Introduction to C++
Lecture 1

programming

- A digital computer is a useful tool for solving a great variety of problems.
- A solution to a problem is called an algorithm; it describes the sequence of steps to be performed for the problem to be solved.
- To make an algorithm intelligible to a computer, it needs to be expressed in a language understood by a computer (machine language).
- Programs expressed in the machine language are said to be executable.
- A further abstraction of machine language is the assembly language.
- High-level languages such as C++ provide a much more convenient notation for implementing algorithms.

Compiler

- A program written in a high-level language is translated to assembly language by a translator called a **compiler.**
- **The assembly** code produced by the compiler is then assembled to produce an executable program.

C++ Compiler

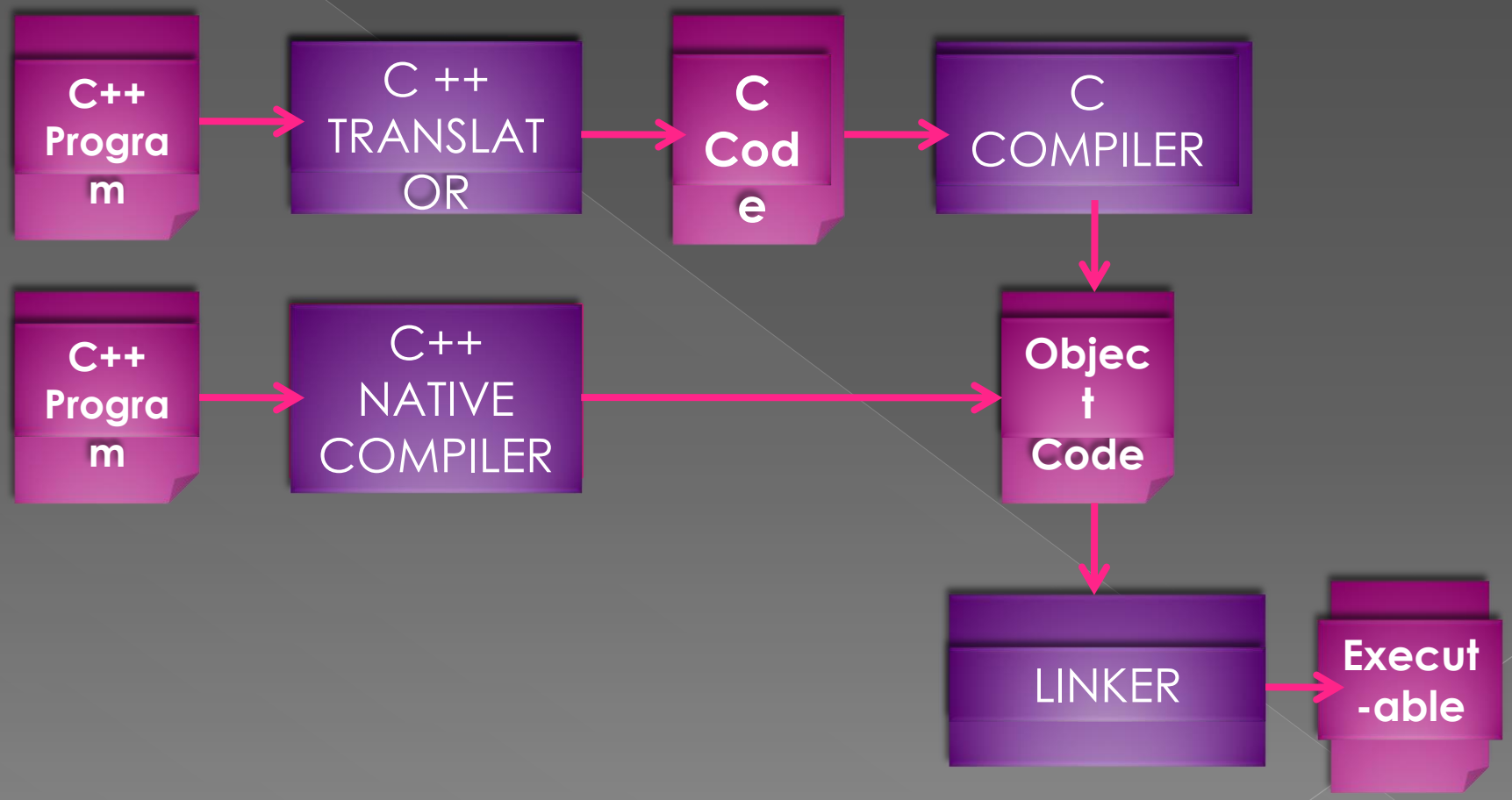
- Compiling a C++ program involves a number of steps :

- First, the C++ **preprocessor goes over the program text and carries out the** instructions specified by the preprocessor directives (e.g., #include). The result is a modified program text which no longer contains any directives.

- Then, the C++ **compiler translates the program code. The compiler may be a** true C++ compiler which generates native (assembly or machine) code, or just a translator which translates the code into C. In the latter case, the resulting C code is then passed through a C compiler to produce native object code. In either case, the outcome may be incomplete due to the program referring to library routines which are not defined as a part of the program.

- Finally, the **linker completes the object code by linking it with the object code** of any library modules that the program may have referred to. The final result is an executable file.

C++ Compilation



Variables

- variable is a symbolic name for a memory location in which data can be stored and subsequently recalled.
- All variables have two important attributes:

- A **type which is established when the variable is defined (e.g., integer, real**, character). Once defined, the type of a C++ variable cannot be changed.

- A **value which can be changed by assigning a new value to the variable. The** kind of values a variable can assume depends on its type. For example, an integer variable can only take integer values (e.g., 2, 100, -12).

Simple variables

```
1   #include <iostream.h>
2   int main (void)
3   {
4           int workDays;
5           float workHours, payRate, weeklyPay;
6           workDays = 5;
7           workHours = 7.5;
8           payRate = 38.55;
9           weeklyPay = workDays * workHours * payRate;
10          cout << "Weekly Pay = ";
11          cout << weeklyPay;
12          cout << '\n';
13  }
```

Initialization

```
#include <iostream.h>
int main (void)
{
        int workDays = 5;
        float workHours = 7.5;
        float payRate = 38.55;
        float weeklyPay = workDays * workHours * payRate;
        cout << "Weekly Pay = ";
        cout << weeklyPay;
        cout << '\n';
}
```

Simple Input/Output

- The most common way in which a program communicates with the outside world is through simple Input/Output (IO) operations.
- C++ provides two useful operators for this purpose: >> for input and << for output.

Comments

- A comment is a piece of descriptive text which explains some aspect of a program.
- Program comments are totally ignored by the compiler and are only intended for human readers.
- C++ provides two types of comment delimiters:
  - Anything after // (until the end of the line on which it appears) is considered a comment.
  - Anything enclosed by the pair /* and */ is considered a comment.

Comments forms

```
#include <iostream.h>

/* This program calculates the weekly gross pay for a worker,
based on the total number of hours worked and the hourly
pay rate. */

int main (void)
{
    int workDays = 5;            // Number of work days per week
    float workHours = 7.5;       // Number of work hours per day
    float payRate = 33.50;       // Hourly pay rate
    float weeklyPay;             // Gross weekly pay
    weeklyPay = workDays * workHours * payRate;
    cout << "Weekly Pay = " << weeklyPay << '\n';
}
```

Arithmetic Operators

Operator	Name	Example
+	Addition	12 + 4.9 // gives 16.9
-	Subtraction	3.98 - 4 // gives -0.02
*	Multiplication	2 * 3.4 // gives 6.8
/	Division	9 / 2.0 // gives 4.5
%	Remainder	13 % 3 // gives 1

Relational Operators

Operator	Name	Example
==	Equality	`5 == 5 // gives 1`
!=	Inequality	`5 != 5 // gives 0`
<	Less Than	`5 < 5.5 // gives 1`
<=	Less Than or Equal	`5 <= 5 // gives 1`
>	Greater Than	`5 > 5.5 // gives 0`
>=	Greater Than or Equal	`6.3 >= 5 // gives 1`

Logical Operators

Operator	Name	Example
!	Logical Negation	! (5 == 5) // gives 0
&&	Logical And	5 < 6 && 6 < 6 // gives 1
||	Logical Or	5 < 6 || 6 < 5 // gives 1

Increment/Decrement Operators

k=5

Operator	Name	Example	
++	Auto Increment (prefix)	++k + 10	// gives 16
++	Auto Increment (postfix)	k++ + 10	// gives 15
--	Auto Decrement (prefix)	--k + 10	// gives 14
--	Auto Decrement (postfix)	k-- + 10	// gives 15

Assignment Operator

Operator	Example	Equivalent To
=	n = 25	
+=	n += 25	n = n + 25
-=	n -= 25	n = n - 25
*=	n *= 25	n = n * 25
/=	n /= 25	n = n / 25
%=	n %= 25	n = n % 25
&=	n &= 0xF2F2	n = n & 0xF2F2
|=	n |= 0xF2F2	n = n | 0xF2F2
^=	n ^= 0xF2F2	n = n ^ 0xF2F2
<<=	n <<= 4	n = n << 4
>>=	n >>= 4	n = n >> 4

Conditional Operator

- The conditional operator takes three operands. It has the general form:

operand1 ? operand2 : operand3

- First operand1 is evaluated, which is treated as a logical condition.
- If the result is nonzero then operand2 is evaluated and its value is the final result.
- Otherwise, operand3 is evaluated and its value is the final result.

- For example:

- int m = 1, n = 2;
- int min = (m < n ? m : n); // min receives 1

Comma Operator

- Multiple expressions can be combined into one expression using the comma operator.

- For example:

```
int m, n, min;
int mCount = 0, nCount = 0;
...//
min = (m < n ? mCount++, m : nCount++, n);
```

- Here when m is less than n, mCount++ is evaluated and the value of m is stored inmin.
- Otherwise, nCount++ is evaluated and the value of n is stored in min.

The sizeof Operator

- C++ provides a useful operator, sizeof, for calculating the size of any data item or type.
- It takes a single operand which may be a type name (e.g., int) or an expression (e.g., 100) and returns the size of the specified entity in bytes.
- The outcome Is totally machine-dependent.

```
cout << "HELLO  size = " << sizeof("HELLO") << " bytes\n";
```

HELLO size = 6 bytes

```
cout << "1.55 size = " << sizeof(1.55) << " bytes\n";
```

1.55 size = 8 bytes

Simple Type Conversion

- A value in any of the built-in types we have see so far can be converted (*type-cast)* to any of the other types.
- For example:

```
int j=75;
cout << (char) j;
```

Output : K

- In some cases, C++ also performs **implicit type conversion. This happens** when values of different types are mixed in an expression. For example:

```
double d = 1; // d receives 1.0
int i = 10.5; // i receives 10
i = i + d; // means: i = int(double(i) + d)
```

Statements

- Simple statement
- Compound Statements
- The if Statement
- The switch Statement
- The while Statement
- The do Statement
- The for Statement
- The continue Statement
- The break Statement
- The goto Statement
- The return Statement

Simple Statements

- A simple statement is a computation terminated by a semicolon. Variable definitions and semicolon terminated expressions are examples:

```
int i; // declaration statement
++i; // this has a side-effect
double d = 10.5; // declaration statement
d + 5; // useless statement!
```

The last example represents a useless statement, because it has no side-effect (d is added to 5 and the result is just discarded).

