

C++

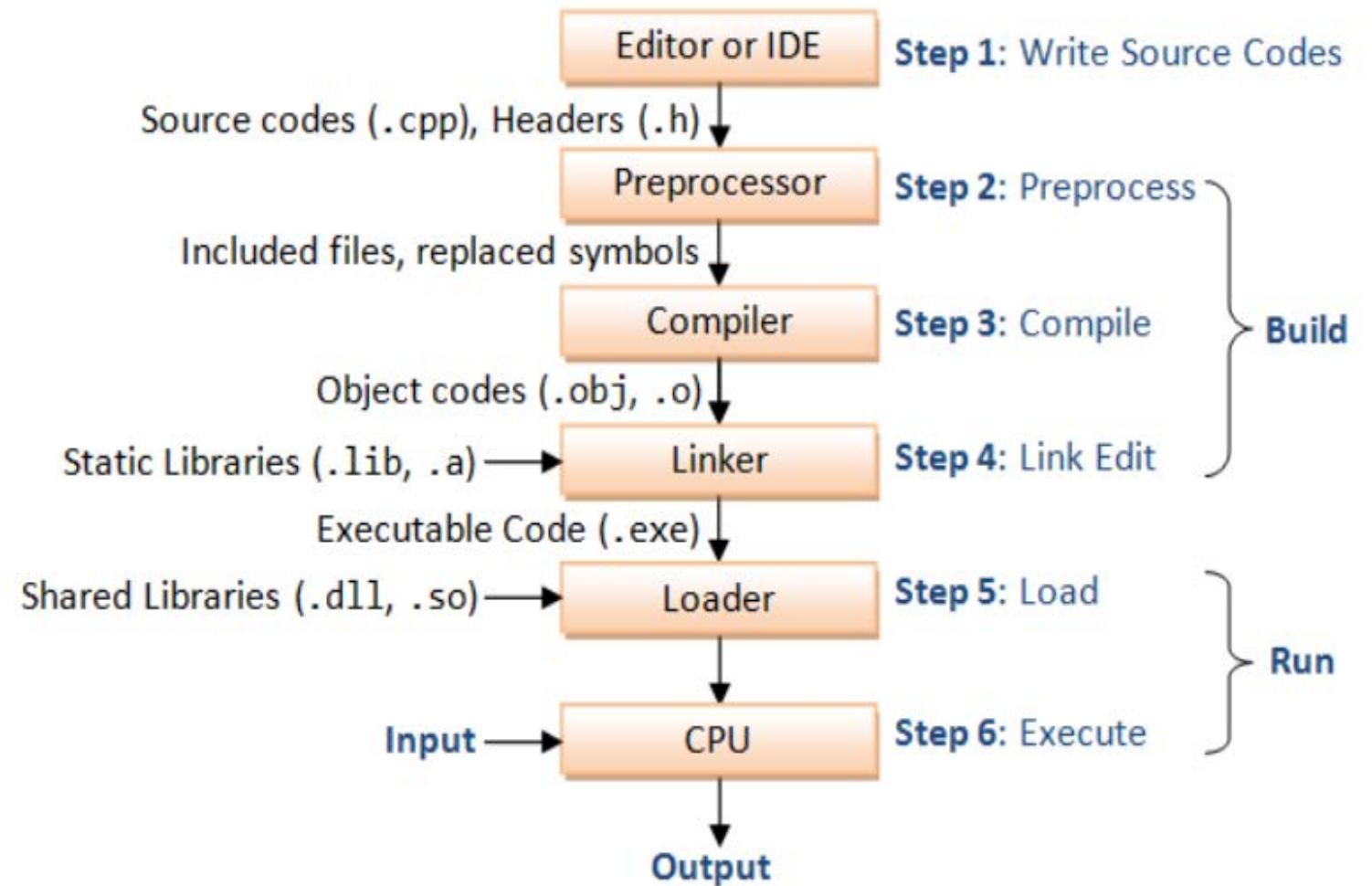
2019

- `/** First C++ program that says hello (hello.cpp) */`
- `#include <iostream> // Needed to perform IO operations`
- `using namespace std;`

```
int main() {                                // Program entry point
    cout << "hello, world" << endl; // Say Hello
    return 0;                            // Terminate main()
}
```

Hello

- The directive "`#include <iostream>`" tells the preprocessor to include the "iostream" header file to support input/output operations.
- The "`using namespace std;`" statement declares `std` as the default namespace used in this program. The names `cout` and `endl`, which is used in this program, belong to the `std` namespace.
- These two lines shall be present in all our programs.



- Step 1: Write the source codes (.cpp) and header files (.h).
- Step 2: Pre-process the source codes according to the preprocessor directives. Preprocessor directives begin with a hash sign (#), e.g., #include and #define.
- Step 3: Compile the pre-processed source codes into object codes (.obj, .o).
- Step 4: Link the compiled object codes with other object codes and the library object codes (.lib, .a) to produce the executable code (.exe).
- Step 5: Load the executable code into computer memory.
- Step 6: Run the executable code, with the input to produce the desired output

Template

- `/*`
- `* Comment to state the purpose of this program (filename.cpp)`
- `*/`

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

```
int main() {
```

```
// Your Programming statements HERE!
```

```
    return 0;
}
```

Output via "cout <<"

```
cout << "hello" << " world, " << "again!" << endl;
```

hello world, again!

```
cout << "hello," << endl << "one more time. " << endl << 5 << 4 << 3 << " " << 2.2 << " "  
<< 1.1 << endl;
```

hello,
one more time.

543 2.2 1.1

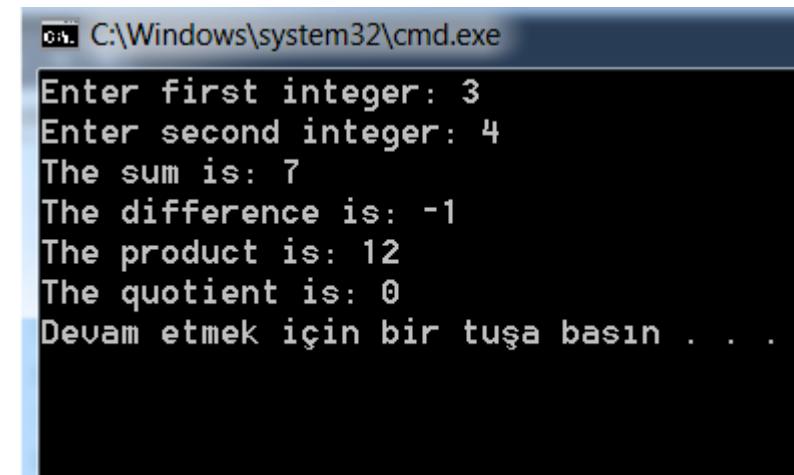
```
cout << "hello world, again!\n";  
cout << "\thello,\nnone\tmore\ttime.\n";
```

hello world, again!
hello,
one more time.

```
#include <iostream>
using namespace std;
int main() {
    int firstInt, secondInt, sum, difference, product,
        quotient;
    cout << "Enter first integer: " // Display a prompting
message
    cin >> firstInt; // Read input from keyboard (cin) into
firstInt
    cout << "Enter second integer: " //Display a prompting
message
    cin >> secondInt; // Read input into secondInt
    sum = firstInt + secondInt;
    difference = firstInt - secondInt;
    product = firstInt * secondInt;
    quotient = firstInt / secondInt;
```

Input via "cin >>"

```
cout << "The sum is: " << sum << endl;
cout << "The difference is: " << difference
<< endl;
cout << "The product is: " << product <<
endl;
cout << "The quotient is: " << quotient <<
endl;
return 0;
}
```



Reading multiple items in one **cin** statement

```
cout << "Enter two integers (separated by space): "; // Put out a prompting  
message  
cin >> firstInt >> secondInt; // Read two values into respective variables  
sum = firstInt + secondInt;  
cout << "The sum is: " << sum << endl;
```

What is a Variable?

NAME	VALUE	TYPE
number	123	int
sum	-456	int
pi	3.1416	double
average	-55.66	double

A variable has a name, stores a value of the declared type

// Syntax: Declare a variable of a type

var-type var-name;

int sum; // Example:

double radius;

// Declare multiple variables of the same type

var-type var-name-1, var-name-2,...;

int sum, difference, product, quotient; // Example:

double area, circumference;

// Declare a variable of a type and assign an initial value

var-type var-name = initial-value;

int sum = 0; // Example:

double pi = 3.14159265;

// Syntax: Declare multiple variables of the same type with initial values

var-type var-name-1 = initial-value-1, var-name-2 = initial-value-2,... ;

int firstNumber = 1, secondNumber = 2; // Example:

Basic Arithmetic Operations

Operator	Meaning	Example
+	Addition	$x + y$
-	Subtraction	$x - y$
*	Multiplication	$x * y$
/	Division	x / y
%	Modulus (Remainder)	$x \% y$
++	Increment by 1 (Unary)	<code>++x</code> or <code>x++</code>
--	Decrement by 1 (Unary)	<code>--x</code> or <code>x--</code>

Type `double` & Floating-Point Numbers

- Recall that a variable in C/C++ has a name and a type, and can hold a value of only that particular type.
- We have so far used a type called **int**. A `int` variable holds only integers (whole numbers), such as 123 and -456.
- In programming, real numbers such as 3.1416 and -55.66 are called floating-point numbers, and belong to a type called **double**.
- You can express **floating-point** numbers in fixed notation (e.g., 1.23, -4.5) or scientific notation (e.g., 1.2e3, -4E5.6) where e or E denote the exponent of base 10.

Mixing int and double, and Type Casting

- Although you can use a double to keep an integer value (e.g., double count = 5), you should use an int for integer. This is because int is far more efficient than double, in terms of **running times and memory requirement**.
- At times, you may need both int and double in your program. For example, keeping the sum from 1 to 100 (=5050) as an int, and their average 50.5 as a double. You need to be extremely careful when different types are mixed.
- It is important to note that:
 - Arithmetic operations ('+', '-', '*', '/') of **two int's produce an int**; while arithmetic operations of **two double's produce a double**. Hence, $1/2 \rightarrow 0$ (take note!) and $1.0/2.0 \rightarrow 0.5$.
 - Arithmetic operations of an **int and a double produce a double**. Hence, $1.0/2 \rightarrow 0.5$ and $1/2.0 \rightarrow 0.5$.
 - You can assign an integer value to a double variable. **The integer value will be converted to a double value automatically**, e.g., $3 \rightarrow 3.0$. For example,

```
int i = 3;  
double d;  
d = i;           // 3 → 3.0, d = 3.0  
d = 88;         // 88 → 88.0, d = 88.0
```

double nought = 0; // 0 → 0.0; there is a subtle difference between int of 0 and double of 0.0

However, if you assign a double value to an int variable, the fractional part will be lost. For example,

```
double d = 55.66;  
int i;  
i = d; // i = 55 (truncated)
```

Type Casting Operators

- If you are certain that you wish to carry out the type conversion, you could use the so-called type cast operator.
 - The type cast operation could take one of these forms in C++, which returns an equivalent value in the new-type specified.
-
- **new-type(expression); // C++ function cast notation**
 - **(new-type)expression; // C-language cast notation**

or example,

```
double d = 5.5;  
int i;  
i = int(d);      // int(d) -> int(5.5) -> 5 (assigned to i)  
i = int(3.1416); // int(3.1416) -> 3 (assigned to i)  
i = (int)3.1416; // same as above
```

Floating-point Literals

- A number with a decimal point, such as 55.66 and -33.44, **is treated as a double**, by default.
- You can also express them in **scientific notation**, e.g., 1.2e3, -5.5E-6, where e or E denotes the exponent in power of 10.
- You could precede the fractional part or exponent with a plus (+) or minus (-) sign. Exponent shall be an integer. There should be no space or other characters (e.g., space) in the number.
- You MUST use a suffix of 'f' or 'F' for float literals, e.g., -1.2345F. For example,
- **float average = 55.66; // Error! RHS is a double. Need suffix 'f' for float.**
- **float average = 55.66f;**

Mixed-Type Operations

Type	Example	Operation
int	2 + 3	int 2 + int 3 → int 5
double	2.2 + 3.3	double 2.2 + double 3.3 → double 5.5
mix	2 + 3.3	int 2 + double 3.3 → double 2.0 + double 3.3 → double 5.3
int	1 / 2	int 1 / int 2 → int 0
double	1.0 / 2.0	double 1.0 / double 2.0 → double 0.5
mix	1 / 2.0	int 1 / double 2.0 → double 1.0 + double 2.0 → double 0.5

Fundamental Types

Category	Type	Description	Bytes (Typical)	Minimum (Typical)	Maximum (Typical)
Integers	int (or signed int)	Signed integer (of at least 16 bits)	4 (2)	-2147483648	2147483647
	unsigned int	Unsigned integer (of at least 16 bits)	4 (2)	0	4294967295
	char	Character (can be either signed or unsigned depends on implementation)	1		
	signed char	Character or signed tiny integer (guarantee to be signed)	1	-128	127
	unsigned char	Character or unsigned tiny integer (guarantee to be unsigned)	1	0	255
	short (or short int) (or signed short) (or signed short int)	Short signed integer (of at least 16 bits)	2	-32768	32767
	unsigned short (or unsigned short int)	Unsigned short integer (of at least 16 bits)	2	0	65535
	long (or long int) (or signed long) (or signed long int)	Long signed integer (of at least 32 bits)	4 (8)	-2147483648	2147483647
	unsigned long (or unsigned long int)	Unsigned long integer (of at least 32 bits)	4 (8)	0	same as above
	long long (or long long int) (or signed long long) (or signed long long int) (C++11)	Very long signed integer (of at least 64 bits)	8	-2 ⁶³	2 ⁶³ -1
	unsigned long long (or unsigned long long int) (C++11)	Unsigned very long integer (of at least 64 bits)	8	0	2 ⁶⁴ -1
Real Numbers	float	Floating-point number, ≈7 digits (IEEE 754 single-precision floating point format)	4	3.4e38	3.4e-38

Real Numbers	<code>float</code>	Floating-point number, \approx 7 digits (IEEE 754 single-precision floating point format)	4	<code>3.4e38</code>	<code>3.4e-38</code>
	<code>double</code>	Double precision floating-point number, \approx 15 digits (IEEE 754 double-precision floating point format)	8	<code>1.7e308</code>	<code>1.7e-308</code>
	<code>long double</code>	Long double precision floating-point number, \approx 19 digits (IEEE 754 quadruple-precision floating point format)	12 (8)		
Boolean Numbers	<code>bool</code>	Boolean value of either true or false	1	<code>false (0)</code>	<code>true (1 or non-zero)</code>
Wide Characters	<code>wchar_t</code> <code>char16_t</code> (C++11) <code>char32_t</code> (C++11)	Wide (double-byte) character	2 (4)		

Character Literals and Escape Sequences

- In C++, characters are represented using 8-bit ASCII code, and can be treated as 8-bit signed integers in arithmetic operations.
- In other words, char and 8-bit signed integer are interchangeable. You can also assign an integer in the range of [-128, 127] to a char variable; and [0, 255] to an unsigned char.
- For example,

```
char letter = 'a';           // Same as 97
char anotherLetter = 98;      // Same as the letter 'b'
cout << letter << endl;      // 'a' printed
cout << anotherLetter << endl; // 'b' printed instead of the number
anotherLetter += 2;          // 100 or 'd'
cout << anotherLetter << endl; // 'd' printed
cout << (int)anotherLetter << endl; // 100 printed
```

Escape Sequence	Description	Hex (Decimal)
\n	New-line (or Line-feed)	0AH (10D)
\r	Carriage-return	0DH (13D)
\t	Tab	09H (9D)
\"	Double-quote (needed to include " in double-quoted string)	22H (34D)
\'	Single-quote	27H (39D)
\\	Back-slash (to resolve ambiguity)	5CH (92D)

String Literals

- A String literal is composed of zero or more characters surrounded by a pair of double quotes, e.g., "Hello, world!", "The sum is ", "". For example,

```
String directionMsg = "Turn Right";  
String greetingMsg = "Hello";  
String statusMsg = ""; // empty string
```

- String literals may contain escape sequences.
- Inside a String, you need to use \" for double-quote to distinguish it from the ending double-quote, e.g. "\"quoted\"". Single quote inside a String does not require escape sequence. For example,

```
cout << "Use \\" to place\n a \" within\t a\tstring" << endl;
```

- Use \" to place
- a " within a string

bool Literals

- There are only two bool literals, i.e., true and false. For example,

```
bool done = true;  
bool gameOver = false;  
int i;  
if (i == 9) { // returns either true or false  
    .....  
}
```

- In an expression, bool values and literals are converted to int 0 for false and 1 (or a non-zero value) for true.

Compound Assignment Operators

Operator	Usage	Description	Example
=	<code>var = expr</code>	Assign the value of the LHS to the variable at the RHS	<code>x = 5;</code>
+=	<code>var += expr</code>	same as <code>var = var + expr</code>	<code>x += 5; same as x = x + 5</code>
-=	<code>var -= expr</code>	same as <code>var = var - expr</code>	<code>x -= 5; same as x = x - 5</code>
*=	<code>var *= expr</code>	same as <code>var = var * expr</code>	<code>x *= 5; same as x = x * 5</code>
/=	<code>var /= expr</code>	same as <code>var = var / expr</code>	<code>x /= 5; same as x = x / 5</code>
%=	<code>var %= expr</code>	same as <code>var = var % expr</code>	<code>x %= 5; same as x = x % 5</code>

Increment/Decrement Operators

Operator	Example	Result
<code>++</code>	<code>x++; ++x</code>	Increment by 1, same as <code>x += 1</code>
<code>--</code>	<code>x--; --x</code>	Decrement by 1, same as <code>x -= 1</code>

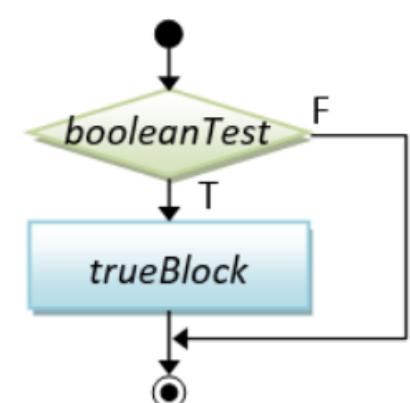
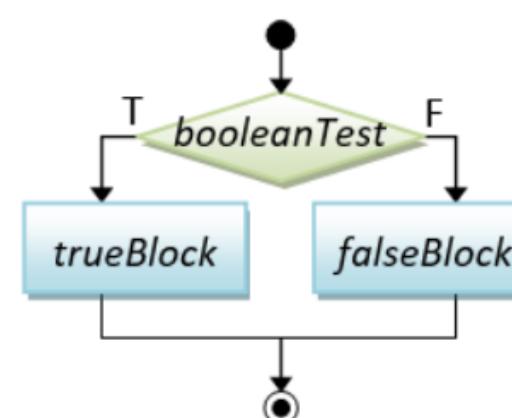
Operator	Description	Example	Result
<code>++var</code>	Pre-Increment Increment <code>var</code> , then use the new value of <code>var</code>	<code>y = ++x;</code>	same as <code>x=x+1; y=x;</code>
<code>var++</code>	Post-Increment Use the old value of <code>var</code> , then increment <code>var</code>	<code>y = x++;</code>	same as <code>oldX=x; x=x+1; y=oldX;</code>
<code>--var</code>	Pre-Decrement	<code>y = --x;</code>	same as <code>x=x-1; y=x;</code>
<code>var--</code>	Post-Decrement	<code>y = x--;</code>	same as <code>oldX=x; x=x-1; y=oldX;</code>

Relational and Logical Operators

Operator	Description	Usage	Example (x=5, y=8)
<code>==</code>	Equal to	<code>expr1 == expr2</code>	$(x == y) \rightarrow \text{false}$
<code>!=</code>	Not Equal to	<code>expr1 != expr2</code>	$(x != y) \rightarrow \text{true}$
<code>></code>	Greater than	<code>expr1 > expr2</code>	$(x > y) \rightarrow \text{false}$
<code>>=</code>	Greater than or equal to	<code>expr1 >= expr2</code>	$(x >= 5) \rightarrow \text{true}$
<code><</code>	Less than	<code>expr1 < expr2</code>	$(y < 8) \rightarrow \text{false}$
<code><=</code>	Less than or equal to	<code>expr1 <= expr2</code>	$(y <= 8) \rightarrow \text{true}$

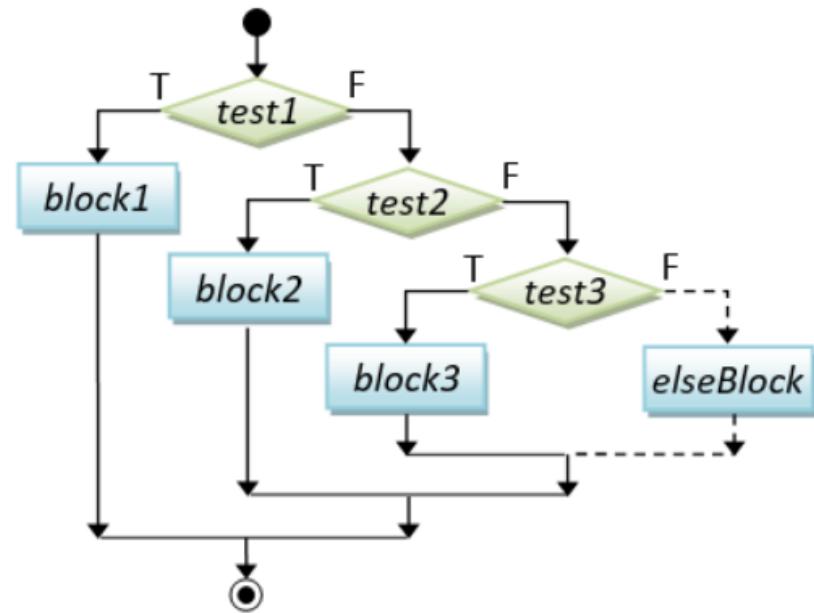
Operator	Description	Usage
<code>&&</code>	Logical AND	<code>expr1 && expr2</code>
<code> </code>	Logical OR	<code>expr1 expr2</code>
<code>!</code>	Logical NOT	<code>!expr</code>
<code>^</code>	Logical XOR	<code>expr1 ^ expr2</code>

Conditional (Decision) Flow Control

Syntax	Example	Flowchart
<pre>// if-then if (booleanExpression) { true-block ; }</pre>	<pre>if (mark >= 50) { cout << "Congratulation!" << endl; cout << "Keep it up!" << endl; }</pre>	
<pre>// if-then-else if (booleanExpression) { true-block ; } else { false-block ; }</pre>	<pre>if (mark >= 50) { cout << "Congratulation!" << endl; cout << "Keep it up!" << endl; } else { cout << "Try Harder!" << endl; }</pre>	

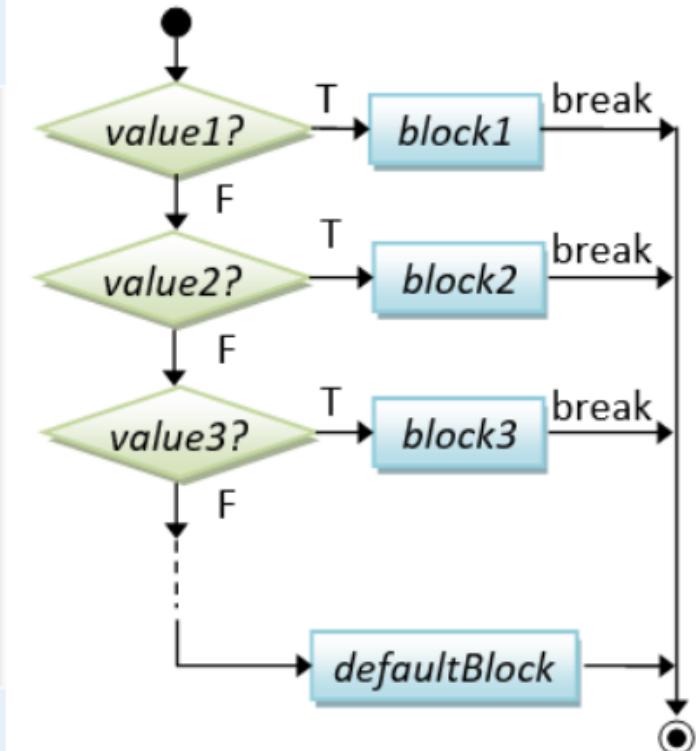
```
// nested-if
if ( booleanExpr-1 ) {
    block-1 ;
} else if ( booleanExpr-2 ) {
    block-2 ;
} else if ( booleanExpr-3 ) {
    block-3 ;
} else if ( booleanExpr-4 ) {
    .....
} else {
    elseBlock ;
}
```

```
if (mark >= 80) {
    cout << "A" << endl;
} else if (mark >= 70) {
    cout << "B" << endl;
} else if (mark >= 60) {
    cout << "C" << endl;
} else if (mark >= 50) {
    cout << "D" << endl;
} else {
    cout << "F" << endl;
}
```



```
// switch-case
switch ( selector ) {
    case value-1:
        block-1; break;
    case value-2:
        block-2; break;
    case value-3:
        block-3; break;
    .....
    case value-n:
        block-n; break;
    default:
        default-block;
}
```

```
char oper; int num1, num2, result;
.....
switch (oper) {
    case '+':
        result = num1 + num2; break;
    case '-':
        result = num1 - num2; break;
    case '*':
        result = num1 * num2; break;
    case '/':
        result = num1 / num2; break;
    default:
        cout << "Unknown operator" << endl;
}
```



Conditional Operator:

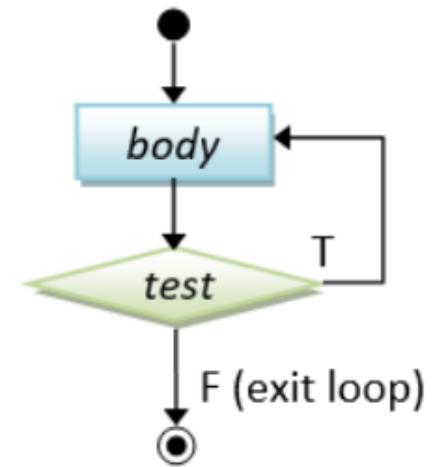
Syntax	Example
<code>booleanExpr ? trueExpr : falseExpr</code>	<pre>cout << (mark >= 50) ? "PASS" : "FAIL" << endl; // return either "PASS" or "FAIL", and put to cout max = (a > b) ? a : b; // RHS returns a or b abs = (a > 0) ? a : -a; // RHS returns a or -a</pre>

Loop Flow Control

Syntax	Example	Flowchart
<pre>// for-loop for (init; test; post-proc) { body ; }</pre>	<pre>// Sum from 1 to 1000 int sum = 0; for (int number = 1; number <= 1000; ++number) { sum += number; }</pre>	<pre>graph TD Start(()) --> init[init] init --> test{test} test -- F --> Exit(()) test -- T --> body[body] body --> update[update] update --> test</pre>
<pre>// while-do while (condition) { body ; }</pre>	<pre>int sum = 0, number = 1; while (number <= 1000) { sum += number; ++number; }</pre>	<pre>graph TD Start(()) --> test{test} test -- F --> Exit(()) test -- T --> body[body] body --> test</pre>

```
// do-while
do {
    body ;
}
while ( condition ) ;
```

```
int sum = 0, number = 1;
do {
    sum += number;
    ++number;
} while (number <= 1000);
```



```
/* *      Sum from 1 to a given upperbound and compute their average (SumNumbers.cpp) */
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    int sum = 0; // Store the accumulated sum
```

```
    int upperbound;
```

```
    cout << "Enter the upperbound: ";
```

```
    cin >> upperbound;
```

```
    for (int number = 1; number <= upperbound; ++number) { // Sum from 1 to the upperbound
```

```
        sum += number;
```

```
}
```

```
    cout << "Sum is " << sum << endl;
```

```
    cout << "Average is " << (double)sum / upperbound << endl;
```

```
    int count = 0; // Sum only the odd numbers // counts of odd numbers
```

```
    sum = 0; // reset sum
```

```
    for (int number=1; number <= upperbound; number=number+2) {
```

```
        ++count;
```

```
        sum += number;
```

```
}
```

```
    cout << "Sum of odd numbers is " << sum << endl;
```

```
    cout << "Average is " << (double)sum / count << endl;
```

```
}
```

```
#include <iostream> /* A mystery series (Mystery.cpp) */  
using namespace std;  
  
int main() {  
    int number = 1;  
  
    while (true) {  
        ++number;  
  
        if ((number % 3) == 0) continue;  
        if (number == 133) break;  
        if ((number % 2) == 0) {  
            number += 3;  
        } else {  
            number -= 3;  
        }  
        cout << number << " ";  
    }  
    cout << endl;  
    return 0; }
```

Interrupting Loop Flow - "break" and "continue"

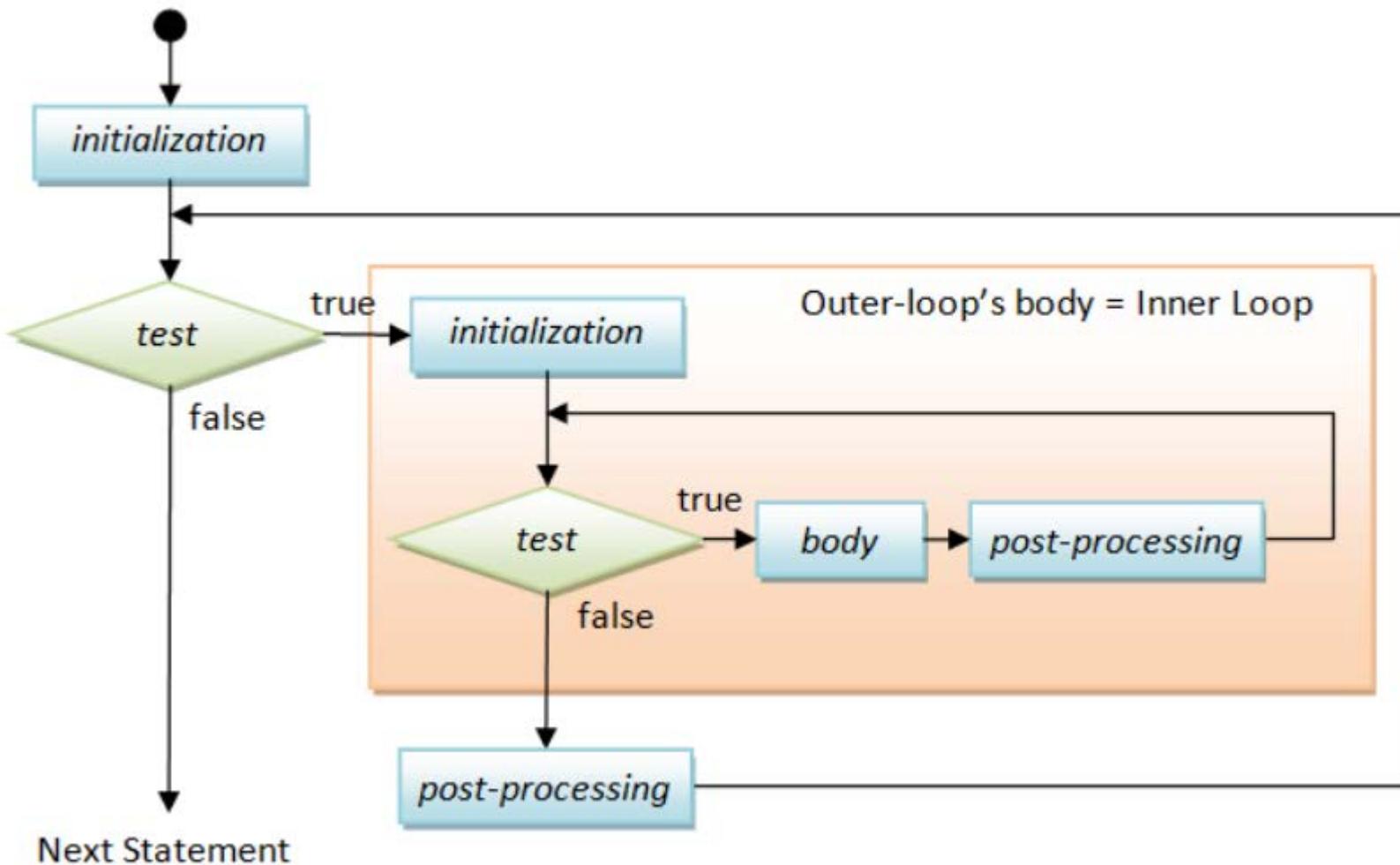
- There are a few ways that you can terminate your program, before reaching the end of the programming statements.
- **exit():** You could invoke the function exit(int exitCode), in <cstdlib> (ported from C's "stdlib.h"), to terminate the program and return the control to the Operating System. By convention, **return code of zero indicates normal termination**; while a **non-zero exitCode (-1) indicates abnormal termination**. For example,
- **abort():** The header <cstdlib> also provide a function called abort(), which can be used to terminate the program abnormally.

```
if (errorCount > 10) {  
    cout << "too many errors" << endl;  
    exit(-1); // Terminate the program// OR abort();  
}
```

- You could also use a "return returnValue" statement in the main() function to terminate the program and return control back to the Operating System.

```
int main() {  
    ...  
    if (errorCount > 10) {  
        cout << "too many errors" << endl;  
        return -1; // Terminate and return control to OS from main()  
    }  
    ...}
```

Nested Loops



```
/*Print square pattern (PrintSquarePattern.cpp). */  
#include <iostream>  
using namespace std;  
  
int main() {  
    int size = 8;  
    for (int row = 1; row <= size; ++row) { // Outer loop to print all the rows  
        for (int col = 1; col <= size; ++col) { // Inner loop to print all the columns of each row  
            cout << "# ";  
        }  
        cout << endl; // A row ended, bring the cursor to the next line  
    }  
  
return 0;  
}
```

# * # * # * # *	# # # # # # #	# # # # # # #	1	1
# * # * # * # *	# # # # # #	# # # # # #	2 1	1 2
# * # * # * # *	# # # # # #	# # # # # #	3 2 1	1 2 3
# * # * # * # *	# # # # #	# # # # #	4 3 2 1	1 2 3 4
# * # * # * # *	# # # #	# # # #	5 4 3 2 1	1 2 3 4 5
# * # * # * # *	# # #	# # #	6 5 4 3 2 1	1 2 3 4 5 6
# * # * # * # *	# #	# #	7 6 5 4 3 2 1	1 2 3 4 5 6 7
# * # * # * # *	#	#	8 7 6 5 4 3 2 1	1 2 3 4 5 6 7 8
(a)	(b)	(c)	(d)	(e)

# # # # # #	# # # # # #	# # # # # #	# # # # # #	# # # # # #
# # #	# #	# #	# #	# #
# # #	# #	# #	# #	# #
# # #	# #	# #	# #	# #
# # #	# #	# #	# #	# #
# # # # # #	# # # # # #	# # # # # #	# # # # # #	# # # # # #
(a)	(b)	(c)	(d)	(e)

Dangling else: The "dangling else" problem can be illustrated as follows:

```
if (i == 0)
    if (j == 0)
        cout << "i and j are zero" << endl;
else cout << "i is not zero" << endl; // intend for the outer-if
```

- The else clause in the above codes is syntactically applicable to both the outer-if and the inner-if.
- The C++ compiler always associate the else clause **with the innermost if** (i.e., the nearest if).
- Dangling else can be resolved by applying explicit parentheses. The above codes are logically incorrect and require explicit parentheses as shown below.

```
if ( i == 0) {
    if (j == 0) cout << "i and j are zero" << endl;
} else {
    cout << "i is not zero" << endl; // non-ambiguous for outer-if
}
```

Strings

- C++ supports two types of strings:
- **the original C-style string:** A string is a char array, terminated with a NULL character '\0' (Hex 0).
- the new string class introduced in C++98.
- The "high-level" string class is recommended. However, avoid C-string unless it is absolutely necessary.

