C++

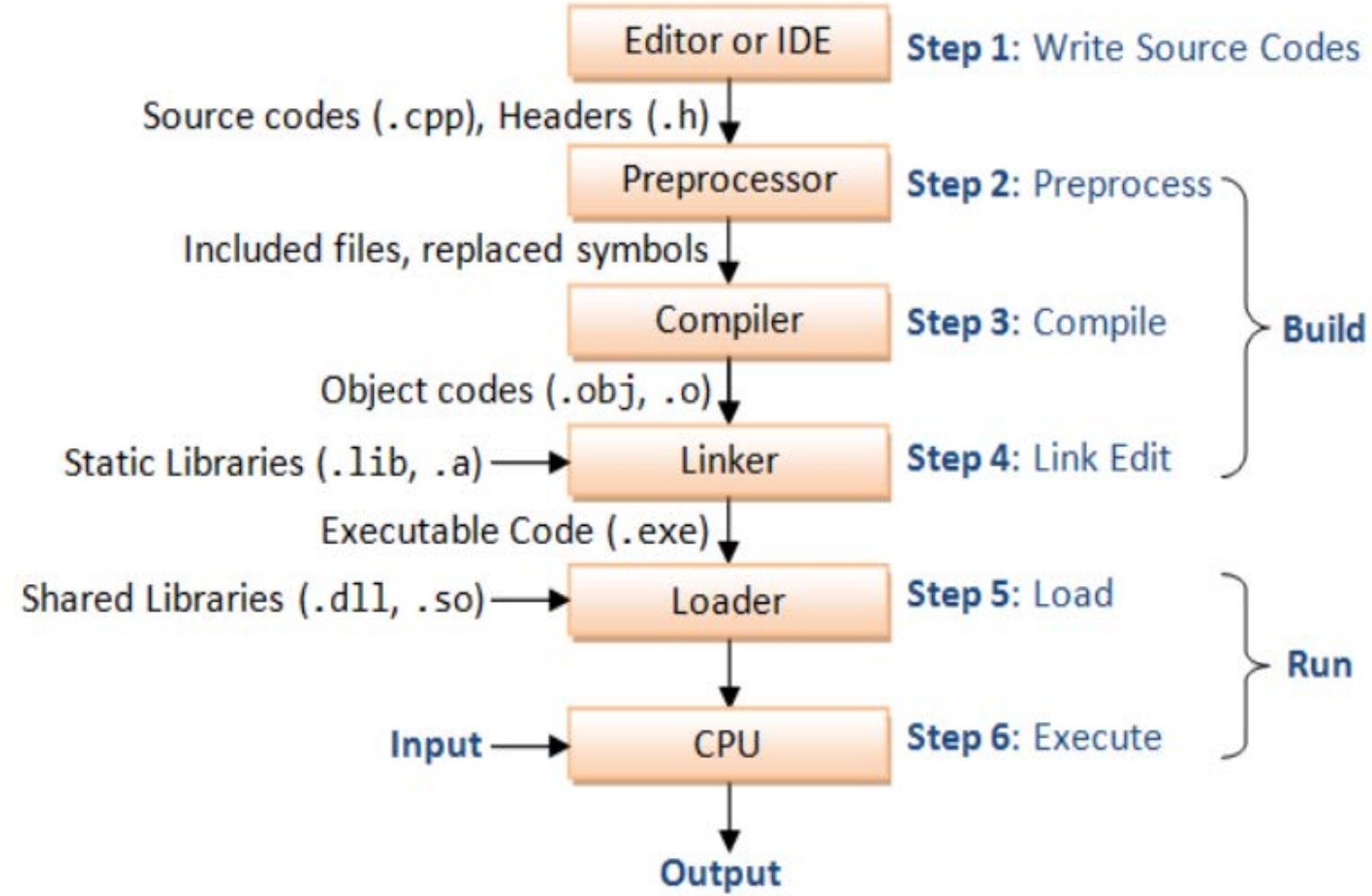
2019

- /** 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 desried 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;
```

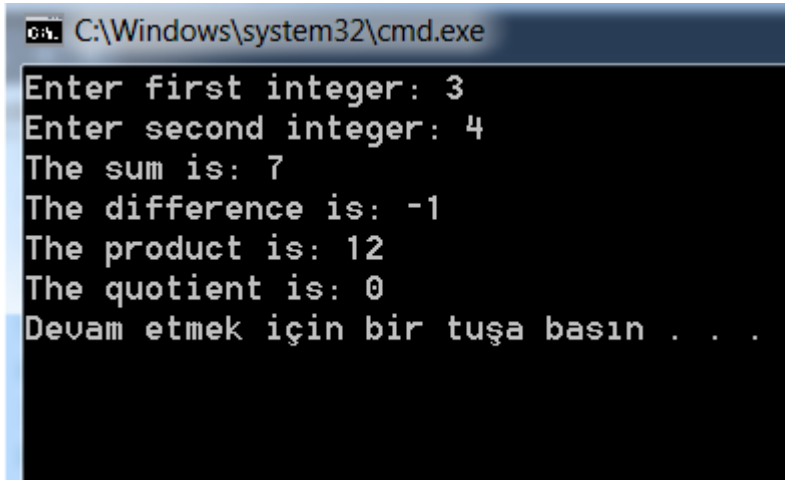
```
hello,
one more time.
543 2.2 1.1
```

```
cout << "hello world, again!\n";
cout << "\thello,\none\tmore\ttime.\n";
```

```
hello world, again!
        hello,
        one     more    time.

_