- The simplest statement is the null statement which consists of just a semicolon:

```
;     // null statement
```

Compound Statements

- Multiple statements can be combined into a **compound statement by enclosing** them within braces.
- For example:

```
{ int min, i = 10, j = 20;
min = (i < j ? i : j);
cout << min << '\n';
}
```

- Compound statements are useful in two ways:
  - (i) they allow us to put multiple statements in places where otherwise only single statements are allowed.
  - (ii) they allow us to introduce a new **scope in the program.**
- **A scope is a part of the** program text within which a variable remains defined. For example, the scope of min, i, and j in the above example is from where they are defined till the closing brace of the compound statement. Outside the compound statement, these variables are not defined.

The if Statement

- It is sometimes desirable to make the execution of a statement dependent upon a condition being satisfied. The if statement provides a way of expressing this, the general form of which is:

```
if (expression)
    Statement1
else
    Statement2
```

- First *expression is evaluated. If the outcome is nonzero then statement is* executed. Otherwise, nothing happens.

- Example

```
if (count != 0)
average = sum / count;
```

The switch statement

- The switch statement provides a way of choosing between a set of alternatives, based on the value of an expression. The general form of the switch statement is:

switch (*expression*) {
    case $constant_1$:
        statements;
    ...
    case $constant_n$:
        statements;
    default:
        statements;
}

```
char operator1;
int operand1=4, operand2=5, result=0;

cin >> operator1;
switch (operator1) {
case '+': result = operand1 + operand2;
break;
case '-': result = operand1 - operand2;
break;
case '*': result = operand1 * operand2;
break;
case '/': result = operand1 / operand2;
break;
default: cout << "unknown operator: " <<
operator1 << '\n';
break;
}
```

The while Statement

The general form of the while statement is:

while (*expression*)
statement;

```
Int i = 1, n=5, sum = 0;
while (i <= n)
sum += i++;
```

- First *expression (called the* ***loop condition) is evaluated. If the outcome is nonzero***
- then *statement (called the* ***loop body) is executed and the whole process is*** repeated.
- Otherwise, the loop is terminated.

The do Statement

The general form of the do statement is:

```
do
statement;
while (expression);
```

```
do {
cin >> n;
cout << n * n << '\n';
} while (n != 0);
```

- First statement is executed and then expression is evaluated.
- If the outcome of the latter is nonzero
- then the whole process is repeated. Otherwise, the loop is terminated.

The for Statement

The general form of the for statement is:

for (*expression1; expression2; expression3)*
statement;

- First *expression1 is evaluated. Each time round the loop, expression2 is* evaluated.
- If the outcome is nonzero then statement is executed and *expression3 is* evaluated.
- Otherwise, the loop is terminated.

The continue Statement

- The continue statement terminates the current iteration of a loop and instead jumps to the next iteration.
- It applies to the loop immediately enclosing the continue statement.
- It is an error to use the continue statement outside a loop.

```
do {
cin  >> num;
if (num < 0) continue;
// process num here...
} while (num != 0);
```

The break Statement

- A break statement may appear inside a loop (while, do, or for) or a switch statement.
- It causes a jump out of these constructs, and hence terminates them.
- Like the continue statement, a break statement only applies to the loop or switch immediately enclosing it.
- It is an error to use the break statement outside a loop or a switch.

- For example, suppose we wish to read in a user password, but would like to allow the user a limited number of attempts:

```
for (i = 0; i < attempts; ++i) {
cout << "Please enter your password: ";
cin >> password;
if (Verify(password)) // check password for correctness
break; // drop out of the loop
cout << "Incorrect!\n";
}
```

The goto Statement

The goto statement has the general form:

goto *label;*

- where *label is an identifier which marks the jump destination of goto. The label* should be followed by a colon and appear before a statement within the same function as the goto statement itself.

- For example

```
for (i = 0; i < attempts; ++i) {
cout << "Please enter your password: ";
cin >> password;
if (Verify(password)) // check password for correctness
goto out; // drop out of the loop
cout << "Incorrect!\n";
}
out:
//etc...
```

The return Statement

The return statement enables a function to return a value to its caller. It has the general form:

return *expression;*

- where *expression denotes the value returned by the function. The type of this value* should match the return type of the function.
- For a function whose return type is void, *expression should be empty:*

return;

Reference

- Sharam Hekmat, “C++ Essentials”, www.pragsoft.com

Questions

- Write a program which inputs a temperature reading expressed in Fahrenheit and outputs its equivalent in Celsius, using the formula:

  °C =5/9*(F-32)

  Compile and run the program. Its behavior should resemble this:

  Temperature in Fahrenheit: **41**

  41 degrees Fahrenheit = 5 degrees Celsius

- Write a program which inputs a positive integer *n* and outputs 2 raised to the power of *n*.

- Write a program which inputs three numbers and outputs the message Sorted if The numbers are in ascending order, and outputs Not sorted otherwise.

- Write a program which inputs a date in the format dd/mm/yy and outputs it in the format month dd, year. For example, 25/12/61 becomes:

  December 25, 1961

FUNCTIONS LECTURE 2

FUNCTION DEFINITION

- A function provides a convenient way of packaging a computational recipe, so that it can be used as often as required.

- A function definition consists of two parts:
  - interface
  - body.

INTERFACE OF A FUNCTION

The interface of a function (function prototype) specifies how it may be used. It consists of three entities:

- The function name. This is simply a unique identifier.
- The function parameters (also called its signature). This is a set of zero or more typed identifiers used for passing values to and from the function.
- The function return type. This specifies the type of value the function returns (A function which returns nothing should have the return type void).

THE BODY OF A FUNCTION

- The body of a function contains the computational steps (statements) that comprise the function.
- Example

```
int power (int base, unsigned int exponent)
{
    int result = 1;
    for (int i = 0; i < exponent; ++i)
        result *= base;
    return result;
}
```

FUNCTIONS CALL

- A **function call consists of:**

  the function name followed by the call operator brackets '()', inside which zero or more comma-separated **arguments appear.**

Example

```
#include <iostream.h>
main (void)
{
cout << "2 ^ 8 = " << Power(2,8) << '\n';
}
```

- **The number of arguments should match the** number of function parameters. Each argument is an expression whose type should match the type of the corresponding parameter in the function interface.

FUNCTION DECLARATION

- function should be declared before its is used. A function declaration simply consists of the function prototype, which specifies the function name, parameter types, and return type. Although a function may be declared without its parameter names.

```
int power (int base, unsigned int exponent);
```

FUNCTIONS EXECUTION

- When a function call is executed, the arguments are first evaluated and their resulting values are assigned to the corresponding parameters. The function body is then executed. Finally, the function return value (if any) is passed to the caller.

PARAMETERS AND ARGUMENTS

C++ supports two styles of parameters:

- Value parameter
- Reference parameters

VALUE PARAMETERS

- A **value parameter** receives a *copy of the value of the argument passed to it. As a result, if the function* makes any changes to the parameter, this will not affect the argument. For example:

```
#include <iostream.h>
void Foo (int num)
{
    num = 0;
    cout << "num = " << num << '\n';
}
int main (void)
{
    int x = 10;
    Foo(x);
    cout << "x = " << x << '\n';
    return 0;
}
```

Output

```
num = 0;
x = 10;
```

VALUE PARAMETERS CONT.

```
#include "stdafx.h"
# include <iostream.h>
int Foo (int num)
{
    num = 0;
    cout << "num = " << num << '\n';
    return num;
}
int main (void)
{
    int x = 10;
    x=Foo(x);
    cout << "x = " << x << '\n';
    return 0;
}
```

Output

```
num = 0;
x = 0;
```

REFERENCE PARAMETERS

A reference parameter receives the argument passed to it and works on it directly. Any changes made by the function to a reference parameter is in effect directly applied to the argument.

GLOBAL AND LOCAL SCOPE

Global Scope

- Everything defined at the program scope level (i.e., outside functions and classes) is said to have a **global scope .**
- Uninitialized global variables are automatically initialized to zero.
- This means that the same global variable or function may not be *defined more than once at the global level*.
- Global entities are generally accessible everywhere in the program.

Local Scope

- the body of a function represents a local scope. The parameters of a function have the same scope as the function body.
- Variables defined within a local scope are visible to that scope only.
- Local scopes may be nested, in which case the inner scopes override the outer scopes.

```
int xyz; // xyz is global
void Foo (int xyz) // xyz is local to the body of Foo
{
if (xyz > 0) {
double xyz; // xyz is local to this block
...//
}
}
```

SCOPE OPERATOR ::

- Because a local scope overrides the global scope, having a local variable with the same name as a global variable makes the latter inaccessible to the local scope. For example, in

```
int error;
void Error (int error)
{
...//
}
```

- the global **error** is inaccessible inside Error, because it is overridden by the local error parameter.

- This problem is overcome using the unary scope operator :: which takes a global entity as argument:

```
int error;
void Error (int error)
{...//
if (::error != 0) // refers to global error
...//
}
```

STATIC VARIABLES AND FUNCTIONS

- A local variable in a function may be defined as static. The variable will remain only accessible within its local scope; however, its lifetime will no longer be confined to this scope, but will instead be global. In other words, a static local variable is a global variable which is only accessible within its local scope. Static local variables are useful when we want the value of a local variable to persist across the calls to the function in which it appears. For example, consider the following program :

STATIC VARIABLES AND FUNCTIONS - EXAMPLE

```
#include <iostream.h>
void Foo (void);
int main (void)
{
Foo();
Foo();
return 0;
}
void Foo (void)
{
static int staticVar=1;
cout <<"staticVar="<< staticVar++<<"\n";
}
```

Output:

```
staticVar=1
staticVar=2
```

SYMBOLIC CONSTANTS

Preceding a variable definition by the keyword const makes that variable read only (i.e., a symbolic constant). A constant must be initialized to some value when it is defined. For example:

```
const int maxSize = 128;
const double pi = 3.141592654;
```

- A constant with no type specifier is assumed to be of type int:

```
const maxSize = 128; // maxSize is of type int
```

SYMBOLIC CONSTANTS

- A function parameter may also be declared to be constant. This may be used to indicate that the function does not change the value of a parameter:

```
int Power (const int base, const unsigned int exponent)
{
...//
}
```

- A function may also return a constant result:

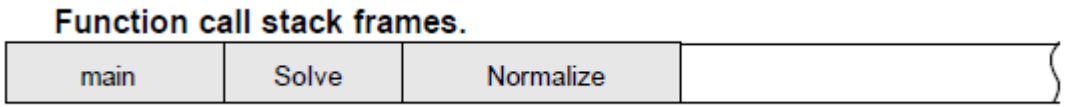
```
const char* SystemVersion (void)
{
return "5.2.1";
}
```

RUNTIME STACK

- C++ function call execution is based on a runtime stack. When a function is called, memory space is allocated on this stack for the function parameters, return value, and local variables, as well as a local stack area for expression evaluation. The allocated space is called a **stack frame. When a function returns, the allocated stack frame is released so that it can** be reused.
- For example, consider a situation where main calls a function called Solve which in turn calls another function called Normalize:

RUNTIME STACK

```
int Normalize (void)
{
...//
}
int Solve (void)
{
...//
Normalize();
...//
}
int main (void)
{
...//
Solve();
...//
}
```



RECURSION

- A function which calls itself is said to be **recursive. Recursion is a general** programming technique applicable to problems which can be defined in terms of themselves.
- Take the factorial problem, for instance, which is defined as:

· Factorial of 0 is 1.

· Factorial of a positive number *n is n times the factorial of n-1*.

- The second line clearly indicates that factorial is defined in terms of itself and hence can be expressed as a recursive function:

```
int Factorial (unsigned int n)
{
return n == 0 ? 1 : n * Factorial(n-1);
}
```

RECURSION

For n set to 3, the table below provides a trace of the calls to Factorial. The stack frames for these calls appear sequentially on the runtime stack, one after the other.