String Declaration and Initialization

- To use the string class, include the <string> header and "using namespace std".
- You can declare and (a) initialize a string with a string literal, (b) initialize to an empty string, or (c) initialize with another string object. For example,

```
#include <string>
using namespace std;
string str1("Hello"); // Initialize with a string literal (Implicit initialization)
string str2 = "world"; // Initialize with a string literal (Explicit initialization via assignment operator)
string str3;           // Initialize to an empty string
string str4(str1);    // Initialize by copying from an existing string object
```

```
#include <iostream>    /* Testing string class input and output (TestStringIO.cpp) */
#include <string>    // Need this header to use string class
#include <limits>
using namespace std; // Also needed for <string>
int main() {
    string message("Hello");
    cout << message << endl;
    cout << "Enter a message (no space): "; // Input a word (delimited by space) into a string
    cin >> message;
    cout << message << endl;
    cin.ignore(numeric_limits<streamsize>::max(), '\n'); // flush cin up to newline (need <limits> header)
    cout << "Enter a message (with spaces): ";           // Input a line into a string
    getline(cin, message);                            // Read input from cin into message
    cout << message << endl;
    return 0;
}
```

String Operations

- Checking the length of a string:

```
int length();
int size();
both of them return the length of the string
```

```
#include <string>
string str("Hello, world");
cout << str.length() << endl; // 12
cout << str.size() << endl; // 12
```

- Check for empty string:

```
bool empty();
Check if the string is empty.
```

```
string str1("Hello, world");
string str2; // Empty string
cout << str1.empty() << endl; // 0 (false)
cout << str2.empty() << endl; // 1 (true)
```

- Copying from another string: Simply use the assignment (=) operator.

```
string str1("Hello, world"), str2;
str2 = str1;
cout << str2 << endl; // Hello, world
```

- Concatenated with another string: Use the plus (+) operator, or compound plus (+=) operator.

```
string str1("Hello,");
string str2(" world");
cout << str1 + str2 << endl; // "Hello, world"
cout << str1 << endl; // "Hello,"
cout << str2 << endl; // " world"
str1 += str2;
cout << str1 << endl; // "Hello, world"
cout << str2 << endl; // " world"
string str3 = str1 + str2;
cout << str3 << endl; // "Hello, world world"
str3 += "again";
cout << str3 << endl; // "Hello, world worldagain"
```

- Read/Write individual character of a string:

char& at(int index);

Return the char at index, index begin at 0. Perform index bound check.

[]

indexing (subscript) operator, no index bound check

```
string str("Hello, world");
cout << str.at(0) << endl; // 'H'
cout << str[1] << endl; // 'e'
cout << str.at(str.length() - 1) << endl; // 'd'

str.at(1) = 'a'; // Write to index 1
cout << str << endl; // "Hallo, world"
```

```
str[0] = 'h';
cout << str << endl; // "hallo, world"
```

- Extracting sub-string:

```
string substr(int beginIndex, int size);
    Return the sub-string starting at beginIndex, of size
```

```
string str("Hello, world");
cout << str.substr(2, 6) << endl; // "llo, w"
```

- Comparing with another string:

```
int compare(string another);
    Compare the content of this string with the given another.
    Return 0 if equals; a negative value if this string is less than another; positive value otherwise.
```

```
== and != Operators
    Compare the contents of two strings
```

```
string str1("Hello"), str2("Hallo"), str3("hello"), str4("Hello");
cout << str1.compare(str2) << endl; // 1  'e' > 'a'
cout << str1.compare(str3) << endl; // -1  'h' < 'H'
cout << str1.compare(str4) << endl; // 0
```

```
// You can also use the operator == or !=
if (str1 == str2) cout << "Same" << endl;
if (str3 != str4) cout << "Different" << endl;
cout << boolalpha; // print bool as true/false
cout << (str1 != str2) << endl;
cout << (str1 == str4) << endl;
```

- Search/Replacing characters: You can use the functions available in the `<algorithm>` such as `replace()`. For example,

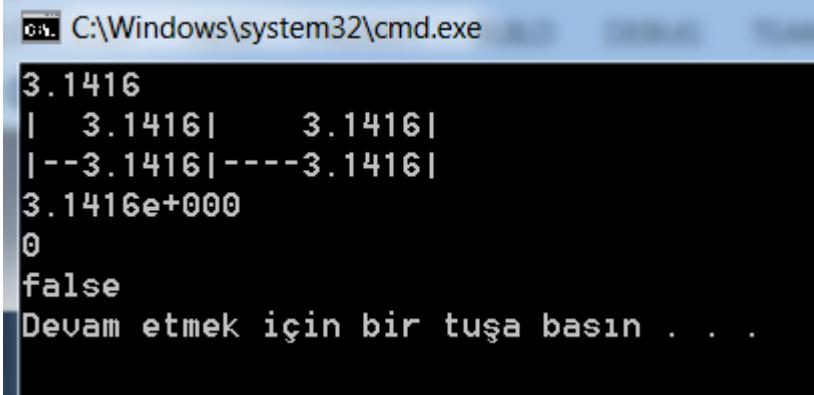
```
#include <algorithm>
.....
string str("Hello, world");
replace(str.begin(), str.end(), 'l', '_');
cout << str << endl; // "He_o, wor_d"
```

Formatting Input/Output using IO Manipulators (Header <iomanip>)

- The **<iomanip>** header provides so-called I/O manipulators for formatting input and output:
- **setw(int field-width):** set the field width for the next IO operation.
- **setw()** is non-sticky and must be issued prior to each IO operation. The field width is reset to the default after each operation
 - **setfill(char fill-char):** set the filled character for padding to the field width.
 - **left | right | internal:** set the alignment
- **fixed/scientific** (for floating-point numbers): use fixed-point notation (e.g., 12.34) or scientific notation (e.g., 1.23e+006).
- **setprecision(int numDecimalDigits)** (for floating-point numbers): specify the number of digits after the decimal point.
- **boolalpha/noboolalpha (for bool):** display bool values as alphabetic string (true/false) or 1/0.

```
#include <iostream> /* Test Formatting Output (TestFormattedOutput.cpp) */
#include <iomanip> // Needed to do formatted I/O
using namespace std;
int main() {
    double pi = 3.14159265; // Floating point numbers
    cout << fixed << setprecision(4); // fixed format with 4 decimal places
    cout << pi << endl;
    cout << " | " << setw(8) << pi << " | " << setw(10) << pi << " | " << endl;
    cout << setfill('-'); // setw() is not sticky, only apply to the next operation.
    cout << " | " << setw(8) << pi << " | " << setw(10) << pi << " | " << endl;
    cout << scientific; // in scientific format with exponent
    cout << pi << endl;
    bool done = false; // booleans
    cout << done << endl; // print 0 (for false) or 1 (for true)
    cout << boolalpha; // print true or false
    cout << done << endl;
    return 0;
}
```

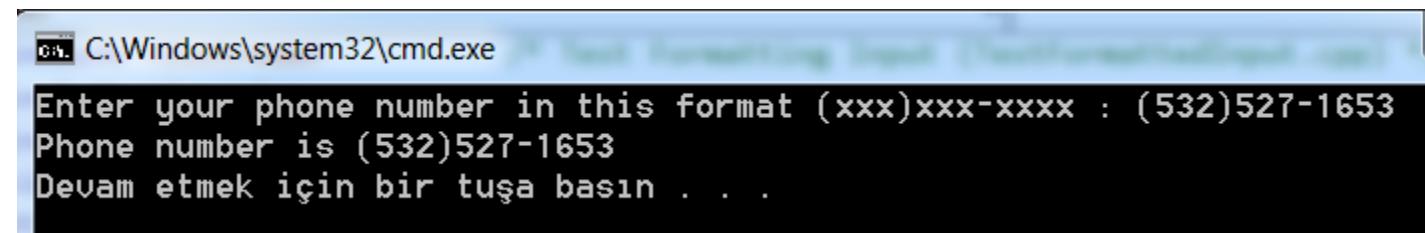
Output Formatting



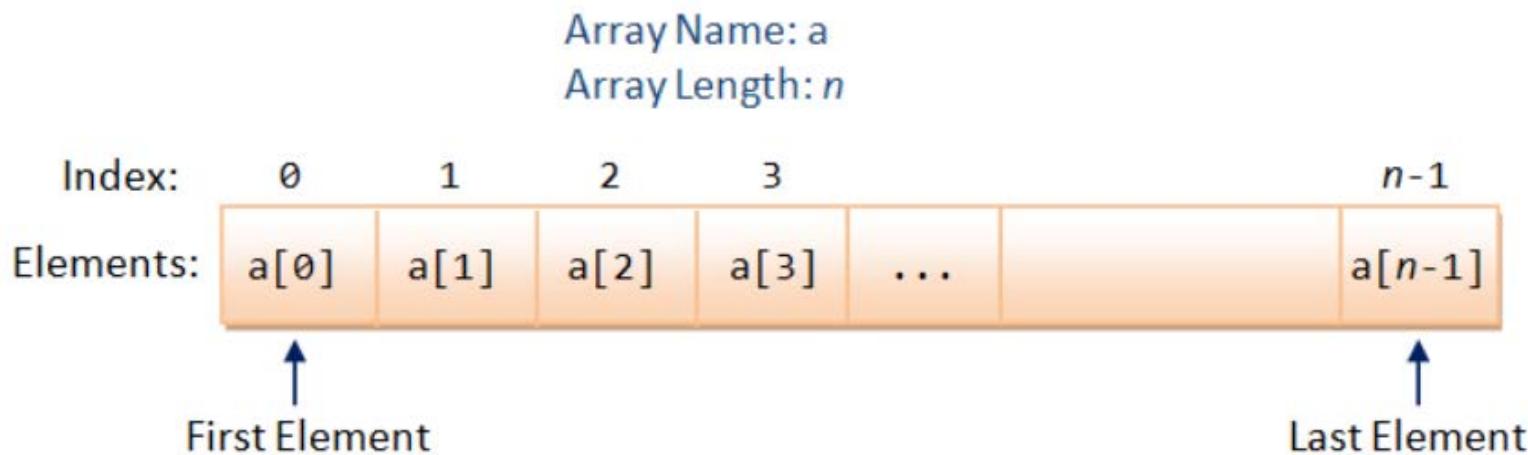
```
C:\Windows\system32\cmd.exe
3.1416
| 3.1416| 3.1416|
|--3.1416|----3.1416|
3.1416e+000
0
false
Devam etmek için bir tuşa basın . . .
```

```
#include <iostream>      /* Test Formatting Input (TestFormattedInput.cpp) */
#include <iomanip>    #include <string>
using namespace std;
int main() {
    string areaCode, phoneCode;
    string inStr;
    cout << "Enter your phone number in this format (xxx)xxx-xxxx : ";
    cin.ignore(); // skip '('
    cin >> setw(3) >> areaCode;
    cin.ignore(); // skip ')'
    cin >> setw(3) >> phoneCode;
    cin.ignore(); // skip '-'
    cin >> setw(4) >> inStr;
    phoneCode += inStr;
    cout << "Phone number is (" << areaCode << ")"
        << phoneCode.substr(0, 3) << "-"
        << phoneCode.substr(3, 4) << endl;
return 0;}
```

Input Formatting



Arrays: Array Declaration and Usage



```
type arrayName[arraylength]; // arraylength can be a literal or a variable
int marks[5];      // Declare an int array called marks with 5 elements
double numbers[10]; // Declare an double array of 10 elements
const int SIZE = 9;
float temps[SIZE]; // Use const int as array length
int size; // Some compilers support an variable as array length, e.g.,
cout << "Enter the length of the array: ";
cin >> size;
float values[size];
```

// Declare and initialize an int array of 3 elements

int numbers[3] = {11, 33, 44};

// If length is omitted, the compiler counts the elements

int numbers[] = {11, 33, 44};

// Number of elements in the initialization shall **be equal to or less than length**

int numbers[5] = {11, 33, 44}; // Remaining elements are zero. Confusing! Don't do this

int numbers[2] = {11, 33, 44}; // ERROR: too many initializers

// Use {0} or {} to initialize all elements to 0

int numbers[5] = {0}; // First element to 0, the rest also to zero

int numbers[5] = {}; // All element to 0 too

```
#include <iostream> /* Find the mean and standard deviation of numbers kept in an array (MeanStdArray.cpp). */  
#include <iomanip>  
#include <cmath>  
#define SIZE 7  
using namespace std;  
int main() {  
    int marks[] = {74, 43, 58, 60, 90, 64, 70};  
    int sum = 0;  
    int sumSq = 0;  
    double mean, stdDev;  
    for (int i = 0; i < SIZE; ++i) {  
        sum += marks[i];  
        sumSq += marks[i]*marks[i];  
    }  
    mean = (double)sum/SIZE;  
    cout << fixed << "Mean is " << setprecision(2) << mean << endl;  
    stdDev = sqrt((double)sumSq/SIZE - mean*mean);  
    cout << fixed << "Std dev is " << setprecision(2) << stdDev << endl;  
    return 0;  
}
```

Array and Loop

Range-based for loop (C++11)

```
#include <iostream> /* Testing For-each loop (TestForEach.cpp) */
using namespace std;
int main() {
    int numbers[] = {11, 22, 33, 44, 55};
    for (int number : numbers) { // For each member called number of array numbers - read only
        cout << number << endl;
    }
    for (int &number : numbers) { // To modify members, need to use reference (&)
        number = 99;
    }
    for (int number : numbers) {
        cout << number << endl;
    }
    return 0;
}
```

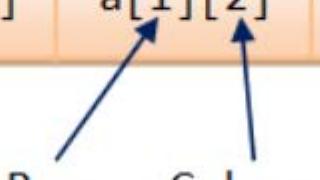
C:\Windows\system32\cmd.exe

```
11
22
33
44
55
99
99
99
99
99
Devam etmek için bir tuşa basın . . .
```

Multi-Dimensional Array

	Column 0	Column 1	Column 2	Column 3	...
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	...
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	...

Row Index Column Index

A diagram illustrating a 2D array structure. It shows two rows and four columns of cells. The first row is labeled "Row 0" and the second row is labeled "Row 1". The columns are labeled "Column 0", "Column 1", "Column 2", "Column 3", and "...". The cells contain expressions like a[0][0], a[0][1], a[0][2], a[0][3], ..., a[1][0], a[1][1], a[1][2], a[1][3], Two blue arrows point from the text "Row Index" and "Column Index" to the labels "Row 0" and "Column 2" respectively.

```
/* Test Multi-dimensional Array (Test2DArray.cpp) */

#include <iostream>
using namespace std;
void printArray(const int[][][3], int);
int main() {
    int myArray[][][3] = {{8, 2, 4}, {7, 5, 2}}; // 2x3 initialized
    printArray(myArray, 2); // Only the first index can be omitted and implied
    return 0;
}
void printArray(const int array[][][3], int rows) {// Print the contents of rows-by-3 array (columns is fixed)
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < 3; ++j) {
            cout << array[i][j] << " ";
        }
        cout << endl;
    }
}
```

Array of Characters - C-String

- In C, a string is a char array terminated by a NULL character '\0' (ASCII code of Hex 0).
- C++ provides a new string class under header <string>.
- The original string in C is known as C-String (or C-style String or Character String). You could allocate a C-string via:

```
char message[256]; // Declare a char array
                    // Can hold a C-String of up to 255 characters terminated by '\0'
char str1[] = "Hello"; // Declare and initialize with a "string literal".
                      // The length of array is number of characters + 1 (for '\0').
char str1char[] = {'H', 'e', 'l', 'l', 'o', '\0'}; // Same as above
char str2[256] = "Hello"; // Length of array is 256, keeping a smaller string.
```

Functions

- At times, a certain portion of codes has to be used many times. Instead of re-writing the codes many times, it is better to put them into a "subroutine", and "call" this "subroutine" many time - for ease of maintenance and understanding.
- Subroutine is called method (in Java) or function (in C/C++).
- The benefits of using functions are:
 - **Divide and conquer:** construct the program from simple, small pieces or components. Modularize the program into self-contained tasks.
 - **Avoid repeating codes:** It is easy to copy and paste, but hard to maintain and synchronize all the copies.
 - **Software Reuse:** you can reuse the functions in other programs, by packaging them into library codes.
 - Two parties are involved in using a function: **a caller** who calls the function, and the **function called**. The caller passes argument(s) to the function. The function receives these argument(s), performs the programmed operations within the function's body, and returns a piece of result back to the caller.

Caller – main()

```
double radius = 1.1;  
double area;  
area = getArea(radius);  
cout << area << endl;
```

Argument(s)

Function – getArea()

```
double getArea(double r) {  
    return r * r * 3.14159265;  
}
```

Result

radius (double)

1.1

r (double)

area (double)

xxxx

xxxx

return-value (double)



Function Definition

The syntax for function definition is as follows:

```
returnValueType functionName ( parameterList ) {  
    functionBody ;  
}
```

The "return" Statement

Inside the function's body, you could use a return statement to return a value (of the *returnValueType* declared in the function's header) and pass the control back to the caller. The syntax is:

```
return expression; // Evaluated to a value of returnValueType declared in function's signature  
return; // For function with return type of void
```

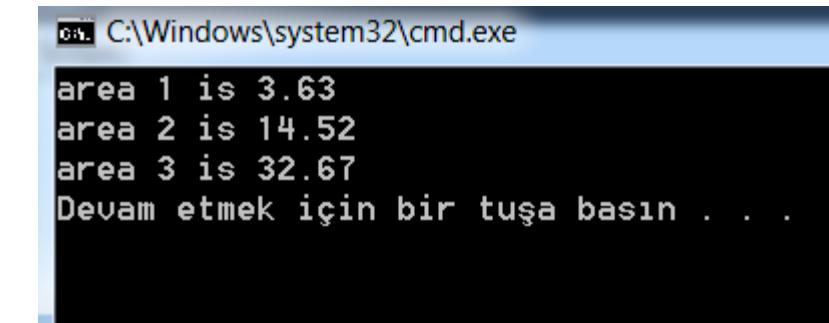
```
#include <iostream>
using namespace std;
const int PI = 3.14159265;
double getArea(double radius); // Function Prototype (Function Declaration)

int main() {
    double radius1 = 1.1, area1, area2;
area1 = getArea(radius1); // call function getArea()
    cout << "area 1 is " << area1 << endl;
area2 = getArea(2.2); // call function getArea()

    cout << "area 2 is " << area2 << endl;
cout << "area 3 is " << getArea(3.3) << endl; // call function getArea()

}

// Function Definition// Return the area of a circle given its radius
double getArea(double radius) {
    return radius * radius * PI;
}
```



Default Arguments

- C++ introduces so-called **default arguments for functions**.
- These default values would be used if the caller omits the corresponding actual argument in calling the function.
- Default arguments are **specified in the function prototype, and cannot be repeated in the function definition**.
- The default arguments are resolved based on their positions. Hence, they can only be used to substitute the trailing arguments to avoid ambiguity.

```
#include <iostream> /* Test Function default arguments (functionDefaultArgument.cpp) */
using namespace std;
int fun1(int = 1, int = 2, int = 3); // Function prototype - Specify the default arguments here
int fun2(int, int, int = 3);
int main() {
    cout << fun1(4, 5, 6) << endl; // No default
    cout << fun1(4, 5) << endl; // 4, 5, 3(default)
    cout << fun1(4) << endl; // 4, 2(default), 3(default)
    cout << fun1() << endl; // 1(default), 2(default), 3(default)
    cout << fun2(4, 5, 6) << endl; // No default
    cout << fun2(4, 5) << endl; // 4, 5, 3(default)
// cout << fun2(4) << endl; // error: too few arguments to function 'int fun2(int, int, int)'
}
int fun1(int n1, int n2, int n3) {
return n1 + n2 + n3; // cannot repeat default arguments in function definition
}
int fun2(int n1, int n2, int n3) {
    return n1 + n2 + n3;
}
```

Function Overloading

- C++ introduces function overloading (or function polymorphism, which means many forms), which allows **you to have multiple versions of the same function name**, differentiated by the parameter list (number, type or order of parameters).
- The version matches the caller's argument list will be selected for execution.

```
/* Test Function Overloading (FunctionOverloading.cpp) */
```

```
#include <iostream>

using namespace std;

void fun(int, int, int);      // Version 1
void fun(double, int);       // Version 2
void fun(int, double);       // Version 3

int main() {
    fun(1, 2, 3); // version 1
    fun(1.0, 2);  // version 2
    fun(1, 2.0);  // version 3
    fun(1.1, 2, 3); // version 1 - double 1.1 casted to int 1 (without warning)
```

```
// fun(1, 2, 3, 4); // error: no matching function for call to 'fun(int, int, int, int)'
// fun(1, 2); // error: call of overloaded 'fun(int, int)' is ambiguous
// note: candidates are:
// void fun(double, int)           // void fun(int, double)
// fun(1.0, 2.0); // error: call of overloaded 'fun(double, double)' is ambiguous
```

{

```
void fun(int n1, int n2, int n3) { // version 1
    cout << "version 1" << endl;
```

}

```
void fun(double n1, int n2) { // version 2
    cout << "version 2" << endl;
```

}

```
void fun(int n1, double n2) { // version 3
    cout << "version 3" << endl;
```

}

Pass-by-Value vs. Pass-by-Reference

- In pass-by-value, a "copy" of argument is created and passed into the function. The invoked function works on the "clone", and **cannot modify the original copy**. - there is no side effect.

```
#include <iostream> /* Fundamental types are passed by value into Function (TestPassByValue.cpp) */
using namespace std;

int inc(int number); // Function prototypes

// Test Driver

int main() {
    int n = 8;
    cout << "Before calling function, n is " << n << endl; // 8
    int result = inc(n);
    cout << "After calling function, n is " << n << endl; // 8
    cout << "result is " << result << endl; // 9
}

int inc(int number) { // Function definitions // Return number+1
    ++number; // Modify parameter, no effect to caller
    return number;
}
```

Pass-by-Reference

- In pass-by-reference, a reference of the caller's variable is passed into the function. In other words, **the invoked function works on the same data**.
- If the invoked function modifies the parameter, the same caller's copy will be modified as well.
- In C/C++, arrays are passed by reference. That is, you can modify the contents of the caller's array inside the invoked function - there could be side effect in passing arrays into function.
- C/C++ does not allow functions to return an array. Hence, if you wish to write a function that modifies the contents of an array you need to rely on pass-by-reference to work on the same copy inside and outside the function.
- Recall that in pass-by-value, the invoked function works on a clone copy and has no way to modify the original copy.

```
/* Function to increment each element of an array (IncrementArray.cpp) */

#include <iostream> using namespace std;

void inc(int array[], int size);
void print(int array[], int size);

int main() {
    int a1[] = {8, 4, 5, 3, 2};
    // Before increment
    print(a1, 5); // {8,4,5,3,2}
    // Array is passed by reference
    // Do increment
    inc(a1, 5);
    // After increment
    print(a1, 5); // {9,5,6,4,3}
```

```
void inc(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        array[i]++;
    }
}

void print(int array[], int size) {
    cout << "{";
    for (int i = 0; i < size; ++i) {
        cout << array[i];
        if (i < size - 1) {
            cout << ",";
        }
    }
    cout << "}" << endl;
}
```

Array is passed into function by reference. That is, the invoked function works on the same copy of the array as the caller. Hence, changes of array inside the function is reflected outside the function.

- Pass-by-reference risks corrupting the original data. If you do not have the intention of modifying the arrays inside the function, you could use the **const** keyword in the function parameter. A **const function argument cannot be modified inside the function.**

```
#include <iostream>
using namespace std;
int linearSearch(const int a[], int size, int key);
int main() {
    const int SIZE = 8;
    int a1[SIZE] = {8, 4, 5, 3, 2, 9, 4, 1};
    cout << linearSearch(a1, SIZE, 8) << endl; // 0
    cout << linearSearch(a1, SIZE, 4) << endl; // 1
    cout << linearSearch(a1, SIZE, 99) << endl; // 8 (not found)
}
// Search the array for the given key// If found, return array index [0, size-1]; otherwise, return size
int linearSearch(const int a[], int size, int key) {
    for (int i = 0; i < size; ++i) {
        if (a[i] == key) return i;
    }
    return size;
}
```

const Function Parameters

```
#include <iostream> /* Test Pass-by-reference for fundamental-type parameter via reference declaration (TestPassByReference.cpp) */

using namespace std;

int squareByValue (int number); // Pass-by-value

void squareByReference (int & number); // Pass-by-reference

int main() {
    int n1 = 8;
    cout << "Before call, value is " << n1 << endl; // 8
    cout << squareByValue(n1) << endl; // no side-effect
    cout << "After call, value is " << n1 << endl; // 8

    int n2 = 9;
    cout << "Before call, value is " << n2 << endl; // 9
    squareByReference(n2); // side-effect
    cout << "After call, value is " << n2 << endl; // 81
}

int squareByValue (int number) { // Pass parameter by value - no side effect
    return number * number;
}

void squareByReference (int & number) { // Pass parameter by reference by declaring as reference (&)// - with side effect to the caller
    number = number * number;
}
```

Pass-by-Reference via "Reference" Parameters

File Input/Output (Header <fstream>)

- The <fstream> header provides ifstream (input file stream) and ofstream (output file stream) for file input and output.
- The steps for file input/output are:
 - Create a ifstream for input, or ofstream for output.
 - Connect the stream to an input or output file via open(filename).
 - Perform formatted output via stream insertion operator <<, or input via stream extraction operator >>, similar to cout << and cin >>.
 - Close the file and free the stream.

```
/* Test File I/O (TestFileIO.cpp)  Read all the integers from an input file and  write the average to an output file */
```

```
#include <iostream>
```

```
#include <fstream> // file stream
```

```
#include <cstdlib>
```

```
using namespace std;
```

```
int main() {
```

```
    ifstream fin; // Input stream
```

```
    ofstream fout; // Output stream
```

```
    fin.open("in.txt"); // Try opening the input file
```

```
    if (!fin.is_open()) {
```

```
        cerr << "error: open input file failed" << endl;
```

```
        abort(); // Abnormally terminate the program (in <cstdlib>)
```

```
}
```

```
    int sum = 0, number, count = 0;
```

```
    while (!(fin.eof())) {
```

```
        fin >> number; // Use >> to read
```

```
        sum += number;
```

```
        ++count;
```

```
}
```

```
double average = double(sum) / count;  
cout << "Count = " << count << " average = " << average << endl;  
fin.close();
```

```
// Try opening the output file  
fout.open("out.txt");  
if (!fout.is_open()) {  
    cerr << "error: open output file failed" << endl;  
    abort();  
}  
// Write the average to the output file using <<  
fout << average;  
fout.close();  
return 0;  
}
```

- When you use different library modules, there is always a potential for name clashes, as different library may use the same name for different purposes.
- This problem can be resolved via the use of **namespace in C++**. A namespace is a collection for identifiers under the same naming scope. (It is known as package in UML and Java.)
- The entity name under a namespace is qualified by the namespace name, followed by **:: (known as scope resolution operator)**, in the form of **namespace::entityName**.

// create a namespace called myNamespace for the enclosed entities

```
namespace myNameSpace {  
    int foo;          // variable  
    int f() { ..... }; // function  
    class Bar { ..... }; // compound type such as class and struct  
}
```

Namespace

// To reference the entities, use

```
myNameSpace::foo  
myNameSpace::f()  
myNameSpace::Bar
```

- A namespace can contain variables, functions, arrays, and compound types such as classes and structures.

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

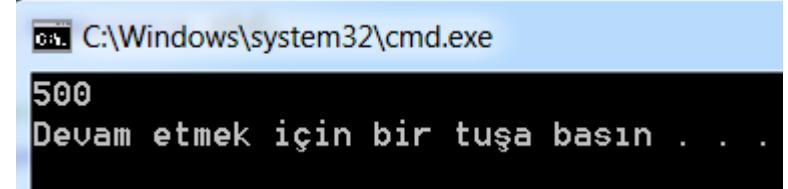
namespace first // Variable created inside namespace
{
    int val = 500;
}

// Global variable
int val = 100;
int main()
{
    int val = 200; // Local variable

    // These variables can be accessed from
    // outside the namespace using the scope
    // operator ::

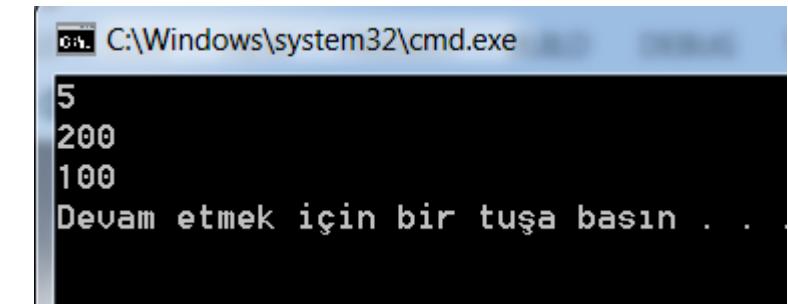
    cout << first::val << '\n';

    return 0;
}
```



```
#include <iostream>
using namespace std;
namespace ns1 {
    int value() { return 5; }
}
namespace ns2 {
    const double x = 100;
    double value() { return 2*x; }
}

int main() {
    cout << ns1::value() << '\n'; // Access value function within ns1
    cout << ns2::value() << '\n'; // Access value function within ns2
    cout << ns2::x << '\n'; // Access variable x directly
    return 0;
}
```



Using Namespace

- For example, all the identifiers in the C++ standard libraries (such as cout, endl and string) are placed under the namespace called std. To reference an identifier under a namespace, you have three options:

- Use the fully qualified names, such as std::cout, std::endl, std::setw() and std::string. For example,

std::cout << std::setw(6) << 1234 << std::endl;

- Missing the "std::" results in "error: 'xxx' was not declared in this scope".

- Use a using declaration to declare the particular identifiers. For example,

using std::cout;

using std::endl;

.....

cout << std::setw(6) << 1234 << endl;

- You can omit the "std::" for cout and endl, but you still have to use "std::" for setw.

- Use a using namespace directive. For example,

using namespace std;

.....

cout << setw(6) << 1234 << endl;

- The using namespace directive effectively brings all the identifiers from the specified namespace to the global scope, as if they are available globally. You can reference them without the scope resolution operator. Take note that the using namespace directive may result in name clashes with identifier in the global scope.

- For long namespace name, you could define a shorthand (or alias) to the namespace, as follows:

namespace shorthand = namespace-name;

Enumeration (enum)

- An enum is a user-defined type of a set of named constants, called enumerators. An enumeration define the complete set of values that can be assigned to objects of that type. For example,

```
enum Color {  
    RED, GREEN, BLUE  
} myColor; // Define an enum and declare a variable of the enum  
.....  
myColor = RED; // Assign a value to an enum  
Color yourColor;  
yourColor = GREEN;
```

- The enumerators are represented internally as integers. You have to use the names in assignment, not the numbers.
- However, it will be promoted to int in arithmetic operations. By default, they are running numbers starting from zero. You can assign different numbers, e.g.,

```
enum Color {  
    RED = 1, GREEN = 5, BLUE  
};
```

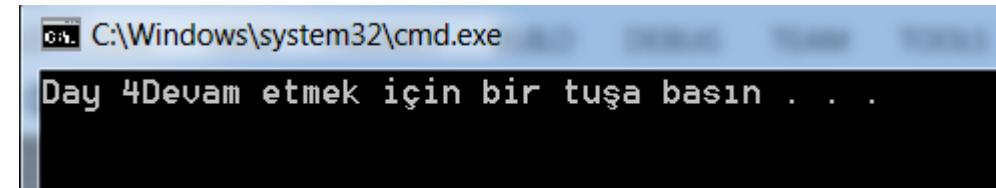
- To print the enumerator names, you may need to define a array of string, indexed by the enumerator numbers.

```
#include <iostream>

using namespace std;

enum week { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

int main()
{
    week today;
    today = Wednesday;
    cout << "Day " << today+1;
    return 0;
}
```



```
#include <iostream>

using namespace std;

enum seasons { spring = 34, summer = 4, autumn = 9, winter = 32};

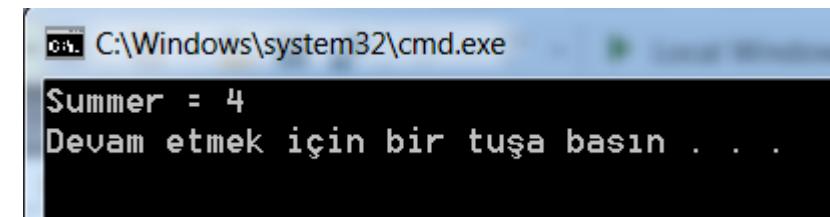
int main() {

    seasons s;

    s = summer;

    cout << "Summer = " << s << endl;

    return 0;
}
```



Structure (struct)

```
struct StructName {  
    type1 var1;  
    type2 var2;  
    .....  
}; // need to terminate by a semi-colon
```

- **struct is an intermediate step towards Object-oriented Programming (OOP).**
- In OOP, we use class that is an user-defined structure containing both data members and member functions.

