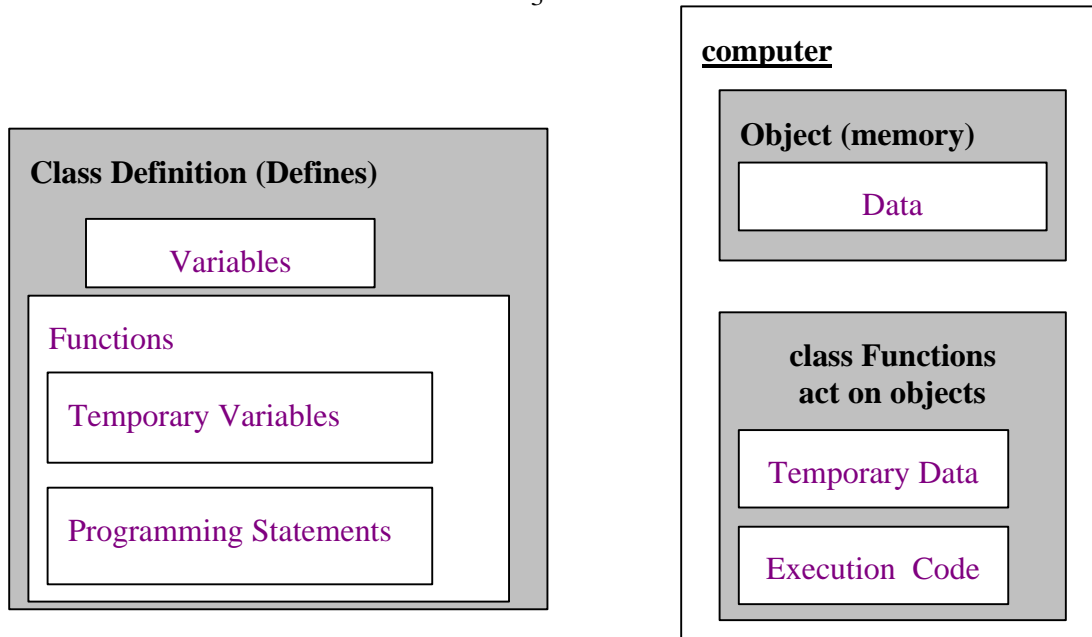
The idea behind object oriented programming is that you can have many **mini programs** each with their own variables and functions. The data part of these mini programs are called objects. The **functions** of the mini program is used to do operations on the data. Objects must be defined before they 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

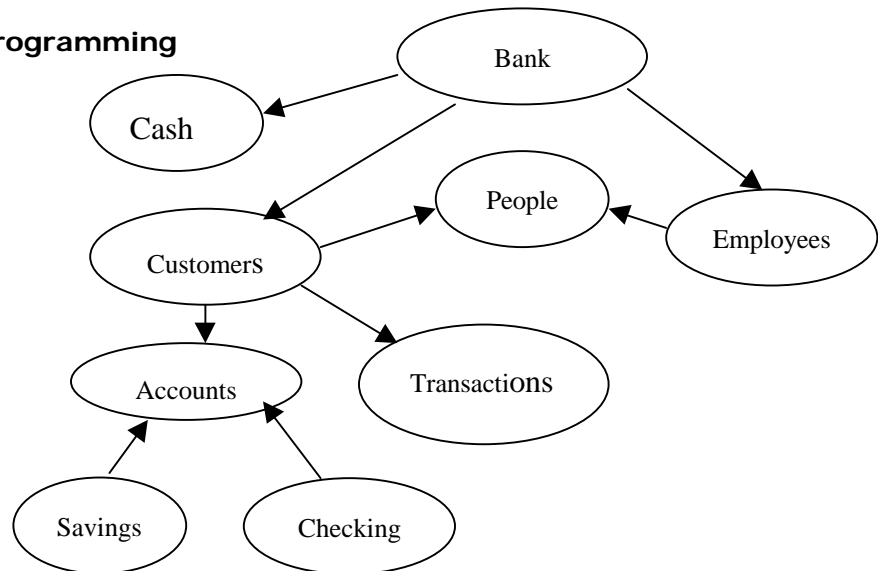
An object is memory for the variables defined in a class. The functions defined in the class are used to do operations on the data in the object.