Factorial(3) execution trace.

Call	n	n == 0	n * Factorial(n-1)	Returns
First	3	0	3 * Factorial(2)	6
Second	2	0	2 * Factorial(1)	2
Third	1	0	1 * Factorial(0)	1
Fourth	0	1		1

recursive function must have at least one **termination condition which can** be satisfied. Otherwise, the function will call itself indefinitely until the runtime stack overflows. The Factorial function, for example, has the termination condition n ==0 which, when satisfied, causes the recursive calls to fold back.

RECURSION

- As a general rule, all recursive functions can be rewritten using iteration. In situations where the number of stack frames involved may be quite large, the iterative version is preferred. In other cases, the elegance and simplicity of the recursive version may give it the edge.
- For factorial, for example, a very large argument will lead to as many stack frames. An iterative version is therefore preferred in this case:

```
int Factorial (unsigned int n)
{
int result = 1;
while (n > 0) result *= n--;
return result;
}
```

VARIABLE NUMBER OF ARGUMENTS - OVERLOADED FUNCTIONS

- An overloaded function appears to perform different activities depending on the kind of data sent to it.
- Consider the repchar() function from the following example. repchar() used a character and a line length that were both specified when the function was called.
- It would be far more convenient to use the same name for three functions, even though they each have different arguments.
- Here’s a program, OVERLOAD, that makes this possible:

VARIABLE NUMBER OF ARGUMENTS - OVERLOADED FUNCTIONS

```
// overload.cpp
// demonstrates function overloading

#include <iostream>
using namespace std;
void repchar(); //declarations
void repchar(char);
void repchar(char, int);

int main()
}
repchar();              //displays 45 asterisks
repchar('=');           // displays 45 copies of specified character
repchar('+', 30);     // displays specified number of copies of specified character
return 0;
{
```

REFERENCES

- C++ essentials, book.
- C++ mastring , book.

Arrays & Structures

Lecture 3

Arrays

- An **array consists of a set of objects (called its elements), all of which are of** the same type and are arranged contiguously in memory. In general, only the array itself has a symbolic name, not its elements. Each element is identified by an **index** which denotes the position of the element in the array. The number of elements in an array is called its **dimension. The dimension of an array is fixed and** predetermined; it cannot be changed during program execution.

- Arrays are suitable for representing composite data which consist of many similar, individual items. Examples include: a list of names, a table of world cities and their current temperatures, or the monthly transactions for a bank account.

Arrays

- An array variable is defined by specifying its dimension and the type of its elements. For example, an array representing 10 height measurements (each being an integer quantity) may be defined as:

  int heights[10];

- The individual elements of the array are accessed by indexing the array. The first array element always has the index 0. Therefore, heights[0] and heights[9] denote, respectively, the first and last element of heights. Each of heights elements can be treated as an integer variable. So, for example, to set the third element to 177, we may write:

  heights[2] = 177;

- Attempting to access a nonexistent array element (e.g., heights[-1] or heights[10]) leads to a serious runtime error (called 'index out of bounds' error).

- Processing of an array usually involves a loop which goes through the array element by element. Listing 5.13 illustrates this using a function which takes an array of integers and returns the average of its elements.

Example

The following example illustrates using a function which takes an array of integers and returns the average of its elements.

```
const int size = 3;
double Average (int nums[size])
{
double average = 0;
for (i = 0; i < size; ++i)
average += nums[i];
return average/size;
}
```

Array Initialization

- an array may have an initializer. Braces are used to specify a list of comma-separated initial values for array elements. For example,

  int nums[3] = {5, 10, 15};

- initializes the three elements of nums to 5, 10, and 15, respectively. When the number of values in the initializer is less than the number of elements, the remaining elements are initialized to zero:

  int nums[3] = {5, 10};

- When a complete initializer is used, the array dimension becomes redundant, because the number of elements is implicit in the initializer. The first definition of nums can therefore be equivalently written as:

  int nums[] = {5, 10, 15}; // no dimension needed

Strings

- A C++ string is simply an array of characters. For example,

  char str[ ] = "HELLO";

- defines str to be an array of six characters: five letters and a null character. The terminating null character is inserted by the compiler. By contrast,

  char str[ ] = {'H', 'E', 'L', 'L', 'O'};

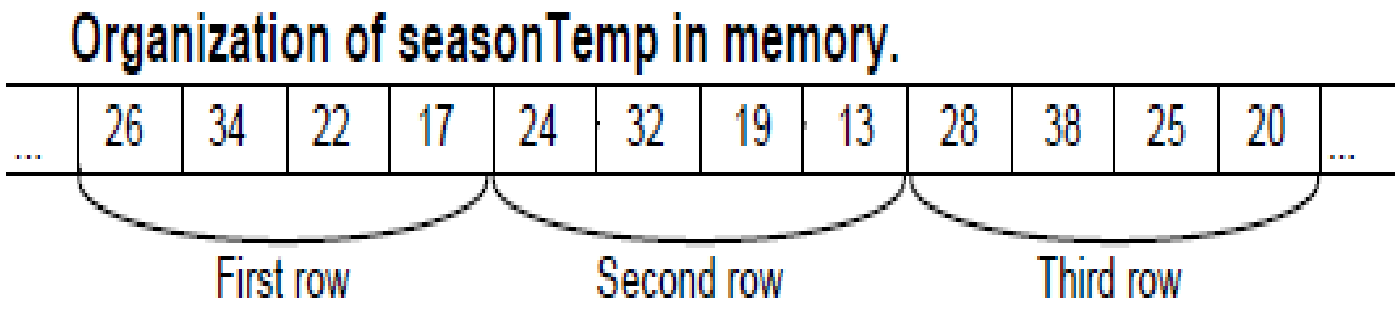
  defines str to be an array of five characters.

Multidimensional Arrays

- An array may have more than one dimension (i.e., two, three, or higher). The organization of the array in memory is still the same (a contiguous sequence of elements), but the programmer's perceived organization of the elements is different, for Example

  int seasonTemp[3][4];

- The organization of this array in memory is as 12 consecutive integer elements. The programmer, however, can imagine it as three rows of four integer entries each.



Multidimensional Arrays Initialization

- The array may be initialized using a nested initializer:

```
int seasonTemp[3][4] = {
{26, 34, 22, 17},
{24, 32, 19, 13},
{28, 38, 25, 10}
};
```

- Because this is mapped to a one-dimensional array of 12 elements in memory, it is equivalent to:

```
int seasonTemp[3][4] = {26, 34, 22, 17, 24, 32, 19, 13, 28, 38, 25, 10};
```

Multidimensional Arrays Processing

- Processing a multidimensional array is similar to a one-dimensional array, but uses nested loops instead of a single loop.:

```
const int rows = 3;
const int columns = 4;
int seasonTemp[rows][columns] = {
{26, 34, 22, 17},
{24, 32, 19, 13},
{28, 38, 25, 10}
};

  int HighestTemp (int temp[rows][columns])
{  int highest = 0;
for (i = 0; i < rows; ++i)
for (j = 0; j < columns; ++j)
if (temp[i][j] > highest)
highest = temp[i][j];
return highest;
}
```

Structures

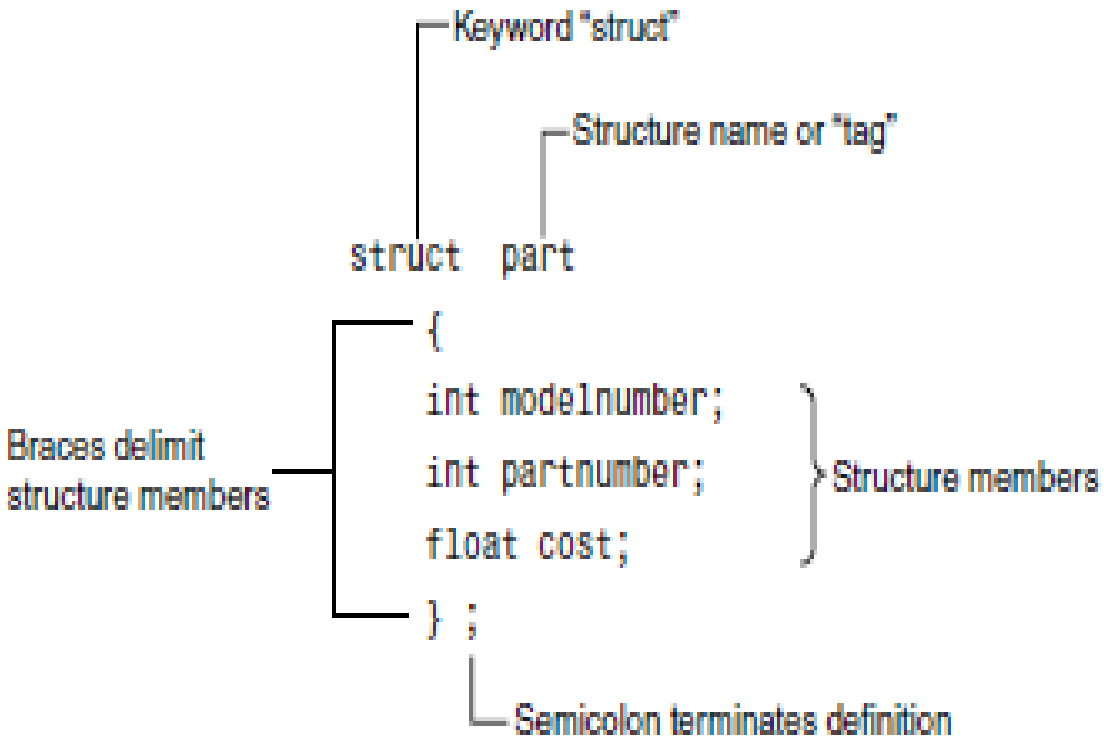
A structure is a collection of simple variables. The variables in a structure can be of different types: Some can be int, some can be float, and so on. (This is unlike the array, which we'll meet later, in which all the variables must be the same type.) The data items in a structure are called the *members of the structure*.

Defining the Structure

The structure definition tells how the structure is organized: It specifies what members the structure will have. Here it is:

```
struct part
{
int modelnumber;
int partnumber;
float cost;
};
```

Syntax of the Structure Definition



Use of the Structure Definition & Defining a Structure Variable

- **Use of the Structure Definition**

- The structure definition serves only as a blueprint for the creation of variables of type part. It does not itself create any structure variables; that is, it does not set aside any space in memory or even name any variables. This is unlike the definition of a simple variable, which does set aside memory. A structure definition is merely a specification for how structure variables will look when they are defined.

- **Defining a Structure Variable**

```
part part1;
```

- defines a variable, called part1, of type structure part. This definition reserves space in memory for part1.

Accessing Structure Members

- Once a structure variable has been defined, its members can be accessed using the dot operator. Here's how the first member is given a value:

  part1.modelnumber = 6244;

- The structure member is written in three parts: the name of the structure variable (part1); the dot operator, which consists of a period (.); and the member name.

Initializing Structure Members

- The next example shows how structure members can be initialized when the structure variable is defined. It also demonstrates that you can have more than one variable of a given structure type.