```
/* Testing struct (TestStruct.cpp) */  
#include <iostream>  
using namespace std;  
  
struct Point {  
    int x;  
    int y;  
};  
  
int main() {  
    Point p1 = {3, 4}; // declare and init members  
    cout << "(" << p1.x << "," << p1.y << ")" << endl; // (3,4)  
  
    Point p2 = {};// declare and init numbers to defaults  
    cout << "(" << p2.x << "," << p2.y << ")" << endl; // (0,0)  
    p2.x = 7;  
    p2.y = 8;  
    cout << "(" << p2.x << "," << p2.y << ")" << endl; // (7,8)  
    return 0;  
}
```

```
#include <iostream> /* * Testing struct (TestStruct.cpp) */

using namespace std;

struct Point {
    int x, y;
};

struct Rectangle {
    Point topLeft;
    Point bottomRight;
};

int main() {
    Point p1, p2;
    p1.x = 0; // p1 at (0, 3)
    p1.y = 3;    p2.x = 4; // p2 at (4, 0)
    p2.y = 0;
    cout << "p1 at (" << p1.x << "," << p1.y << ")" << endl;
    cout << "p2 at (" << p2.x << "," << p2.y << ")" << endl;
    Rectangle rect;
    rect.topLeft = p1;    rect.bottomRight = p2;
    cout << "Rectangle top-left at (" << rect.topLeft.x    << "," << rect.topLeft.y << ")" << endl;
    cout << "Rectangle bottom-right at (" << rect.bottomRight.x   << "," << rect.bottomRight.y << ")" << endl;
    return 0;
}
```

Inline Functions

- Function call has its overhead (handling the argument, managing function stack, branch and return). For simple and short functions, you may use inline functions to remove the function call overhead. The keyword **inline** (before the function's return-type) suggest to the compiler to **"expand" the function code in place, instead of performing a function call**. For example,

```
// TestInlineFunction.cpp
```

```
#include <iostream>
```

```
using namespace std;
```

```
inline int max(int n1, int n2) {  
    return (n1 > n2) ? n1 : n2;  
}
```

```
int main() {  
    int i1 = 5, i2 = 6;  
    cout << max(i1, i2) << endl; // inline request to expand to (i1 > i2) ? i1 : i2  
    return 0;  
}
```

```
#include <iostream> /* Test #define macro (TestMacro.cpp) */  
using namespace std;  
#define SQUARE(x) x*x // Macro with argument  
inline int square(int x) { return x*x; } // inline function  
int main() {  
    cout << SQUARE(5) << endl; // expand to 5*5 (25)  
    int x = 2, y = 3;  
    cout << SQUARE(x) << endl; // expand to x*x (4)  
    cout << SQUARE(5+5) << endl; // expand to 5+5*5+5 - wrong answer// Problem with the expansions  
    cout << square(5+5) << endl; // Okay square(10)  
    cout << SQUARE(x+y) << endl; // expand to x+y*x+y - wrong answer  
    cout << square(x+y) << endl; // Okay  
    // can be fixed using #define SQUARE(x) (x)*(x)  
    cout << SQUARE(++x) << endl; // expand to ++x*++x (16) - x increment twice  
    cout << x << endl; // x = 4  
    cout << square(++y) << endl; // Okay ++y, (y*y) (16)  
    cout << y << endl; // y =  
}
```

Inline Function vs. #define Macro

Inline function is preferred over macro expansion

```
#include <iostream> /* * TestEllipses.cpp */
#include <cstdarg>
using namespace std;
int sum(int, ...);
int main() {
    cout << sum(3, 1, 2, 3) << endl;      // 6
    cout << sum(5, 1, 2, 3, 4, 5) << endl; // 15
    return 0;
}
int sum(int count, ...) {
    int sum = 0;
    va_list lst; // Declare a va_list// Ellipses are accessed thru a va_list
    // Use function va_start to initialize the va_list, // with the list name and the number of
    // parameters.
    va_start(lst, count);
    for (int i = 0; i < count; ++i) {
        // Use function va_arg to read each parameter from va_list, // with the type.
        sum += va_arg(lst, int);
    }
    va_end(lst); // Cleanup the va_list.
    return sum;
}
```

Ellipses (...)

Scope Resolution Operator

- The symbol :: is known as scope resolution operator. If a global variable is hidden by a local variable of the same name, you could use the scope resolution operator to retrieve the hidden global variable.

```
#include <iostream> // TestScopeResolutionOperator.cpp
using namespace std;
int x = 5; // Global variable

int main() {
    // A local variable having the Same name as a global variable, // which hides the global
variable
    float x = 55.5f;
    // Local
    cout << x << endl;
    // Use unary scope resolution operator to retrieve the global variable
    cout << ::x << endl;
    return 0;
}
```

Object-Oriented Programming (OOP) in C++

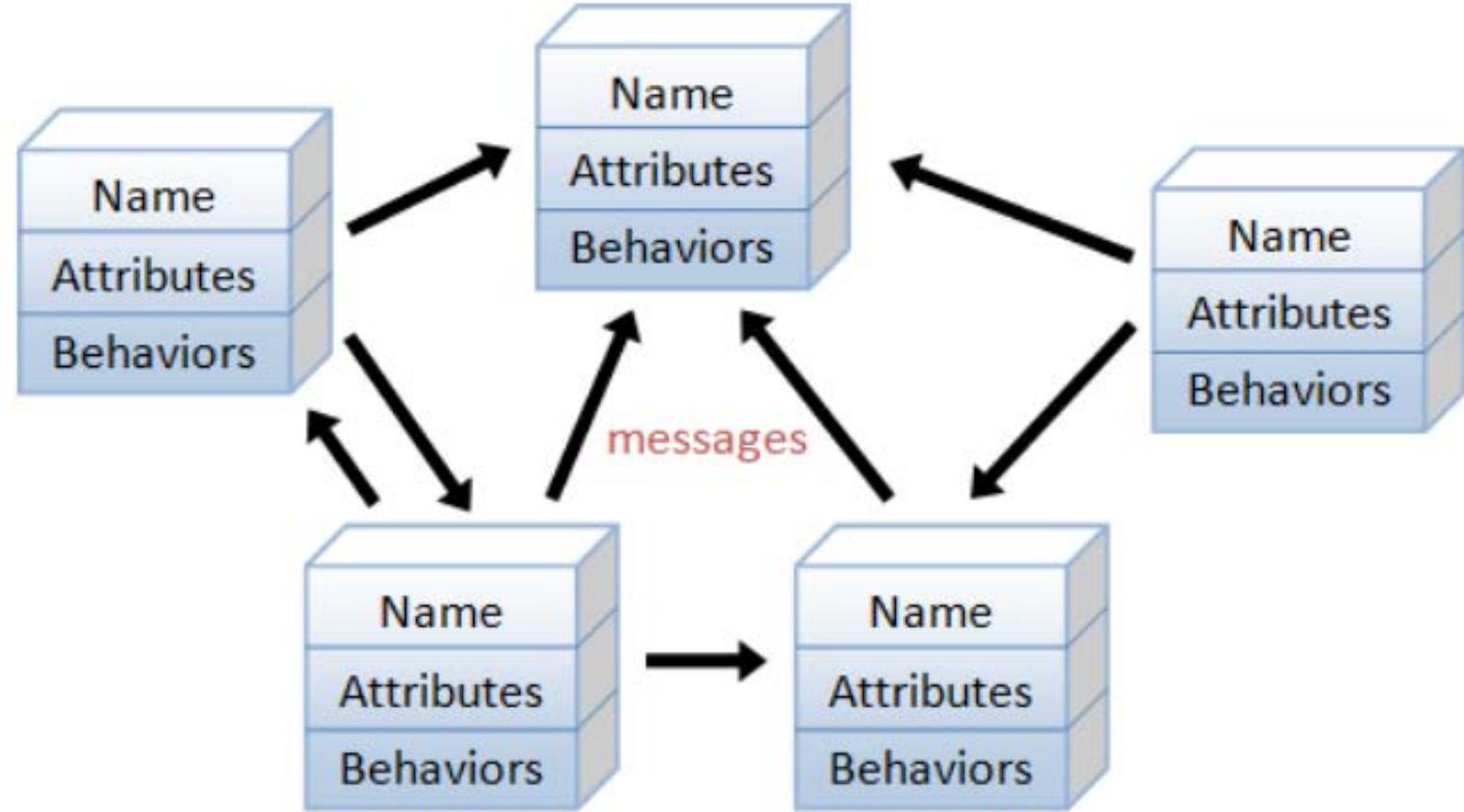
- Suppose that you want to assemble your own PC, you go to a hardware store and pick up a motherboard, a processor, some RAMs, a hard disk, a casing, a power supply, and put them together. You turn on the power, and the PC runs. You need not worry whether the motherboard is a 4-layer or 6-layer board, whether the hard disk has 4 or 6 plates; 3 inches or 5 inches in diameter, whether the RAM is made in Japan or Korea, and so on. You simply put the hardware components together and expect the machine to run. Of course, you have to make sure that you have the correct interfaces, i.e., you pick an IDE hard disk rather than a SCSI hard disk, if your motherboard supports only IDE; you have to select RAMs with the correct speed rating, and so on. Nevertheless, it is not difficult to set up a machine from hardware components.
- Similarly, a car is assembled from parts and components, such as chassis, doors, engine, wheels, brake, and transmission. The components are reusable, e.g., a wheel can be used in many cars (of the same specifications).
- Hardware, such as computers and cars, are assembled from parts, which are reusable components.
- How about software? Can you "assemble" a software application by picking a routine here, a routine there, and expect the program to run? The answer is obviously no! Unlike hardware, it is very difficult to "assemble" an application from software components. Since the advent of computer 60 years ago, we have written tons and tons of programs. However, for each new application, we have to re-invent the wheels and write the program from scratch.
- **Why re-invent the wheels?**

Traditional Procedural-Oriented languages

- Traditional procedural-oriented languages (such as C and Pascal) suffer some notable drawbacks **in creating reusable software components**:
- The programs are made up of functions. Functions are often not reusable. It is very difficult to copy a function from one program and reuse in another program because the function is likely to reference the headers, global variables and other functions.
- In other words, **functions are not well-encapsulated as a self-contained reusable unit**.
- **The procedural languages are not suitable of high-level abstraction** for solving real life problems. For example, C programs uses constructs such as if-else, for-loop, array, function, pointer, which are low-level and hard to abstract real problems such as a Customer Relationship Management (CRM) system or a computer soccer game.
- In brief, **the traditional procedural-languages separate the data structures and algorithms of the software entities**.

Object-Oriented Programming Languages

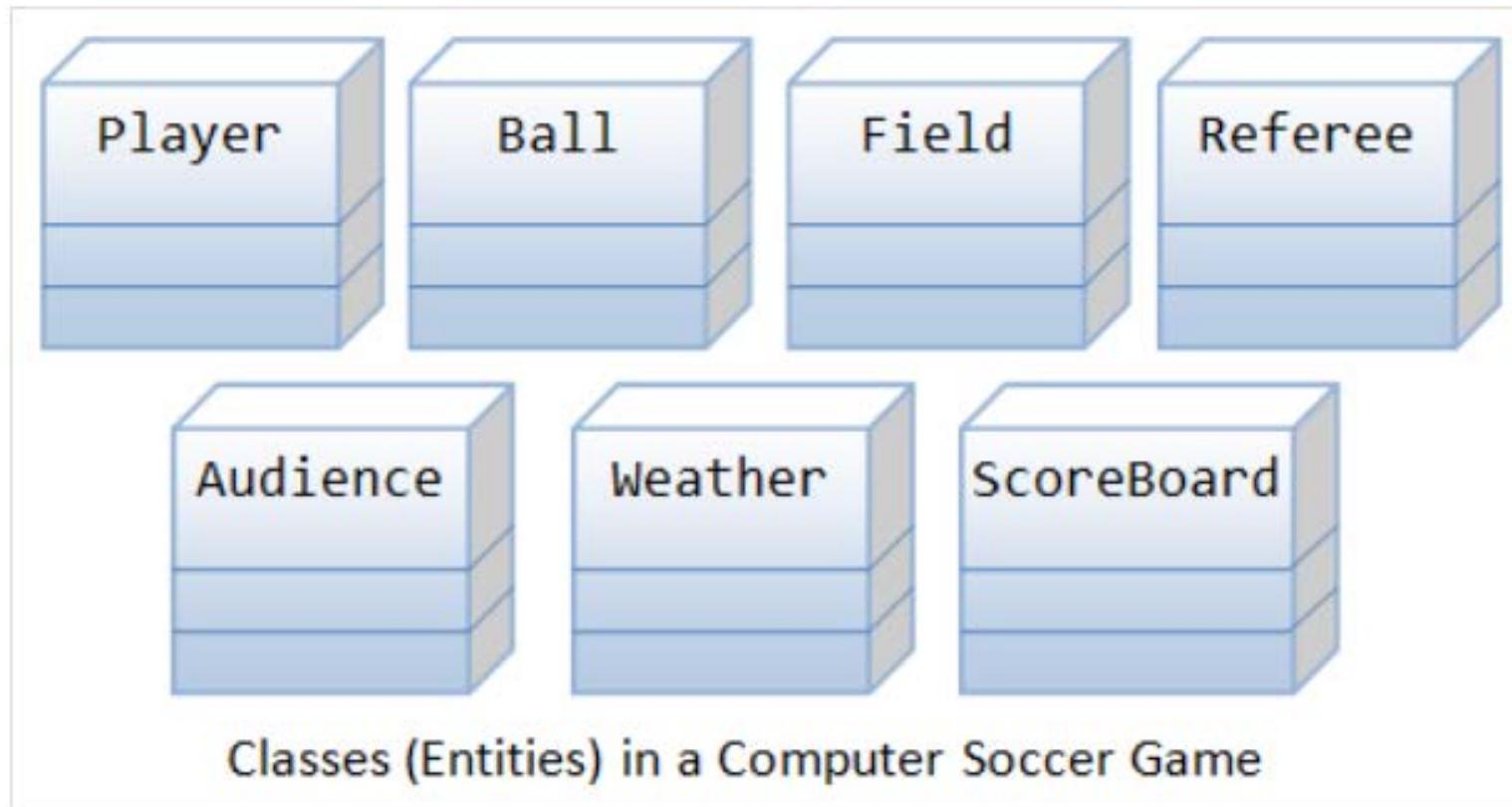
- The basic unit of OOP is a **class**, which encapsulates both the static attributes and dynamic behaviors within a "box", and specifies the public interface for using these boxes.
- Since the class is well-encapsulated (compared with the function), it is easier to reuse these classes.
- In other words, OOP combines the data structures and algorithms of a software entity inside the same box.
- OOP languages permit higher level of abstraction for solving real-life problems.
- The traditional procedural language (such as C and Pascal) forces you to think in terms of the structure of the computer (e.g. memory bits and bytes, array, decision, loop) rather than thinking in terms of the problem you are trying to solve.
- The OOP languages (such as Java, C++, C#) let you think in the problem space, and use software objects to represent and abstract entities of the problem space to solve the problem.



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

Example:

- As an example, suppose you wish to write a computer soccer games. It is quite difficult to model the game in procedural-oriented languages. But using OOP languages, you can easily model the program accordingly to the "real things" appear in the soccer games.
- **Player:** attributes include name, number, location in the field, and etc; operations include run, jump, kick-the-ball, and etc.
- **Ball:**
- **Reference:**
- **Field:**
- **Audience:**
- **Weather:**
- Most importantly, some of these classes (such as Ball and Audience) can be reused in another application, e.g., computer basketball game, with little or no modification.



Benefits of OOP

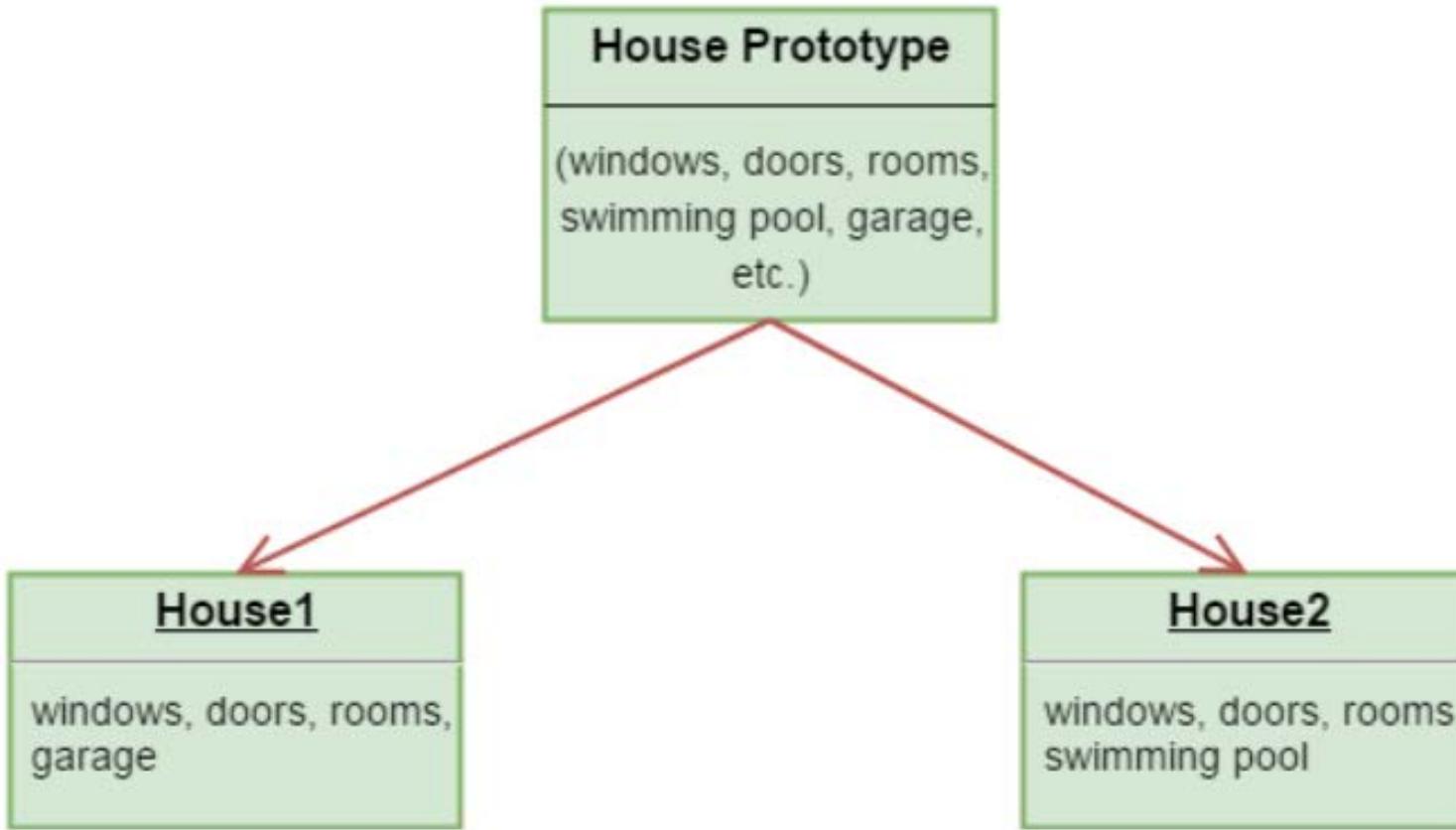
- The procedural-oriented languages focus on procedures, with function as the basic unit. You need to first figure out all the functions and then think about how to represent data.
- The object-oriented languages focus on components that the user perceives, with objects as the basic unit. **You figure out all the objects by putting all the data and operations that describe the user's interaction with the data.**

Object-Oriented technology has many benefits:

- **Ease in software design** as you could think in the problem space rather than the machine's bits and bytes. You are dealing with high-level concepts and abstractions. Ease in design leads to more productive software development.
- **Ease in software maintenance:** object-oriented software are easier to understand, therefore easier to test, debug, and maintain.
- **Reusable software:** you don't need to keep re-inventing the wheels and re-write the same functions for different situations. The fastest and safest way of developing a new application is to reuse existing codes - fully tested and proven codes.

Classes & Instances

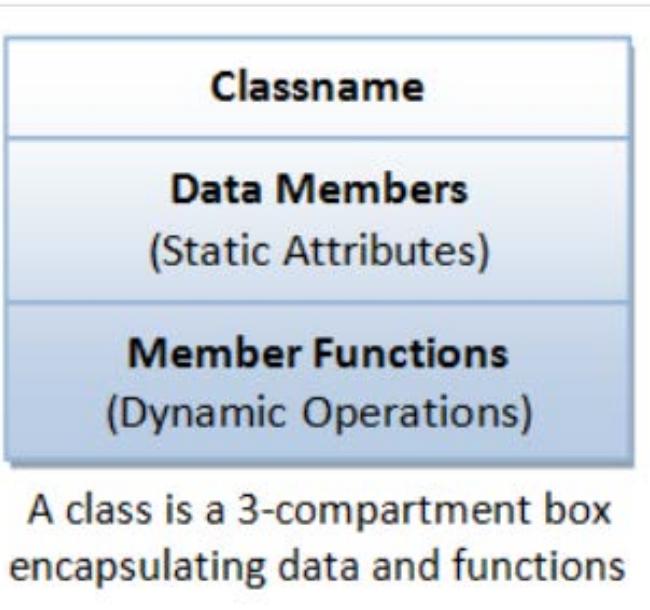
- **Class:** A class is a definition of objects of the same kind.
- In other words, a class is a blueprint, template, or prototype that defines and describes the static attributes and dynamic behaviors common to all objects of the same kind.
- Classes also determine the forms of objects. The data and methods contained in a class are known as class members. **A class is a user-defined data type.** To access the class members, we use an instance of the class. You can see a class as a blueprint for an object.
- **Instance:** An instance is a realization of a particular item of a class.
- In other words, an instance is an instantiation of a class. All the instances of a class have similar properties, as described in the class definition. For example, you can define a class called "Student" and create three instances of the class "Student" for "Peter", "Paul" and "Pauline".
- The term "**object**" usually refers to **instance**. But it is often used quite loosely, which may refer to a class or an instance.



In the above figure, we have a single house prototype. From this prototype, we have created two houses with different features.

A Class is a 3-Compartment Box encapsulating Data and Functions

- A class can be visualized as a three-compartment box, as illustrated:
 - **Classname (or identifier):** identifies the class.
 - **Data Members or Variables (or attributes, states, fields):** contains the static attributes of the class.
 - **Member Functions (or methods, behaviors, operations):** contains the dynamic operations of the class.
- In other words, a **class encapsulates the static attributes (data) and dynamic behaviors (operations that operate on the data) in a box.**
- **Class Members:** The data members and member functions are collectively called class members.



Classname (Identifier)	Student	Circle
Data Member (Static attributes)	name grade	radius color
Member Functions (Dynamic Operations)	getName() printGrade()	getRadius() getArea()

SoccerPlayer	Car
name number xLocation yLocation	plateNumber xLocation yLocation speed
run() jump() kickBall()	move() park() accelerate()

Examples of classes

- **Unified Modeling Language (UML) Class and Instance Diagrams:** The class diagrams are drawn according to the UML notations.
- A **class** is represented as a 3-compartment box, containing name, data members (variables), and member functions, respectively. **classname** is shown in bold and centralized.
- An **instance (object)** is also represented as a 3-compartment box, with instance name shown as **instanceName:Classname** and underlined.

Classname	<u>paul:Student</u>	<u>peter:Student</u>
Data Members	name="Paul Lee" grade=3.5	name="Peter Tan" grade=3.9
Member Functions	getName() printGrade()	getName() printGrade()

Two instances of the **Student** class

- **Class Naming Convention:** A classname shall be a noun or a noun phrase made up of several words. All the words shall be initial-capitalized (camel-case).
- Use a singular noun for classname. Choose a meaningful and self-descriptive classname.

```
class Circle {      // classname  
private:  
    double radius;    // Data members (variables)  
    string color;  
public:  
    double getRadius(); // Member functions  
    double getArea();  
}
```

```
class SoccerPlayer { // classname  
private:  
    int number;      // Data members (variables)  
    string name;  
    int x, y;  
public:  
    void run();      // Member functions  
    void kickBall();  
}
```

Class Definition

Creating Instances of a Class

- To create an instance of a class, you have to:
- Declare an instance identifier (name) of a particular class.
- **Invoke a constructor** to construct the instance (i.e., allocate storage for the instance and initialize the variables).
- For examples, suppose that we have a class called **Circle**, we can create instances of Circle as follows:

```
// Construct 3 instances of the class Circle: c1, c2, and c3
```

```
Circle c1(1.2, "red"); // radius, color
```

```
Circle c2(3.4);      // radius, default color
```

```
Circle c3;          // default radius and color
```

- Alternatively, you can **invoke the constructor explicitly** using the following syntax:
 - **Circle c1 = Circle(1.2, "red"); // radius, color**
 - **Circle c2 = Circle(3.4); // radius, default color**
 - **Circle c3 = Circle(); // default radius and color**

Dot (.) Operator

- To reference a member of a object (data member or member function), you must:
- **Use the dot operator (.)** to reference the member, **instanceName.memberName**.
- For example, suppose that we have a class called Circle, with two data members (radius and color) and two functions (getRadius() and getArea()).
- We have created three instances of the class Circle, namely, c1, c2 and c3.

// Declare and construct instances c1 and c2 of the class Circle

Circle c1(1.2, "blue");

Circle c2(3.4, "green");

// Invoke member function via dot operator

cout << c1.getArea() << endl;

cout << c2.getArea() << endl;

// Reference data members via dot operator

c1.radius = 5.5;

c2.radius = 6.6;

```
class MyClass {    // The class
public:          // Access specifier
    int myNum;    // Attribute (int variable)
    string myString; // Attribute (string variable)
};

int main() {
    MyClass myObj; // Create an object of MyClass

    // Access attributes and set values
    myObj.myNum = 15;
    myObj.myString = "Some text";

    // Print attribute values
    cout << myObj.myNum << "\n";
    cout << myObj.myString;
    return 0;
}
```

```
class Car {    // Create a Car class with some attributes
public:
    string brand;
    string model;
    int year;
};

int main() {
    Car carObj1;    // Create an object of Car
    carObj1.brand = "BMW";
    carObj1.model = "X5";
    carObj1.year = 1999;

    Car carObj2;    // Create another object of Car
    carObj2.brand = "Ford";
    carObj2.model = "Mustang";
    carObj2.year = 1969;

    // Print attribute values
    cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
    cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
    return 0;
}
```

Data Members (Variables)

- A **data member (variable)** has a name (or identifier) and a type; and holds a value of that particular type. A data member can also be an instance of a certain class .
- **Data Member Naming Convention:** A data member name shall be a noun or a noun phrase made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case), e.g., **fontSize**, **roomNumber**, **xMax**, **yMin** and **xTopLeft**. Take note that variable name begins with an lowercase, while classname begins with an uppercase.

Member Functions

- A member function :
 - receives parameters from the caller,
 - performs the operations defined in the function body, and
 - returns a piece of result (or void) to the caller.
- **Member Function Naming Convention:** A function name shall be a verb, or a verb phrase made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case). For example, **getRadius()**, **getParameterValues()**.
- Take note that data member name is a noun (**denoting a static attribute**), while function name is a verb (**denoting an action**). They have the same naming convention. Functions take arguments in parentheses (possibly zero argument with empty parentheses), but variables do not.

```
#include <bits/stdc++.h>
using namespace std;

class Geeks {
public: // Access specifier
    string geekname; // Data Members
    void printname() // Member Functions()
    {
        cout << "Geekname is: " << geekname;
    }
};

int main() {
    Geeks obj1; // Declare an object of class geeks
    obj1.geekname = "Abhi"; // accessing data member
    obj1.printname(); // accessing member function
    return 0;
}
```

```
#include <iostream>
using namespace std;
class Geeks {
public:
    string geekname;
    int id;
void printname(); // printname is not defined inside
class definition
// printid is defined inside class definition
void printid() {
    cout << "Geek id is: " << id;
}
};

void Geeks::printname() { // Definition of
printname using scope resolution operator ::

    cout << "Geekname is: " << geekname;
}
```

```
int main() {
    Geeks obj1;
    obj1.geekname = "xyz";
    obj1.id=15;
    obj1.printname(); // call printname()
    cout << endl;
    obj1.printid(); // call printid()
    return 0;
}
```

```
class MyClass {      // The class
public:        // Access specifier
    void myMethod() { // Method/function defined inside the class
        cout << "Hello World!";
    }
};

int main() {
    MyClass myObj;    // Create an object of MyClass
    myObj.myMethod(); // Call the method
    return 0;
}
```

```
class MyClass {    // The class
public:        // Access specifier
    void myMethod(); // Method/function declaration
};
```

// Method/function definition outside the class

```
void MyClass::myMethod() {
    cout << "Hello World!";
}
```

```
int main() {
```

```
    MyClass myObj; // Create an object of MyClass
```

```
    myObj.myMethod(); // Call the method
```

```
    return 0;
}
```

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

class Car {
public:
    int speed(int maxSpeed);
};

int Car::speed(int maxSpeed) {
    return maxSpeed;
}

int main() {
    Car myObj; // Create an object of Car
    cout << myObj.speed(200); // Call the method with an argument
    return 0;
}
```

An OOP Example

Class Definition

Circle
-radius:double=1.0
-color:String="red"
+getRadius():double
+getColor():String
+getArea():double

Instances

<u>c1:Circle</u>
-radius=2.0
-color="blue"
+getRadius()
+getColor()
+getArea()

<u>c2:Circle</u>
-radius=2.0
-color="red"
+getRadius()
+getColor()
+getArea()

<u>c3:Circle</u>
-radius=1.0
-color="red"
+getRadius()
+getColor()
+getArea()

- A class called Circle is to be defined as illustrated in the class diagram.
- It contains two data members: **radius** (of type double) and **color** (of type String); and three member functions: **getRadius()**, **getColor()**, and **getArea()**.
- Three instances of Circles called **c1**, **c2**, and **c3** shall then be constructed with their respective data members, as shown in the instance diagrams.
- In this example, we shall keep all the codes in a single source file called **CircleAIO.cpp**.

```
#include <iostream> // using IO functions          /* The Circle class (All source codes in one file) (CircleAIO.cpp) */  
#include <string> // using string  
using namespace std;  
  
class Circle {  
  
private:  
    double radius; // Data member (Variable)  
    string color; // Data member (Variable)  
  
public:  
    Circle(double r = 1.0, string c = "red") { // Constructor with default values for data members  
        radius = r;  
        color = c;  
    }  
    double getRadius() { // Member function (Getter)  
        return radius;  
    }  
    string getColor() { // Member function (Getter)  
        return color;  
    }  
    double getArea() { // Member function  
        return radius*radius*3.1416;}  
}; // need to end the class declaration with a semi-colon
```

```
int main() {  
    // Construct a Circle instance, object  
  
    Circle c1(1.2, "blue");  
  
    cout << "Radius=" << c1.getRadius() << " Area=" << c1.getArea() << endl;  
  
    // Construct another Circle instance  
  
    Circle c2(3.4); // default color  
  
    cout << "Radius=" << c2.getRadius() << " Area=" << c2.getArea() << endl;  
  
    // Construct a Circle instance using default no-arg constructor  
  
    Circle c3; // default radius and color  
  
    cout << "Radius=" << c3.getRadius() << " Area=" << c3.getArea() << endl;  
  
    return 0;  
}
```

Constructors

- A constructor is a special function that has the function name same as the classname.
In the above Circle class, we define a constructor as follows:

// Constructor has the same name as the class

```
Circle(double r = 1.0, string c = "red") {  
    radius = r;  
    color = c;  
}
```

- A constructor is used to construct and initialize all the data members.
- A constructor in C++ is a special method that is automatically called when an object of a class is created.
- To create a new instance of a class, you need to declare the name of the instance and *invoke the constructor*. For example,

```
Circle c1(1.2, "blue");
```

```
Circle c2(3.4); // default color
```

```
Circle c3; // default radius and color  
// Take note that there is no empty bracket ()
```

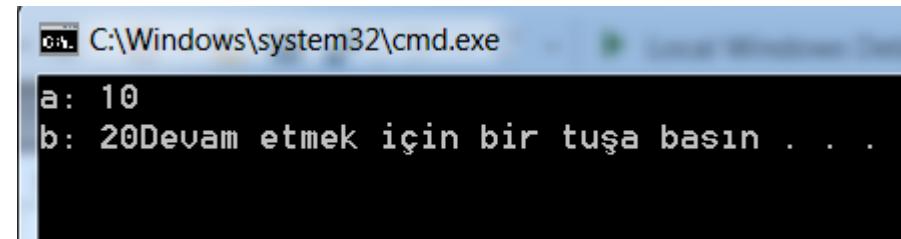
```
class MyClass { // The class
public:      // Access specifier
MyClass() { // Constructor
    cout << "Hello World!";
}
};
```

```
int main() {
    MyClass myObj; // Create an object of MyClass (this will call the constructor)
    return 0;
}
```

// Cpp program to illustrate the concept of Constructors

```
#include <iostream>
using namespace std;
class construct {
public:
    int a, b;
construct() { // Default Constructor
    a = 10;
    b = 20;
}
};
```

```
int main() {
// Default constructor called automatically when the object is created
construct c;
cout << "a: " << c.a << endl
    << "b: " << c.b;
return 1;
}
```



```
// CPP program to illustrate parameterized constructors
#include <iostream>      using namespace std;
class Point {
private:
    int x, y;
public:
// Parameterized Constructor
Point(int x1, int y1)  {
    x = x1;
    y = y1;
}
int getX()  {
    return x;
}
int getY()  {
    return y;
}
};
```

```
int main() {
// Constructor called
Point p1(10, 15);
// Access values assigned by constructor
cout << "p1.x = " << p1.getX() << ","
p1.y = " << p1.getY();
return 0;
}
```

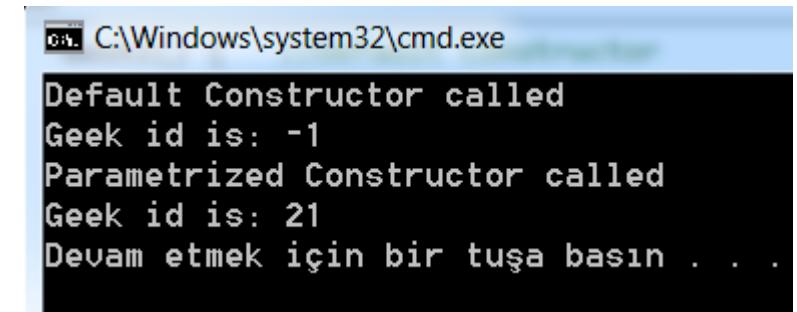


```
// C++ program to demonstrate constructors
```

```
#include <iostream>
using namespace std;
class Geeks {
public:
    int id;
    Geeks() { //Default Constructor
        cout << "Default Constructor called" << endl;
        id=-1;
    }
    //Parametrized Constructor
    Geeks(int x)
    {
        cout << "Parametrized Constructor called" << endl;
        id=x;
    }
};
```

```
int main() {
    // obj1 will call Default Constructor
    Geeks obj1;
    cout << "Geek id is: " <<obj1.id << endl;

    // obj2 will call Parametrized Constructor
    Geeks obj2(21);
    cout << "Geek id is: " <<obj2.id << endl;
    return 0;
}
```



```
C:\Windows\system32\cmd.exe
Default Constructor called
Geek id is: -1
Parametrized Constructor called
Geek id is: 21
Devam etmek için bir tuşa basın . . .
```

```
class Car {    // The class
public:      // Access specifier
    string brand; // Attribute
    string model; // Attribute
    int year;    // Attribute
    Car(string x, string y, int z) { // Constructor with parameters
        brand = x;
        model = y;
        year = z;
    }
};

int main() {
    Car carObj1("BMW", "X5", 1999); // Create Car objects and call the constructor with different
values
    Car carObj2("Ford", "Mustang", 1969);
    // Print values
    cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
    cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
    return 0;
}
```

Just like functions, constructors can also be defined outside the class.

```
class Car {      // The class
    public:      // Access specifier
        string brand; // Attribute  string model; // Attribute  int year;   // Attribute
        Car(string x, string y, int z); // Constructor declaration
    };
Car::Car(string x, string y, int z) { // Constructor definition outside the class
    brand = x;
    model = y;
    year = z;
}
int main() {
    Car carObj1("BMW", "X5", 1999); // Create Car objects and call the constructor with different
values
    Car carObj2("Ford", "Mustang", 1969);
    // Print values
    cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
    cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
    return 0;
}
```

- A constructor function is different from an ordinary function in the following aspects:
- The name of the constructor is the same as the classname.
- Constructor has no return type (or implicitly returns void). Hence, no return statement is allowed inside the constructor's body.
- Constructor can only be invoked once to initialize the instance constructed.
- You cannot call the constructor afterwards in your program.
- Constructors are not inherited (to be explained later).

```
#include <iostream>
using namespace std;
class HelloWorld{
public:
HelloWorld() { //Constructor
    cout<<"Constructor is called"\;
}
~HelloWorld() { //Destructor
    cout<<"Destructor is called"\;
}
void display(){//Member function
    cout<<"Hello World!"\;
}
int main(){
HelloWorld obj; //Object created
obj.display(); //Member function called
return 0;
}
```

- A destructor is a special member function that works just opposite to constructor.