```

Input via "cin >>"

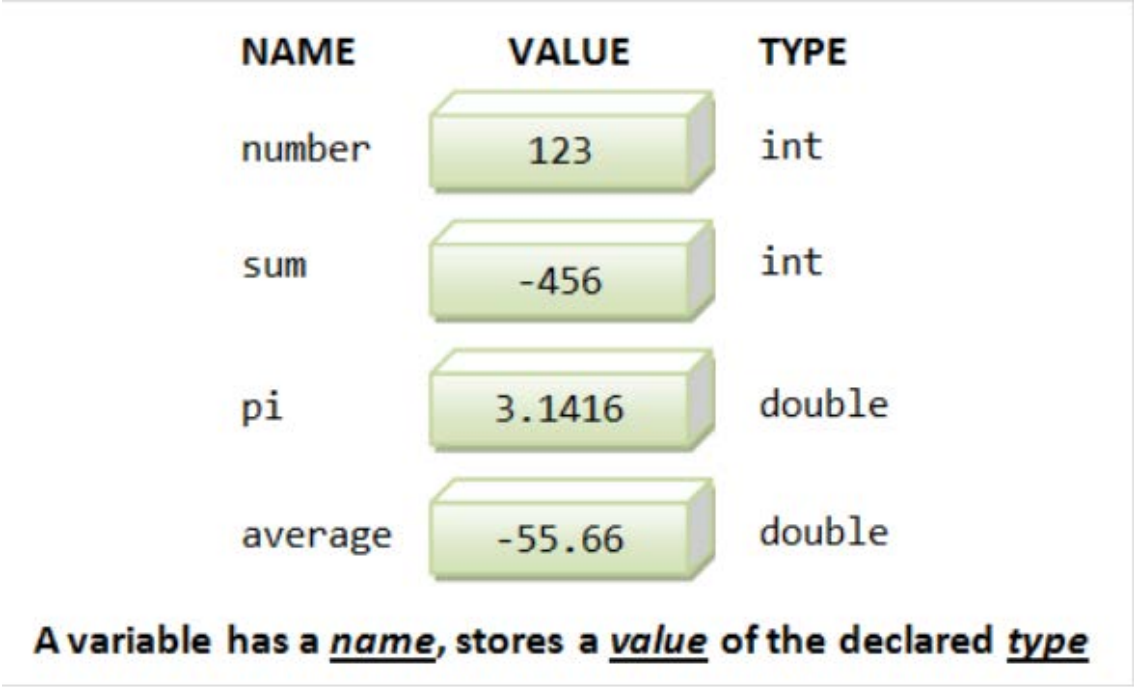
```
#include <iostream>
 using namespace std;
int main() {
   int firstInt, secondInt,  sum, difference, product,
quotient;
   cout << "Enter first integer: ";   // Display a prompting
message
   cin >> firstInt;   // Read input from keyboard (cin) into
firstInt
   cout << "Enter second integer: "; 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