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

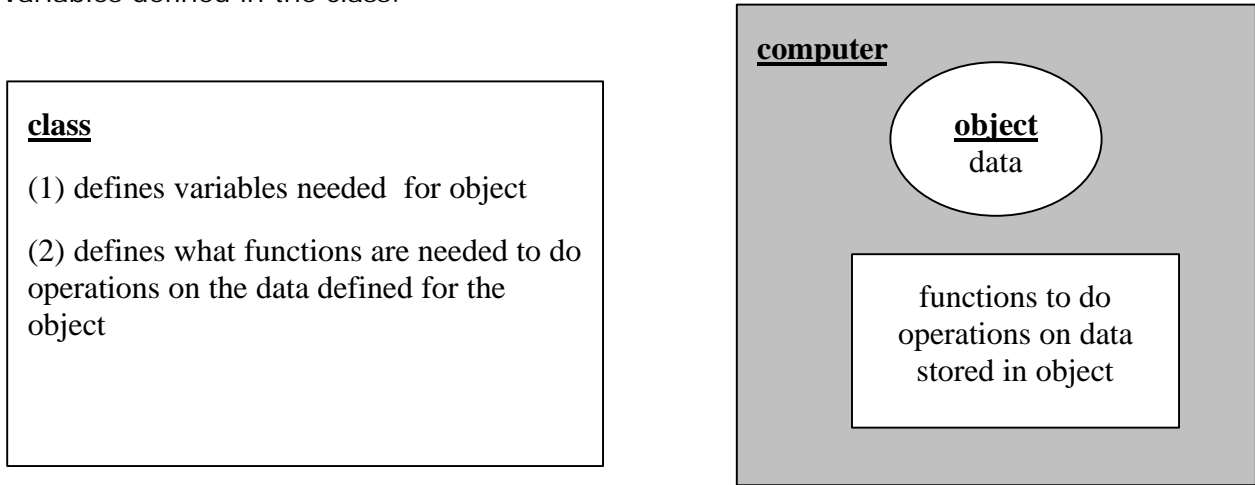
A good example is a banking system. Here we have many objects. A bank has customers, accounts and cash. Accounts may be checking or savings. Customers have accounts and do transactions with the accounts. Banks have Customers and Employees. Customers and Employees are people. Classes may be sub classes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

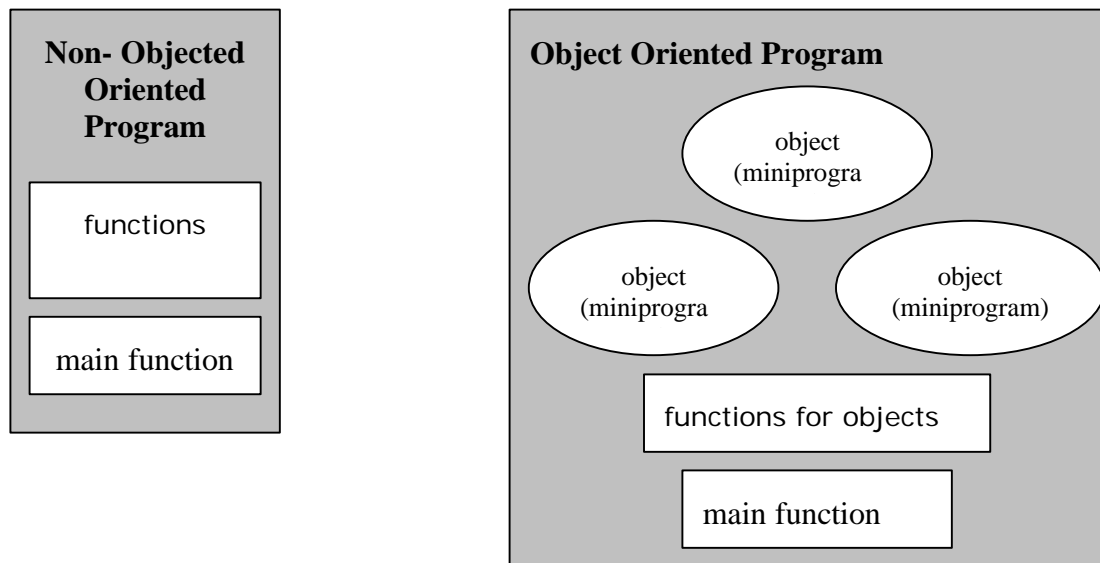
What is the difference between a class and an object ?