Unlike constructors that are used for initializing an object, destructors destroy (or delete) the object.

When does the destructor get called?

A destructor is automatically called when:

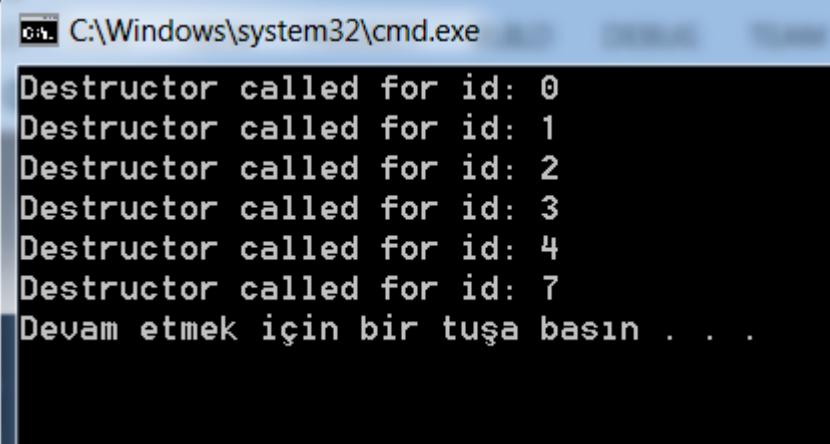
- 1) The program finished execution.
- 2) When a scope (the {} parenthesis) containing local variable ends.
- 3) When you call the delete operator.

- Destructor is another special member function that is called by the compiler when the scope of the object ends.

```
#include <iostream>    using namespace std;
class Geeks {
public:
    int id;
~Geeks() { //Definition for Destructor
    cout << "Destructor called for id: " << id << endl;
}
};

int main() {
    Geeks obj1;
    obj1.id=7;
    int i = 0;
    while ( i < 5 ) {
        Geeks obj2;
        obj2.id=i;
        i++;
    } // Scope for obj2 ends here
    return 0;
} // Scope for obj1 ends here
```

Destructors



```
C:\Windows\system32\cmd.exe
Destructor called for id: 0
Destructor called for id: 1
Destructor called for id: 2
Destructor called for id: 3
Destructor called for id: 4
Destructor called for id: 7
Devam etmek için bir tuşa basın . . .
```

- In C++, you can specify the default value for the trailing arguments of a function (including constructor) in the function header. For example,

```
#include <iostream>          /* Test function default arguments (TestFnDefault.cpp) */  
using namespace std;  
// Function prototype  
int sum(int n1, int n2, int n3 = 0, int n4 = 0, int n5 = 0);  
  
int main() {  
    cout << sum(1, 1, 1, 1, 1) << endl; // 5  
    cout << sum(1, 1, 1, 1) << endl; // 4  
    cout << sum(1, 1, 1) << endl; // 3  
    cout << sum(1, 1) << endl; // 2  
    // cout << sum(1) << endl; // error: too few arguments  
}  
  
// Function definition// The default values shall be specified in function prototype,  
// not the function implementation  
int sum(int n1, int n2, int n3, int n4, int n5) {  
    return n1 + n2 + n3 + n4 + n5;  
}
```

Access Specifiers

- In C++, there are three access specifiers:
- **public** - members are accessible from outside the class
- **private** - members cannot be accessed (or viewed) from outside the class
- **protected** - members cannot be accessed from outside the class, however, they can be accessed in inherited classes. You will learn more about Inheritance later.

```
class MyClass { // The class  
public:    // Access specifier  
    // class members goes here  
};
```

```
class MyClass {  
    public: // Public access specifier  
        int x; // Public attribute  
    private: // Private access specifier  
        int y; // Private attribute  
};
```

```
int main() {  
    MyClass myObj;  
    myObj.x = 25; // Allowed (public)  
    myObj.y = 50; // Not allowed (private)  
    return 0;  
}
```

"public" vs. "private" Access Control Modifiers

- An access control modifier **can be used to control the visibility of a data member** or a member function within a class. We begin with the following two access control modifiers:
- **public:** The member (data or function) is accessible and available to all in the system.
- **private:** The member (data or function) is accessible and available within this class only.
- For example, in the above Circle definition, the data member **radius is declared private**. As the result, radius is accessible inside the Circle class, but NOT outside the class.
- In other words, **you cannot use "c1.radius" to refer to c1's radius in main()**.
- The function `getRadius()` is declared public in the Circle class. Hence, it can be invoked in the `main()`.
- UML Notation: In UML notation, public members are denoted with a "+", while private members with a "-" in the class diagram.

- By default, all members of a class are private if you don't specify an access specifier:

- `class MyClass {`
- `int x; // Private attribute`
- `int y; // Private attribute`
- `};`

// C++ program to demonstrate private access modifier

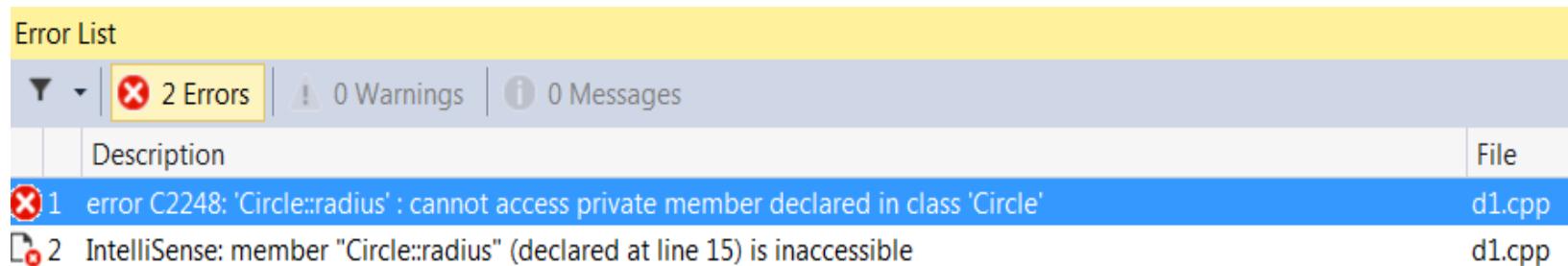
```
#include<iostream>
using namespace std;
class Circle {
    // private data member
private:
    double radius;

    // public member function
public:
    double compute_area()
    {   // member function can access private
        // data member radius
        return 3.14*radius*radius;
    }
};
```

```
int main()
{
    // creating object of the class
    Circle obj;

    // trying to access private data member
    // directly outside the class
    obj.radius = 1.5;

    cout << "Area is:" << obj.compute_area();
    return 0;
}
```



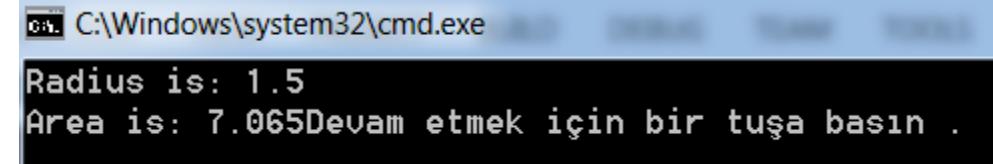
/ C++ program to demonstrate private access modifier

```
#include<iostream>
using namespace std;
class Circle {
private: // private data member
    double radius;
public: // public member function
    void compute_area(double r)
    { // member function can access private
        // data member radius
        radius = r;
        double area = 3.14*radius*radius;
        cout << "Radius is: " << radius << endl;
        cout << "Area is: " << area;
    }
};
```

```
int main()
{
    // creating object of the class
    Circle obj;

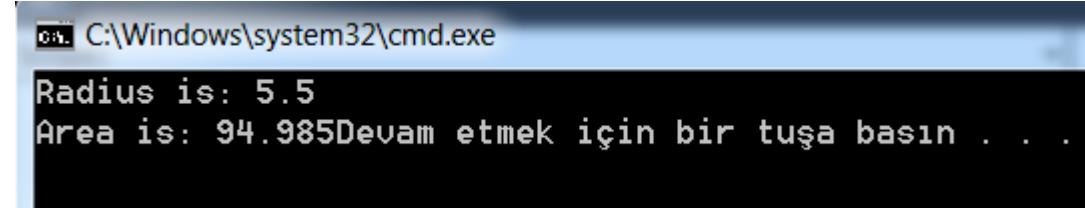
    // trying to access private data member
    // directly outside the class
    obj.compute_area(1.5);

    return 0;
}
```



// C++ program to demonstrate public access modifier

```
#include<iostream>
using namespace std;
class Circle { // class definition
public:
    double radius;
    double compute_area() {
        return 3.14*radius*radius;
    }
};
int main() {
    Circle obj;
    obj.radius = 5.5; // accessing public datamember outside class
    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
    return 0;
}
```



Information Hiding and Encapsulation

- A class **encapsulates the static attributes and the dynamic behaviors** into a "3-compartment box".
- Once a class is defined, you can seal up the "box" and put the "box" on the shelf for others to use and reuse.
- Anyone can pick up the "box" and use it in their application.
- Data member of a class are typically hidden from the outside world, with private access control modifier.
- This follows the **principle of information hiding**.
- That is, **objects communicate with each others using well-defined interfaces (public functions)**.
- Objects are not allowed to know the implementation details of others. **The implementation details are hidden or encapsulated within the class.**
Information hiding facilitates reuse of the class.
- Rule of Thumb: Do not make any data member public, unless you have a good reason.

Getters and Setters

- To allow others to read the value of a private data member says **xxx**, you shall provide **a get** function (or getter or accessor function) called **getXxx()**.
- It can process the data and limit the view of the data others will see. Getters shall not modify the data member.
- To allow other classes to modify the value of a private data member says **xxx**, you shall provide **a set** function (or setter or mutator function) called **setXxx()**.
- A setter could provide data validation (such as range checking), and transform the raw data into the internal representation.

- For example, in our Circle class:
- The data members radius and color are declared private.
- That is to say, they are only available within the Circle class and not visible outside the Circle class - including main().
- You cannot access the private data members radius and color from the main() directly - via says **c1.radius or c1.color**.
- The Circle class provides two public accessor functions, namely, getRadius() and getColor(). These functions are declared public.
- The main() can invoke these public accessor functions to retrieve the radius and color of a Circle object, via says c1.getRadius() and c1.getColor().
- There is no way you can change the radius or color of a Circle object, after it is constructed in main().
- You cannot issue statements such as c1.radius = 5.0 to change the radius of instance c1, as radius is declared as private in the Circle class and is not visible to other including main().
- If the designer of the Circle class permits the change the radius and color after a Circle object is constructed, he has to provide the appropriate setter, e.g.,

```
// Setter for color  
void setColor(string c) {  
    color = c;  
}
```

```
// Setter for radius  
void setRadius(double r) {  
    radius = r;  
}
```

```
#include<iostream> // c++ program to explain Encapsulation  
using namespace std;  
class Encapsulation {  
private:  
  
    int x; // data hidden from outside world  
public:  
void set(int a) { // function to set value of // variable x  
    x =a;  
}  
int get() { // function to return value of // variable x  
    return x;  
}  
};
```

```
int main() {  
    Encapsulation obj;  
    obj.set(5);  
    cout<<obj.get();  
    return 0;  
}
```

```
#include <iostream>
using namespace std;
class Employee {
    private:
        int salary; // Private attribute
    public:
        void setSalary(int s) { // Setter
            salary = s;
        }
        int getSalary() { // Getter
            return salary;
        }
    };
int main() {
    Employee myObj;
    myObj.setSalary(50000);
    cout << myObj.getSalary();
    return 0;
}
```

- One of the main usage of keyword this is to resolve ambiguity between the names of data member and function parameter.

```
class Circle {  
private:  
    double radius;      // Member variable called "radius"  
    .....  
public:  
    void setRadius(double radius) { // Function's argument also called "radius"  
        this->radius = radius;  
        // "this.radius" refers to this instance's member variable    // "radius" resolved to the function's argument.  
    }  
    .....  
}
```

Keyword "this"

- In the above codes, there are two identifiers called radius - a data member and the function parameter.
- This causes naming conflict. To resolve the naming conflict, you could name the function parameter r instead of radius.
- However, radius is more approximate and meaningful in this context.
- You can use keyword this to resolve this naming conflict. "this->radius" refers to the data member; while "radius" resolves to the function parameter.
- "this" is actually a pointer to this object.

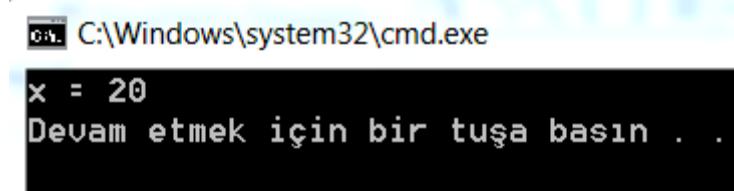
- Alternatively, you could use a prefix (such as m_) or suffix (such as _) to name the data members to avoid name crashes. For example,

```
class Circle {  
private:  
    double m_radius; // or radius_  
.....  
public:  
    void setRadius(double radius) {  
        m_radius = radius; // or radius_ = radius  
    }  
.....  
}
```

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

/* local variable is same as a member's name */
class Test
{
private:
    int x;
public:
    void setX (int x)
    {
        // The 'this' pointer is used to retrieve the object's x
        // hidden by the local variable 'x'
        this->x = x;
    }
    void print() { cout << "x = " << x << endl; }
};
```

```
int main()
{
    Test obj;
    int x = 20;
    obj.setX(x);
    obj.print();
    return 0;
}
```



```
#include <iostream>
using namespace std;
class Box {
public: // Constructor definition
Box(double l = 2.0, double b = 2.0, double h = 2.0)
{
cout << "Constructor called." << endl;
length = l;
breadth = b;
height = h;
}
double Volume() {
return length * breadth * height;
}
int compare(Box box) {
return this->Volume() > box.Volume();
}
private:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
};
int main(void) {
Box Box1(3.3, 1.2, 1.5); // Declare box1
Box Box2(8.5, 6.0, 2.0); // Declare box2
if (Box1.compare(Box2)) {
cout << "Box2 is smaller than Box1"
<< endl;
}
else {
cout << "Box2 is equal to or larger
than Box1" << endl;
}
return 0;
}
```

"const" Member Functions

- A const member function, identified by a **const** keyword at the end of the member function's header, cannot modify any data member of this object. For example,

```
double getRadius() const { // const member function  
    radius = 0;  
    // error: assignment of data-member 'Circle::radius' in read-only structure  
    return radius;  
}
```


Convention for Getters/Setters and Constructors

- The constructor, getter and setter functions for a private data member called `xxx` of type `T` in a class `Aaa` have the following conventions:

class Aaa {

private:

T xxx; // A private variable named xxx of type T

public:

Aaa(T x) { xxx = x; } // Constructor

Aaa(T xxx) { this->xxx = xxx; } // OR

Aaa(T xxx) : xxx(xxx) { } // OR using member initializer list

// A getter for variable xxx of type T receives no argument and return a value of type T

T getXxx() const { return xxx; }

// A setter for variable xxx of type T receives a parameter of type T and return void

void setXxx(T x) { xxx = x; }

void setXxx(T xxx) { this->xxx = xxx; }

}

- For a bool variable xxx, the getter shall be named isXxx(), instead of getXxx(), as follows:

private:

```
// Private boolean variable  
bool xxx;
```

public:

```
// Getter  
bool isXxx() const { return xxx; }
```

```
// Setter
```

```
void setXxx(bool x) { xxx = x; }
```

```
// OR
```

```
void setXxx(bool xxx) { this->xxx = xxx; }
```

Default Constructor

- A default constructor is a constructor with no parameters, or having default values for all the parameters. For example, Circle's constructor can be served as default constructor with all the parameters default.

Circle c1; // Declare c1 as an instance of Circle, and invoke the default constructor

Circle c1(); // Error!

// (This declares c1 as a function that takes no parameter and returns a Circle instance)

- If C++, if you did not provide ANY constructor, the compiler automatically provides a default constructor that does nothing. That is,

ClassName::ClassName() {} // Take no argument and do nothing

- Compiler will not provide a default constructor if you define any constructor(s).
- If all the constructors you defined require arguments, invoking no-argument default constructor results in error.
- This is to allow class designer to make it impossible to create an uninitialized instance, by NOT providing an explicit default constructor.

Constructor's Member Initializer List

- Instead of initializing the private data members inside the body of the constructor, as follows:

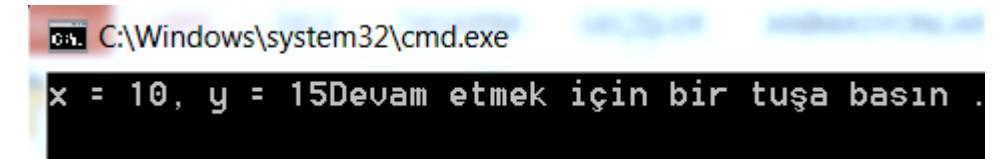
```
Circle(double r = 1.0, string c = "red") {  
    radius = r;  
    color = c;  
}
```

- We can use an alternate syntax called member initializer list as follows:

```
Circle(double r = 1.0, string c = "red") : radius(r), color(c) { }
```

- Member initializer list is placed after the constructor's header, separated by a colon (:).
- Each initializer is in the form of data_member_name(parameter_name).
- For fundamental type, it is equivalent to data_member_name = parameter_name.
- For object, the constructor will be invoked to construct the object. The constructor's body (empty in this case) will be run after the completion of member initializer list.
- It is recommended to use member initializer list to initialize all the data members, as it is often more efficient than doing assignment inside the constructor's body.

```
#include<iostream>      using namespace std;
class Point {
private:
    int x;
    int y;
public:
    Point(int i = 0, int j = 0) :x(i), y(j) {}
    /* The above use of Initializer list is optional as the constructor can also be written as:
    Point(int i = 0, int j = 0) {
        x = i;
        y = j;
    }
    int getX() const { return x; }
    int getY() const { return y; }
};
int main() {
    Point t1(10, 15);
    cout << "x = " << t1.getX() << ", ";
    cout << "y = " << t1.getY();
    return 0;
}
```



Destructor

- A destructor, similar to constructor, is a special function that has the same name as the classname, with a prefix ~, e.g., ~Circle().
- Destructor is called implicitly when an object is destroyed.
- If you do not define a destructor, the compiler provides a default, which does nothing.

```
class MyClass {  
public:  
    // The default destructor that does nothing  
    ~MyClass() {}  
.....  
}
```

Copy Constructor

- A copy constructor constructs a new object by copying an existing object of the same type. In other words, a **copy constructor takes an argument, which is an object of the same class**.
- If you do not define a copy constructor, the compiler provides a default which copies all the data members of the given object. For example,

```
Circle c4(7.8, "blue");
```

```
cout << "Radius=" << c4.getRadius() << " Area=" << c4.getArea() << " Color=" << c4.getColor()  
<< endl;
```

```
// Radius=7.8 Area=191.135 Color=blue
```

// Construct a new object by copying an existing object via the so-called default copy constructor

```
Circle c5(c4);
```

```
cout << "Radius=" << c5.getRadius() << " Area=" << c5.getArea() << " Color=" << c5.getColor()  
<< endl;
```

```
// Radius=7.8 Area=191.135 Color=blue
```

- The copy constructor is particularly important. When an object is passed into a function by value, the copy constructor will be used to make a clone copy of the argument.

What is a copy constructor?

- A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

ClassName (const ClassName &old_obj);

```
#include<iostream>    using namespace std;
class Point {
private:
    int x, y;
public:
    Point(int x1, int y1) { x = x1; y = y1; }
    Point(const Point &p2) {x = p2.x; y = p2.y; } // Copy constructor
    int getX()      { return x; }
    int getY()      { return y; }
};
int main() {
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY(); // Let us access values assigned
by constructors
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
    return 0;
}
```

Advanced Notes

- Pass-by-value for object means calling the copy constructor.
- To avoid the overhead of creating a clone copy, it is usually better to pass-by-reference-to-const, which will not have side effect on modifying the caller's object.
- The copy constructor has the following signature:

```
class MyClass {  
private:  
    T1 member1;  
    T2 member2;  
public:  
    // The default copy constructor which constructs an object via memberwise copy  
    MyClass(const MyClass & rhs) {  
        member1 = rhs.member1;  
        member2 = rhs.member2;  
    }  
    ....  
}
```

Copy Assignment Operator (=)

- The compiler also provides a default assignment operator (=), which can be used to assign one object to another object of the same class via memberwise copy. For example, using the Circle class defined earlier,

```
Circle c6(5.6, "orange"), c7;  
cout << "Radius=" << c6.getRadius() << " Area=" << c6.getArea()  
    << " Color=" << c6.getColor() << endl;  
    // Radius=5.6 Area=98.5206 Color=orange  
cout << "Radius=" << c7.getRadius() << " Area=" << c7.getArea()  
    << " Color=" << c7.getColor() << endl;  
    // Radius=1 Area=3.1416 Color=red (default constructor)
```

```
c7 = c6; // memberwise copy assignment  
cout << "Radius=" << c7.getRadius() << " Area=" << c7.getArea()  
    << " Color=" << c7.getColor() << endl;  
    // Radius=5.6 Area=98.5206 Color=orange
```

Separating Header and Implementation

- For better software engineering, it is recommended that the class declaration and implementation be kept in 2 separate files: declaration is a header file ".h"; while implementation in a ".cpp".
- This is known as separating the public interface (header declaration) and the implementation.
- Interface is defined by the designer, implementation can be supplied by others.
- While the interface is fixed, different vendors can provide different implementations.
- Furthermore, only the header files are exposed to the users, the implementation can be provided in an object file ".o" (or in a library).
- The source code needs not given to the users.

Example: The Circle Class

- Instead of putting all the codes in a single file. We shall "separate the interface and implementation" by placing the codes in 3 files.

Circle
-radius:double = 1.0
-color:string = "red"
+Circle(radius:double,color:string)
+getRadius():double
+setRadius(radius:double):void
+getColor():string
+setColor(color:string):void
+getArea():double

- Circle.h: defines the public interface of the Circle class.
- Circle.cpp: provides the implementation of the Circle class.
- TestCircle.cpp: A test driver program for the Circle class.

```
• /* The Circle class Header (Circle.h) */  
#include <string> // using string  
using namespace std;  
  
class Circle {// Circle class declaration  
private: // Accessible by members of this class only  
    double radius; // private data members (variables)  
    string color;  
public: // Accessible by ALL  
    // Declare prototype of member functions // Constructor with default values  
    Circle(double radius = 1.0, string color = "red");  
    // Public getters & setters for private data members  
    double getRadius() const;  
    void setRadius(double radius);  
    string getColor() const;  
    void setColor(string color);  
    // Public member Function  
    double getArea() const;  
};
```

```
#include "Circle.h" // user-defined header in the same directory

Circle::Circle(double r, string c) {// Constructor// default values shall only be specified in the declaration,// cannot be repeated in definition
    radius = r;      color = c;
}

/* The Circle class Implementation (Circle.cpp) */

double Circle::getRadius() const // Public getter for private data member radius
{
    return radius;
}

void Circle::setRadius(double r) // Public setter for private data member radius
{
    radius = r;
}

string Circle::getColor() const // Public getter for private data member color
{
    return color;
}

void Circle::setColor(string c) // Public setter for private data member color
{
    color = c;
}

double Circle::getArea() const // A public member function
{
    return radius*radius*3.14159265;
}
```

```
/* A test driver for the Circle class (TestCircle.cpp) */
```

```
#include <iostream>
#include "Circle.h" // using Circle class
using namespace std;
int main() {
    Circle c1(1.2, "red"); // Construct an instance of Circle c1
    cout << "Radius=" << c1.getRadius() << " Area=" << c1.getArea()
        << " Color=" << c1.getColor() << endl;
    c1.setRadius(2.1); // Change radius and color of c1
    c1.setColor("blue");
    cout << "Radius=" << c1.getRadius() << " Area=" << c1.getArea()
        << " Color=" << c1.getColor() << endl;
    // Construct another instance using the default constructor
    Circle c2;
    cout << "Radius=" << c2.getRadius() << " Area=" << c2.getArea()
        << " Color=" << c2.getColor() << endl;
    return 0;
}
```

Program Notes:

- The implementation file provides the definition of the functions, which are omitted from the declaration in the header file.

#include "Circle.h"

- The compiler searches the headers in double quotes (such as "Circle.h") in the current directory first, then the system's include directories. For header in angle bracket (such as <iostream>), the compiler does NOT search the current directory, but only the system's include directories. Hence, use double quotes for user-defined headers.

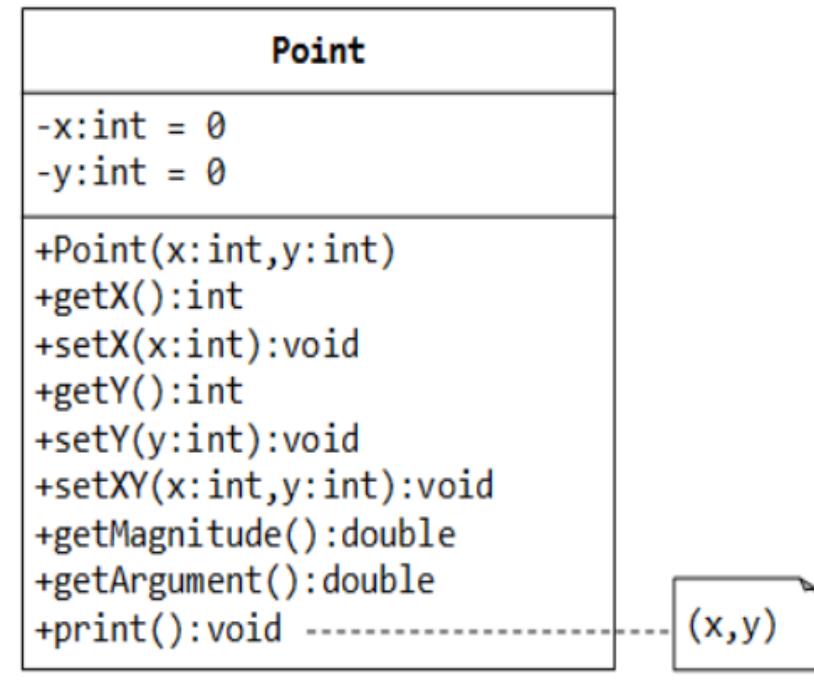
Circle::Circle(double r, string c) {

- You need to include the **className::** (**called class scope resolution operator**) in front of all the members names, so as to inform the compiler this member belong to a particular class.
- (Class Scope: Names defined inside a class have so-called class scope. They are visible within the class only. Hence, you can use the same name in two different classes. To use these names outside the class, the class scope resolution operator **className::** is needed.)
- You CANNOT place the default arguments in the implementation (they shall be placed in the header). For example,

Circle::Circle(double r = 1.0, string c = "red") { // error!

Example: The Point Class

- The Point class, as shown in the class diagram, models 2D points with x and y co-ordinates.
- In the class diagram, "-" denotes private member; "+" denotes public member. "= xxx" specifies the default value of a data member.
- The Point class contains the followings:
 - Private data members x and y (of type int), with default values of 0.
 - A constructor, getters and setters for private data member x and y.
 - A function setXY() to set both x and y coordinates of a Point.
 - A function print() which prints "(x,y)" of this instance.



- /* The Point class Header (Point.h) */

```
#ifndef POINT_H
#define POINT_H
class Point { // Point class declaration
private:
    int x; // private data members (variables)
    int y;
public:
    // Declare member function prototypes
    Point(int x = 0, int y = 0); // Constructor with default values
    int getX() const;
    void setX(int x);
    int getY() const;
    void setY(int y);
    void setXY(int x, int y);
    double getMagnitude() const;
    double getArgument() const;
    void print() const;
};
#endif
```

```
/* The Point class Implementation (Point.cpp) */
#include "Point.h" // user-defined header in the same directory
#include <iostream>
#include <cmath>
using namespace std;
// Constructor (default values can only be specified in the declaration)
Point::Point(int x, int y) : x(x), y(y) {} // Use member initializer list
int Point::getX() const { // Public getter for private data member x
    return x;
}
void Point::setX(int x) { // Public setter for private data member x
    this->x = x;
}
int Point::getY() const { // Public getter for private data member y
    return y;
}
```

```
void Point::setY(int y) { // Public setter for private data member y
    this->y = y;
}

void Point::setXY(int x, int y) { // Public member function to set both x and y
    this->x = x;
    this->y = y;
}

double Point::getMagnitude() const { // Public member function to return the magnitude
    return sqrt(x*x + y*y); // sqrt in <cmath>
}

double Point::getArgument() const { // Public member function to return the argument
    return atan2(y, x); // atan2 in <cmath>
}

void Point::print() const { // Public member function to print description about this point
    cout << "(" << x << "," << y << ")" << endl;
}
```

- /* A test driver for the Point class (TestPoint.cpp) */

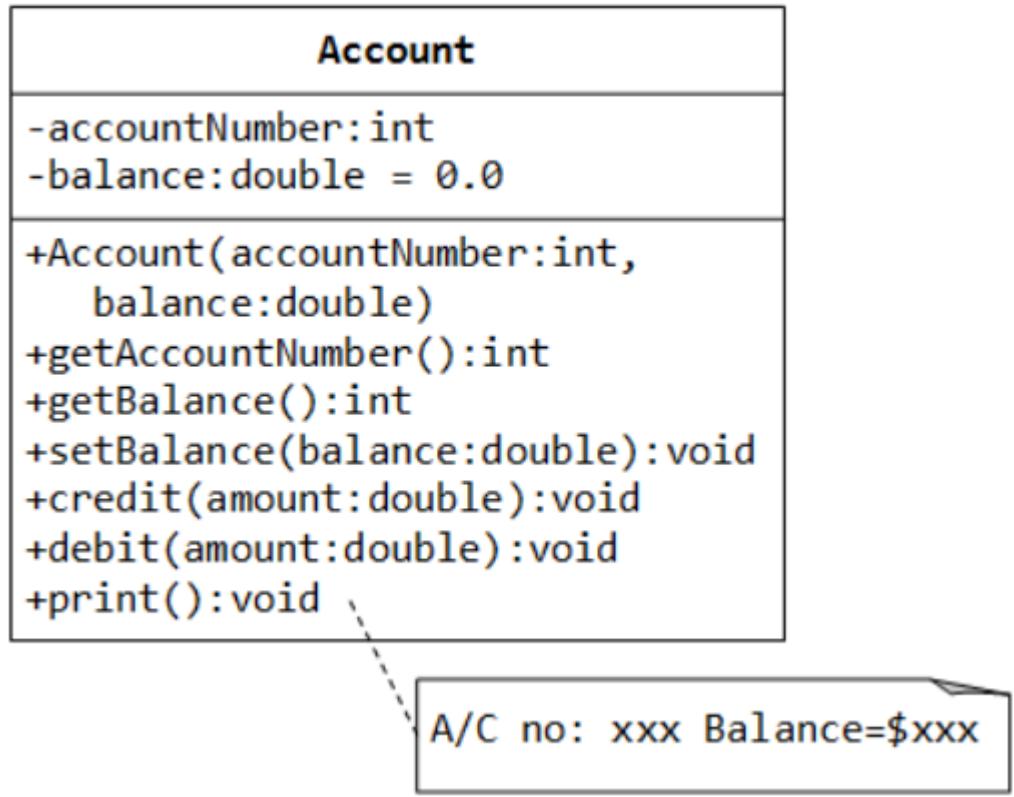
```
#include <iostream>
#include <iomanip>
#include "Point.h" // using Point class
using namespace std;

int main() {
    Point p1(3, 4); // Construct an instance of Point p1
    p1.print();
    cout << "x = " << p1.getX() << endl;
    cout << "y = " << p1.getY() << endl;
    cout << fixed << setprecision(2);
    cout << "mag = " << p1.getMagnitude() << endl;
    cout << "arg = " << p1.getArgument() << endl;
    p1.setX(6);
    p1.setY(8);
    p1.print();
    p1.setXY(1, 2);
    p1.print();