// 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)	++x or x++
--	Decrement by 1 (Unary)	--x or x--

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 → 0 (take note!) and 1.0/2.0 → 0.5.
- Arithmetic operations of an **int and a double produce a double**. Hence, 1.0/2 → 0.5 and 1/2.0 → 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 → 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 shot 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	float	Floating-point number, ≈7 digits (IEEE 754 single-precision floating point format)	4	3.4e38	3.4e-38
	double	Double precision floating-point number, ≈15 digits (IEEE 754 double-precision floating point format)	8	1.7e308	1.7e-308
	long double	Long double precision floating-point number, ≈19 digits (IEEE 754 quadruple-precision floating point format)	12 (8)		
Boolean Numbers	bool	Boolean value of either true or false	1	false (0)	true (1 or non-zero)
Wide Characters	wchar_t char16_t (C++11) char32_t (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 a 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 of 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 contains 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\ta\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
=	*var = expr*	Assign the value of the LHS to the variable at the RHS	x = 5;
+=	*var += expr*	same as *var = var + expr*	x += 5; same as x = x + 5
-=	*var -= expr*	same as *var = var - expr*	x -= 5; same as x = x - 5
*=	*var *= expr*	same as *var = var * expr*	x *= 5; same as x = x * 5
/=	*var /= expr*	same as *var = var / expr*	x /= 5; same as x = x / 5
%=	*var %= expr*	same as *var = var % expr*	x %= 5; same as x = x % 5

Increment/Decrement Operators

Operator	Example	Result
++	x++; ++x	Increment by 1, same as x += 1
--	x--; --x	Decrement by 1, same as x -= 1

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

Relational and Logical Operators

Operator	Description	Usage	Example (x=5, y=8)
==	Equal to	*expr1* == *expr2*	(x == y) → false
!=	Not Equal to	*expr1* != *expr2*	(x != y) → true
>	Greater than	*expr1* > *expr2*	(x > y) → false
>=	Greater than or equal to	*expr1* >= *expr2*	(x >= 5) → true
<	Less than	*expr1* < *expr2*	(y < 8) → false
<=	Less than or equal to	*expr1* >= *expr2*	(y <= 8) → true

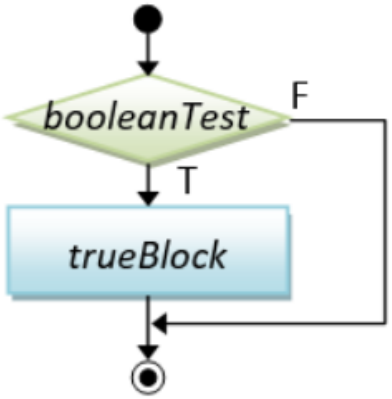
Operator	Description	Usage
&&	Logical AND	*expr1* && *expr2*
||	Logical OR	*expr1* || *expr2*
!	Logical NOT	!*expr*
^	Logical XOR	*expr1* ^ *expr2*