Example

```
#include <iostream>
using namespace std;
struct part //specify a structure
{
int modelnumber; //ID number of widget
int partnumber; //ID number of widget part
float cost; //cost of part
};
int main()
{
part part1 = { 6244, 373, 217.55F }; //initialize variable
part part2; //define variable
cout << "Model" << part1.modelnumber; //display first variable
cout << "part" << part1.partnumber;
cout << "costs $" << part1.cost << endl;
part2 = part1; //assign first variable to second
cout << "Model " << part2.modelnumber; //display second variable
cout << ", part " << part2.partnumber;
cout << ", costs $" << part2.cost << endl;
return 0;
}
```

Here's the output:

```
Model 6244, part 373, costs $217.55
Model 6244, part 373, costs $217.55
```

Structures as Arguments

The following example features a function that uses an argument of type structure named Distance:

```
#include <iostream>
using namespace std;
struct Distance //English distance
{
int feet;
float inches;
};
```

Structures as Arguments – cont.

```
Void disp( Distance ); //declaration
int main()
{
Distance d1, d2; //define two lengths
cout << "Enter feet: "; cin >> d1.feet;
cout << "Enter inches: "; cin >> d1.inches;
cout << "\nEnter feet: "; cin >> d2.feet;
cout << "Enter inches: "; cin >> d2.inches;
cout << "\nd1 = ";
disp(d1); //display length 1
cout << "\nd2 = ";
disp(d2); //display length 2
cout << endl;
return 0;
}

Void disp( Distance dd ) //parameter dd of type Distance
{
cout << dd.feet << "\'-" << dd.inches << "\"";
}
```

```
Enter feet: 6
Enter inches: 4

Enter feet: 5
Enter inches: 4.25

d1 = 6'-4"
d2 = 5'-4.25"
```

Returning Structure Variables

```
#include <iostream>
using namespace std;
struct Distance //English distance
{
int feet;
float inches;
};
Distance add (Distance, Distance); //declarations
Void disp(Distance);
```

Returning Structure Variables

```
int main()
{
Distance d1, d2, d3; //define three lengths
cout << "\nEnter feet: "; cin >> d1.feet;
cout << "Enter inches: "; cin >> d1.inches;
cout << "\nEnter feet: "; cin >> d2.feet;
cout << "Enter inches: "; cin >> d2.inches;
d3 = add (d1, d2); //d3 is sum of d1 and d2
cout << endl;
disp(d1); cout << " + "; //display all lengths
disp(d2); cout << " = ";
disp(d3); cout << endl;
return 0;
}
```

Returning Structure Variables – cont.

```
Distance add ( Distance dd1, Distance dd2 )
{
Distance dd3; //define a new structure for sum
dd3.inches = dd1.inches + dd2.inches; //add the inches
dd3.feet = 0; //(for possible carry)
if(dd3.inches >= 12.0) //if inches >= 12.0,
{ //then decrease inches
dd3.inches -= 12.0; //by 12.0 and
dd3.feet++; //increase feet
} //by 1
dd3.feet += dd1.feet + dd2.feet; //add the feet
return dd3; //return structure
}

Void disp( Distance dd )
{
cout << dd.feet << "\'-" << dd.inches << "\"";
}
```

Enter feet: 4
Enter inches: 5.5
Enter feet: 5
Enter inches: 6.5

4’ 5.5” + 5’ 6.5”= 10’

Pointers and References

Lecture 4

Pointers

- A pointer is simply the address of a variable in memory. Generally, objects can be accessed in two ways: directly by their symbolic name, or indirectly through a pointer. The act of getting to an object via a pointer to it, is called dereferencing the pointer. Pointer objects are defined to point to values of a specific type so that when the pointer is dereferenced, a typed value is obtained.

- Pointers are useful for creating dynamic variables during program execution.

- Unlike normal (global and local) variables which are allocated storage on the runtime stack, a dynamic variable is allocated memory from a different storage area called the heap.

- Dynamic variables do not obey the normal scope rules. Their scope is explicitly controlled by the programmer.

Pointer Definition

- A pointer variable is defined to 'point to' data of a specific type. For example:

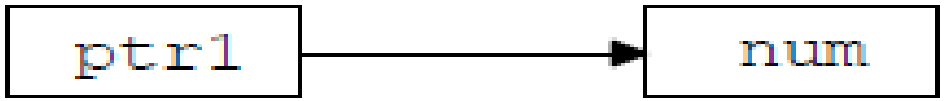
  int *ptr1; // pointer to an int

  char *ptr2; // pointer to a char
- The value of a pointer variable is the address to which it points. For example, given the definitions

  int num;
- we can write:

  ptr1 = #
- The symbol & is the address operator; it takes a variable as argument and returns the memory address of that variable. The effect of the above assignment is that the address of num is assigned to ptr1. Therefore, we say that ptr1 points to num.

A simple integer pointer.



Pointer Values

- Given that ptr1 points to num, the expression
  *ptr1
- dereferences ptr1 to get to what it points to, and is therefore equivalent to num. The symbol * is the dereference operator; it takes a pointer as argument and returns the contents of the location to which it points.

- In general, the type of a pointer must match the type of the data it is set to point to.

- Regardless of its type, a pointer may be assigned the value 0 (called the null pointer). The null pointer is used for initializing pointers, and for marking the end of pointer-based data structures (e.g., linked lists).

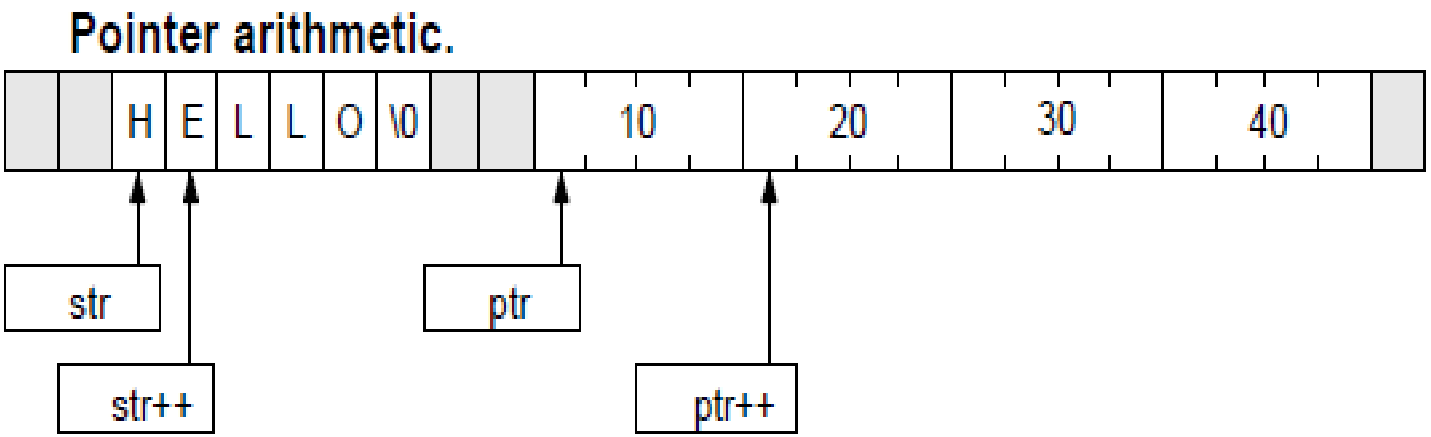
Pointer Arithmetic

- In C++ one can add an integer quantity to or subtract an integer quantity from a pointer. This is frequently used by programmers and is called pointer arithmetic. Pointer arithmetic is not the same as integer arithmetic, because the outcome depends on the size of the object pointed to. For example, suppose that an int is represented by 4 bytes. Now, given

```
char *str = "HELLO";
int nums[] = {10, 20, 30, 40};
int *ptr = &nums[0]; // pointer to first element
```

- str++ advances str by one char (i.e., one byte) so that it points to the second character of "HELLO", whereas ptr++ advances ptr by one int (i.e., four bytes) o that it points to the second element of nums.

Pointer Arithmetic - cont.



It follows, therefore, that the elements of "HELLO" can be referred to as *str, *(str + 1), *(str + 2), etc. Similarly, the elements of nums can be referred to as *ptr, *(ptr + 1), *(ptr + 2), and *(ptr + 3).

Example

- Pointer arithmetic is very handy when processing the elements of an array. The following example shows a string copying function similar to strcpy.

```
void CopyString (char *dest, char *src)
{
while (*dest++ = *src++);
}
```

Example

The following example shows how to pass a string argument and return a string to the main program:

```
#include "stdafx.h"
#include <iostream>
using namespace std;
char* stf(char *str3);
int main()
{
char *str1 = "ABC";
const int MAX = 10; //size of str2 buffer
char str2[MAX]; //empty string
char *st;
strcpy(str2, str1); //copy str1 to str2
cout << str2 << endl; //display str2
st=stf(str1);
cout << str1 << endl; //display str1
cout << st << endl; //display str1
return 0;
}
```

```
char* stf(char *str3)
{
    char* pointer;
    char *str1 =“XYZ";
    str3=str1;
    pointer=&str3[0];
    cout << str3 << endl;
    return pointer;
}
```

Output :

```
ABC
XYZ
ABC
XYZ
```

References

- A reference provides an alternative symbolic name (alias) for an object. Accessing an object through a reference is exactly the same as accessing it through its original name. References offer the power of pointers and the convenience of direct access to objects. They are used to support the call-by-reference style of function parameters, especially when large objects are being passed to functions.

Example

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

void main()
{
int      z=1;
int &x = z;
++x;
int y = x + z;
cout <<y;
   return ;
}
```

The following example shows the difference of passing argument by value, pointer and reference

```
#include<iostream.h>
void Swap1 (int x, int y);
void Swap2 (int *x, int *y);
void Swap3 (int &x, int &y);
int main (void)
{ int i = 10, j = 20;
Swap1(i, j);
cout << "return from swap1=";
 cout << i << ", " << j << '\n';
Swap2(&i, &j);
cout<< "return from swap2=  ";
cout << i << ", " << j << '\n';
Swap3(i, j);
cout<<"return from swap3=  ";
cout << i << ", " << j << '\n';
    return 0; }
```

```
// pass-by-value (objects)
void Swap1 (int x, int y)
{ int temp = x;
x = y;
y = temp; }

// pass-by-value (pointers)
void Swap2 (int *x, int *y)
{ int temp = *x;
*x = *y;
*y = temp; }

// pass-by-reference
void Swap3 (int &x, int &y)
{ int temp = x;
x = y;
y = temp; }
```

Typedefs

- Typedef is a syntactic facility for introducing symbolic names for data types. Just as a reference defines an alias for an object, a typedef defines an alias for a type. Its main use is to simplify otherwise complicated type declarations as an aid to improved readability. Here are a few examples:

```
typedef char *String;
Typedef char Name[12];
typedef unsigned int uint;
```

- The effect of these definitions is that String becomes an alias for char*, Name becomes an alias for an array of 12 chars, and uint becomes an alias for unsigned int. Therefore:

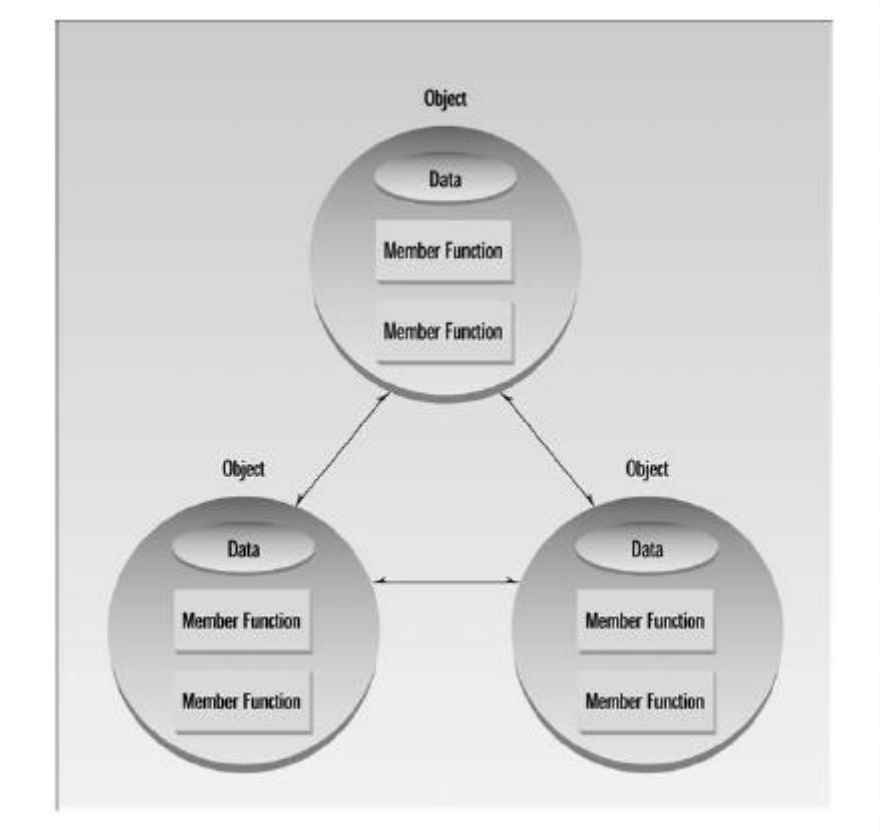
```
String str; // is the same as: char *str;
Name name; // is the same as: char name[12];
uint n; // is the same as: unsigned int n;
```

OOP, Classes, and Objects
Lecture 5

The Object-Oriented Approach

- The fundamental idea behind object-oriented languages is to combine into a single unit both data and the functions that operate on that data. Such a unit is called an object. An object's functions, called member functions in C++, typically provide the only way to access its data. If you want to read a data item in an object, you call a member function in the object. It will access the data and return the value to you. You can't access the data directly. The data is hidden, so it is safe from accidental alteration. Data and its functions are said to be encapsulated into a single entity. Data encapsulation and data hiding are key terms in the description of object-oriented languages. If you want to modify the data in an object, you know exactly what functions interact with it: the member functions in the object. No other functions can access the data. This simplifies writing, debugging, and maintaining the program. A C++ program typically consists of a number of objects, which communicate with each other by calling one another's member functions.

The organization of an OOP C++ program



OOP: An Approach to Organization

- Object-oriented programming is not primarily concerned with the details of program operation. Instead, it deals with the overall organization of the program. Most individual program statements in C++ are similar to statements in procedural languages, and many are identical to statements in C. Indeed, an entire member function in a C++ program may be very similar to a procedural function in C. It is only when you look at the larger context that you can determine whether a statement or a function is part of a procedural C program or an object-oriented C++ program.

Characteristics of Object-Oriented Languages

- Objects
- Classes
- Inheritance
- Reusability
- Creating New Data Types
- Polymorphism and Overloading

Objects

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

Classes

- In OOP we say that objects are members of classes. What does this mean? Almost all computer languages have built-in data types. For instance, a data type int, meaning integer, is predefined in C++ . You can declare as many variables of type int as you need in your program:

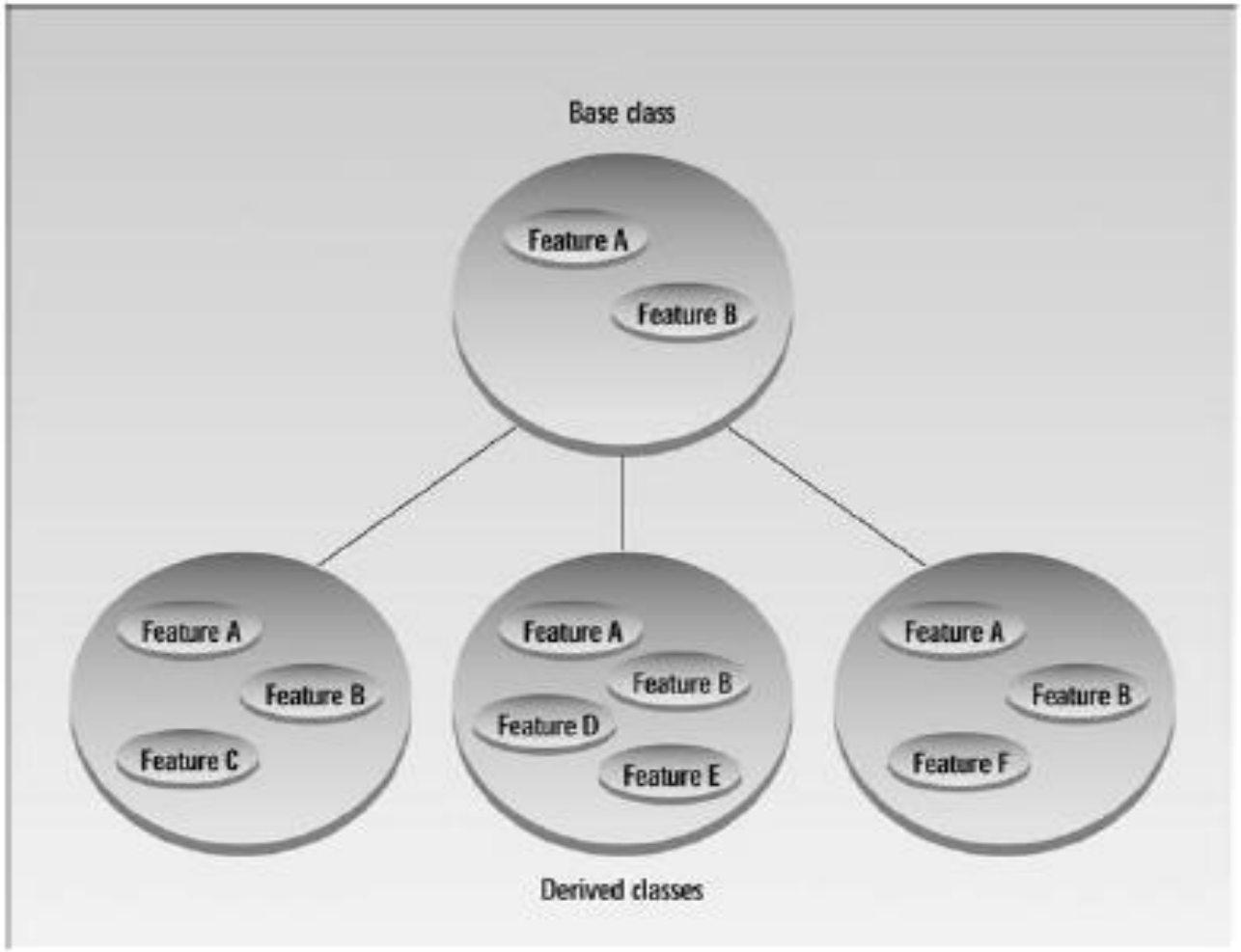
```
int day;
int count;
int divisor;
int answer;
```

- In a similar way, you can define many objects of the same class. It specifies what data and what functions will be included in objects of that class. Defining the class doesn't create any objects, just as the mere existence of data type int doesn't create any variables. A class is thus a description of a number of an object is often called an "instance" of a class.

Inheritance

- The idea of classes leads to the idea of inheritance. In our daily lives, we use the concept of classes divided into subclasses. We know that the animal class is divided into mammals, amphibians, insects, birds, and so on. The vehicle class is divided into cars, trucks, buses, motorcycles, and so on.
- In C++ the original class is called the base class; other classes can be defined that share its characteristics, but add their own as well. These are called derived classes.
- Don't confuse the relation of objects to classes, on the one hand, with the relation of a base class to derived classes, on the other. Objects, which exist in the computer's memory, each embody the exact characteristics of their class, which serves as a template. Derived classes inherit some characteristics from their base class, but add new ones of their own.

Inheritance – cont.



Reusability

- Once a class has been written, created, and debugged, it can be distributed to other programmers for use in their own programs. This is called reusability. It is similar to the way a library of functions in a procedural language can be incorporated into different programs.

Creating New Data Types

- One of the benefits of objects is that they give the programmer a convenient way to construct new data types. Suppose you work with two-dimensional positions (such as x and y coordinates, or latitude and longitude) in your program. You would like to express operations on these positional values with normal arithmetic operations, such as:

  position1 = position2 + origin

- where the variables position1, position2, and origin each represent a pair of independent numerical quantities. By creating a class that incorporates these two values, and declaring position1, position2, and origin to be objects of this class, we can, in effect, create a new data type. Many features of C++ are intended to facilitate the creation of new data types in this manner.

Polymorphism and Overloading

- Note that the = (equal) and + (plus) operators, used in the position arithmetic shown above, don't act the same way they do in operations on built-in types such as int. The objects position1 and so on are not predefined in C++, but are programmer-defined objects of class Position. How do the = and + operators know how to operate on objects? The answer is that we can define new behaviors for these operators. These operations will be member functions of the Position class. Using operators or functions in different ways, depending on what they are operating on, is called polymorphism (one thing with several distinct forms). When an existing operator, such as + or =, is given the capability to operate on a new data type, it is said to be overloaded. Overloading is a kind of polymorphism; it is also an important feature of OOP.

A Simple Class

```
#include <iostream>
using namespace std;
class smallobj    //define a class
{
private:
int somedata; //class data
public:
void setdata(int d) //member function to set data
{ somedata = d; }
void showdata() //member function to display data
{ cout << “Data is “ << somedata << endl; }
};
```

A Simple Class

```
int main()
{
smallobj s1, s2;          //define two objects of class smallobj
s1.setdata(1066);     //call member function to set data
s2.setdata(1776);
s1.showdata();        //call member function to display data
s2.showdata();
return 0;
}
```

Classes and Objects

- an object has the same relationship to a class that a variable has to a data type. An object is said to be an instance of a class. In SMALLOBJ, the class—whose name is smallobj—is defined in the first part of the program. Later, in main(), we define two objects—s1 and s2—that are instances of that class.
- Each of the two objects is given a value, and each displays its value. Here's the output of the program:
- Data is 1066 object s1 displayed this
- Data is 1776 object s2 displayed this

Defining the Class

- Here's the definition (sometimes called a specifier) for the class smallobj, copied from the SMALLOBJ listing:

```
class smallobj //define a class
{
private:
int somedata; //class data
public:
void setdata(int d) //member function to set data
{ somedata = d; }
void showdata() //member function to display data
{ cout << "\nData is " << somedata; }
};
```

- The definition starts with the keyword class, followed by the class name—smallobj in this example. Like a structure, the body of the class is delimited by braces and terminated by a semicolon.

private and public

- The body of the class contains two unfamiliar keywords: private and public. What is their purpose?
- A key feature of object-oriented programming is data hiding. This term does not refer to the activities of particularly paranoid programmers; rather it means that data is concealed within a class so that it cannot be accessed mistakenly by functions outside the class. The primary mechanism for hiding data is to put it in a class and make it private. Private data or functions can only be accessed from within the class. Public data or functions, on the other hand, are accessible from outside the class.

Class Data

- The smallobj class contains one data item: somedata, which is of type int. The data items within a class are called data members (or sometimes member data). There can be any number of data members in a class, just as there can be any number of data items in a structure. The data member somedata follows the keyword private, so it can be accessed from within the class, but not from outside.

Member Functions

- Member functions are functions that are included within a class. There are two member functions in smallobj: setdata() and showdata().