    Point p2; // Construct an instance of Point using default constructor
    p2.print();
}
```

The Account Class

- A class called Account, which models a bank account, is designed as shown in the class diagram. It contains:
 - Two private data members: **accountNumber (int)** and **balance (double)**, which maintains the current account balance.
 - Public functions **credit()** and **debit()**, which adds or subtracts the given amount from the balance, respectively. The **debit()** function shall print "amount withdrawn exceeds the current balance!" if amount is more than balance.
 - A public function **print()**, which shall print "A/C no: xxx Balance=xxx" (e.g., A/C no: 991234 Balance=\$88.88), with balance rounded to two decimal places.



- /* Header for Account class (Account.h) */

```
#ifndef ACCOUNT_H  
#define ACCOUNT_H
```

```
class Account {
```

```
private:
```

```
    int accountNumber;
```

```
    double balance;
```

```
public:
```

```
    Account(int accountNumber, double balance = 0.0);
```

```
    int getAccountNumber() const;
```

```
    double getBalance() const;
```

```
    void setBalance(double balance);
```

```
    void credit(double amount);
```

```
    void debit(double amount);
```

```
    void print() const;
```

```
};
```

```
#endif
```

- /* Implementation for the Account class (Account.cpp) */

```
#include <iostream>
#include <iomanip>
#include "Account.h"
using namespace std;

// Constructor
Account::Account(int no, double b) : accountNumber(no), balance(b) { }

// Public getter for private data member accountNumber
int Account::getAccountNumber() const {
    return accountNumber;
}

// Public getter for private data member balance
double Account::getBalance() const {
    return balance;
}
```

```
// Public setter for private data member balance
void Account::setBalance(double b) {
    balance = b;
}

// Adds the given amount to the balance
void Account::credit(double amount) {
    balance += amount;
}

// Subtract the given amount from the balance
void Account::debit(double amount) {
    if (amount <= balance) {
        balance -= amount;
    } else {
        cout << "Amount withdrawn exceeds the current balance!" << endl;
    }
}

// Print description for this Account instance
void Account::print() const {
    cout << fixed << setprecision(2);
    cout << "A/C no: " << accountNumber << " Balance=$" << balance << endl;
}
```

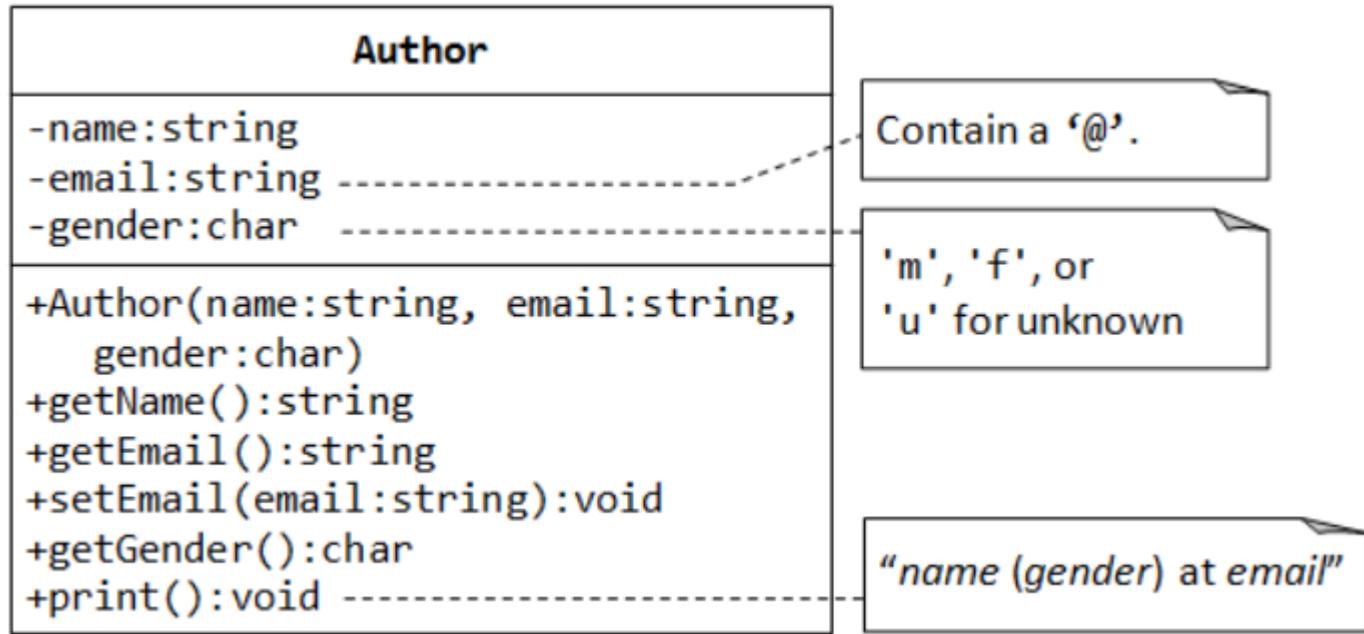
```
• /* Test Driver for Account class (TestAccount.cpp) */

#include <iostream>
#include "Account.h"
using namespace std;

int main() {
    Account a1(8111, 99.99);
    a1.print(); // A/C no: 8111 Balance=$99.99
    a1.credit(20);
    a1.debit(10);
    a1.print(); // A/C no: 8111 Balance=$109.99

    Account a2(8222); // default balance
    a2.print(); // A/C no: 8222 Balance=$0.00
    a2.setBalance(100);
    a2.credit(20);
    a2.debit(200); // Amount withdrawn exceeds the current balance!
    a2.print(); // A/C no: 8222 Balance=$120.00
    return 0;
}
```

The Author and Book Classes (for a Bookstore)



- Let's begin with a class called **Author**, designed as shown in the class diagram. It contains:
 - Three private data members: `name` (`string`), `email` (`string`), and `gender` (`char` of '`m`', '`f`' or '`u`' for unknown).
 - A constructor to initialize the `name`, `email` and `gender` with the given values. There are no default values for data members.
 - Getters for `name`, `email` and `gender`, and setter for `email`. There is no setter for `name` and `gender` as we assume that these attributes cannot be changed.
 - A `print()` member function that prints "`name (gender) at email`", e.g., "`Peter Jones (m) at peter@somewhere.com`".

- /* Header for the Author class (Author.h) */

```
#ifndef AUTHOR_H
#define AUTHOR_H
#include <string>
using namespace std;

class Author {
private:
    string name;
    string email;
    char gender; // 'm', 'f', or 'u' for unknown

public:
    Author(string name, string email, char gender);
    string getName() const;
    string getEmail() const;
    void setEmail(string email);
    char getGender() const;
    void print() const;
};

#endif
```

- /* Implementation for the Author class (Author.cpp) */

```
#include <iostream>
#include "Author.h"
using namespace std;
```

// Constructor, with input validation

```
Author::Author(string name, string email, char gender) {
    this->name = name;
    setEmail(email); // Call setter to check for valid email
    if (gender == 'm' || gender == 'f') {
        this->gender = gender;
    } else {
        cout << "Invalid gender! Set to 'u' (unknown)." << endl;
        this->gender = 'u';
    }
}
string Author::getName() const {
    return name;
}
```

```
string Author::getEmail() const {
    return email;
}

void Author::setEmail(string email) {
    // Check for valid email. Assume that a valid email contains // a '@' that is not the first nor last character.
    size_t atIndex = email.find('@');
    if (atIndex != string::npos && atIndex != 0 && atIndex != email.length()-1) {
        this->email = email;
    } else {
        cout << "Invalid email! Set to empty string." << endl;
        this->email = "";
    }
}

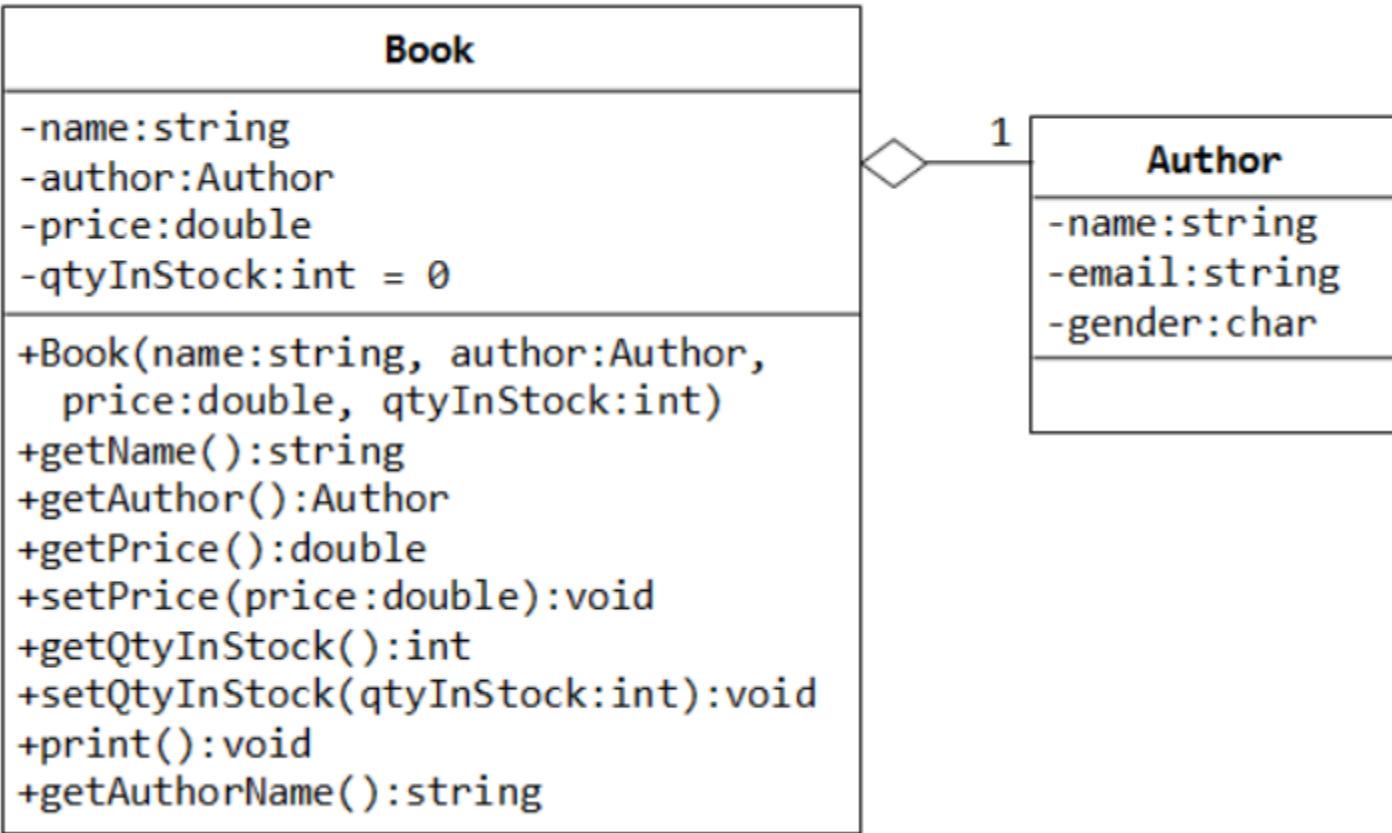
char Author::getGender() const {
    return gender;
}

void Author::print() const { // print in the format "name (gender) at email"
    cout << name << " (" << gender << ") at " << email << endl;
}
```

- /* Test Driver for the Author class (TestAuthor.cpp) */

```
#include "Author.h"
int main() {
    // Declare and construct an instance of Author
    Author peter("Peter Jones", "peter@somewhere.com", 'm');
    peter.print();
    // Peter Jones (m) at peter@somewhere.com
    peter.setEmail("peter@xyz.com");
    peter.print();
    // Peter Jones (m) at peter@xyz.com
    Author paul("Paul Jones", "@somewhere.com", 'n');
    // Invalid email! Set to empty string.
    // Invalid gender! Set to 'u' (unknown).
    paul.setEmail("paul@");
    // Invalid email! Set to empty string.
    paul.print();
    // Paul Jones (u) at
}
```


A Book is written by an Author - Using an "Object" Data Member



- Let's design a Book class. Assume that a book is written by one and only one author. The Book class (as shown in the class diagram) contains the following members:
- Four private data members: name (string), author (an instance of the class Author that we have created earlier), price (double), and qtyInStock (int, with default value of 0). The price shall be positive and the qtyInStock shall be zero or positive.
- Take note that data member author is an instance (object) of the class Author, instead of a fundamental types (such as int, double). In fact, name is an object of the class string too.
- The public getters and setters for the private data members. Take note that getAuthor() returns an object (an instance of class Author).
- A public member function print(), which prints "book-name" by author-name (gender) @ email".
- A public member function getAuthorName(), which returns the name of the author of this Book instance.
- The hollow diamond shape in the class diagram denotes aggregation (or has-a) association relationship. That is, a Book instance has one (and only one) Author instance as its component.

- /* Header for the class Book (Book.h) */

```
#ifndef BOOK_H
#define BOOK_H
#include <string>
#include "Author.h" // Use the Author class
using namespace std;
class Book {
private:
    string name;
    Author author; // data member author is an instance of class Author
    double price;
    int qtyInStock;
public:
    Book(string name, Author author, double price, int qtyInStock = 0);
    string getName() const; // To receive an instance of class Author as argument
    Author getAuthor() const; // Returns an instance of the class Author
    double getPrice() const;
    void setPrice(double price);
    int getQtyInStock() const;
    void setQtyInStock(int qtyInStock);
    void print() const;
    string getAuthorName() const;
};
#endif
```

```
• /* Implementation for the class Book (Book.cpp) */

#include <iostream>
#include "Book.h"
using namespace std;

// Constructor, with member initializer list to initialize the // component Author instance
Book::Book(string name, Author author, double price, int qtyInStock)
    : name(name), author(author) { // Must use member initializer list to construct object
setPrice(price); // Call setters to validate price and qtyInStock
    setQtyInStock(qtyInStock);
}

string Book::getName() const {
    return name;
}

Author Book::getAuthor() const {
    return author;
}

double Book::getPrice() const {
    return price;
}
```

```
void Book::setPrice(double price) // Validate price, which shall be positive
{
    if (price > 0) {
        this->price = price;
    } else {
        cout << "price should be positive! Set to 0" << endl;
        this->price = 0;
    }
}

int Book::getQtyInStock() const {
    return qtyInStock;
}

void Book::setQtyInStock(int qtyInStock) // Validate qtyInStock, which cannot be negative
{
    if (qtyInStock >= 0) {
        this->qtyInStock = qtyInStock;
    } else {
        cout << "qtyInStock cannot be negative! Set to 0" << endl;
        this->qtyInStock = 0;
    }
}

void Book::print() const // print in the format ""Book-name" by author-name (gender) at email"
{
    cout << "" << name << " by ";
    author.print();
}

string Book::getAuthorName() const // Return the author' name for this Book
{
    return author.getName(); // invoke the getName() on instance author
}
```

```
#include <iostream> /* Test Driver for the Book class (TestBook.cpp) */
#include "Book.h"
using namespace std;
int main() {
    // Declare and construct an instance of Author
    Author peter("Peter Jones", "peter@somewhere.com", 'm');
    peter.print(); // Peter Jones (m) at peter@somewhere.com
    // Declare and construct an instance of Book
    Book cppDummy("C++ for Dummies", peter, 19.99);
    cppDummy.setQtyInStock(88);
    cppDummy.print();
    // 'C++ for Dummies' by Peter Jones (m) at peter@somewhere.com
    cout << cppDummy.getQtyInStock() << endl; // 88
    cout << cppDummy.getPrice() << endl; // 19.99
    cout << cppDummy.getAuthor().getName() << endl; // "Peter Jones"
    cout << cppDummy.getAuthor().getEmail() << endl; // "peter@somewhere.com"
    cout << cppDummy.getAuthorName() << endl; // "Peter Jones"
    Book moreCpp("More C++ for Dummies", peter, -19.99);
    // price should be positive! Set to 0
    cout << moreCpp.getPrice() << endl; // 0
}
```

Pointers, References and Dynamic Memory Allocation

- Pointers, References and Dynamic Memory Allocation are the most powerful features in C/C++ language, which allows programmers to directly manipulate memory to efficiently manage the memory - the most critical and scarce resource in computer - for best performance.
- However, "pointer" is also the most complex and difficult feature in C/C++ language.
- Pointers are extremely powerful because they allow you to access addresses and manipulate their contents. But they are also extremely complex to handle. Using them correctly, they could greatly improve the efficiency and performance.
- On the other hand, using them incorrectly could lead to many problems, from unreadable and un-maintainable codes, to infamous bugs such as memory leaks and buffer overflow, which may expose your system to hacking.
- Many new languages (such as Java and C#) remove pointer from their syntax to avoid the pitfalls of pointers, by providing automatic memory management.
- Although you can write C/C++ programs without using pointers, however, it is difficult not to mention pointer in teaching C/C++ language.

Pointer Variables

Computer		Programmers		
Address	Content	Name	Type	Value
90000000	00	sum	int (4 bytes)	000000FF (255 ₁₀)
90000001	00			
90000002	00			
90000003	FF			
90000004	FF		short (2 bytes)	
90000005	FF		FFFF (-1 ₁₀)	
90000006	1F	average	double (8 bytes)	1FFFFFFFFFFFFFFF (4.45015E-308 ₁₀)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90		ptrSum	int* (4 bytes)
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

- A computer memory location has an address and holds a content. The address is a numerical number (often expressed in hexadecimal), which is hard for programmers to use directly. Typically, each address location holds 8-bit (i.e., 1-byte) of data. It is entirely up to the programmer to interpret the meaning of the data, such as integer, real number, characters or strings.
- To ease the burden of programming using numerical address and programmer-interpreted data, early programming languages (such as C) introduce the concept of variables. A variable is a named location that can store a value of a particular type. Instead of numerical addresses, names (or identifiers) are attached to certain addresses. Also, types (such as int, double, char) are associated with the contents for ease of interpretation of data.
- Each address location typically hold 8-bit (i.e., 1-byte) of data. A 4-byte int value occupies 4 memory locations. A 32-bit system typically uses 32-bit addresses. To store a 32-bit address, 4 memory locations are required.

- Pointers must be declared before they can be used, just like a normal variable. The syntax of declaring a pointer is to place a * in front of the name. A pointer is associated with a type (such as int and double) too.

type *ptr; // Declare a pointer variable called ptr as a pointer of type

// or

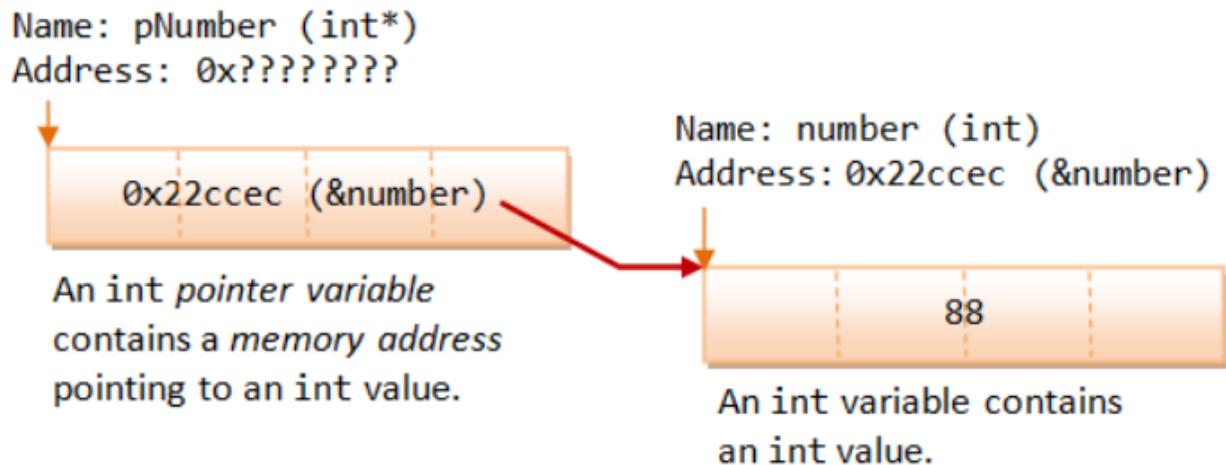
type* ptr;

// or

Declaring Pointers

type * ptr; // I shall adopt this convention

- For example,
- **int * iPtr; // Declare a pointer variable called iPtr pointing to an int (an int pointer)**
- **double * dPtr; // Declare a double pointer**
- Take note that you need to place a * in front of each pointer variable, in other words, * applies only to the name that followed. The * in the declaration statement is not an operator, but indicates that the name followed is a pointer variable. For example,
- **int *p1, *p2, i; // p1 and p2 are int pointers. i is an int**
- **int* p1, p2, i; // p1 is a int pointer, p2 and i are int**
- **int * p1, * p2, i; // p1 and p2 are int pointers, i is an int**
- Naming Convention of Pointers: Include a "p" or "ptr" as prefix or suffix, e.g., iPtr, numberPtr, pNumber, pStudent.



- **Initializing Pointers via the Address-Of Operator (&)**
- When you declare a pointer variable, its content is not initialized. In other words, it contains an address of "somewhere", which is of course not a valid location. This is dangerous! You need to initialize a pointer by assigning it a valid address.
- The address-of operator (&) operates on a variable, and returns the address of the variable.
- You can use the address-of operator to get the address of a variable, and assign the address to a pointer variable.:

```

int number = 88; // An int variable with a value
int * pNumber; // Declare a pointer variable called pNumber pointing to an int (or int pointer)
pNumber = &number; // Assign the address of the variable number to pointer pNumber
int * pAnother = &number; // Declare another int pointer and init to address of the variable number
  
```

Indirection or Dereferencing Operator (*)

- The indirection operator (or dereferencing operator) (*) operates on a pointer, and returns the value stored in the address kept in the pointer variable.

```
int number = 88;  
  
int * pNumber = &number; // Declare and assign the address of variable number to pointer pNumber (0x22cc0c)  
  
cout << pNumber << endl; // Print the content of the pointer variable, which contain an address (0x22cc0c)  
  
cout << *pNumber << endl; // Print the value "pointed to" by the pointer, which is an int (88)  
  
*pNumber = 99; // Assign a value to where the pointer is pointed to, NOT to the pointer variable  
  
cout << *pNumber << endl; // Print the new value "pointed to" by the pointer (99)  
  
cout << number << endl; // The value of variable number changes as well (99)
```

- Take note that pNumber stores a memory address location, whereas *pNumber refers to the value stored in the address kept in the pointer variable, or the value pointed to by the pointer.
- A variable (such as number) directly references a value, whereas a pointer indirectly references a value through the memory address it stores.
- The indirection operator (*) can be used in both the RHS (temp = *pNumber) and the LHS (*pNumber = 99) of an assignment statement.

Pointer has a Type Too

- A pointer is associated with a type (of the value it points to), which is specified during declaration.
- A pointer can only hold an address of the declared type; it cannot hold an address of a different type.

```
int i = 88;  
double d = 55.66;  
int * iPtr = &i; // int pointer pointing to an int value  
double * dPtr = &d; // double pointer pointing to a double value
```

```
iPtr = &d; // ERROR, cannot hold address of different type  
dPtr = &i; // ERROR  
iPtr = i; // ERROR, pointer holds address of an int, NOT int value
```

```
int j = 99;  
iPtr = &j; // You can change the address stored in a pointer
```

```
/* Test pointer declaration and initialization (TestPointerInit.cpp) */  
#include <iostream>  
using namespace std;  
int main() {  
    int number = 88; // Declare an int variable and assign an initial value  
    int * pNumber; // Declare a pointer variable pointing to an int (or int pointer)  
    pNumber = &number; // assign the address of the variable number to pointer pNumber  
  
    cout << pNumber << endl; // Print content of pNumber (0x22ccf0)  
    cout << &number << endl; // Print address of number (0x22ccf0)  
    cout << *pNumber << endl; // Print value pointed to by pNumber (88)  
    cout << number << endl; // Print value of number (88)  
    *pNumber = 99; // Re-assign value pointed to by pNumber  
    cout << pNumber << endl; // Print content of pNumber (0x22ccf0)  
    cout << &number << endl; // Print address of number (0x22ccf0)  
    cout << *pNumber << endl; // Print value pointed to by pNumber (99)  
    cout << number << endl; // Print value of number (99)  
        // The value of number changes via pointer  
    cout << &pNumber << endl; // Print the address of pointer variable pNumber (0x22cce0)  
}
```

Uninitialized Pointers

- The following code fragment has a serious logical error!

```
int * iPtr;  
*iPtr = 55;  
cout << *iPtr << endl;
```

- The pointer iPtr was declared without initialization, i.e., it is pointing to "somewhere" which is of course an invalid memory location.
- The *iPtr = 55 corrupts the value of "somewhere"! You need to initialize a pointer by assigning it a valid address. Most of the compilers does not signal an error or a warning for uninitialized pointer?!

Null Pointers

- You can initialize a pointer to 0 or NULL, i.e., it points to nothing. It is called a **null pointer**. Dereferencing a null pointer (*p) causes an **STATUS_ACCESS_VIOLATION** exception.

```
int * iPtr = 0;      // Declare an int pointer, and initialize the pointer to point to nothing  
cout << *iPtr << endl; // ERROR! STATUS_ACCESS_VIOLATION exception  
int * p = NULL;     // Also declare a NULL pointer points to nothing
```

- Initialize a pointer to null during declaration is a good software engineering practice.
- C++11 introduces a new keyword called `nullptr` to represent null pointer.

Reference Variables

- C++ added the so-called reference variables (or references in short). A reference is an alias, or an alternate name to an existing variable. For example, suppose you make peter a reference (alias) to paul, you can refer to the person as either peter or paul.
- The main use of references is acting as function formal parameters to support pass-by-reference. In an reference variable is passed into a function, **the function works on the original copy (instead of a clone copy in pass-by-value)**. Changes inside the function are reflected outside the function.
- A reference is similar to a pointer. a reference can be used as an alternative to pointer, in particular, for the function parameter.

References (or Aliases) (&)

- Recall that C/C++ use & to denote the address-of operator in an expression. C++ assigns an additional meaning to & in declaration to declare a reference variable.
- The meaning of symbol & is different in an expression and in a declaration. When it is used in an expression, & denotes the address-of operator, which returns the address of a variable, e.g., if number is an int variable, &number returns the address of the variable number (this has been described in the above section).
- However, when & is used in a declaration (including function formal parameters), it is part of the type identifier and is used to declare a reference variable (or reference or alias or alternate name). It is used to provide another name, or another reference, or alias to an existing variable.

`type &newName = existingName; // or`

`type& newName = existingName; // or`

`type & newName = existingName; // I shall adopt this convention`

- It shall be read as "newName is a reference to existingName", or "newName is an alias of existingName". You can now refer to the variable as newName or existingName.

- /* Test reference declaration and initialization (TestReferenceDeclaration.cpp) */

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

```
int main() {
```

```
    int number = 88;      // Declare an int variable called number
```

```
    int & refNumber = number; // Declare a reference (alias) to the variable number
                            // Both refNumber and number refer to the same value
```

```
    cout << number << endl; // Print value of variable number (88)
```

```
    cout << refNumber << endl; // Print value of reference (88)
```

```
    refNumber = 99;          // Re-assign a new value to refNumber
```

```
    cout << refNumber << endl;
```

```
    cout << number << endl; // Value of number also changes (99)
```

```
    number = 55;            // Re-assign a new value to number
```

```
    cout << number << endl;
```

```
    cout << refNumber << endl; // Value of refNumber also changes (55)
```

```
}
```

References vs. Pointers

- Pointers and references are equivalent, **except:**
- A reference is a name constant for an address. You need to initialize the reference during declaration.
- **int & iRef;** // Error: 'iRef' declared as reference but not initialized
- Once a reference is established to a variable, you cannot change the reference to reference another variable.
- To get the value pointed to by a pointer, you need to use the dereferencing operator * (e.g., if pNumber is a int pointer, *pNumber returns the value pointed to by pNumber. It is called dereferencing or indirection).
- To assign an address of a variable into a pointer, you need to use the address-of operator & (e.g., pNumber = &number).
- On the other hand, referencing and dereferencing are done on the references implicitly:
- For example, if refNumber is a reference (alias) to another int variable, refNumber returns the value of the variable. No explicit dereferencing operator * should be used. Furthermore, to assign an address of a variable to a reference variable, no address-of operator & is needed.

```
/* References vs. Pointers (TestReferenceVsPointer.cpp) */

#include <iostream>

using namespace std;

int main() {
    int number1 = 88, number2 = 22;

    int * pNumber1 = &number1; // Create a pointer pointing to number1 // Explicit referencing
    *pNumber1 = 99;          // Explicit dereferencing

    cout << *pNumber1 << endl; // 99
    cout << &number1 << endl; // 0x22ff18
    cout << pNumber1 << endl; // 0x22ff18 (content of the pointer variable - same as above)
    cout << &pNumber1 << endl; // 0x22ff10 (address of the pointer variable)

    pNumber1 = &number2;      // Pointer can be reassigned to store another address

    int & refNumber1 = number1; // Create a reference (alias) to number1
    refNumber1 = 11;          // Implicit referencing (NOT &number1)// Implicit dereferencing (NOT *refNumber1)

    cout << refNumber1 << endl; // 11
    cout << &number1 << endl; // 0x22ff18
    cout << &refNumber1 << endl; // 0x22ff18
    //refNumber1 = &number2; // Error! Reference cannot be re-assigned

    refNumber1 = number2; // error: invalid conversion from 'int*' to 'int'// refNumber1 is still an alias to number1. // Assign value of number2 (22) to refNumber1 (and number1).

    number2++;

    cout << refNumber1 << endl; // 22
    cout << number1 << endl; // 22
    cout << number2 << endl; // 23
}
```

- A reference variable provides a new name to an existing variable.
- It is dereferenced implicitly and does not need the dereferencing operator * to retrieve the value referenced.
- On the other hand, a pointer variable stores an address.
- You can change the address value stored in a pointer.
- To retrieve the value pointed to by a pointer, you need to use the indirection operator *, which is known as explicit dereferencing. Reference can be treated as a const pointer. It has to be initialized during declaration, and its content cannot be changed.
- Reference is closely related to pointer. In many cases, it can be used as an alternative to pointer.
- A reference allows you to manipulate an object using pointer, but without the pointer syntax of referencing and dereferencing.

Pass-By-Reference into Functions with Reference Arguments vs. Pointer Arguments

- **Pass-by-Value**
- In C/C++, by default, arguments are passed into functions by value (except arrays which is treated as pointers).
- That is, a clone copy of the argument is made and passed into the function. **Changes to the clone copy inside the function has no effect to the original argument in the caller.**
- In other words, the called function has no access to the variables in the caller. For example,

```
/* Pass-by-value into function (TestPassByValue.cpp) */
```

```
#include <iostream>
using namespace std;
int square(int);
int main() {
    int number = 8;
    cout << "In main(): " << &number << endl; // 0x22ff1c
    cout << number << endl;      // 8
    cout << square(number) << endl; // 64
    cout << number << endl;      // 8 - no change
}
```

```
int square(int n) { // non-const
    cout << "In square(): " << &n << endl; // 0x22ff00
    n *= n;      // clone modified inside the function
    return n;
}
```

Pass-by-Reference with Pointer Arguments

- In many situations, we may wish to modify the original copy directly (especially in passing huge object or array) to avoid the overhead of cloning.
- This can be done by passing a pointer of the object into the function, known as pass-by-reference. For example,

```
/* Pass-by-reference using pointer (TestPassByPointer.cpp) */  
#include <iostream>  
using namespace std;  
  
void square(int *);  
int main() {  
    int number = 8;  
    cout << "In main(): " << &number << endl; // 0x22ff1c  
    cout << number << endl; // 8  
    square(&number); // Explicit referencing to pass an address  
    cout << number << endl; // 64  
}
```

```
void square(int * pNumber) { // Function takes an int pointer (non-const)  
    cout << "In square(): " << pNumber << endl; // 0x22ff1c  
    *pNumber *= *pNumber; // Explicit de-referencing to get the value pointed-to  
}
```

- The called function operates on the same address, and can thus modify the variable in the caller.

Pass-by-Reference with Reference Arguments

- Instead of passing pointers into function, you could also pass references into function, to avoid the clumsy syntax of referencing and dereferencing.