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

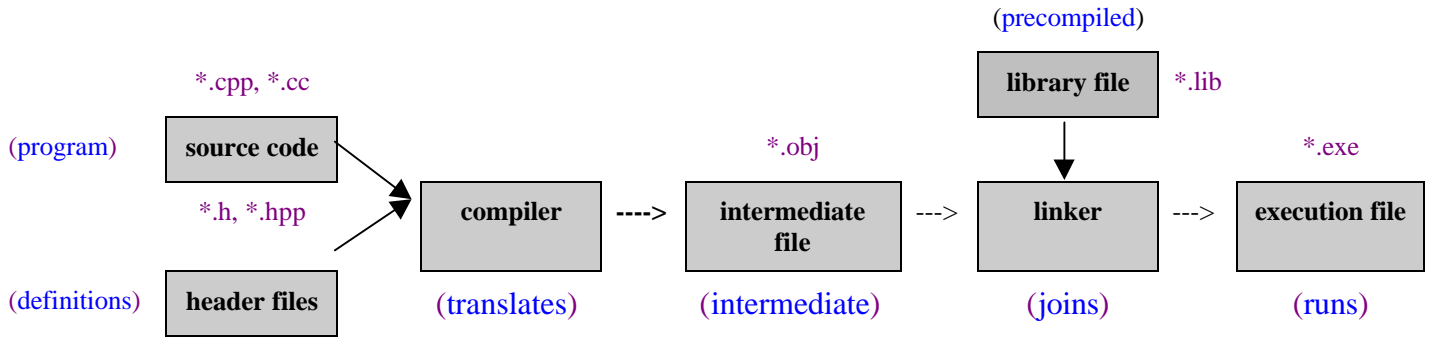
A C++ program is made up of programming components usually arranged in a predefined format. By following a proven format, your program will be more organized and easier to read.

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 preceded by a `/*` and followed by a `*/`. This style is usually used to explain a section of programming lines. The comment may span multiple lines.

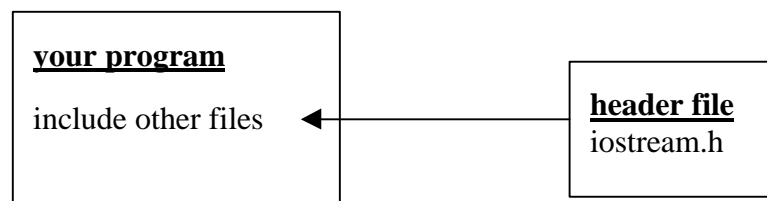
```
/* My First C++ Program */
```

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.

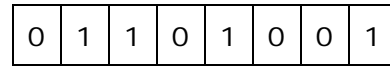
1 byte is 8 bits

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.

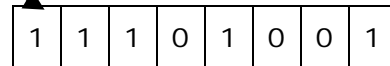
Signed **Positive** numbers start with a 0.



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:

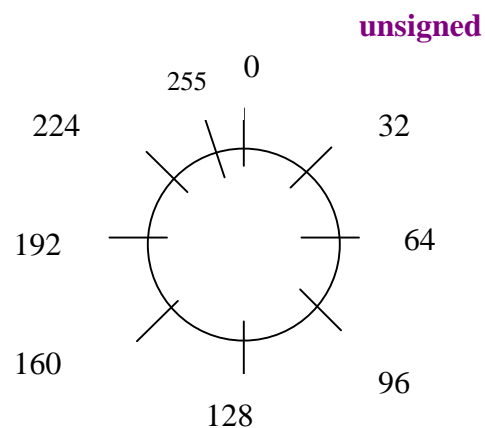
N = 8 bits

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

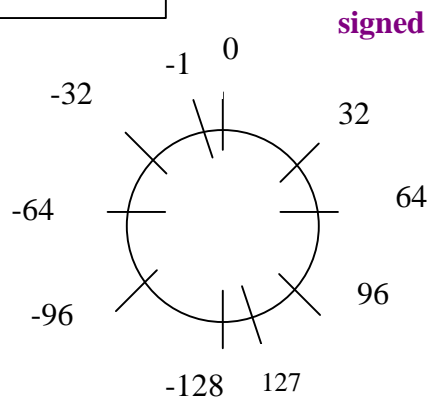
To calculate the range of a signed number for N = 8 then:

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

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	$1.12365 * 10^{12}$

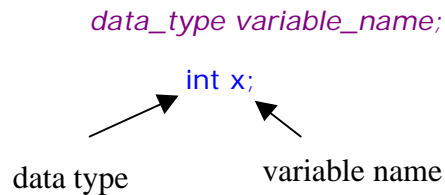
VARIABLES

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.