```
void setdata(int d)
{
somedata = d;
}
```

- And

```
void showdata()
{
cout << "\nData is " << somedata;
}
```

- Because setdata() and showdata() follow the keyword public, they can be accessed from outside the class.

Defining Objects

- The first statement in main()
- smallobj s1, s2;
- defines two objects, s1 and s2, of class smallobj. Remember that the definition of the class smallobj does not create any objects. It only describes how they will look when they are created, just as a structure definition describes how a structure will look but doesn't create any structure variables. It is objects that participate in program operations. Defining an object is similar to defining a variable of any data type: Space is set aside for it in memory. Defining objects in this way means creating them. This is also called instantiating them. The term instantiating arises because an instance of the class is created. An object is an instance (that is, a specific example) of a class. Objects are sometimes called instance variables.

Calling Member Functions

- The next two statements in main() call the member function setdata():

```
s1.setdata(1066);
s2.setdata(1776);
```

- These statements don't look like normal function calls. Why are the object names s1 and s2 connected to the function names with a period? This strange syntax is used to call a member function that is associated with a specific object. Because setdata() is a member function of the smallobj class, it must always be called in connection with an object of this class. It doesn't make sense to say setdata(1066);

- Some object-oriented languages refer to calls to member functions as messages.

Example2

```
#include "stdafx.h"
#include <iostream.h>
class Point {
int xVal, yVal;
public:
void SetPt (int, int);
void OffsetPt (int, int);
};
void Point::SetPt (int x, int y)
{
xVal = x;
yVal = y;
cout<< "x="<<x <<"  "<<
    "y="<<y<<endl;
}
```

```
void Point::OffsetPt (int x, int y)
{
xVal += x;
yVal += y;
cout<< "x="<<xVal <<"  "<<
    "y="<<yVal<<endl;;
}
int main(int argc, char* argv[])
{

Point pt; // pt is an object of class Point
pt.SetPt(10,20); // pt is set to (10,20)
pt.OffsetPt(2,2); // pt becomes (12,22)
    return 0;
}
```

OOP
Constructors & Destructors
Lecture 6

Example1

```
#include <iostream>
using namespace std;
class part //define class
{
private:
int modelnumber; //ID number of widget
int partnumber; //ID number of widget
    part
float cost; //cost of part
public:
void setpart(int mn, int pn, float c) //set
    data
{
modelnumber = mn;
partnumber = pn;
cost = c;
}
void showpart() //display data
{
cout << “Model “ << modelnumber;
cout << “, part “ << partnumber;
cout << “, costs $” << cost << endl;
}
};
int main()
{
part part1; //define object
// of class part
part1.setpart(6244, 373, 217.55F); //call
     member function
part1.showpart(); //call member function
return 0;
}
```

Example2

```
#include <iostream>
using namespace std;
class Distance //English Distance
    class
{
private:
int feet;
float inches;
public:
void setdist(int ft, float in) //set
    Distance to args
{ feet = ft; inches = in; }
void getdist() //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >>
    inches;
}
void showdist() //display distance
{ cout << feet << "\'-" << inches <<
"\""; }
};
int main()
{
Distance dist1, dist2; //define two
    lengths
dist1.setdist(11, 6.25); //set dist1
dist2.getdist(); //get dist2 from user
//display lengths
cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << endl;
return 0;
}
```

Constructor Functions

- Constructor is a special member functions that is used to automatically initialize objects of a class. A constructor always has the same name as the class itself. For example,

```
class Point {
int xVal, yVal;
public:
Point (int x,int y) {xVal = x; yVal = y;} // constructor
void OffsetPt (int,int);
};
```

- is an alternative definition of the Point class, where SetPt has been replaced by a constructor.
- Now we can define objects of type Point and initialize them at once. This is in fact compulsory for classes that contain constructors that require arguments:

```
Point pt1 = Point(10,20);
Point pt2; // illegal!
```

Example 3

```
#include <iostream>
using namespace std;

class Counter
{
private:
int count; //count
public:
Counter() : count(0) //constructor
{ /*empty body*/ }
void inc_count() //increment
    count
{ count++; }
int get_count() //return count
{ return count; }
};
```

```
int main()
{
Counter c1, c2; //define and
    initialize
cout << "\nc1=" <<
    c1.get_count(); //display
cout << "\nc2=" <<
    c2.get_count();
c1.inc_count(); //increment c1
c2.inc_count(); //increment c2
c2.inc_count(); //increment c2
cout << "\nc1="<< c1.get_count();
    //display again
cout << "\nc2=" <<
    c2.get_count();
cout << endl;
return 0;
}
```

Output:

C1=0
C2=0
C1=1
C2=2

Automatic Initialization

- When an object of type Counter is first created, we want its count to be initialized to 0. After all, most counts start at 0. We could provide a set_count() function to do this and call it with an argument of 0, or we could provide a zero_count() function, which would always set count to 0. However, such functions would need to be executed every time we created a Counter object.

```
Counter c1; //every time we do this,
c1.zero_count(); //we must do this too
```

- This is mistake prone, because the programmer may forget to initialize the object after creating it. It's more reliable and convenient, especially when there are a great many objects of a given class, to cause each object to initialize itself when it's created. In the Counter class, the constructor Counter() does this. This function is called automatically whenever a new object of type Counter is created. Thus in main() the statement

```
Counter c1, c2;
```

- creates two objects of type Counter. As each is created, its constructor, Counter(), is executed. This function sets the count variable to 0. So the effect of this single statement is to not only create two objects, but also to initialize their count variables to 0.

Aspects of constructor functions

- There are some aspects of constructor functions:
- First, they have exactly the same name as the class of which they are members. This is one way the compiler knows they are constructors.
- Second, no return type is used for constructors. the constructor is called automatically by the system, there's no program for it to return anything to. This is the second way the compiler knows they are constructors.

Constructor for Example2 objects as function Arguments

```
#include <iostream>
using namespace std;
class Distance //English Distance class
{
private:
int feet;
float inches;
public: //constructor (no args)
Distance() : feet(0), inches(0.0)
{}
//constructor (two args)
Distance(int ft, float in) : feet(ft),
    inches(in)
{}
void getdist() //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() //display distance
{ cout << feet << "\'-" << inches <<
    '\"'; }
void add_dist( Distance, Distance );
//declaration
};
```

Write add_dist function.

```
int main()
{
Distance dist1, dist3; //define two lengths
Distance dist2(11, 6.25); //define and
initialize dist2
dist1.getdist(); //get dist1 from user
dist3.add_dist(dist1, dist2); //dist3 = dist1
+ dist2
//display all lengths
cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << "\ndist3 = "; dist3.showdist();
cout << endl;
return 0;
}
```

Destructors

- We’ve seen that a special member function—the constructor—is called automatically when an object is first created. You might guess that another function is called automatically when an object is destroyed. This is indeed the case. Such a function is called a destructor. A destructor has the same name as the constructor (which is the same as the class name) but is preceded by a tilde:

```
class Foo
{
private:
int data;
public:
Foo() : data(0) //constructor (same name as class)
{ }
~Foo() //destructor (same name with tilde)
{ }
};
```

Aspects of Destructors

- destructors do not have a return value.
- They also take no arguments (the assumption being that there's only one way to destroy an object).

- The most common use of destructors is to deallocate memory that was allocated for the object by the constructor.

Classes, Objects, and Memory

Lecture 6

Objects as Function Arguments

```
#include <iostream>
class Distance //English Distance class
{
private:
int feet;
float inches;
public: //constructor (no args)
Distance() : feet(0), inches(0.0)
{ }
//constructor (two args)
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist() //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() //display distance
{ cout << feet << "\'-" << inches << "\""; }
void add_dist( Distance, Distance ); //declaration
};
//add lengths d2 and d3
void Distance::add_dist(Distance d2, Distance d3)
{ inches = d2.inches + d3.inches; //add the inches
feet = 0; //(for possible carry)
if(inches >= 12.0) //if total exceeds 12.0,
{ //then decrease inches
inches -= 12.0; //by 12.0 and
feet++; //increase feet
} //by 1
feet += d2.feet + d3.feet; //add the feet
}
int main()
{
Distance dist1, dist3; //define two lengths
Distance dist2(11, 6.25); //define and initialize dist2
dist1.getdist(); //get dist1 from user
dist3.add_dist(dist1, dist2); //dist3 = dist1 + dist2
//display all lengths
cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << "\ndist3 = "; dist3.showdist();
cout << endl;
    return 0;
}
```

Returning Objects from Functions

we'll see an example of a function that returns an object. We'll modify the add_dis function in the above program:

```
Distance Distance::add_dist(Distance d2)
{
Distance temp; //temporary variable
temp.inches = inches + d2.inches; //add the inches
if(temp.inches >= 12.0) //if total exceeds 12.0,
{ //then decrease inches
temp.inches -= 12.0; //by 12.0 and
temp.feet = 1; //increase feet
} //by 1
temp.feet += feet + d2.feet; //add the feet
return temp;
}
```

The calling of add_dis in the main program will be as follows:

```
d3= d1. add_dis(d2);
```

Member functions and Data in Memory

- each object created from a class contains separate copies of that class's data and member functions.
- all the objects in a given class use the same member functions. The member functions are created and placed in memory only once—when they are defined in the class definition. There is no point in duplicating all the member functions in a class every time you create another object of that class, since the functions for each object are identical.
- The data items, however, will hold different values, so there must be a separate instance of each data item for each object. Data is therefore placed in memory when each object is defined, so there is a separate set of data for each object.

Static Class Data

- If a data item in a class is declared as static, only one such item is created for the entire class, no matter how many objects there are. A static data item is useful when all objects of the same class must share a common item of information. A member variable defined as static has characteristics similar to a normal static variable: It is visible only within the class, but its lifetime is the entire program. It continues to exist even if there are no objects of the class.
- However, while a normal static variable is used to retain information between calls to a function, static class member data is used to share information among the objects of a class.
- As an example, suppose an object needed to know how many other objects of its class were in the program. All the objects would have access to this variable.

Example of static variable

```
#include <iostream.h>
class ooc
{
    private:
            static int count;
    public:
            ooc()
            {count++;};
            void show()
            {
            cout << "no. of objects: ";
            cout << count;
            }
};
int ooc::count=0;

int main()
{
    ooc o1, o2, o3;
    o2.show();
    return 0;
}
```

Output :

no. of objects: 3

Separate Declaration and Definition

- Static member data requires an unusual format. Ordinary variables are usually declared (the compiler is told about their name and type) and defined (the compiler sets aside memory to hold the variable) in the same statement. Static member data, on the other hand, requires two separate statements. The variable's declaration appears in the class definition, but the variable is actually defined outside the class, in much the same way as a global variable.

- static int count; (declaration)

- int ooc::count; (definition)

const and Classes

- we can introduce some uses of const: on member functions, on member function arguments, and on objects.

const Member Functions

- A const member function guarantees that it will never modify any of its class's member data.