Conditional (Decision) Flow Control

Syntax	Example	Flowchart

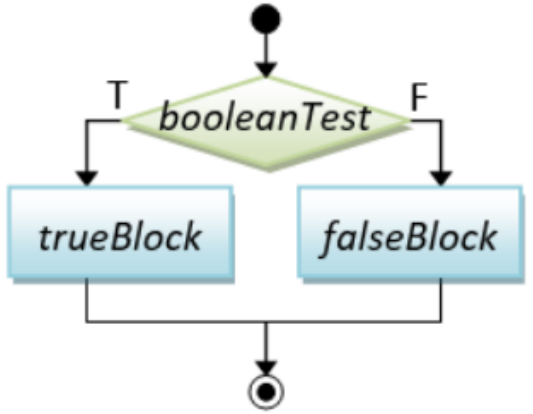
```
// if-then
if ( booleanExpression ) {
   true-block ;
}
```

```
if (mark >= 50) {
   cout << "Congratulation!" << endl;
   cout << "Keep it up!" << endl;
}
```



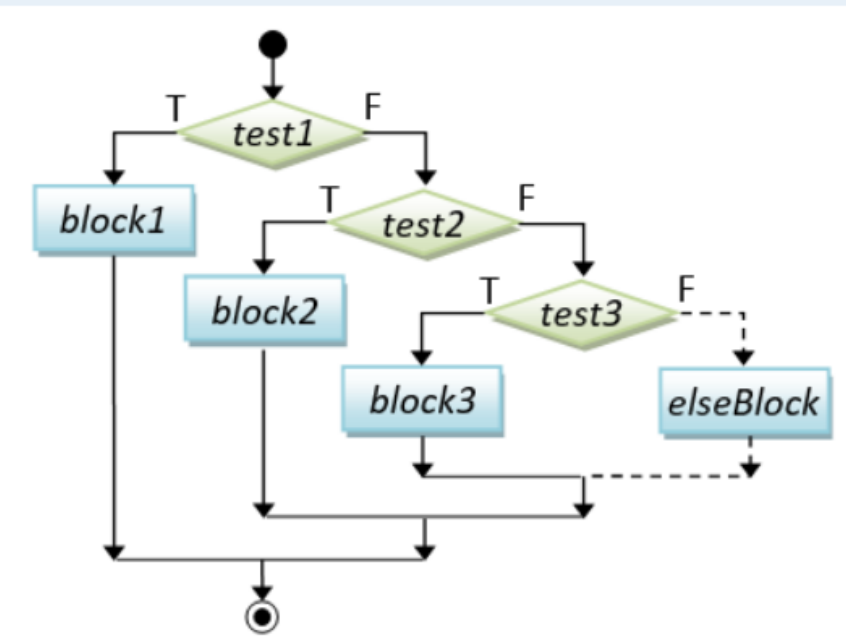

```
// if-then-else
if ( booleanExpression ) {
   true-block ;
} else {
   false-block ;
}
```

```
if (mark >= 50) {
   cout << "Congratulation!" << endl;
   cout << "Keep it up!" << endl;
} else {
   cout << "Try Harder!" << endl;
}
```