**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 declaring 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 3
```

```
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;
```

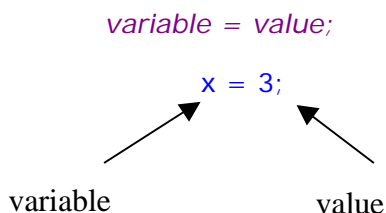
```
int i=10,j=20,k=30; // declare variables i, j, k of data type int
```

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
then 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)

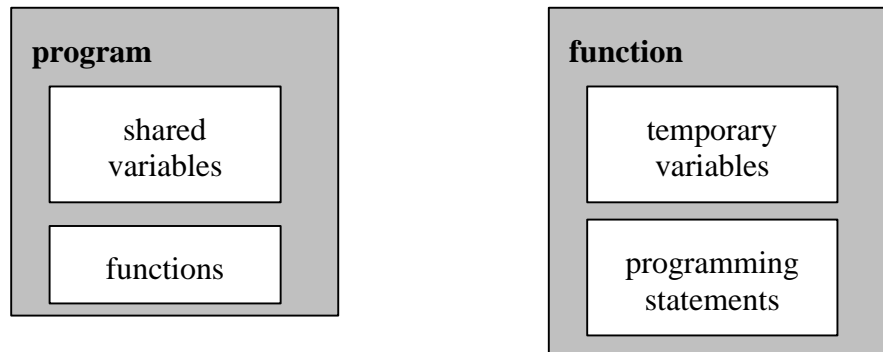
The letter 'A' is represented in computer memory by the numeric value 65. Each letter on the keyboard gets a numeric value. All the numeric values are listed in a chart known as the ASCII chart.

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**.

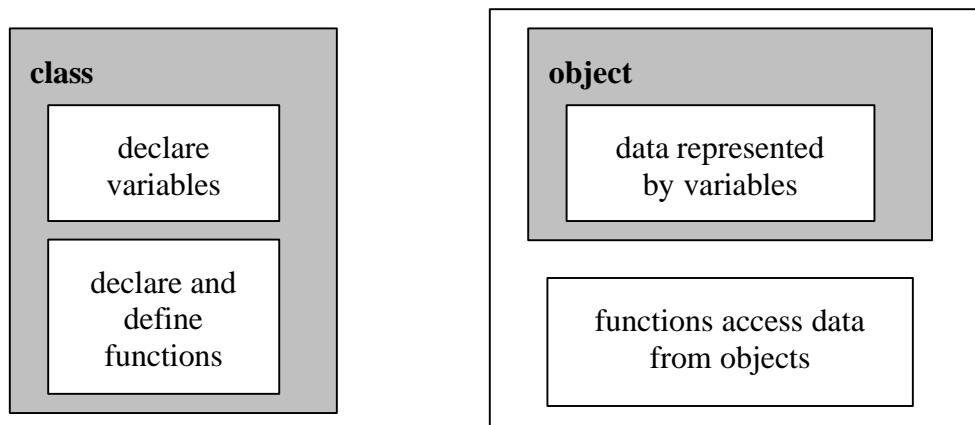
Variables in Programs and Functions

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

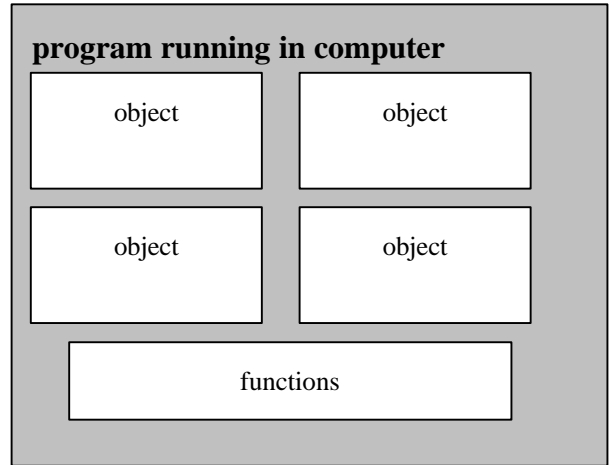
An object is allocated memory for the variables defined in a class. The data lasts for the life time of the object. The functions defined in a class share all the variables defined in the class.



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

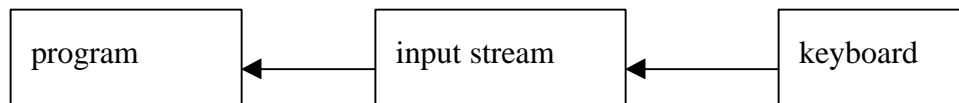
Object Oriented Programming allows programs to be very organized and data to be secure. One program may define many classes. By having many classes, each having their own data variables, makes programming tasks very easy. Each class will have a dedicated purpose. The variables in the class will store 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.



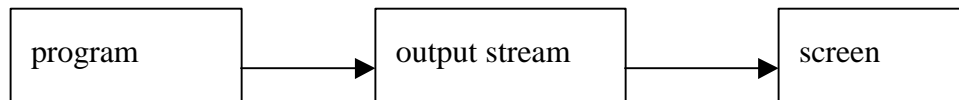
Do not be too worried of understanding classes and functions for now. You just need to be familiar with the terminology. You just need to know something about classes and objects to understand input and output stream classes. Just remember classes are what you type in and objects are allocated memory 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.