```
class aClass
{
    private:
            int alpha;
    public:
            void nonFunc() //non-const member function
            { alpha = 99; } //OK
            void conFunc() const //const member function
            { alpha = 99; } //ERROR: can't modify a member
};
```

- The non-const function nonFunc() can modify member data alpha, but the constant function conFunc() can't. If it tries to, a compiler error results.

const Objects

- we can apply const to objects of classes. When an object is declared as const, you can't modify it. It follows that you can use only const member functions with it, because they're the only ones that guarantee not to modify it.

Example

```
#include <iostream>
class Distance //English Distance class
{
private:
int feet;
float inches;
public: //2-arg constructor
Distance(int ft, float in) : feet(ft), inches(in)
{ }

void getdist() //user input; non-const func
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() const //display distance; const
    func
{ cout << feet << "\'-" << inches << '\"'; }
};
```

```
int main()
{
const Distance football(300, 0);
// football.getdist(); //ERROR: getdist()
    not const
cout << "football = ";
football.showdist(); //OK
cout << endl;
return 0;
};
```

Friend Functions & Friend Classes

Lecture 8

Class Object Members

- A data member of a class may be of a user-defined type, that is, an object of another class. For example, a D3 class may be defined using 2 data members. One of them is of type int and the second one of type point which is a user defined data type.
- The constructor for D3 should also initialize the data members of the class of the both types.

Constructor of Class Object Members

```
#include <iostream.h>
class point
{
    int xVal;
    int yVal;
    public:
    point(int x, int y): xVal(x), yVal(y)
    {}
};
class D3
{
private:
int z;
point td;
public:
    D3(int x, int y, int zz): td(x,y), z(zz)
    {}
};
```

```
int main()
{
D3 d1(4,5,6), d2(3,2,8);
cout << endl;
    return 0;
}
```

Friend Functions

- The concepts of encapsulation and data hiding dictate that nonmember functions should not be able to access an object's private or protected data. The policy is, if you're not a member, you can't get in. However, there are situations where such rigid discrimination leads to considerable inconvenience.
- Imagine that you want a function to operate on objects of two different classes. Perhaps the function will take objects of the two classes as arguments, and operate on their private data. In this situation there's nothing like a friend function. Here's a simple example, FRIEND, that shows how friend functions can act as a bridge between two classes:

Friend Function Example

```
#include <iostream>
using namespace std;

class beta; //needed for
    frifunc declaration
class alpha
{
private:
int data;
public:
alpha() : data(3) { }
friend int frifunc(alpha,
    beta); //friend function
};
class beta
{
private:
int data;
public:
beta() : data(7) { }
friend int frifunc(alpha,
    beta); //friend
    function
};
int frifunc(alpha a, beta
    b) //function
    definition
{
return( a.data + b.data
    );
}

int main()
{
alpha aa;
beta bb;
cout << frifunc(aa, bb)
    << endl; //call the
    function
return 0;
}
```

Friend Function Example- cont.

- In this program, the two classes are alpha and beta. The constructors in these classes initialize their single data items to fixed values (3 in alpha and 7 in beta).
- We want the function frifunc() to have access to both of these private data members, so we make it a friend function. It's declared with the friend keyword in both classes:

  friend int frifunc(alpha, beta);

- This declaration can be placed anywhere in the class; it doesn't matter whether it goes in the public or the private section.

- Remember that a class can't be referred to until it has been declared. Class beta is referred to in the declaration of the function frifunc() in class alpha, so beta must be declared before alpha. Hence the declaration

  class beta;
  at the beginning of the program.

Friend Function Example 2

```
#include <iostream.h>
class D3;
class point {
    int xVal;
    int yVal;
    public:
    point(int x, int y): xVal(x), yVal(y)
    {}
    friend void func1(point a, D3 b); };
class D3 {
private:
int zVal;
//point td;
public:
    D3(int z): zVal(z) {}
    friend void func1(point a, D3 b); };
```

```
void func1(point a, D3 b) {
cout << "\nxVal=" << a.xVal; cout <<
     "\nyVal=" << a.yVal; cout <<
     "\nzVal=" << b.zVal; }
int main()
{
D3 d1(6);
point d2(3,2);
func1(d2,d1);
cout << endl;
     return 0;
}
```

English Distance Example

```
#include <iostream.h>
class Distance //English Distance class
{
private:
int feet;
float inches;
public:
Distance() : feet(0), inches(0.0)
{ }
//constructor (two args)
Distance(int ft, float in) : feet(ft),
    inches(in)
{ }
void showdist() //display distance
{ cout << feet << "\'-" << inches << "\""; }
friend float square(Distance); //friend
    function
};
float square(Distance d) //return square of
{ //this Distance
    float fltfeet = d.feet + d.inches/12; //convert
    to float
float feetsqrd = fltfeet * fltfeet; //find the square
return feetsqrd; //return square feet
}

int main()
{
Distance dist(3, 6.0);
float sqft;
sqft = square(dist); //return square of dist
cout << "\nDistance = "; dist.showdist();
cout << "\nSquare = " << sqft << " square
    feet\n";
return 0;
}
```

Output :
12.25

Friend Classes

- The member functions of a class can all be made friends at the same time when you make the entire class a friend.

- The extreme case of having all member functions of a class A as friends of another class B can be expressed in an abbreviated form:

```
class A;
class B
  {
  .//
  friend class A; // abbreviated form
  };
```

Friend Class Example1

```
#include <iostream.h>
class alpha
{
private:
int data1;
public:
alpha() : data1(99) { }
friend class beta; //beta is a friend class
};
class beta
{               //all member functions can
public:  //access private alpha data
void func1(alpha a) { cout << "\ndata1=" <<
    a.data1; }
void func2(alpha a) { cout << "\ndata1=" <<
    a.data1; }
};
```

```
int main()
{
alpha a;
beta b;
b.func1(a);
b.func2(a);
cout << endl;
return 0;
}
```

Friend Class Example1 – cont.

- In class alpha the entire class beta is proclaimed a friend. Now all the member functions of beta can access the private data of alpha (in this program, the single data item data1).
- Note that in the friend declaration we specify that beta is a class using the class keyword:

  friend class beta;

- We could have also declared beta to be a class before the alpha class specifier, as in previous examples

- class beta;

- and then, within alpha, referred to beta without the class keyword: friend beta;

Example2 of Friend Class

```
#include <iostream.h>
class point
{
    int xVal;
    int yVal;
    public:
    point(int x, int y): xVal(x), yVal(y)
    {}
    friend class D3;
};
class D3
{
private:
int zVal;
public:
D3(int zz): zVal(zz)
{}
void func1(point a)
{ cout << "\nxVal=" << a.xVal; cout <<
    "\nyVal=" << a.yVal; cout << "\nzVal="
    << zVal; }
};
```

```
int main()
{
D3 d1(6);

point d2(3,2);

d1.func1(d2);
cout << endl;
    return 0;
}
```

Output:

3
2
6

Class object member example modification

```
#include <iostream.h>
class point
{
    int xVal;
    int yVal;
    public:
    point(int x, int y): xVal(x), yVal(y)
    {}
    friend class D3;
};

class D3
{
private:
int zVal;
point td;
public:
D3(int x, int y, int z): td(x,y), zVal(z) {}
void func()
{  cout << td.xVal << td.yVal  << zVal;}
};
```

```
int main()
{
D3 d1(4,5,6), d2(3,2,8);
d1.func();
Cout << endl;
d2.func();
cout << endl;
    return 0;
}
```

ARRAYS AND STRINGS
LECTURE 9

Object Arrays

- An array of a user-defined type is defined and used much in the same way as an array of a built-in type. For example, a pentagon can be defined as an array of 5 points:

  point pentagon[5];

- This definition assumes that point has an 'argument-less' constructor (i.e., one which can be invoked without arguments). The constructor is applied to each element of the array.
- The array can also be initialized using a normal array initializer. Each entry in the initialization list would invoke the constructor with the desired arguments. When the initializer has less entries than the array dimension, the remaining elements are initialized by the argument-less constructor. For example,

  point pentagon[5] = { Point(10,20), Point(10,30), Point(20,30), Point(30,20) };

- initializes the first four elements of pentagon to explicit points, and the last element is initialized to (0,0).

Arrays as Class Member Data

```
#include <iostream.h>
class arr
{
private:
enum { MAX = 3 }; //constant definition
int ar[MAX]; // array of integers
public:
arr()
    { int i;
    for (i=0;i<MAX;i++)
    ar[i]=0;}
arr(int x, int y, int z)
    { ar[0]=x; ar[1]=y; ar[2]=z;};
void get() //put number on stack
{
    int x, i;
    for (i=0;i<MAX;i++)
    { cin >> x;
    ar[i] = x;  };
}
void show()
{
    int  i;
    for (i=0;i<MAX; i++)
    cout << ar[i];
}
}; // end class


int main()
{
arr a1;
arr a2(3,4,6);
a1.show();
a2.show();
a1.get();
a1.show();
return 0;
}
```

Arrays as Class Member Data

- This definition of MAX is unusual. In keeping with the philosophy of encapsulation, it's preferable to define constants that will be used entirely within a class, as MAX is here, within the class. Thus the use of global const variables for this purpose is non-optimal.

- To place items on the Array , the function get() is called.
- To print the items of the array , the function show() is called.

- The main() program creates and constructs objects ,a1 and a2 of the class arr.
- It reads 3 items onto the array, and then displays them.

Arrays of Objects

- We can also create an array of objects. We'll look at a situations: an array of English distances.

Arrays of Objects- Example

```
#include <iostream.h>
class Distance //English Distance class
{
private:
int feet;
float inches;
public:
void getdist() //get length from user
{
cout << "\n Enter feet: "; cin >> feet;
cout << " Enter inches: "; cin >> inches;
}
void showdist() const //display distance
{ cout << feet << "\"-" << inches << "\""; }
};
```

```
int main()
{
Distance dist[10]; //array of distances
int n=0; //count the entries
char ans; //user response (‘y’ or ‘n’)
cout << endl;

do {                    //get distances from user
cout << "Enter distance number " << n+1;
dist[n++].getdist(); //store distance in array
cout << "Enter another (y/n)?: ";
cin >> ans;
} while( ans != 'n' ); //quit if user types ‘n’
for(int j=0; j<n; j++) //display all di"“;
dist[j].showdist();
cout << endl;
return 0;
}
```

Output

```
Enter distance number 1
Enter feet: 5
Enter inches: 4
Enter another (y/n)? y
Enter distance number 2
Enter feet: 6
Enter inches: 2.5
Enter another (y/n)? y
Enter distance number 3
Enter feet: 5
Enter inches: 10.75
Enter another (y/n)? N

Distance number 1 is 5'-4"
Distance number 2 is 6'-2.5"
Distance number 3 is 5'-10.75"
```

Accessing Objects in an Array

- in the main() program we define an array of such objects:

  Distance dist[MAX];
- Here the data type of the dist array is Distance, and it has MAX elements.
- A class member function that is an array element is accessed similarly to a structure member that is an array element.
- Here's how the showdist() member function of the jth element of the array dist is invoked:

  dist[j].showdist();

- A member function of an object that is an array element is accessed using the dot operator: The array name followed by the index in brackets is joined, using the dot operator, to the member function name followed by parentheses. This is similar to accessing a structure (or class) data member, except that the function name and parentheses are used instead of the data name.
- Notice that when we call the getdist() member function to put a distance into the array, we take the opportunity to increment the array index n:

- dist[n++].getdist();

Strings as Class Members

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

class part
{
private:
char partname[30]; //name of widget part
int partnumber; //ID number of widget part
double cost; //cost of part
public:
void setpart(char pname[], int pn, double c)
{
strcpy(partname, pname);
partnumber = pn;
cost = c;
}
void showpart() //display data
{
cout << "\nName=" << partname;
cout << ", number=" << partnumber;
cout << ", cost=$" << cost;
}
};
int main()
{
part part1, part2;
part1.setpart("ABC", 4473, 217.55);
part2.setpart("XYZ", 9924, 419.25);
cout << "\nFirst part: "; part1.showpart();
cout << "\nSecond part: "; part2.showpart();
cout << endl;
      return 0;
}
```

Strings as Class Members – cont.

- This program defines two objects of class part and gives them values with the setpart() member function. Then it displays them with the showpart() member function. Here's the output:
- First part:
- Name= ABC, number=4473, cost=$217.55
- Second part:
- Name= XYZ, number=9924, cost=$419.25

- In the setpart() member function, we use the strcpy() string library function to copy the string from the argument pname to the class data member partname. Thus this function serves the same purpose with string variables that an assignment statement does with simple variables. (A similar function, strncpy(), takes a third argument, which is the maximum number of characters it will copy. This can help prevent overrunning the array.)
- Besides those we've seen, there are library functions to add a string to another, compare strings, search for specific characters in strings, and perform many other actions.