```
/* Pass-by-reference using reference (TestPassByReference.cpp) */
```

```
#include <iostream>
using namespace std;
void square(int &);

int main() {
    int number = 8;
    cout << "In main(): " << &number << endl; // 0x22ff1c
    cout << number << endl; // 8
    square(number); // Implicit referencing (without '&')
    cout << number << endl; // 64
}

void square(int & rNumber) { // Function takes an int reference (non-const)
    cout << "In square(): " << &rNumber << endl; // 0x22ff1c
    rNumber *= rNumber; // Implicit de-referencing (without '*')
}
```

- Again, the output shows that the called function operates on the same address, and can thus modify the caller's variable.
- Take note referencing (in the caller) and dereferencing (in the function) are done implicitly. The only coding difference with pass-by-value is in the function's parameter declaration.
- Recall that references are to be initialized during declaration. In the case of function formal parameter, the references are initialized when the function is invoked, to the caller's arguments.
- References are primarily used in passing reference in/out of functions to allow the called function accesses variables in the caller directly.

"const" Function Reference/Pointer Parameters

- A const function formal parameter cannot be modified inside the function. Use const whenever possible as it protects you from inadvertently modifying the parameter and protects you against many programming errors.
- A const function parameter can receive both const and non-const argument. On the other hand, a non-const function reference/pointer parameter can only receive non-const argument.

Dynamic Memory Allocation

new and delete Operators

- Instead of define an int variable (int number), and assign the address of the variable to the int pointer (int *pNumber = &number), the storage can be dynamically allocated at runtime, via a new operator.
- In C++, whenever you allocate a piece of memory dynamically via new, you need to use delete to remove the storage (i.e., to return the storage to the heap).
- The new operation returns a pointer to the memory allocated. The delete operator takes a pointer (pointing to the memory allocated via new) as its sole argument.

```
int number = 88; // Static allocation
```

```
int * p1 = &number; // Assign a "valid" address into pointer // Dynamic Allocation
```

```
int * p2; // Not initialize, points to somewhere which is invalid
```

```
cout << p2 << endl; // Print address before allocation
```

```
p2 = new int; // Dynamically allocate an int and assign its address to pointer  
// The pointer gets a valid address with memory allocated
```

```
*p2 = 99;
```

```
cout << p2 << endl; // Print address after allocation
```

```
cout << *p2 << endl; // Print value point-to
```

```
delete p2; // Remove the dynamically allocated storage
```

- Observe that new and delete operators work on pointer.

- To initialize the allocated memory, you can use an initializer for fundamental types, or invoke a constructor for an object. For example,

```
// use an initializer to initialize a fundamental type (such as int, double)
```

```
int * p1 = new int(88);
```

```
double * p2 = new double(1.23);
```

```
// C++11 brace initialization syntax
```

```
int * p1 = new int {88};
```

```
double * p2 = new double {1.23};
```

```
// invoke a constructor to initialize an object (such as Date, Time)
```

```
Date * date1 = new Date(1999, 1, 1);
```

```
Time * time1 = new Time(12, 34, 56);
```

- You can dynamically allocate storage for global pointers inside a function. Dynamically allocated storage inside the function remains even after the function exits. For example,

```
// Dynamically allocate global pointers (TestDynamicAllocation.cpp)
#include <iostream>
using namespace std;
int * p1, * p2; // Global int pointers
// This function allocates storage for the int*/ which is available outside the function
void allocate() {
    p1 = new int;    // Allocate memory, initial content unknown
    *p1 = 88;       // Assign value into location pointed to by pointer
    p2 = new int(99); // Allocate and initialize
}

int main() {
    allocate();
    cout << *p1 << endl; // 88
    cout << *p2 << endl; // 99
    delete p1; // Deallocate
    delete p2;
    return 0;
}
```

- **The main differences between static allocation and dynamic allocations are:**
- In static allocation, the compiler allocates and deallocates the storage automatically, and handle memory management.
- Whereas in dynamic allocation, you, as the programmer, handle the memory allocation and deallocation yourself (via new and delete operators).
- You have full control on the pointer addresses and their contents, as well as memory management.
- Static allocated entities are manipulated through named variables. Dynamic allocated entities are handled through pointers.

/* Test dynamic allocation of array (TestDynamicArray.cpp) */ Dynamic array is allocated at runtime rather than compile-time, via the new[] operator. To remove the storage, you need to use the delete[] operator (instead of simply delete). For example,

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
    const int SIZE = 5;
    int * pArray;
    pArray = new int[SIZE]; // Allocate array via new[] operator
    for (int i = 0; i < SIZE; ++i) { // Assign random numbers between 0 and 99
        *(pArray + i) = rand() % 100;
    }
    for (int i = 0; i < SIZE; ++i) { // Print array
        cout << *(pArray + i) << " ";
    }
    cout << endl;
    delete[] pArray; // Deallocate array via delete[] operator
    return 0;
}
```

new[] and delete[] Operators

Pointer, Array and Function

- Array is Treated as Pointer
- In C/C++, an array's name is a pointer, pointing to the first element (index 0) of the array.
- For example, suppose that numbers is an int array, numbers is also an int pointer, pointing at the first element of the array.
- That is, numbers is the same as &numbers[0]. Consequently, *numbers is number[0]; *(numbers+i) is numbers[i].
- For example,

```
/* Pointer and Array (TestPointerArray.cpp) */  
#include <iostream>  
using namespace std;  
  
int main() {  
    const int SIZE = 5;  
    int numbers[SIZE] = {11, 22, 44, 21, 41}; // An int array  
  
    // The array name numbers is an int pointer, pointing at the  
    // first item of the array, i.e., numbers = &numbers[0]  
    cout << &numbers[0] << endl; // Print address of first element (0x22fef8)  
    cout << numbers << endl; // Same as above (0x22fef8)  
    cout << *numbers << endl; // Same as numbers[0] (11)  
    cout << *(numbers + 1) << endl; // Same as numbers[1] (22)  
    cout << *(numbers + 4) << endl; // Same as numbers[4] (41)  
}
```

• Pointer Arithmetic

- As seen from the previous section, if numbers is an int array, it is treated as an int pointer pointing to the first element of the array. (numbers + 1) points to the next int, instead of having the next sequential address. Take note that an int typically has 4 bytes. That is (numbers + 1) increases the address by 4, or sizeof(int). For example,

```
int numbers[] = {11, 22, 33};  
int * iPtr = numbers;  
cout << iPtr << endl;      // 0x22cd30  
cout << iPtr + 1 << endl;  // 0x22cd34 (increase by 4 - sizeof int)  
cout << *iPtr << endl;    // 11  
cout << *(iPtr + 1) << endl; // 22  
cout << *(iPtr + 2) << endl; // 33
```

sizeof Array

- The operation sizeof(arrayName) returns the total bytes of the array. You can derive the length (size) of the array by dividing it with the size of an element (e.g. element 0). For example,

```
int numbers[100];  
cout << sizeof(numbers) << endl;    // Size of entire array in bytes (400)  
cout << sizeof(numbers[0]) << endl;  // Size of first element of the array in bytes (4)  
cout << "Array size is " << sizeof(numbers) / sizeof(numbers[0]) << endl; // (100)
```

Passing Array In/Out of a Function

- An array is passed into a function as a pointer to the first element of the array. You can use array notation (e.g., int[]) or pointer notation (e.g., int*) in the function declaration.
- The compiler always treats it as pointer (e.g., int*). For example, the following declarations are equivalent:

```
int max(int numbers[], int size);  
int max(int *numbers, int size);  
int max(int number[50], int size);
```

- They will be treated as int* by the compiler, as follow. The size of the array given in [] is ignored.
- **int max(int*, int);**
- Array is passed by reference into the function, because a pointer is passed instead of a clone copy.
- If the array is modified inside the function, the modifications are applied to the caller's copy.
- You could declare the array parameter as const to prevent the array from being modified inside the function.
- The size of the array is not part of the array parameter, and needs to be passed in another int parameter. Compiler is not able to deduce the array size from the array pointer, and does not perform array bound check.
- Example: Using the usual array notation.

```
/* Passing array in/out function (TestArrayPassing.cpp) */

#include <iostream>

using namespace std;

int max(const int arr[], int size); // Function prototypes

void replaceByMax(int arr[], int size);

void print(const int arr[], int size);

int main() {

    const int SIZE = 4;

    int numbers[SIZE] = {11, 22, 33, 22};

    print(numbers, SIZE);

    cout << max(numbers, SIZE) << endl;

    replaceByMax(numbers, SIZE);

    print(numbers, SIZE);

}

int max(const int arr[], int size) // Return the maximum value of the given array.// The array is declared const, and cannot be modified inside the function.

int max = arr[0];

for (int i = 1; i < size; ++i) {

    if (max < arr[i]) max = arr[i];

}

return max;

}

void replaceByMax(int arr[], int size) // Replace all elements of the given array by its maximum value// Array is passed by reference. Modify the caller's copy.

int maxValue = max(arr, size);

for (int i = 0; i < size; ++i) {

    arr[i] = maxValue;

}

}

// Print the array's content

void print(const int arr[], int size) {

    cout << "{";

    for (int i = 0; i < size; ++i) {

        cout << arr[i];

        if (i < size - 1) cout << ",";

    }

    cout << "}" << endl;

}
```

- Take note that you can modify the contents of the caller's array inside the function, as array is passed by reference.
 - To prevent accidental modification, you could apply const qualifier to the function's parameter.
 - Recall that const inform the compiler that the value should not be changed.
 - For example, suppose that the function print() prints the contents of the given array and does not modify the array, you could apply const to both the array name and its size, as they are not expected to be changed inside the function.
-
- `void print(const int arr[], int size);`
 - Compiler flags out an error "assignment of read-only location" if it detected a const value would be changed.

```
/* Passing array in/out function using pointer (TestArrayPassingPointer.cpp) */
```

```
#include <iostream>
using namespace std;
// Function prototype
int max(const int *arr, int size);
int main() {
    const int SIZE = 5;
    int numbers[SIZE] = {10, 20, 90, 76, 22};
    cout << max(numbers, SIZE) << endl;
}
int max(const int *arr, int size) {// Return the maximum value of the given array
    int max = *arr;
    for (int i = 1; i < size; ++i) {
        if (max < *(arr+i)) max = *(arr+i);
    }
    return max;
}
```

```
/* Test sizeof array (TestSizeofArray.cpp) */  
#include <iostream>  
using namespace std;  
// Function prototypes  
void fun(const int *arr, int size);  
// Test Driver  
int main() {  
    const int SIZE = 5;  
    int a[SIZE] = {8, 4, 5, 3, 2};  
    cout << "sizeof in main() is " << sizeof(a) << endl;  
    cout << "address in main() is " << a << endl;  
    fun(a, SIZE);  
}  
// Function definitions  
void fun(const int *arr, int size) {  
    cout << "sizeof in function is " << sizeof(arr) << endl;  
    cout << "address in function is " << arr << endl;  
}
```

Operating on a Range of an Array/* Function to compute the sum of a range of an array (SumArrayRange.cpp) */

```
#include <iostream>
using namespace std;
int sum(const int *begin, const int *end); // Function prototype
int main() { // Test Driver
    int a[] = {8, 4, 5, 3, 2, 1, 4, 8};
    cout << sum(a, a+8) << endl;      // a[0] to a[7]
    cout << sum(a+2, a+5) << endl;    // a[2] to a[4]
    cout << sum(&a[2], &a[5]) << endl; // a[2] to a[4]
}
```

// Function definition// Return the sum of the given array of the range from// begin to end, exclude end.

```
int sum(const int *begin, const int *end) {
    int sum = 0;
    for (const int *p = begin; p != end; ++p) {
        sum += *p;
    }
    return sum;
}
```


Function Pointer

- In C/C++, functions, like all data items, have an address.
- The name of a function is the starting address where the function resides in the memory, and therefore, can be treated as a pointer.
- We can pass a function pointer into function as well. The syntax for declaring a function pointer is:

// Function-pointer declaration

return-type (* function-ptr-name) (parameter-list)

// Examples

double (*fp)(int, int) // fp points to a function that takes two ints and returns a double (function-pointer)

double *dp; // dp points to a double (double-pointer)

double *fun(int, int) // fun is a function that takes two ints and returns a double-pointer

double f(int, int); // f is a function that takes two ints and returns a double

fp = f; // Assign function f to fp function-pointer

```
/* Test Function Pointers (TestFunctionPointer.cpp) */

#include <iostream>
using namespace std;
int arithmetic(int, int, int (*)(int, int));
    // Take 3 arguments, 2 int's and a function pointer // int (*)(int, int), which takes two int's and return
an int

int add(int, int);
int sub(int, int);
int add(int n1, int n2) { return n1 + n2; }
int sub(int n1, int n2) { return n1 - n2; }
int arithmetic(int n1, int n2, int (*operation)(int, int)) {
    return (*operation)(n1, n2);
}
int main() {
    int number1 = 5, number2 = 6;
    // add
    cout << arithmetic(number1, number2, add) << endl;
    // subtract
    cout << arithmetic(number1, number2, sub) << endl;
}
```

Generic Pointer or void Pointer (`void *`)

- A void pointer can hold address of any data type (except function pointer).
- We cannot operate on the object pointed to by void pointer, as the type is unknown.
- We can use a void pointer to compare with another address.

Constant Pointer vs. Constant Pointed-to Data

Non-constant pointer to non-constant data: Data pointed to CAN be changed; and pointer CAN be changed to point to another data. For example,

```
int i1 = 8, i2 = 9;  
int * iptr = &i1; // non-constant pointer pointing to non-constant data  
*iptr = 9; // okay      iptr = &i2; // okay
```

Non-constant pointer to constant data: Data pointed to CANNOT be changed; but pointer CAN be changed to point to another data. For example,

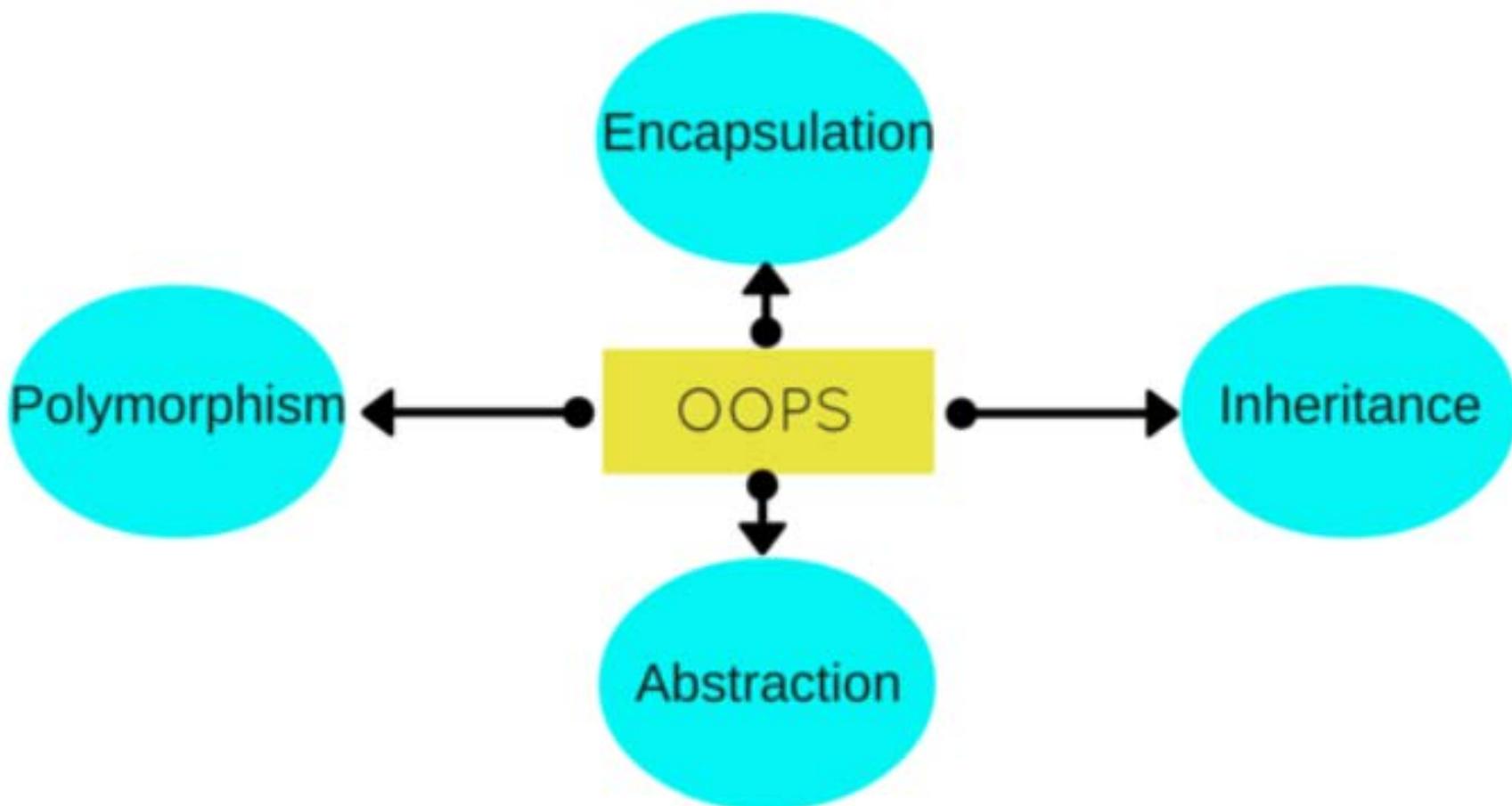
```
int i1 = 8, i2 = 9;  
const int * iptr = &i1; // non-constant pointer pointing to constant data  
// *iptr = 9; // error: assignment of read-only location  
iptr = &i2; // okay
```

Constant pointer to non-constant data: Data pointed to CAN be changed; but pointer CANNOT be changed to point to another data. For example,

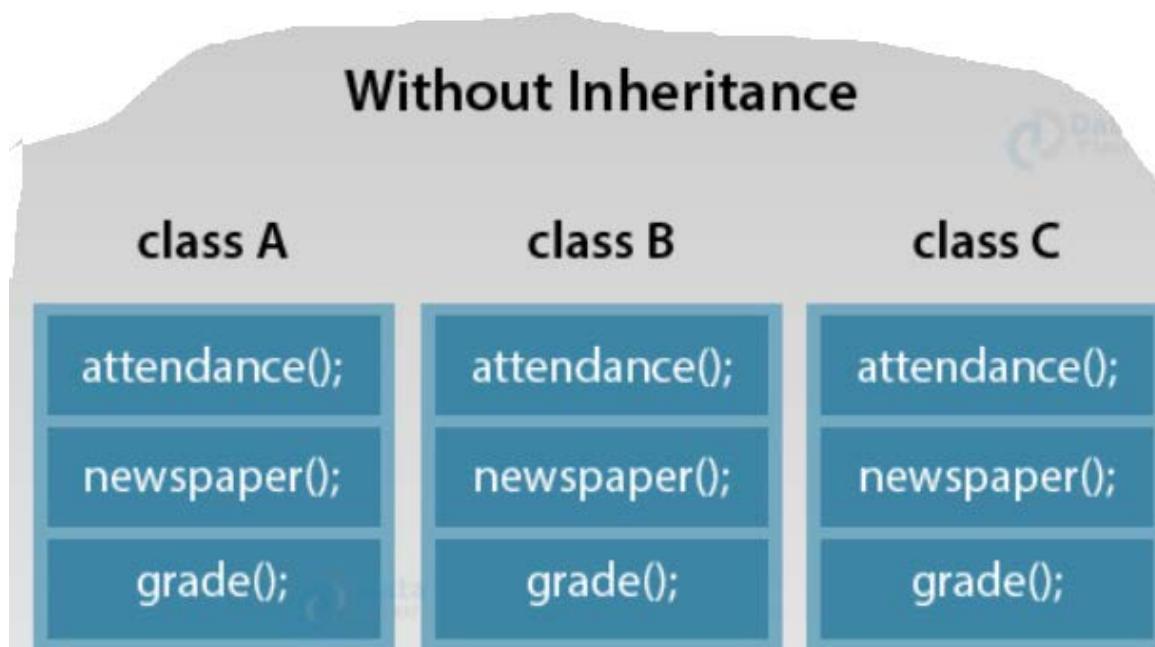
```
int i1 = 8, i2 = 9;  
int * const iptr = &i1; // constant pointer pointing to non-constant data // constant pointer must be initialized during declaration  
*iptr = 9; // okay  
// iptr = &i2; // error: assignment of read-only variable
```

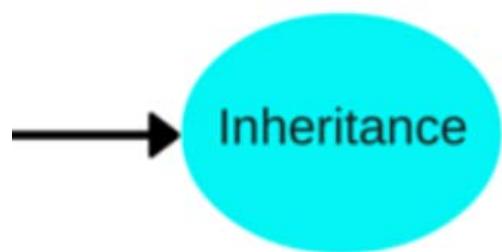
Constant pointer to constant data: Data pointed to CANNOT be changed; and pointer CANNOT be changed to point to another data. For example,

```
int i1 = 8, i2 = 9;  
const int * const iptr = &i1; // constant pointer pointing to constant data  
// *iptr = 9; // error: assignment of read-only variable  
// iptr = &i2; // error: assignment of read-only variable
```

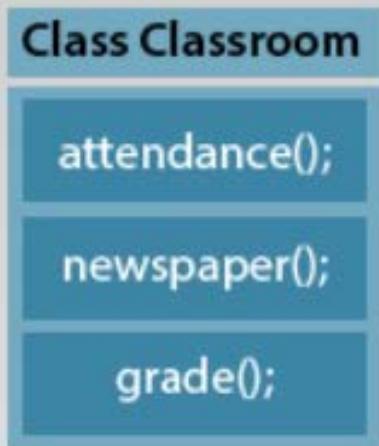


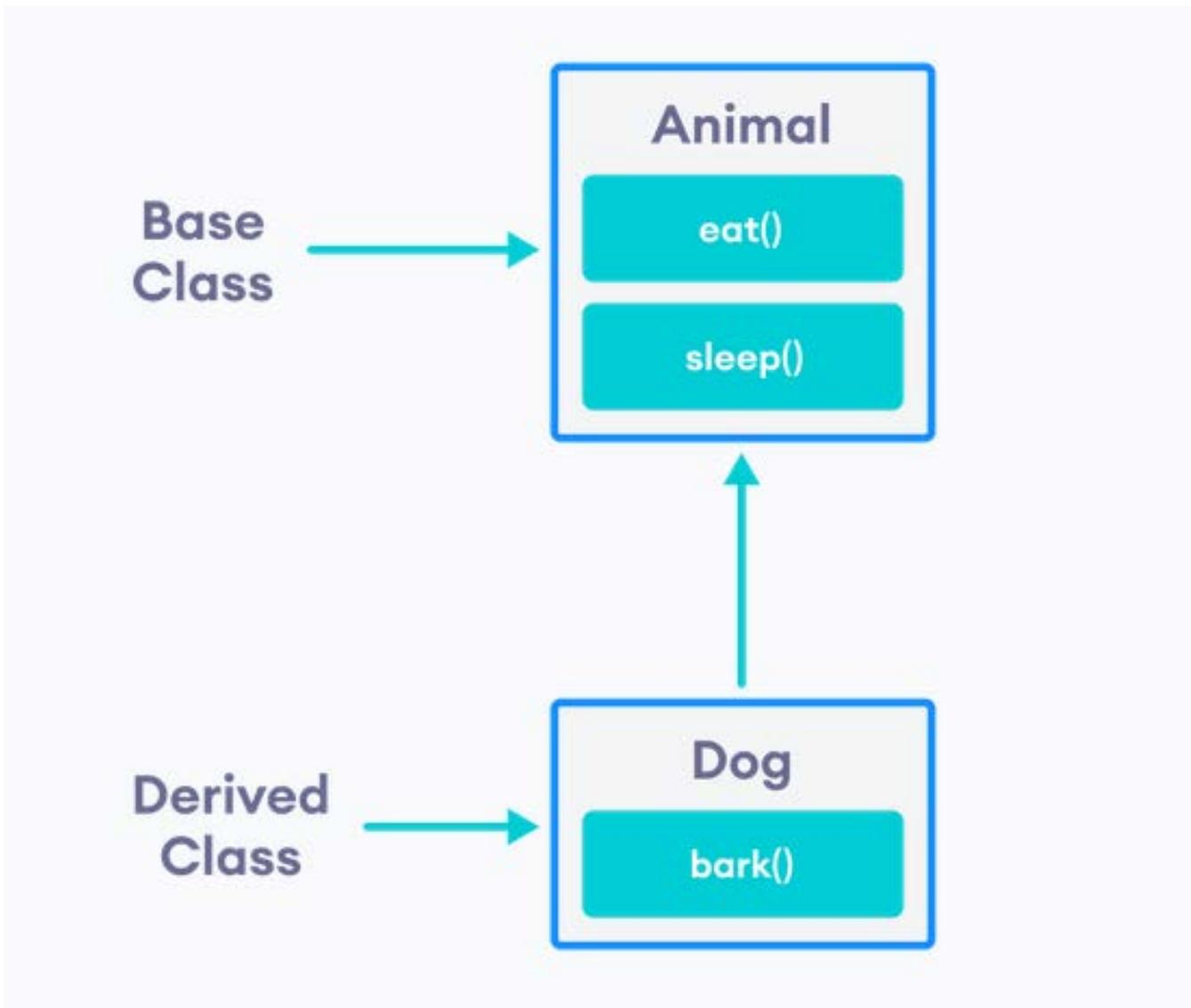
- Suppose there are 3 sections in the 12th grade of your school: A, B, and C.
- The functions that we need to perform in each class are taking the attendance, distributing newspapers to students who subscribed for it, and giving them their final grades.





With Inheritance





- It drastically reduces code redundancy as it allows the derived class to inherit all the properties of the base class, leaving no room for duplicate data to achieve the same task.
- Inheritance in C++ offers the feature of code reusability. We can use the same fragment of code multiple times.
- To sum it up, inheritance helps us save our development time, maintain data in a simplified manner and gives us the provision to make our code extensible.
- It increases the code reliability by providing a definite body to the program.

- The capability of a class to derive properties and characteristics from another class is called **Inheritance**.

- **Sub Class:**

The class that inherits properties from another class is called Sub class or **Derived Class**.

- **Super Class:**

- The class whose properties are inherited by sub class is called **Base Class or Super class**.

Class Bus

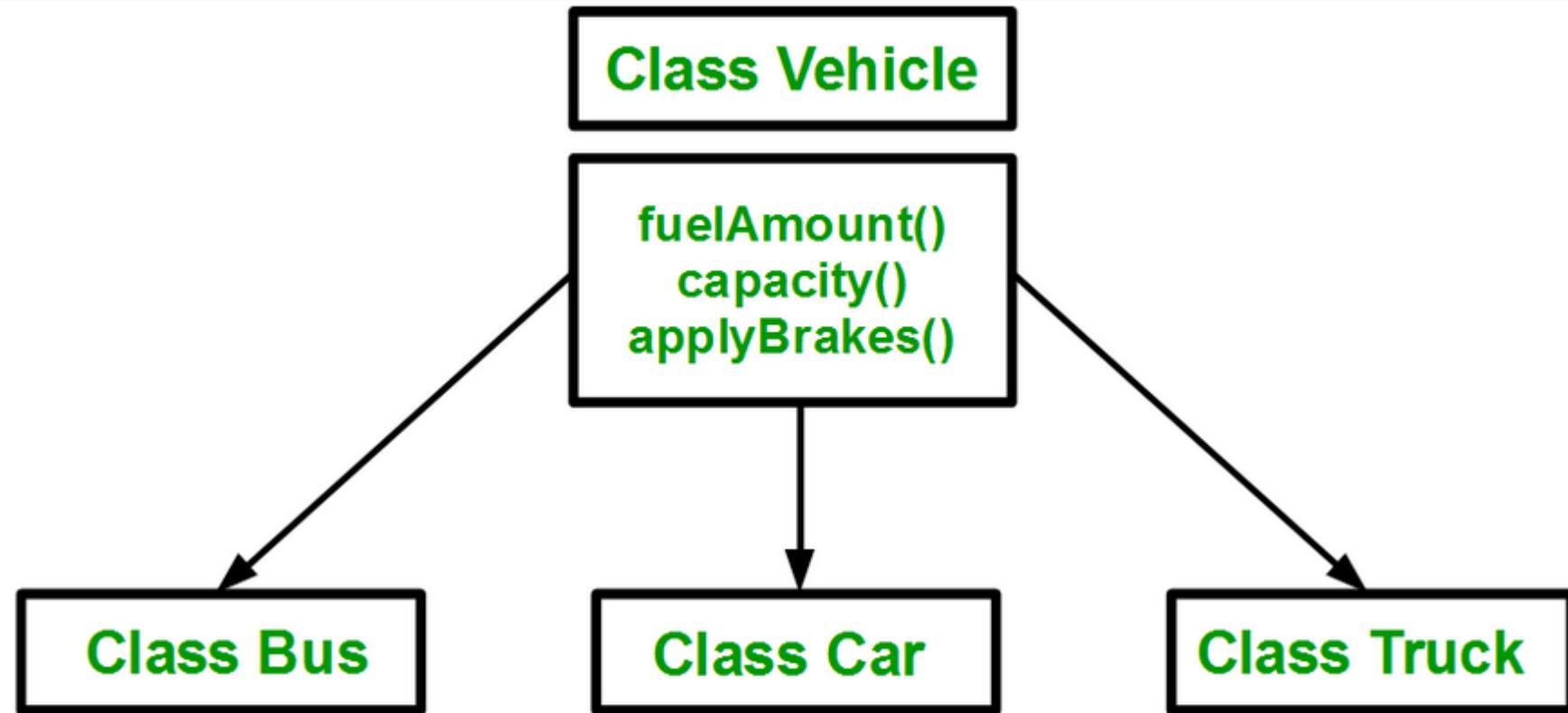
**fuelAmount()
capacity()
applyBrakes()**

Class Car

**fuelAmount()
capacity()
applyBrakes()**

Class Truck

**fuelAmount()
capacity()
applyBrakes()**



- **Using inheritance,**

- we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).