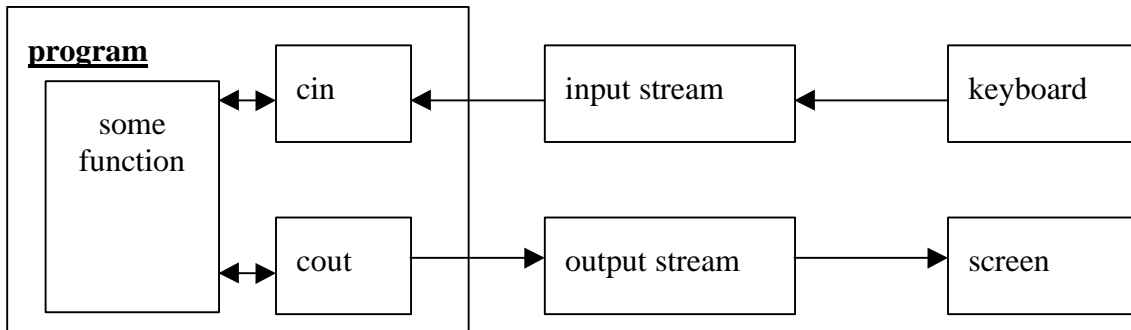


Output streams allow your program to send data to the computer screen or to a data file.



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++ programs 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

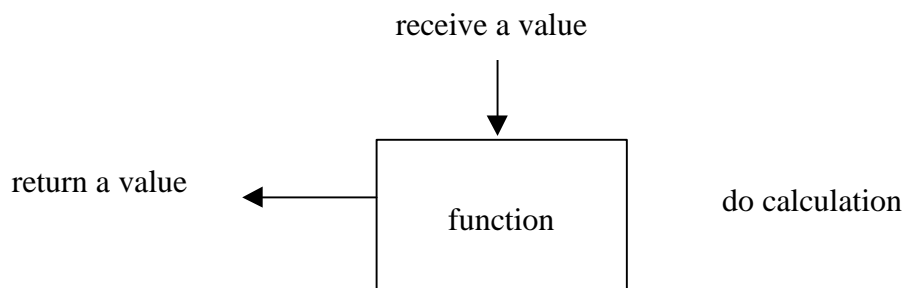
A simple C++ program just only requires a main function and no classes. A **function** is a group of programming statements identified by a function name. The **main** function is the first function to executed in your program. The job of the main function is to create objects from class definitions 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	const int Sun = 0;
#define Mon	1	const int Mon = 1;
#define Tue	2	const int Tue = 2;
#define Wed	3	const int Wed = 3;
#define Thur	4	const int Thur = 4;
#define Fri	5	const int Fri = 5;
#define Sat	6	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
#define
const
data type
variable
typedef
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 preceding the variable or value you want to force.

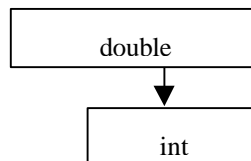
```
new_data_type = (typecast) old_data_type.
```

```
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 **days** 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**.

IMPORTANT

You should use all the material in all the lessons to do the questions and exercises. 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 exercises to:

students@cstutoring.com

Order your next Lesson from:

<http://www.cstutoring.com/cpp>

This lesson is copyright (C) 1998-2001 by The Computer Science Tutoring Center "cstutoring"
This document is not to be copied or reproduced in any form. For use of student only.