References, Pointers in OOP

Lecture 10

Implicit Member Argument

- When a class member function is called, it receives an implicit argument which denotes the particular object (of the class) for which the function is invoked. For example, in:

```
Point pt(10,20);
pt.OffsetPt(2,2);
```

- pt is an implicit argument to OffsetPt. Within the body of the member function, one can refer to this implicit argument explicitly as this, which denotes a pointer to the object for which the member is invoked. Using this, OffsetPt can be rewritten as:

```
Point::OffsetPt (int x, int y)
{
this->xVal += x; // equivalent to: xVal += x;
this->yVal += y; // equivalent to: yVal += y;
}
```

this pointer

- The this pointer can be used for referring to member functions in exactly the same way as it is used for data members. It is important to bear in mind, however, that this is defined for use within member functions of a class only. In particular, it is undefined for global functions (including global friend functions).

(this) is a pointer named which points to the object itself. Thus any member function can find out the address of the object of which it is a member.

```
#include <iostream.h>

class where
{
private:
char charray[10];
public:
void reveal()
{ cout << "\nMy object's
   address is " << this; }
};
```

```
int main()
{
where w1, w2, w3;
w1.reveal();
w2.reveal();
w3.reveal();
cout << endl;
return 0;
}
```

Accessing Member Data with this

```
#include <iostream.h>
class what
{
private:
int alpha;
public:
void tester()
{
this->alpha = 11;
cout << this->alpha;
}
};
```

```
int main()
{
what w;
w.tester();
cout << endl;
return 0;
}
```

References Members

- A class data member may defined as reference. For example:

```
class Image {
    int width;
    int height;
    int &widthRef;
    ...//
};
```

- A data member reference cannot be initialized using the same syntax as for other references:

```
class Image {
int width;
int height;
int &widthRef = width; // illegal!
...//
};
```

The correct way to initialize a data member reference is as the following example

Example

```
# include <iostream.h>
class image {
private:
int width;
int height;
int &widthRef;
public:
image (int w, int h): width(w), widthRef(width), height(h)
            {}
void show ()
{cout << width << "  " << widthRef << "  " <<height << endl;}
};
int main()
{
image o(10, 4);
o.show();
return 0;
}
```

Memory Management: new and delete

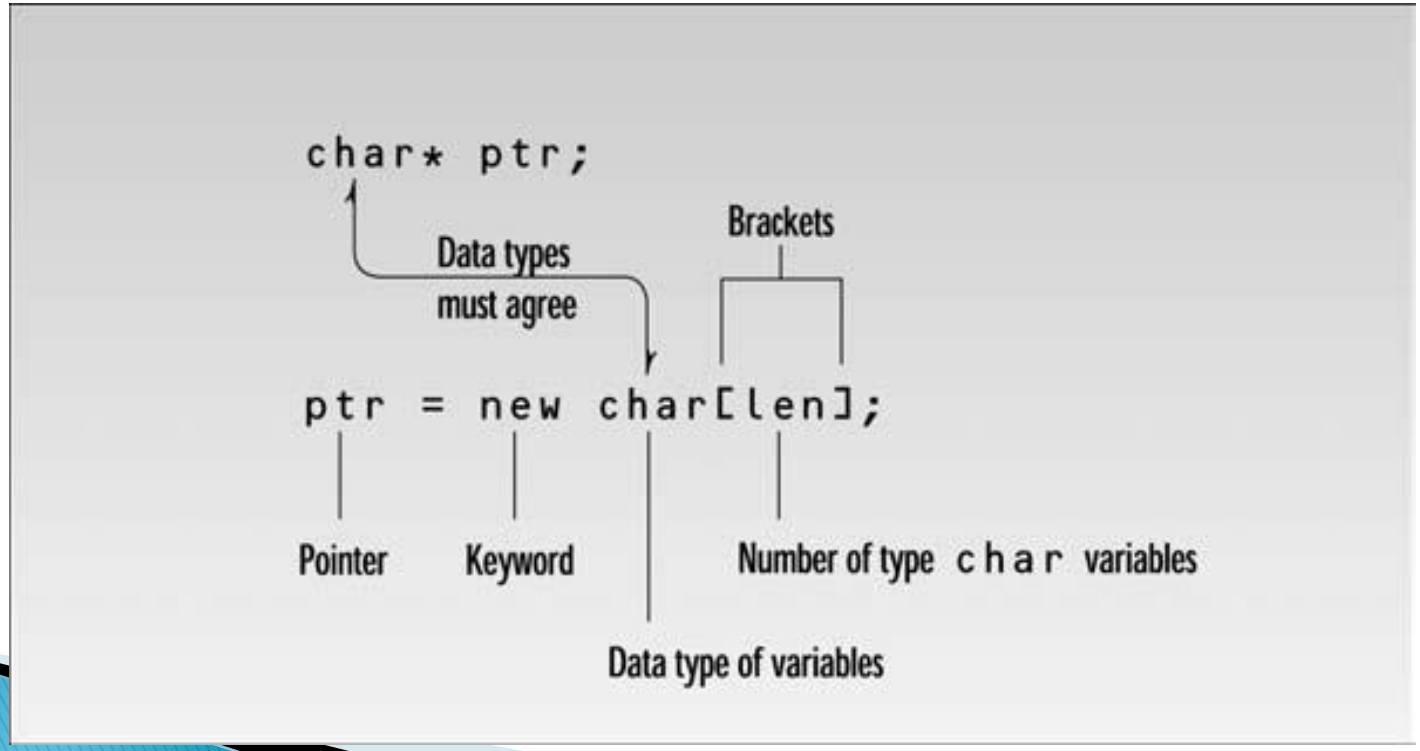
- We've seen many examples where arrays are used to set aside memory. The statement
- int arr1[100];
- reserves memory for 100 integers. Arrays are a useful approach to data storage, but they have a serious drawback: We must know at the time we write the program how big the array will be. We can't wait until the program is running to specify the array size. The following approach won't work:

```
cin >> size; // get size from user
int arr[size]; // error; array size must be a constant
```

- The compiler requires the array size to be a constant. But in many situations we don't know how much memory we need until runtime. We might want to store a string that was typed in by the user, for example. In this situation we can define an array sized to hold the largest string we expect, but this wastes memory.

The new Operator

- C++ provides a different approach to obtaining blocks of memory: the new operator. This versatile operator obtains memory from the operating system and returns a pointer to its starting point. The following example shows how new is used:



Example

```
#include <iostream.h>
#include <cstring> //for strlen
int main()
{
char* str = "ABC";
int len = strlen(str); //get length of str
char* ptr; //make a pointer to char
ptr = new char[len+1]; //set aside memory: string + '\0'
strcpy(ptr, str); //copy str to new memory area ptr
cout << "ptr=" << ptr << endl;
delete[] ptr; //release ptr's memory
return 0;
}
```

The delete Operator

- If your program reserves many chunks of memory using new, eventually all the available memory will be reserved and the system will crash. To ensure safe and efficient use of memory, the new operator is matched by a corresponding delete operator that returns memory to the operating system. In the above example the statement

  delete[] ptr;

- returns to the system whatever memory was pointed to by ptr. Actually, there is no need for this operator in the above example, since memory is automatically returned when the program terminates. However, suppose you use new in a function. If the function uses a local variable as a pointer to this memory, the pointer will be destroyed when the function terminates, but the memory will be left as an orphan, taking up space that is inaccessible to the rest of the program.

The delete Operator

- The brackets following delete indicate that we're deleting an array. If you create a single object with new, you don't need the brackets when you delete it.

```
..
delete ptr; // no brackets following delete
```

new and delete operators in OOP

- The new operator often appears in constructors. The following example, we'll show the String class, Which uses new to obtain exactly the right amount of memory. delete operators appears in destructors to release the reserved memory space .

Example

```
#include <iostream.h>
#include <cstring> //for strcpy(), etc

class String //user-defined string type
{
private:
char* str; //pointer to string
public:
String(char* s) //constructor, one arg
{
int length = strlen(s); //length of string
    argument
str = new char[length+1]; //get
    memory
strcpy(str, s); //copy argument to it
}
~String() //destructor
{
cout << "Deleting str.\n";
delete[] str; //release memory
}
void display() //display the String
{
cout << str << endl;
}
};

int main()
{
String s1 = "Who knows nothing doubts
    nothing.";
cout << "s1="; //display string
s1.display();
return 0;
}
```

Pointers to Objects

- Pointers can point to objects as well as to simple data types and arrays. We've seen many examples of objects defined and given a name, in statements like
- Distance dist;
- where an object called dist is defined to be of the Distance class. Sometimes, however, we don't know, at the time that we write the program, how many objects we want to create. When this is the case we can use new to create objects while the program is running. As we've seen, new returns a pointer to an unnamed object.

Example

```
#include <iostream.h>
class Distance //English Distance class
{
private:
int feet;
float inches;
public:
void getdist() //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >>
    inches;
}
void showdist() //display distance
{ cout << feet << "\'-" << inches <<
    "\""; }
};
```

```
int main()
{
Distance dist; //define a named
    Distance object
dist.getdist(); //access object members
dist.showdist(); // with dot operator
Distance* distptr; //pointer to Distance
distptr = new Distance; //points to new
    Distance object
distptr->getdist(); //access object
    members
distptr->showdist(); // with -> operator
cout << endl;
return 0;
}
```

Referring to Members

- The dot operator requires the identifier on its left to be a variable. Since distptr is a pointer to a variable, we need another syntax. One approach is to dereference (get the contents of the variable pointed to by) the pointer:

- (*distptr).getdist(); // ok but inelegant

- However, this is slightly cumbersome because of the parentheses. (The parentheses are necessary because the dot operator (.) has higher precedence than the dereference operator (*). An equivalent but more concise approach is furnished by the membership-access operator, which consists of a hyphen and a greater-than sign:
- distptr->getdist(); // better approach
- As you can see in the above example, the -> operator works with pointers to objects in just the same way that the . operator works with objects.

An Array of Pointers to Objects

- A common programming construction is an array of pointers to objects. This arrangement allows easy access to a group of objects, and is more flexible than placing the objects themselves in an array.

Example

```
#include <iostream.h>
class person //class of persons
{
private:
char name[40]; //person's name
public:
void setName() //set the name
{
cout << "Enter name: ";
cin >> name;
}
void printName() //get the name
{
cout << "\n Name is: " << name;
}
};
int main()
{
person* persPtr[100]; //array of
    pointers to persons
int n = 0; //number of persons in array
char choice;
do //put persons in array
{
persPtr[n] = new person; //make new
    object
persPtr[n]->setName(); //set person's
    name
n++; //count new person
cout << "Enter another (y/n)? "; //enter
    another
cin >> choice; //person?
}
while( choice=='y' ); //quit on 'n'
for(int j=0; j<n; j++) //print names of
{ //all persons
cout << "\nPerson number " << j+1;
persPtr[j]->printName();
}
cout << endl;
return 0;
} //end main()
```

Accessing Member Functions

- We need to access the member functions setName() and printName() in the person objects pointed to by the pointers in the array persPtr. Each of the elements of the array persPtr is specified in array notation to be persPtr[j]. The elements are pointers to objects of type person. To access a member of an object using a pointer, we use the -> operator. Putting this all together, we have the following syntax for getname():

  persPtr[j]->getName()

- This executes the getname() function in the person object pointed to by element j of the persPtr array.