- **Implementing inheritance in C++:**

- For creating a sub-class which is inherited from the base class we have to follow the below syntax.

```
class subclass_name : access_mode  base_class_name  
{  
//body of subclass  
};
```

- **subclass_name** is the name of the sub class,
- **access_mode** is the mode in which you want to inherit this sub class for example: public, private etc.
- **base_class_name** is the name of the base class from which you want to inherit the sub class.
- A derived class doesn't inherit access to private data members. However, it does inherit a full parent object, which contains any private members which that class declares

```
// C++ program to demonstrate implementation  
// of Inheritance
```

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

```
//Base class
```

```
class Parent
```

```
{
```

```
public:
```

```
    int id_p;
```

```
};
```

```
// Sub class inheriting from Base Class(Parent)
```

```
class Child : public Parent
```

```
{
```

```
public:
```

```
    int id_c;
```

```
};
```

```
int main()
```

```
{
```

```
    Child obj1;
```

```
    // An object of class child has all data members
```

```
    // and member functions of class parent
```

```
    obj1.id_c = 7;
```

```
    obj1.id_p = 91;
```

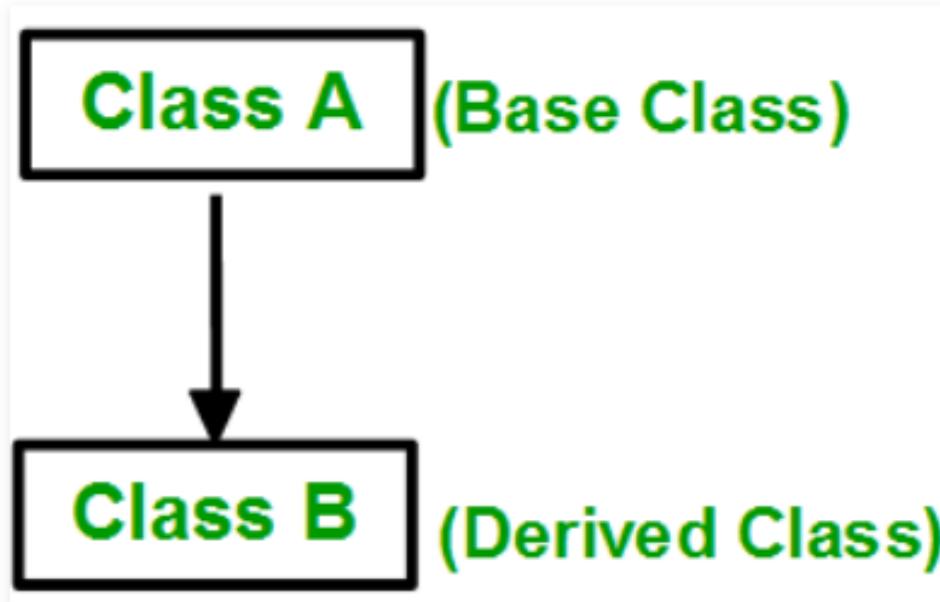
```
    cout << "Child id is " << obj1.id_c << endl;
```

```
    cout << "Parent id is " << obj1.id_p << endl;
```

```
    return 0;
```

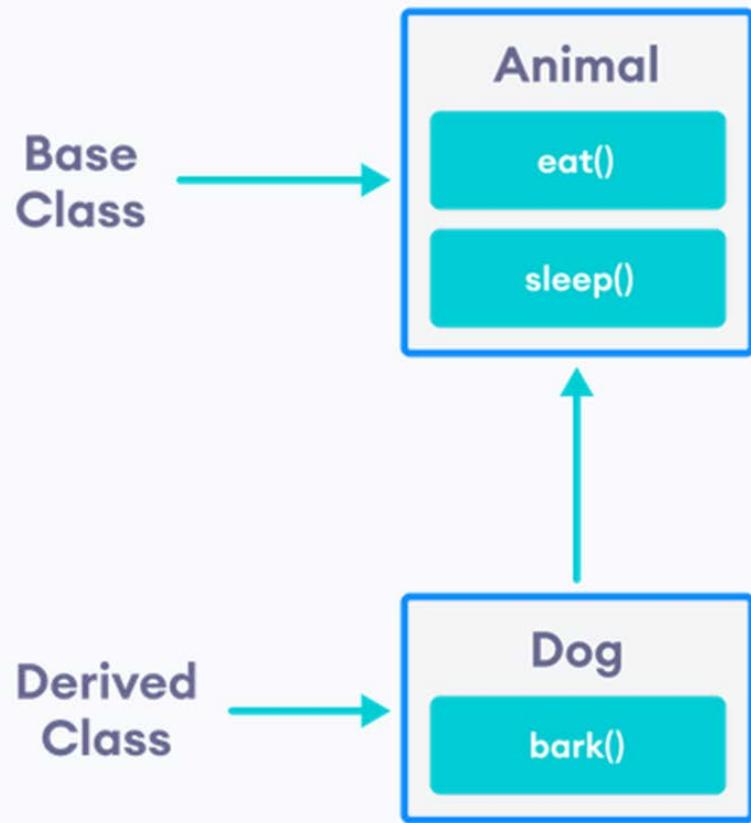
```
}
```

1. Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



Syntax:

```
class subclass_name : access_mode base_class
{
    //body of subclass
};
```



```
class Animal {  
    // eat() function  
    // sleep() function  
};  
  
class Dog : public Animal {  
    // bark() function  
};
```

```
// C++ program to explain // Single inheritance
#include <iostream>
using namespace std;

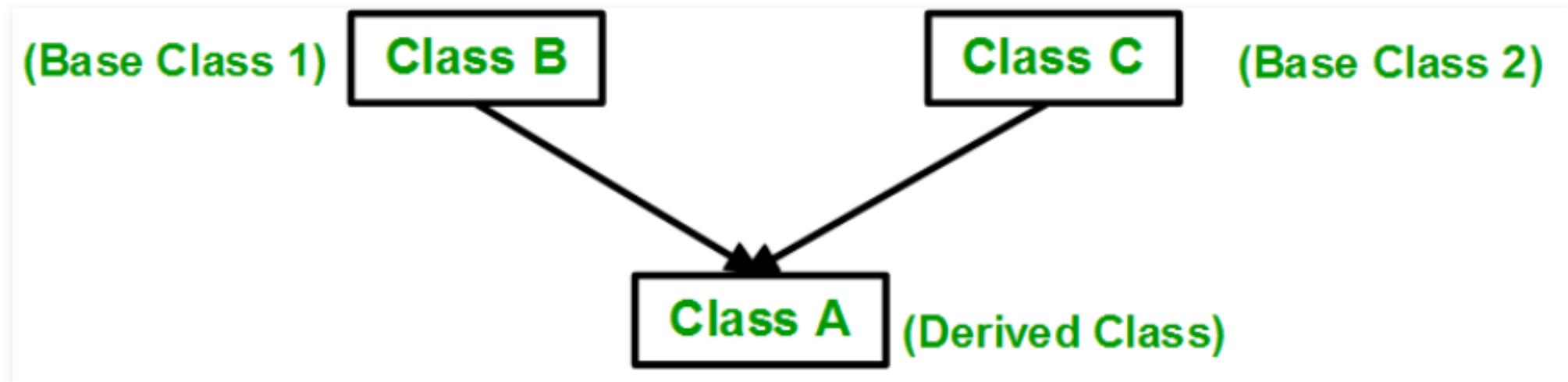
class Vehicle { // base class
public:
    Vehicle() {
        cout << "This is a Vehicle" << endl;
    }
};

class Car: public Vehicle{ // sub class derived from two base classes
};

int main() {
    // creating object of sub class will // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

```
class Base{  
public:  
int base_value;  
    void base_input(){  
        cout<<"Enter the integer value of base class: ";  
        cin>>base_value;  
    }  
};  
class Derived : public Base {  
int derived_value; // private by default  
public:  
    void derived_input(){  
        cout<<"Enter the integer value of derived class: ";  
        cin>>derived_value;  
    }  
void sum(){  
    cout << "The sum of the two integer values is: " <<  
    base_value + derived_value<<endl;  
}  
};  
  
int main(){  
    cout<<"Welcome "<<endl<<endl;  
    Derived d; // Object of the derived class  
    d.base_input();  
    d.derived_input();  
    d.sum();  
    return 0;  
}
```

2. Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.



Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....  
{  
    //body of subclass  
};
```

Here, the number of base classes will be separated by a comma (',') and access mode for every base class must be specified.

```
#include <iostream> using namespace std; // C++ program to explain // multiple inheritance
class Vehicle { // first base class
public:
    Vehicle() {
        cout << "This is a Vehicle" << endl;
    }
};
class FourWheeler { // second base class
public:
    FourWheeler() {
        cout << "This is a 4 wheeler Vehicle" << endl;
    }
};
class Car: public Vehicle, public FourWheeler { // sub class derived from two base classes
};
int main() {
    // creating object of sub class will // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

```
class A{
public:
int A_value;

void A_input(){
cout<<"Enter the integer value of class A: ";
cin>>A_value;
}

};

class B{
public:
int B_value;

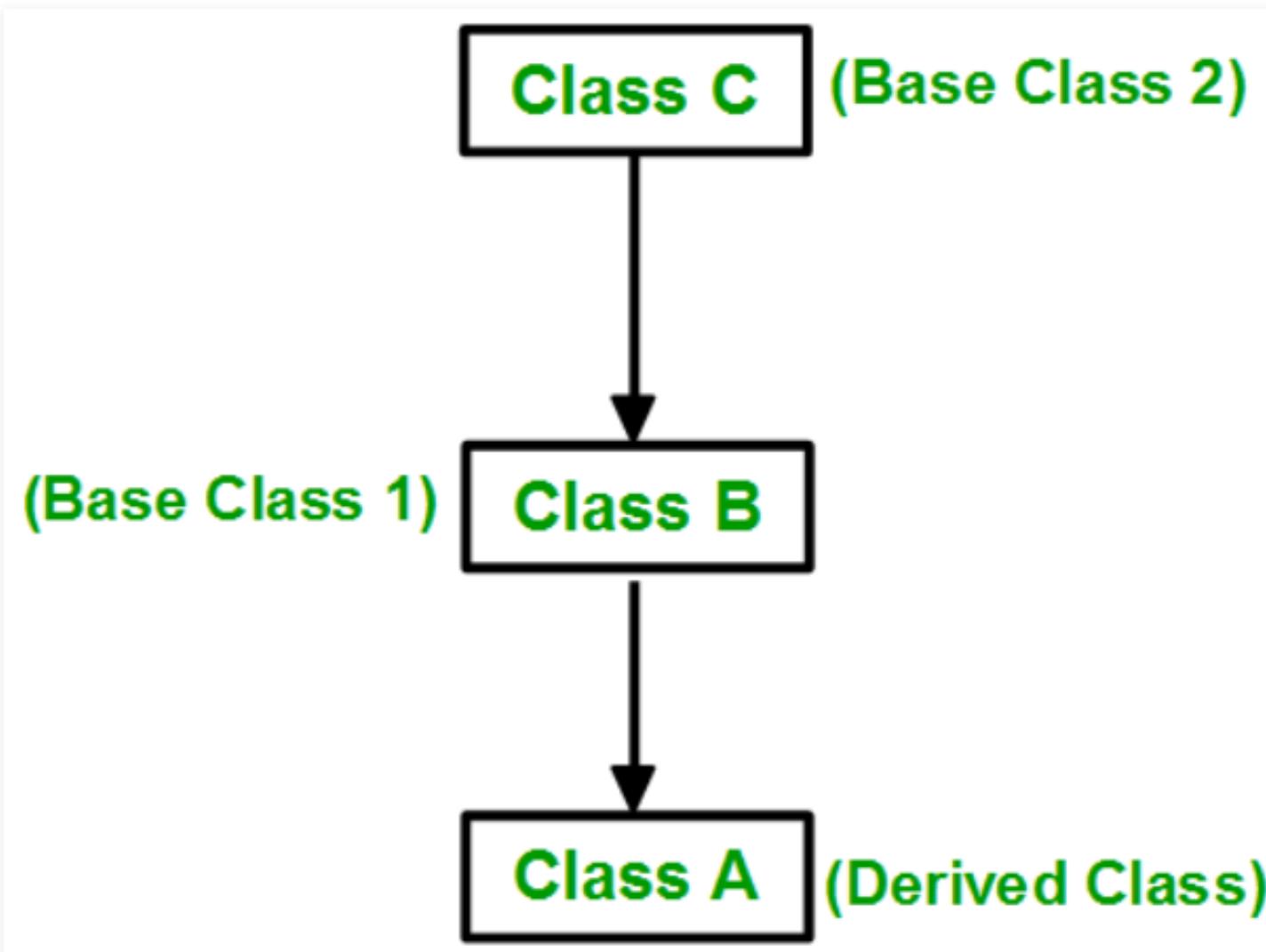
void B_input(){
cout<<"Enter the integer value of class B: ";
cin>>B_value;
}

};

class C : public A, public B{ //C is a derived class from classes A and B
public:
void difference(){
cout<<"The difference between the two values is: " << A_value - B_value<<endl;
}
};
```

```
int main()
{
cout<<"Welcome"<<endl<<endl;
C c; // c is an Object of derived class C
c.A_input();
c.B_input();
c.difference();
return 0;
}
```

3. Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.



```
#include <iostream>      using namespace std;
class Vehicle { // base class 1
public:
    Vehicle() {
        cout << "This is a Vehicle" << endl;
    }
};
class fourWheeler: public Vehicle {
public:
    fourWheeler() { // base class 2
        cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};
class Car: public fourWheeler{ // sub class derived from
two base classes
public:
    car() {
        cout<<"Car has 4 Wheels"<<endl;
    }
};
// C++ program to implement
// Multilevel Inheritance
```

```
int main() {
//creating object of sub class will
//invoke the constructor of base classes
    Car obj;
    return 0;
}
```

```
class Base{
public:
int base_value;

void Base_input(){
cout<<"Enter the integer value of base class: ";
cin>>base_value;
}

};

class Derived1 : public Base{ // Derived class of base class
public:
int derived1_value;

void Derived1_input(){
cout<<"Enter the integer value of first derived class: ";
cin>>derived1_value;
}

};

class Derived2 : public Derived1{ // Derived class of Derived1 class
int derived2_value;
public:
void Derived2_input(){
cout<<"Enter the integer value of the second derived class: ";
cin>>derived2_value;
}
};

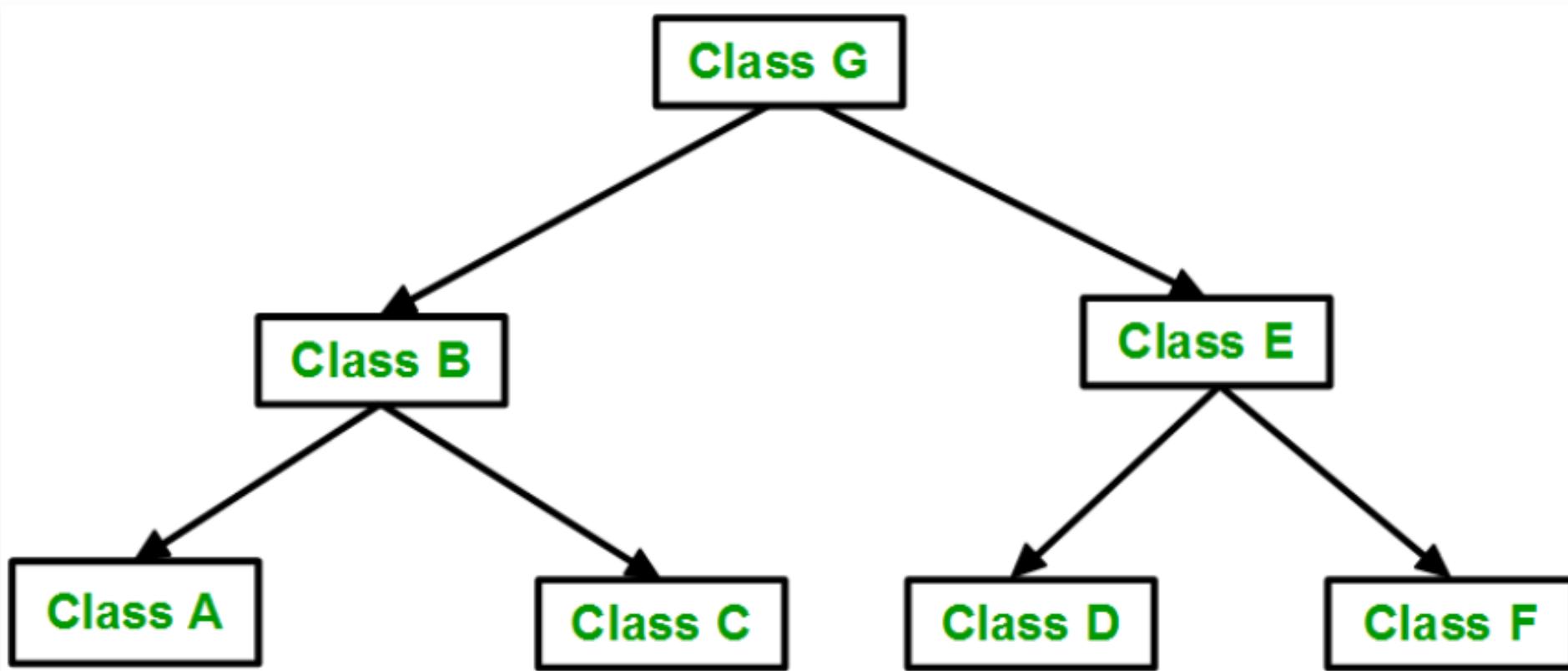
void sum(){
cout << "The sum of the three intger : " << base_value +
derived1_value + derived2_value<<endl;
}

int main(){

cout<<"Welcome "<<endl<<endl;

Derived2 d2; // Object d2 of second derived class
d2.Base_input();
d2.Derived1_input();
d2.Derived2_input();
d2.sum();
return 0;
}
```

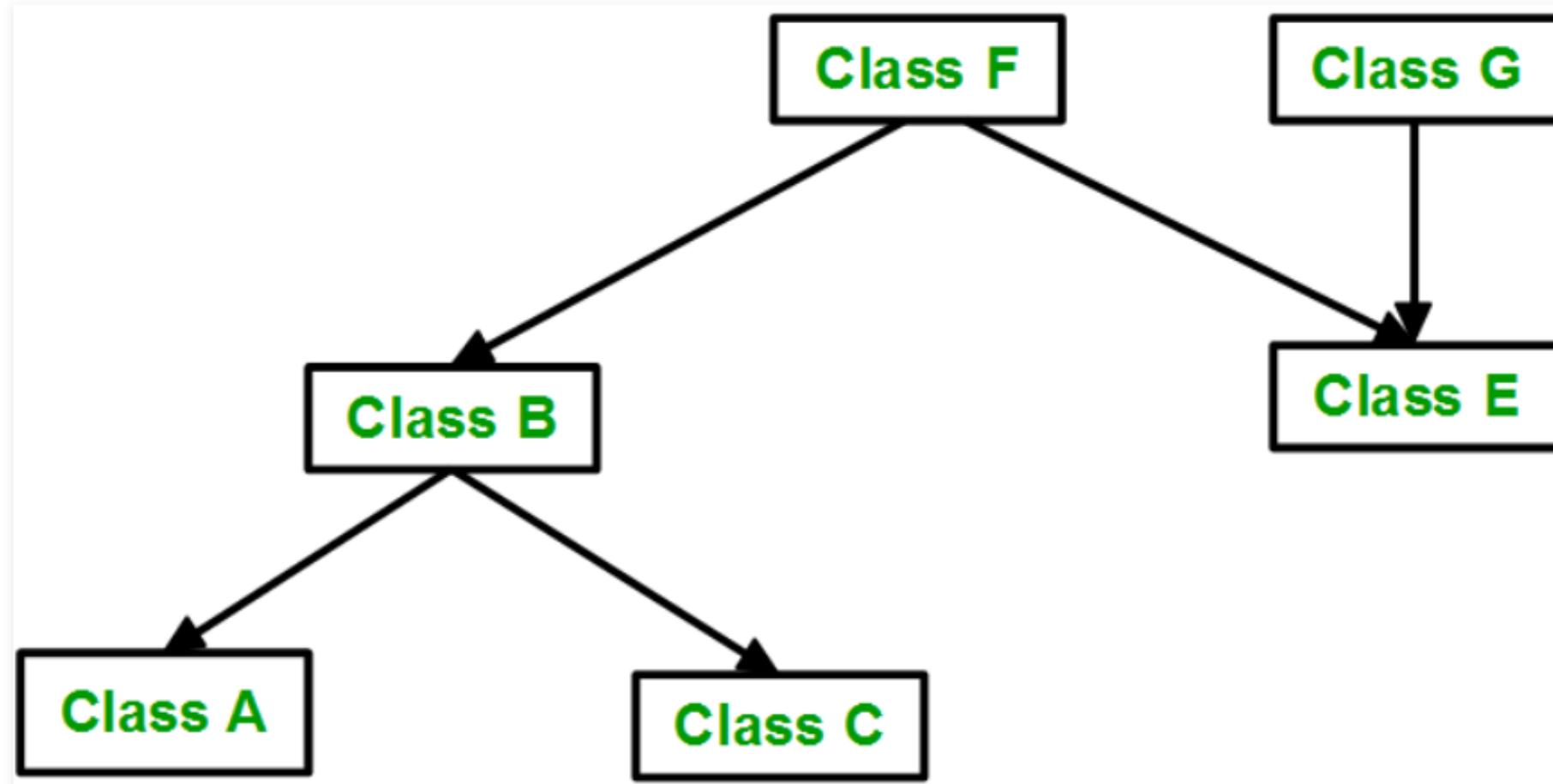
4. Hierarchical Inheritance: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.



```
#include <iostream> // C++ program to implement // Hierarchical Inheritance
using namespace std;
class Vehicle { // base class
public:
    Vehicle() {
        cout << "This is a Vehicle" << endl;
    }
};
class Car: public Vehicle { // first sub class
};
class Bus: public Vehicle { // second sub class
};
int main() {
    // creating object of sub class will // invoke the constructor of base class
    Car obj1;
    Bus obj2;
    return 0;
}
```

```
class A {  
public:  
int x, y;  
void A_input(){  
cout<<"Enter two values of class A: ";  
cin>>x>>y;  
}  
};  
class B : public A { // B is derived from A  
public:  
void product(){  
cout<<"The Product of the two values is: "<< x * y<<endl;  
}  
};  
class C : public A{ //C is derived from A  
public:  
void division(){  
cout<<"The Division of the two values is: "<< x / y<<endl;  
}  
};  
  
int main(){  
cout<<"Welcome"<<endl<<endl;  
  
B b; // Object b of derived class B  
C c; // Object c of derived class C  
b.A_input();  
b.product();  
c.A_input();  
c.division();  
return 0;  
}
```

5. Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.
Below image shows the combination of hierarchical and multiple inheritance:



```
#include <iostream>      using namespace std;
class Vehicle { // base class
public:
    Vehicle() {
        cout << "This is a Vehicle" << endl;
    }
};
class Fare { //base class
public:
    Fare() {
        cout<<"Fare of Vehicle\n";
    }
};
class Car: public Vehicle { // first sub class
};
class Bus: public Vehicle, public Fare { // second sub class
};

// C++ program for Hybrid Inheritance
```

```
int main()
{
    // creating object of sub class will
    // invoke the constructor of base
    // class
    Bus obj2;
    return 0;
}
```

```
#include<iostream>
using namespace std;
class A {
public:
    A() { cout << "A's constructor called" << endl; }
};

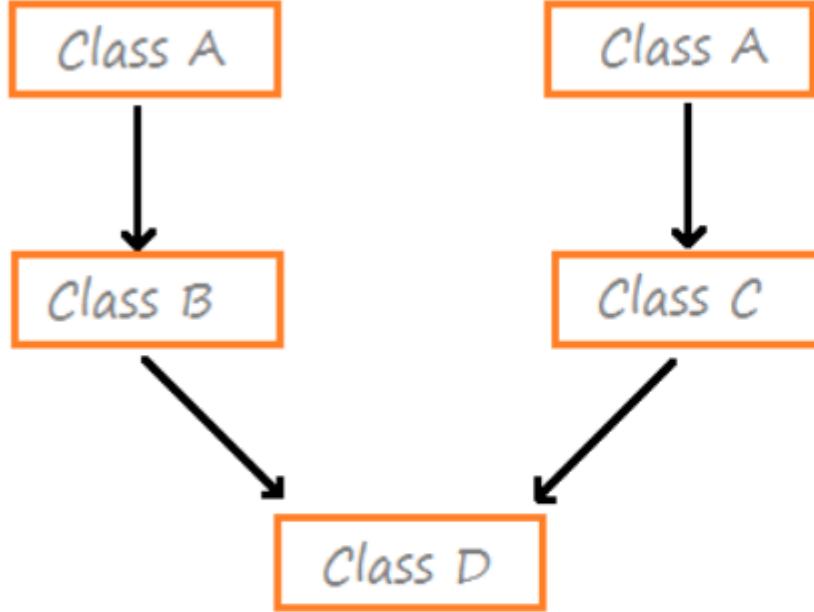
class B {
public:
    B() { cout << "B's constructor called" << endl; }
};

class C: public B, public A { // Note the order
public:
    C() { cout << "C's constructor called" << endl; }
};

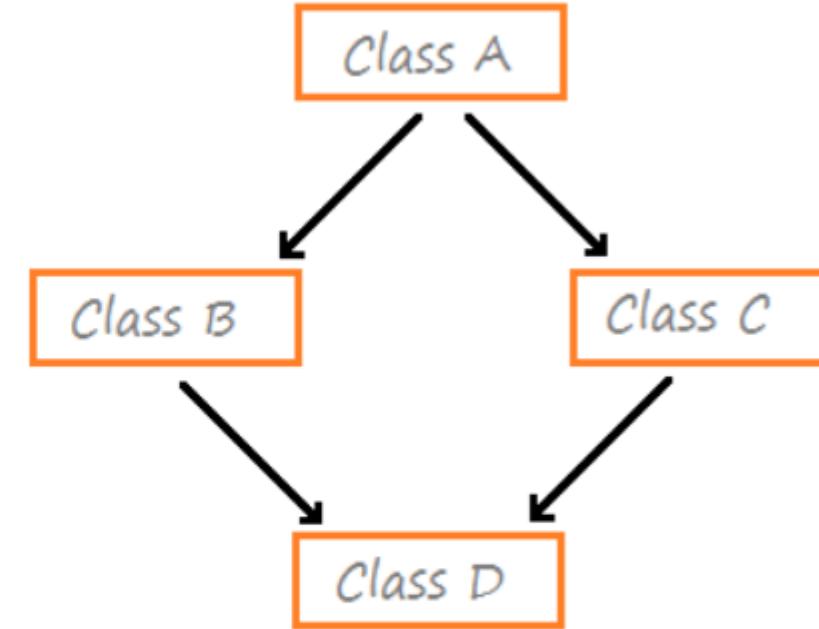
int main() {
    C c;
    return 0;
}
```

Output:
B's constructor called
A's constructor called
C's constructor called

The diamond problem



Without Virtual Inheritance



With Virtual Inheritance

```
using namespace std;
class A{
public:
    A(){}
    void name(){
        cout << "This is class A \n";
    }
};

class B: public A{
public:
    B(){}
};

class C: public A{
public:
    C(){}
};

class D: public B, public C{
public:
    D(){}
};
```

```
int main(){
    D d;
    d.name();
    return 0;
}
```

4 inheritance are taking place in the above example:

Class B inherits Class A.
Class C inherits Class A.
Class D inherits Class B and Class C

- There is no syntactical error in the above program but still, if you try to compile then it will give the following compilation error:

line 35 | error: request for member ‘name’ is ambiguous

- This is because there are two instances of name() method, one inherited by class B and the other one inherited by class C.
- So, the compiler gets confused and is not able to decide which name() method to call?
- This ambiguity arose because of the presence of multiple instances of ‘class A’.
- Virtual inheritance is a technique in C++ that ensures that only one copy or instance of base class’s member variables is inherited by grandchild derived class.
- It means that there will be only one instance of class A i.e B::A and C::A are same, and the same would be inherited and called by D.
- To create a virtual inheritance **virtual** keyword is used.

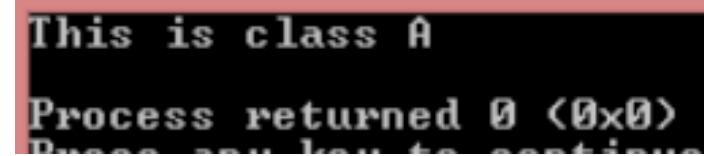
```
class A{
public:
    A(){}
    void name(){
        cout << "This is class A \n";
    }
};

class B: virtual public A{
public:
    B(){}
};

class C: virtual public A{
public:
    C(){}
};

class D: public B, public C{
public:
    D(){}
};
```

```
int main()
{
    D d;
    d.name();
    return 0;
}
```



```
This is class A
Process returned 0 (0x0)
Press any key to continue . . .
```


Modes of Inheritance

- **Public mode:** If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
- **Protected mode:** If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
- **Private mode:** If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.
- **Note :** The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed.

Base class member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types of Inheritance in C++

// C++ Implementation to show that a derived class // doesn't inherit access to private data members. // However, it does inherit a full parent object

```
class A {  
public:  
    int x;  
protected:  
    int y;  
private:  
    int z;  
};  
  
class B : public A {  
    // x is public  // y is protected  // z is not accessible from B  
};  
  
class C : protected A {  
    // x is protected  // y is protected  // z is not accessible from C  
};  
  
class D : private A { // 'private' is default for classes  
    // x is private  // y is private  // z is not accessible from D  
};
```

```
class Shape {      // Base class Shape
public:
    void setWidth(int w) {
        width = w;
    }
    void setHeight(int h) {
        height = h;
    }
protected:
    int width;
    int height;
};

class PaintCost { // Base class PaintCost
public:
    int getCost(int area) {
        return area * 70;
    }
};

class Rectangle: public Shape, public PaintCost { // Derived class
public:
    int getArea() {
        return (width * height);
    }
};
```

ÖR:

```
int main(void) {
    Rectangle Rect;
    int area;

    Rect.setWidth(5);
    Rect.setHeight(7);

    area = Rect.getArea();

    // Print the area of the object.
    cout << "Total area: " << Rect.getArea() << endl;

    // Print the total cost of painting
    cout << "Total paint cost: $" << Rect.getCost(area) << endl;

    return 0;
}
```

Order of Constructor Call with Inheritance in C++

- Base class constructors are always called in the derived class constructors.
- Whenever you create derived class object,
 - first the base class default constructor is executed
 - and then the derived class's constructor finishes execution.
- Whether derived class's default constructor is called or parameterised is called, base class's default constructor is always called inside them.
- To call base class's parameterised constructor inside derived class's parameterised constructor, we must mention it **explicitly** while declaring derived class's parameterized constructor

```
class Base{
    int x;
public:
    Base() { // default constructor
        cout << "Base default constructor\n";
    }
};

class Derived : public Base{
    int y;
public:
    Derived() { // default constructor
        cout << "Derived default constructor\n";
    }
    Derived(int i) { // parameterized constructor
        cout << "Derived parameterized constructor\n";
    }
};

int main(){
    Base b;
    Derived d1;
    Derived d2(10);
}
```

OUTPUT:

```
Base default constructor
Base default constructor
Derived default constructor
Base default constructor
Derived parameterized constructor
```

```
class Parent1 { // first base class
    public:
        Parent1() { // first base class's Constructor
            cout << "Inside first base class" << endl;
        }
    };
class Parent2 { // second base class
    public:
        Parent2() { // second base class's Constructor
            cout << "Inside second base class" << endl;
        }
    };
class Child : public Parent1, public Parent2 { // child class inherits Parent1 and Parent2
    public:
        Child() { // child class's Constructor
            cout << "Inside child class" << endl;
        }
    };
int main() { // main function
    Child obj1; // creating object of class Child
    return 0;
}
```

Output:

```
Inside first base class
Inside second base class
Inside child class
```

```
class Base{  
    int x;  
    public:  
        Base(int i) { // parameterized constructor  
            x = i;  
            cout << "Base Parameterized Constructor\n";  
        }  
};  
class Derived : public Base{  
    int y;  
    public:  
        Derived(int j):Base(j) { // parameterized constructor  
            y = j;  
            cout << "Derived Parameterized Constructor\n";  
        }  
};  
int main(){  
    Derived d(10);  
}
```

OUTPUT:

Base Parameterized Constructor
Derived Parameterized Constructor

Base class Parameterized Constructor in Derived class Constructor:

We can explicitly mention to call the Base class's parameterized constructor when Derived class's parameterized constructor is called.

Whenever the derived class's default constructor is called, the base class's default constructor is called automatically.

To call the parameterised constructor of base class inside the parameterised constructor of sub class, we have to mention it explicitly.

The parameterised constructor of base class cannot be called in default constructor of sub class, it should be called in the parameterised constructor of sub class.

```
#include <iostream>
using namespace std;
class MyBaseClass {
public:
    MyBaseClass(int x) {
        cout << "Constructor of base class: " << x << endl;
    }
};
class MyDerivedClass : public MyBaseClass { //base constructor as initializer list
public:
    MyDerivedClass(int y) : MyBaseClass(50) {
        cout << "Constructor of derived class: " << y << endl;
    }
};
int main() {
    MyDerivedClass derived(100);
}
```

Output

```
Constructor of base class: 50
Constructor of derived class: 100
```

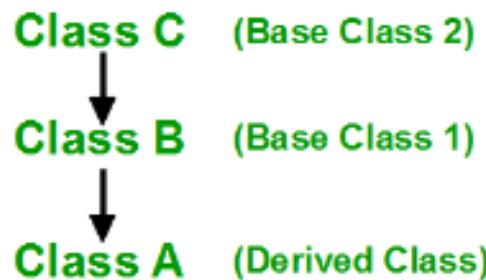
```
class Rectangle{  
private :  
    float length;  
    float width;  
public:  
    Rectangle ()  
    {  
        length = 0;  
        width = 0;  
    }  
    Rectangle (float len,  
    float wid) {  
        length = len;  
        width = wid;  
    }  
    float area() {  
        return length * width ;  
    }  
};
```

```
klas Box : public Rectangle{  
private :  
    float height;  
public:  
    Box () {  
        height = 0;  
    }  
    Box (float len, float wid, float ht) : Rectangle(len, wid) {  
        height = ht;  
    }  
    float volume() {  
        return area() * height;  
    }  
};  
  
int main ()  
{  
    Box bx;  
    Box cx(4,8,5);  
    cout << bx.volume() << endl;  
    cout << cx.volume() << endl;  
    return 0;  
}
```

output :
0
160

Order of constructor and Destructor call for a given order of Inheritance

Order of Inheritance



Order of Constructor Call

1. **C()** (Class C's Constructor)

2. **B()** (Class B's Constructor)

3. **A()** (Class A's Constructor)

Order of Destructor Call

1. **~A()** (Class A's Destructor)

2. **~B()** (Class B's Destructor)

3. **~C()** (Class C's Destructor)

Functions that are never Inherited

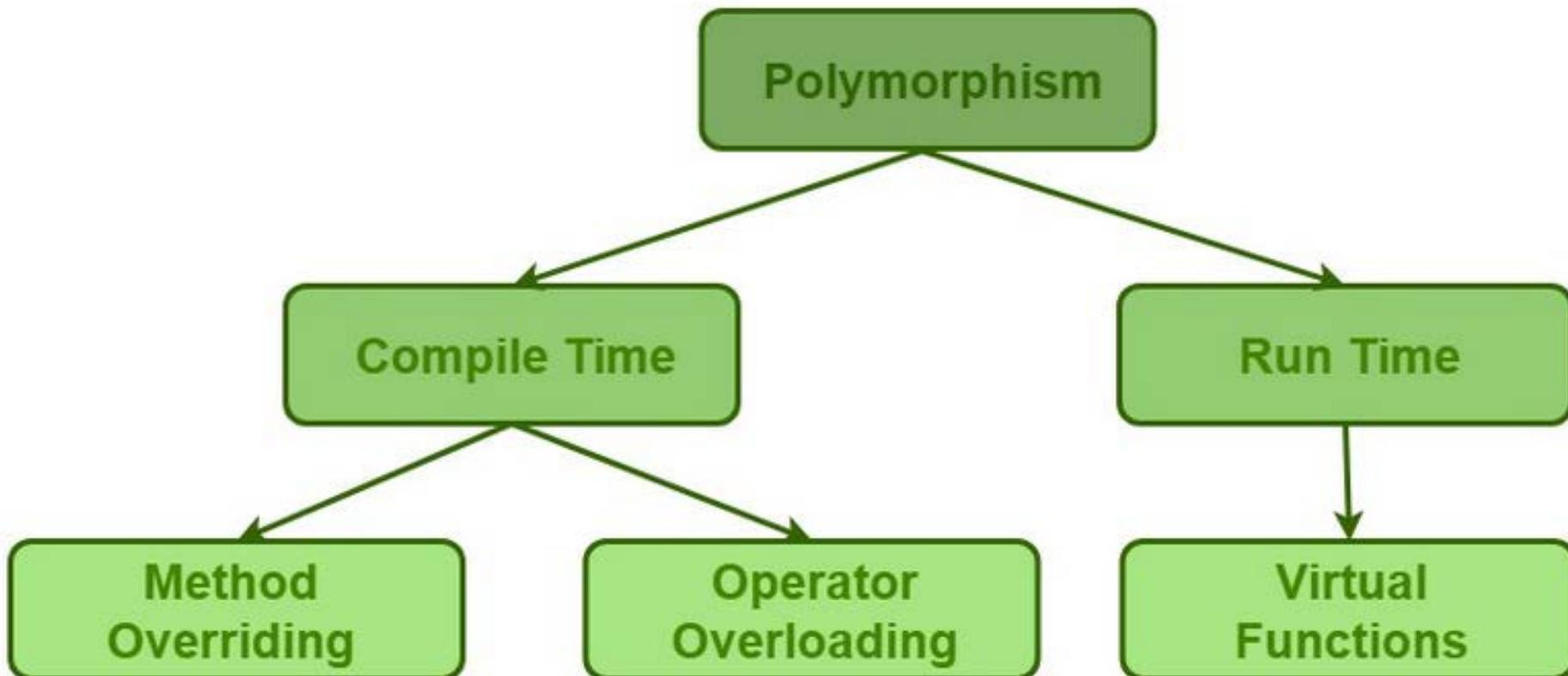
- Constructors and Destructors are never inherited and hence never overrided.(We will study the concept of function overriding in the next tutorial)
- Also, assignment operator = is never inherited. It can be overloaded but can't be inherited by sub class

Polymorphism in C++

- The word polymorphism means having many forms.
- In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
- **Real life example of polymorphism, a person at the same time can have different characteristic. Like a man at the same time is**
 - a father,
 - a husband,
 - an employee.
- **So the same person posses different behavior in different situations. This is called polymorphism.**
- Polymorphism is considered as one of the important features of Object Oriented Programming.
- In C++ polymorphism is mainly divided into two types:

Compile time Polymorphism

Runtime Polymorphism



- **Compile time polymorphism:** This type of polymorphism is achieved by function overloading or operator overloading.
- **Function Overloading:** When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**.
- Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

```
#include <iostream>
using namespace std;
class printData {
public:
    void print(int i) {
        cout << "Printing int: " << i << endl;
    }
    void print(double f) {
        cout << "Printing float: " << f << endl;
    }
    void print(char* c) {
        cout << "Printing character: " << c << endl;
    }
};
int main(void) {
    printData pd;
    pd.print(5); // Call print to print integer
    pd.print(500.263); // Call print to print float
    pd.print("Hello C++"); // Call print to print character
    return 0;
}
```

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

```
#include<iostream>      using namespace std;
class Geeks {
public:
void func(int x)  { // function with 1 int parameter
    cout << "value of x is " << x << endl;
}
void func(double x) { // function with same name but 1 double parameter
    cout << "value of x is " << x << endl;
}
void func(int x, int y) { // function with same name and 2 int parameters
    cout << "value of x and y is " << x << ", " << y << endl;
}
int main() {
    Geeks obj1;
    obj1.func(7); // Which function is called will depend on the parameters passed
obj1.func(9.132); // The second 'func' is called
obj1.func(85,64); // The third 'func' is called
    return 0;
}
```

```
using namespace std;
class Sum {
public:
    int add(int num1,int num2){
        return num1 + num2;
    }
    int add(int num1, int num2, int num3){
        return num1 + num2 + num3;
    }
};
int main(void) {
    //Object of class Sum
    Sum obj;

    //This will call the second add function
    cout<<obj.add(10, 20, 30)<<endl;

    //This will call the first add function
    cout<<obj.add(11, 22);
    return 0;
}
```

It's a **compile-time polymorphism** because the compiler knows which function to execute before the program is compiled.

- **Operator Overloading:** C++ also provide option to overload operators.
- For example, we can make the operator ('+') for string class to concatenate two strings.
- We know that this is the addition operator whose task is to add two operands.
- So a single operator ‘+’ when placed between integer operands , adds them and when placed between string operands, concatenates them.

- In C++, we can overload an operator as long as we are operating on user-defined types like objects or structures.
- We cannot use operator overloading for basic types such as int, double, etc.
- Operator overloading is basically function overloading, where different operator functions have the same symbol but different operands.
- And, depending on the operands, different operator functions are executed.

```
#include <iostream>
using namespace std;
class OperatorOverload {
private:
    int x;

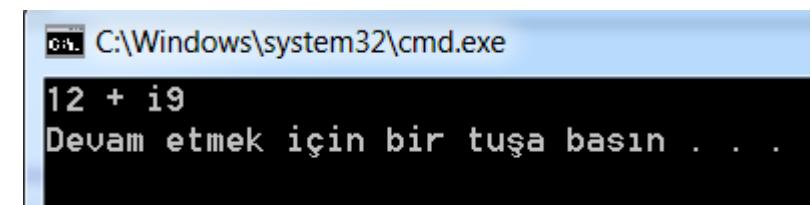
public:
    OperatorOverload() : x(10) {}
    void operator ++() {
        x = x + 2;
    }
    void Print() {
        cout << "The Count is: " << x;
    }
};

int main() {
    OperatorOverload ov;
    ++ov;
    ov.Print();
    return 0;
}
```

The Count is: 12

```
#include<iostream>      using namespace std;          // CPP program to illustrate // Operator Overloading
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i = 0) {real = r;  imag = i;}
    Complex operator + (Complex const &obj) { // This is automatically called when '+' is used with between two
                                                // Complex objects
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << " + " << imag << endl; }
};
int main() {
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

// CPP program to illustrate // Operator Overloading



What is the difference between operator functions and normal functions?

- Operator functions are same as normal functions.
- The only differences are, name of an operator function is always operator keyword followed by symbol of operator and operator
- functions are called when the corresponding operator is used.
- Can we overload all operators?
- Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.

. (dot)

::

?:

sizeof

- **Runtime polymorphism:** This type of polymorphism is achieved by Function Overriding.
- Function overriding on the other hand occurs when a derived class has a definition for one of the member functions of the base class.
- **That base function is said to be overridden.**
- In C++ inheritance, we can have the same function in the base class as well as its derived classes.
- When we call the function using an object of the derived class, the function of the derived class is executed instead of the one in the base class.
- So, different functions are executed depending on the object calling the function.
- This is known as function overriding in C++

```
#include <iostream>
using namespace std;
class BaseClass {
public:
    void disp(){
        cout<<"Function of Parent Class";
    }
};
class DerivedClass: public BaseClass{
public:
    void disp() {
        cout<<"Function of Child Class";
    }
};
int main() {
    DerivedClass obj = DerivedClass();
    obj.disp();
    return 0;
}
```

Function Overriding

To override a function you must have the same signature in child class.

By signature I mean the data type and sequence of parameters.

Here we don't have any parameter in the parent function so we didn't use any parameter in the child function.

Output:

Function of Child Class

```
#include <iostream>
using namespace std;
class Mammal {
public:
    void eat() {
        cout << "Mammals eat...";
    }
};
class Cow: public Mammal {
public:
    void eat() {
        cout << "Cows eat grass...";
    }
};
int main(void) {
    Cow c = Cow();
    c.eat();
    return 0;
}
```

Cows eat grass...

```
class A {  
    int a, b, c;  
  
public:  
  
void add(int x, int y) {  
    a=x;b=y;  
  
    cout<<"addition of a+b is:"<<(a+b)<<endl;  
}  
  
void add(int x, int y, int z) {  
    a=x;b=y;c=z;  
  
    cout<<"addition of a+b+c is:"<<(a+b+c)<<endl;  
}  
  
void print() {  
    cout<<"Class A's method is running"<<endl;  
};  
  
class B : public A {  
  
public:  
  
void print() {  
    cout<<"Class B's method is running"<<endl; } };
```

```
int main() {  
    A a1;  
  
    //method overloading (Compile-time polymorphism)  
    a1.add(6, 5);  
  
    //method overloading (Compile-time polymorphism)  
    a1.add(1, 2, 3);  
  
    B b1;  
    //Method overriding (Run-time polymorphism)  
    b1.print();  
}
```

S.NO	INHERITANCE	POLYMORPHISM
1.	Inheritance is one in which a new class is created (derived class) that inherits the features from the already existing class(Base class).	Whereas polymorphism is that which can be defined in multiple forms.
2.	It is basically applied to classes.	Whereas it is basically applied to functions or methods.
3.	Inheritance supports the concept of reusability and reduces code length in object-oriented programming.	Polymorphism allows the object to decide which form of the function to implement at compile-time (overloading) as well as run-time (overriding).
4.	Inheritance can be single, hybrid, multiple, hierarchical and multilevel inheritance.	Whereas it can be compiled-time polymorphism (overload) as well as run-time polymorphism (overriding).
5.	It is used in pattern designing.	While it is also used in pattern designing.

Compile-Time Polymorphism Vs. Run-Time Polymorphism

Here are the major differences between the two:

Compile-time polymorphism	Run-time polymorphism
It's also called early binding or static polymorphism	It's also called late/dynamic binding or dynamic polymorphism
The method is called/invoked during compile time	The method is called/invoked during run time
Implemented via function overloading and operator overloading	Implemented via method overriding and virtual functions
Example, method overloading. Many methods may have similar names but different number or types of arguments	Example, method overriding. Many methods may have a similar name and the same prototype.
Faster execution since the methods discovery is done during compile time	Slower execution since method discoverer is done during runtime.
Less flexibility for problem-solving is provided since everything is known during compile time.	Much flexibility is provided for solving complex problems since methods are discovered during runtime.

Defining a Pointer of Class type

```
class Simple{  
public:  
    int a;  
};  
  
int main(){  
    Simple obj;  
    Simple* ptr; // Pointer of class type  
    ptr = &obj;  
  
    cout << obj.a;  
    cout << ptr->a; // Accessing member with pointer  
}
```

```
class Data{
public:
int a;
void print()
{
    cout << "a is "<< a;
}
};

int main(){
Data d, *dp;
dp = &d; // pointer to object

int Data::*ptr=&Data::a; // pointer to data member 'a'

d.*ptr=10;
d.print();

dp->*ptr=20;
dp->print();
}
```

Pointer to Data Members of Class

Pointer to Member Functions of Class

return_type (class_name::*ptr_name) (argument_type) = &class_name::function_name;

```
class Data{  
public:  
int f(float)  
{  
    return 1;  
}  
};
```

```
int (Data::*fp1) (float) = &Data::f; // Declaration and assignment  
int (Data::*fp2) (float); // Only Declaration
```

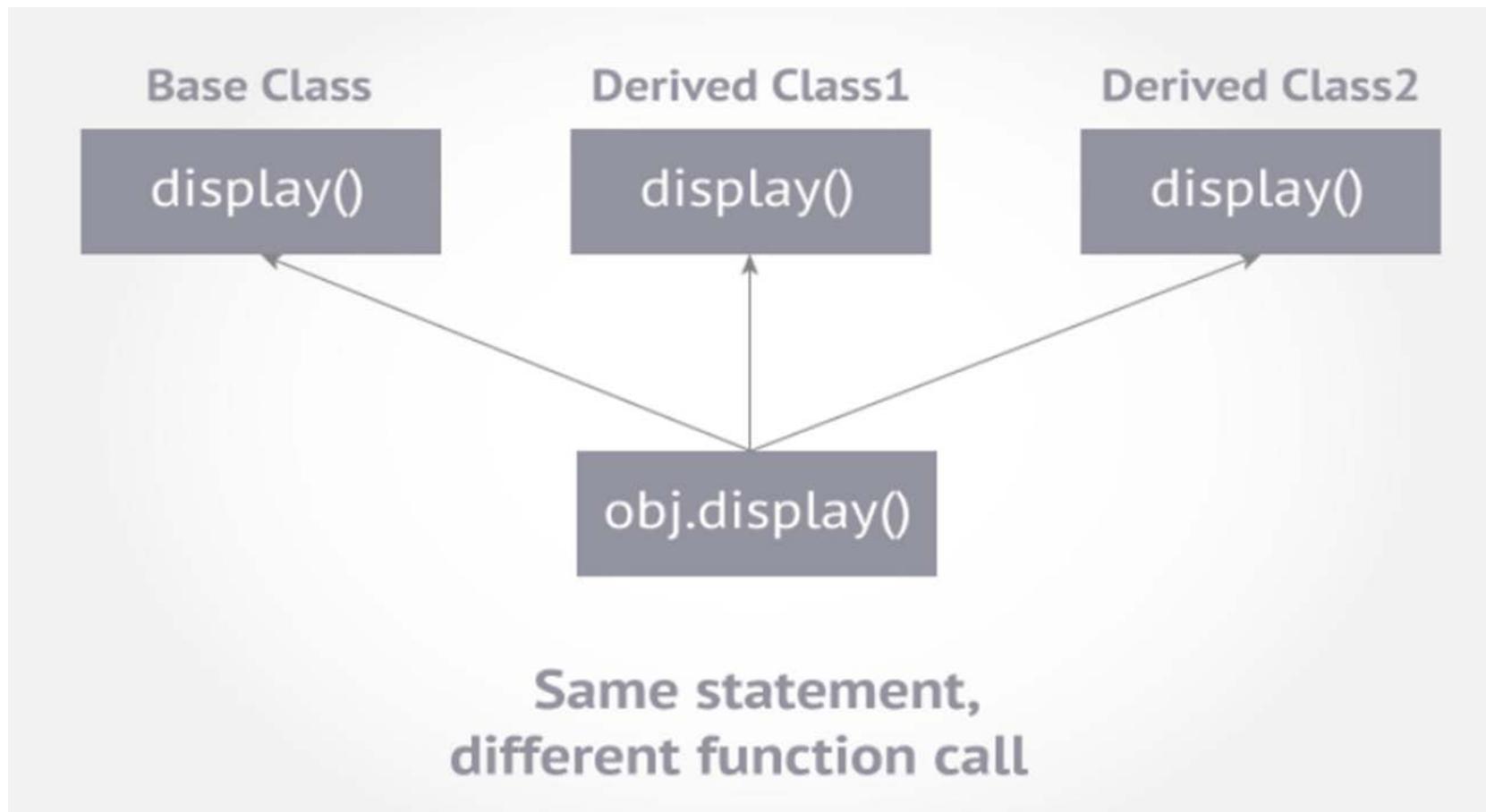
```
int main(){  
    fp2 = &Data::f; // Assignment inside main()  
}
```


C++ Virtual Functions

- A virtual function is another way of **implementing run-time polymorphism** in C++.
- It is a special function defined in a base class and redefined in the derived class.
- **virtual functions allow derived classes to provide different versions of a base class function**
- To declare a virtual function, you should **use the virtual keyword**.
- Using virtual functions in the base class ensures that the function can be overridden in these cases.
- Thus, virtual functions actually fall under **function overriding**.
- Virtual Function is a function in base class, which is overridden in the derived class, and which tells the compiler to perform Late Binding on this function.

- A virtual function a member function which is declared within a base class and is re-defined (Overridden) by a derived class.
- When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.
- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at Run-time.

- Virtual functions cannot be static and also cannot be a friend function of another class.
- Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
- The prototype of virtual functions should be same in base as well as derived class.
- They are always defined in base class and overridden in derived class.
- It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
- A class may have virtual destructor but it cannot have a virtual constructor



```
class Base{
public:
void show()  {
    cout << "Base class";
}
};

class Derived:public Base{
public:
void show()
{
    cout << "Derived Class";
}
};

int main(){
Base* b;    //Base class pointer
Derived d;  //Derived class object
b = &d;
b->show(); //Early Binding Occurs
}
```

OUTPUT:
Base class

```
class Base{
public:
virtual void show()
{
    cout << "Base class\n";
}
};

class Derived:public Base{
public:
void show()
{
    cout << "Derived Class";
}
};

int main(){
Base* b;    //Base class pointer
Derived d;  //Derived class object
b = &d;
b->show(); //Late Binding Occurs
}
```

OUTPUT:
Derived class

```
class base {  
public:  
    void fun_1() { cout << "base-1\n"; }  
    virtual void fun_2() { cout << "base-2\n"; }  
    virtual void fun_3() { cout << "base-3\n"; }  
    virtual void fun_4() { cout << "base-4\n"; }  
};
```

```
class derived : public base {  
public:  
    void fun_1() { cout << "derived-1\n"; }  
    void fun_2() { cout << "derived-2\n"; }  
    void fun_4(int x) { cout << "derived-4\n"; }  
};
```

```
// CPP program to illustrate // working of Virtual Functions  
  
int main() {  
    base* p;  
    derived obj1;  
    p = &obj1;  
    // Early binding because fun1() is non-virtual in base  
    p->fun_1();  
    // Late binding (RTP)  
    p->fun_2();  
    // Late binding (RTP)  
    p->fun_3();  
    // Late binding (RTP)  
    p->fun_4();  
    // Early binding but this function call is illegal  
    // (produces error) because pointer is of base type and  
    // function is of derived class  
    // p->fun_4(5);  
}
```



```

class base {
public:
    virtual void print()  {
        cout << "print base class" << endl;
    }

    void show()  {
        cout << "show base class" << endl;
    }
};

class derived : public base {
public:
    void print()  {
        cout << "print derived class" << endl;
    }

    void show()  {
        cout << "show derived class" << endl;
    }
};

int main() {
    base* bptr;
    derived d;
    bptr = &d;
    bptr->print(); // virtual function, binded at runtime
    bptr->show(); // Non-virtual function, binded at compile time
}

```

Explanation:

- Runtime polymorphism is achieved only through a pointer (or reference) of base class type.
- **Also, a base class pointer can point to the objects of base class as well as to the objects of derived class.**
- In above code, base class pointer 'bptr' contains the address of object 'd' of derived class.
- Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer)
- Early binding(Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be bound at run-time (output is print derived class as pointer is pointing to object of derived class) and
- show() is non-virtual so it will be bound during compile time(output is show base class as pointer is of base type).


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

class Animal{
public:
virtual void my_features()
{
cout << "I am an animal.";
}
};

class Mammal : public Animal{
public:
void my_features()
{
cout << "\nI am a mammal.";
}
};
```

```
class Reptile : public Animal{
public:
void my_features() {
cout << "\nI am a reptile.";
}
};

//intermediate function
void intermediate_func(Animal *a1){
a1->my_features();
}
```

Ör:

```
int main(){
Animal *obj1 = new Animal;
Mammal *obj2 = new Mammal;
Reptile *obj3 = new Reptile;

intermediate_func(obj1);
intermediate_func(obj2);
intermediate_func(obj3);
return 0;
}
```

I am an animal.
I am a mammal.
I am a reptile.

// C++ program to demonstrate the use of virtual function

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

class Animal {
    private:
        string type;

    public:
        // constructor to initialize type
        Animal() : type("Animal") {}

        // declare virtual function
        virtual string getType() {
            return type;
        }
};
```

```
class Dog : public Animal {  
    private:  
        string type;  
    public:  
        Dog() : type("Dog") {} // constructor to initialize type  
        string getType() override {  
            return type;  
        }  
};  
class Cat : public Animal {  
    private:  
        string type;  
    public:  
        Cat() : type("Cat") {} // constructor to initialize type  
        string getType() override {  
            return type;  
        }  
};
```

```
void print(Animal* ani) {  
    cout << "Animal: " << ani->getType() << endl;  
}  
}
```

```
int main() {  
    Animal* animal1 = new Animal();  
    Animal* dog1 = new Dog();  
    Animal* cat1 = new Cat();  
  
    print(animal1);  
    print(dog1);  
    print(cat1);  
  
    return 0;  
}
```

We then call the `print()` function using these pointers:

When `print(animal1)` is called, the pointer points to an `Animal` object. So, the virtual function in `Animal` class is executed inside of `print()`.

When `print(dog1)` is called, the pointer points to a `Dog` object. So, the virtual function is overridden and the function of `Dog` is executed inside of `print()`.

When `print(cat1)` is called, the pointer points to a `Cat` object. So, the virtual function is overridden and the function of `Cat` is executed inside of `print()`.



```
#include <iostream>
using namespace std;
class ClassA {
    public:
        virtual void show() {
            cout << "The show() function in base class invoked..." << endl;
        }
};
class ClassB :public ClassA {
public:
    void show() {
        cout << "The show() function in derived class invoked...";
    }
};
int main() {
    ClassA* a;
    ClassB b;
    a = &b;
    a->show();
}
```


- A **pure virtual function** (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it.
- A pure virtual function is declared by assigning 0 in declaration.
- See the following example.

```
// An abstract class
class Test
{
    // Data members of class
public:
    // Pure Virtual Function
    virtual void show() = 0;

    /* Other members */
};
```

Pure virtual functions are used
if a function doesn't have any use in the base class
but the function must be implemented by all its derived classes

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

class Base {
    int x;
public:
    virtual void fun() = 0;
    int getX() { return x; }
};

// This class inherits from Base and implements fun()
class Derived: public Base {
    int y;
public:
    void fun() { cout << "fun() called"; }
};

int main(void) {
    Derived d;
    d.fun();
    return 0;
}
```

```
#include<iostream>
using namespace std;
class B {
public:
    virtual void s() = 0; // Pure Virtual Function
};

class D:public B {
public:
    void s() {
        cout << "Virtual Function in Derived class\n";
    }
};

int main() {
    B *b;
    D dobj;
    b = &dobj;
    b->s();
}
```

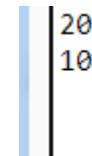
Output

```
Virtual Function in Derived class
```

- Suppose, we have derived **Triangle**, **Square** and **Circle** classes from the **Shape** class, and we want to calculate the area of all these shapes.
- In this case, we can create a pure virtual function named **calculateArea()** in the Shape.
- Since it's a pure virtual function, all derived classes Triangle, Square and Circle must include the **calculateArea()** function with implementation.
- A pure virtual function doesn't have the function body and it must end with = 0.

```
class Shape {  
public:  
    // creating a pure virtual function  
    virtual void calculateArea() = 0;  
};
```

```
class Polygon {  
protected:  
    int width, height;  
public:  
    Polygon (int a, int b) : width(a), height(b) {}  
    virtual int area (void) =0;  
    void printarea()  
    { cout << this->area() << '\n'; }  
};  
class Rectangle: public Polygon {  
public:  
    Rectangle(int a,int b) : Polygon(a,b) {}  
    int area()  
    { return width*height; }  
};  
class Triangle: public Polygon {  
public:  
    Triangle(int a,int b) : Polygon(a,b) {}  
    int area()  
    { return width*height/2; }  
};  
  
int main () {  
    Polygon * ppoly1 = new Rectangle (4,5);  
    Polygon * ppoly2 = new Triangle (4,5);  
    ppoly1->printarea();  
    ppoly2->printarea();  
    delete ppoly1;  
    delete ppoly2;  
    return 0;  
}
```



Friend class and function in C++

Friend Class

- A friend class can access private and protected members of other class in which it is declared as friend.
- It is sometimes useful to allow a particular class to access private members of other class.
- For example a `LinkedList` class may be allowed to access private members of `Node`.

```
class Node {  
private:  
    int key;  
    Node* next;  
    /* Other members of Node Class */  
    // Now class LinkedList can  
    // access private members of Node  
    friend class LinkedList;  
};
```

Friend Function

- Like friend class, a friend function can be given special grant to access private and protected members.

A friend function can be:

- a) A method of another class
- b) A global function

```
class Node {  
private:  
    int key;  
    Node* next;  
  
    /* Other members of Node Class */  
    friend int LinkedList::search();  
    // Only search() of linkedList  
    // can access internal members  
};
```

Following are some important points about friend functions and classes:

- 1) Friends should be used only for limited purpose.
- 2) Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.
- 3) Friendship is not inherited
- 4) The concept of friends is not there in Java.

```
#include <iostream>

class A {
private:
    int a;
public:
    A() { a = 0; }
    friend class B; // Friend Class
};

class B {
private:
    int b;
public:
    void showA(A& x) {
        // Since B is friend of A, it can access      // private members of A
        std::cout << "A::a=" << x.a;
    }
};

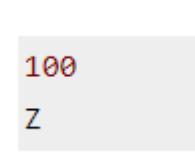
int main() {
    A a;
    B b;
    b.showA(a);
    return 0;
}
```

```
#include <iostream>

class B;
    class A {
        public:
            void showB(B&);

    };
class B {
    private:
        int b;
    public:
        B() { b = 0; }
        friend void A::showB(B& x); // Friend function
    };
void A::showB(B& x) {
    // Since showB() is friend of B, it can    // access private members of B
    std::cout << "B::b = " << x.b;
}
int main() {
    A a;
    B x;
    a.showB(x);
    return 0;
}
```

```
class XYZ {  
private:  
    int num=100;  
    char ch='Z';  
public:  
    friend void disp(XYZ obj);  
};  
//Global Function  
void disp(XYZ obj){  
    cout<<obj.num<<endl;  
    cout<<obj.ch<<endl;  
}  
int main() {  
    XYZ obj;  
    disp(obj);  
    return 0;  
}
```



100
z

```
class XYZ {  
private:  
    char ch='A';  
    int num = 11;  
public:  
/* This statement would make class ABC * a friend class of XYZ, this means that * ABC can access the private and protected  
* members of XYZ class. */  
friend class ABC;  
};  
class ABC {  
public:  
void disp(XYZ obj){  
    cout<<obj.ch<<endl;  
    cout<<obj.num<<endl;  
}  
};  
int main() {  
    ABC obj;  
    XYZ obj2;  
    obj.disp(obj2);  
    return 0;  
}
```

A

11

```
class Rectangle {  
    int width, height;  
public:  
    Rectangle() {}  
    Rectangle (int x, int y) : width(x), height(y) {}  
    int area() {return width * height;}  
    friend Rectangle duplicate (const Rectangle&);  
};
```

```
Rectangle duplicate (const Rectangle& param)  
{  
    Rectangle res;  
    res.width = param.width*2;  
    res.height = param.height*2;  
    return res;  
}
```

```
int main () {  
    Rectangle foo;  
    Rectangle bar (2,3);  
    foo = duplicate (bar);  
    cout << foo.area() << '\n';  
    return 0;  
}
```

A non-member function can access the private and protected members of a class if it is declared a friend of that class.

That is done by including a declaration of this external function within the class, and preceding it with the keyword friend:

The duplicate function is a friend of class Rectangle.

Therefore, function duplicate is able to access the members width and height (which are private) of different objects of type Rectangle.

It simply has access to its private and protected members without being a member.

```
// friend class
class Square;

class Rectangle {
    int width, height;
public:
    int area ()
    {return (width * height);}
    void convert (Square a);
};

class Square {
    friend class Rectangle;
private:
    int side;
public:
    Square (int a) : side(a) {}
};

void Rectangle::convert (Square a) {
    width = a.side;
    height = a.side;
}
```

```
int main () {
    Rectangle rect;
    Square sqr (4);
    rect.convert(sqr);
    cout << rect.area();
    return 0;
}
```

16

Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another class:

In this example, class Rectangle is a friend of class Square allowing Rectangle's member functions to access private and protected members of Square.

More concretely, Rectangle accesses the member variable Square::side, which describes the side of the square.

Templates in C++

- A template is a simple and yet very powerful tool in C++.
- The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types.
- For example, a software company may need sort() for different data types.
- Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.
- C++ adds two new keywords to support templates: 'template' and 'typename'.
- The second keyword can always be replaced by keyword 'class'.

How templates work?

- Templates are expanded at compiler time. This is like macros.
- The difference is, compiler does type checking before template expansion.
- The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class

Compiler internally generates
and adds below code

```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}
```

```
int main()
{
    cout << myMax<int>(3, 7) << endl;
    cout << myMax<char>('g', 'e') << endl;
    return 0;
}
```

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates
and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

Function Templates

- We write a generic function that can be used for different data types.
- Examples of function templates are sort(), max(), min(), printArray().

// One function works for all data types. This would work // even for user defined types if operator '>' is overloaded

```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}
```

```
int main() {
    cout << myMax<int>(3, 7) << endl; // Call myMax for int
    cout << myMax<double>(3.0, 7.0) << endl; // call myMax for double
    cout << myMax<char>('g', 'e') << endl; // call myMax for char
```

```
return 0;
}
```

```
#include <iostream>      using namespace std;
// A template function to implement bubble sort. // We can use this for any data type that supports
// comparison operator < and swap works for it.
template <class T>
void bubbleSort(T a[], int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = n - 1; i < j; j--)
            if (a[j] < a[j - 1])
                swap(a[j], a[j - 1]);
}
int main() { // Driver Code
    int a[5] = {10, 50, 30, 40, 20};
    int n = sizeof(a) / sizeof(a[0]);
    bubbleSort(a, 5); // calls template function
    cout << " Sorted array : ";
    for (int i = 0; i < n; i++)
        cout << a[i] << " ";
    cout << endl;
    return 0;
}
```

// CPP code for bubble sort // using template function

Class Templates

- Like function templates, class templates are useful when a class defines something that is independent of the data type.
- Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.
- Following is a simple example of template Array class

```
template <typename T>
class Array {
private:
    T *ptr;
    int size;
public:
    Array(T arr[], int s);
    void print();
};

template <typename T>
Array<T>::Array(T arr[], int s) {
    ptr = new T[s];
    size = s;
    for(int i = 0; i < size; i++)
        ptr[i] = arr[i];
}

template <typename T>
void Array<T>::print() {
    for (int i = 0; i < size; i++)
        cout<<" "<<*(ptr + i);
    cout<<endl;
}

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    Array<int> a(arr, 5);
    a.print();
    return 0;
}
```

- Can there be more than one arguments to templates?
- Yes, like normal parameters, we can pass more than one data types as arguments to templates. The following example demonstrates the same.

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

template<class T, class U>
class A {
    T x;
    U y;
public:
    A() { cout<<"Constructor Called"<<endl; }
};

int main() {
    A<char, char> a;
    A<int, double> b;
    return 0;
}
```

Can we specify default value for template arguments?

- Yes, like normal parameters, we can specify default arguments to templates.
- The following example demonstrates the same.

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

template<class T, class U = char>
class A {
public:
    T x;
    U y;
    A() { cout<<"Constructor Called"<<endl; }
};

int main() {
    A<char> a; // This will call A<char, char>
    return 0;
}
```


Interfaces and Abstract Classes in C++

Abstract Classes

- Classes that have one or more pure virtual functions are said to be abstract.
- Up to this point, all the classes that we have created or used have been concrete. The difference between concrete and abstract classes is that you may make an instance of a concrete class but you may not make an instance of an abstract class.
- Although you can't have an instance of an abstract class, there are some things that an abstract class can still do. An abstract class can:
 - be a base (parent or super) class
 - have concrete features (both variables and functions) that can be inherited by derived (child or sub) classes
 - participate in polymorphism

Designing Strategy

- An object-oriented system might use an abstract base class to provide a common and standardized interface appropriate for all the external applications.
- Then, through inheritance from that abstract base class, derived classes are formed that operate similarly.
- The capabilities (i.e., the public functions) offered by the external applications are provided as pure virtual functions in the abstract base class.
- The implementations of these pure virtual functions are provided in the derived classes that correspond to the specific types of the application.
- This architecture also allows new applications to be added to a system easily, even after the system has been defined.

```
#include<iostream>
using namespace std;
class Animal{
public:
virtual void sound() = 0; //Pure Virtual Function

void sleeping() {//Normal member Function
    cout<<"Sleeping";
}
};

class Dog: public Animal{
public:
void sound() {
    cout<<"Woof"<<endl;
}
};

int main(){
Dog obj;
obj.sound();
obj.sleeping();
return 0;
}
```

Rules of Abstract Class

- 1) As we have seen that any class that has a pure virtual function is an abstract class.
- 2) We cannot create the instance of abstract class. For example: If I have written this line Animal obj; in the above program, it would have caused compilation error.
- 3) We can create pointer and reference of base abstract class points to the instance of child class. For example, this is valid:

```
Animal *obj = new Dog();  
obj->sound();
```

- 4) Abstract class can have constructors.
- 5) If the derived class does not implement the pure virtual function of parent class then the derived class becomes abstract.

```
#include <iostream>
using namespace std;
// Base class
class Shape {
public:
    // pure virtual function providing interface framework.
    virtual int getArea() = 0;
    void setWidth(int w) {
        width = w;
    }
    void setHeight(int h) {
        height = h;
    }
protected:
    int width;
    int height;
};
```

// Derived classes

```
class Rectangle: public Shape {  
public:  
    int getArea() {  
        return (width * height);  
    }  
};  
  
class Triangle: public Shape {  
public:  
    int getArea() {  
        return (width * height)/2;  
    }  
};  
•
```

```
int main(void) {  
    Rectangle Rect;  
    Triangle Tri;  
  
    Rect.setWidth(5);  
    Rect.setHeight(7);  
  
    // Print the area of the object.  
    cout << "Total Rectangle area: " << Rect.getArea() << endl;  
  
    Tri.setWidth(5);  
    Tri.setHeight(7);  
  
    // Print the area of the object.  
    cout << "Total Triangle area: " << Tri.getArea() << endl;  
  
    return 0;  
}
```

Total Rectangle area: 35
Total Triangle area: 17

```
class Apple{
public:
virtual void price() = 0; // Pure Virtual Function declaration
void ringtone(){ // Member functions
cout<<"The ringtone is: Reflection" << endl;
}
};

class iPhoneX: public Apple{
public:
void price() {
cout<<"The price is: 65,500" << endl;
}
};

int main(){
cout<<"Welcome to DataFlair tutorials" << endl << endl;

iPhoneX i;
i.price();
i.ringtone();
return 0;
}
```

```
// abstract base class  
  
#include <iostream>  
using namespace std;  
  
class Polygon {  
protected:  
    int width, height;  
public:  
    void set_values (int a, int b)  
    { width=a; height=b; }  
    virtual int area (void) =0;  
};  
  
class Rectangle: public Polygon {  
public:  
    int area (void)  
    { return (width * height); }  
};  
  
class Triangle: public Polygon {  
public:  
    int area (void)  
    { return (width * height / 2); }  
};  
  
int main () {  
    Rectangle rect;  
    Triangle trgl;  
    Polygon * ppoly1 = &rect;  
    Polygon * ppoly2 = &trgl;  
    ppoly1->set_values (4,5);  
    ppoly2->set_values (4,5);  
    cout << ppoly1->area() << '\n';  
    cout << ppoly2->area() << '\n';  
    return 0;  
}
```


Interface

- An interface has no implementation.
- An interface class contains only a virtual destructor and pure virtual functions.
- An interface class is a class that specifies the polymorphic interface i.e. pure virtual function declarations into a base class.
- The programmer using a class hierarchy can then do so via a base class that communicates only the interface of classes in the hierarchy.
- Every interface class should have a virtual destructor.
- Virtual destructor makes sure that when a shape is deleted polymorphically, correct destructor of the derived class is invoked.

Differences

- 1 - interfaces can have no state or implementation
- 2 - a class that implements an interface must provide an implementation of all the method of that interface
- 3 - abstract classes may contain state (data members) and/or implementation (methods)
- 4 - abstract classes can be inherited without implementing the abstract methods (though such a derived class is abstract itself)
- 5 -interfaces may be multiple-inherited, abstract classes may not (this is probably the key concrete reason for interfaces to exist separately from abstract classes - they permit an implementation of multiple inheritance that removes many of the problems of general MI).
- 6- If you anticipate creating multiple versions of your component, create an abstract class. Abstract classes provide a simple and easy way to version your components. By updating the base class, all inheriting classes are automatically updated with the change.
- Interfaces, on the other hand, cannot be changed once created. If a new version of an interface is required, you must create a whole new interface.
- 7-If the functionality you are creating will be useful across a wide range of disparate objects, use an interface. Abstract classes should be used primarily for objects that are closely related, whereas interfaces are best suited for providing common functionality to unrelated classes.

```
1 class shape // An interface class
2 {
3     public:
4         virtual ~shape();
5         virtual void move_x(int x) = 0;
6         virtual void move_y(int y) = 0;
7         virtual void draw() = 0;
8 //...
9 };
```

```
class Player {  
public:  
    virtual void play() = 0;  
    virtual void stop() = 0;  
    virtual void pause() = 0;  
    virtual void reverse() = 0;  
};
```

an interface is a class

```
class Recorder: public Player {  
public:  
    virtual void record() = 0;  
};
```

```
class TapePlayer: public Recorder {  
public:  
    void play();  
    void stop();  
    void pause();  
    void reverse();  
    void record();  
};
```