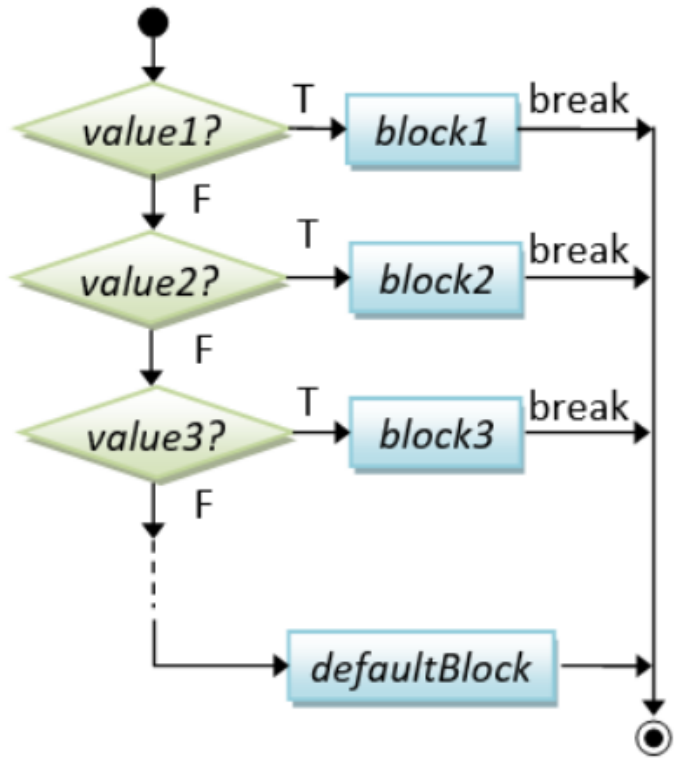
```
// nested-if
if ( booleanExpr-1 ) {
   block-1 ;
} else if ( booleanExpr-2 ) {
   block-2 ;
} else if ( booleanExpr-3 ) {
   block-3 ;
} else if ( booleanExpr-4 ) {
   ......
} else {
   elseBlock ;
}
```

```
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

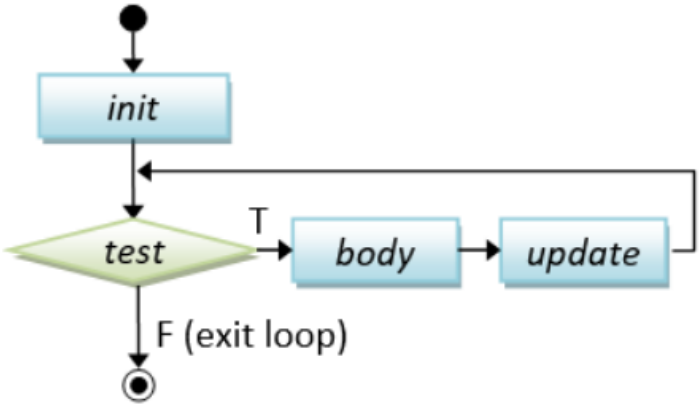
booleanExpr **?** *trueExpr* **:** *falseExpr*	`cout << (mark >= 50) ? "PASS" : "FAIL" << endl;`
	`  // return either "PASS" or "FAIL", and put to cout`
	`max = (a > b) ? a : b;  // RHS returns a or b`
	`abs = (a > 0) ? a : -a;  // RHS returns a or -a`

Loop Flow Control

Syntax	Example	Flowchart

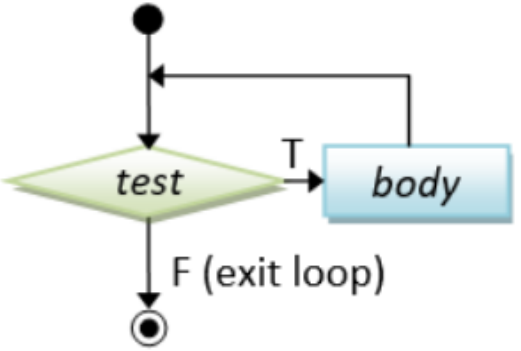
```
// for-loop
for (init; test; post-proc) {
   body ;
}
```

```
// Sum from 1 to 1000
int sum = 0;
for (int number = 1; number <= 1000; ++number) {
   sum += number;
}
```



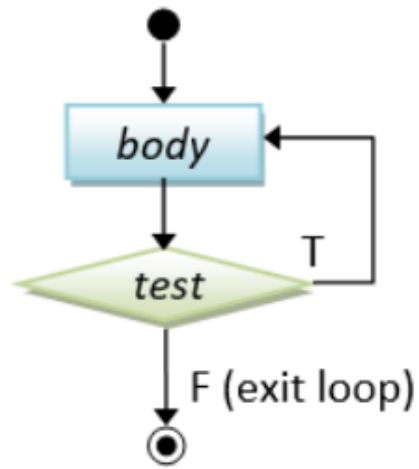

```
// while-do
while ( condition ) {
   body ;
}
```

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



```
// 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.

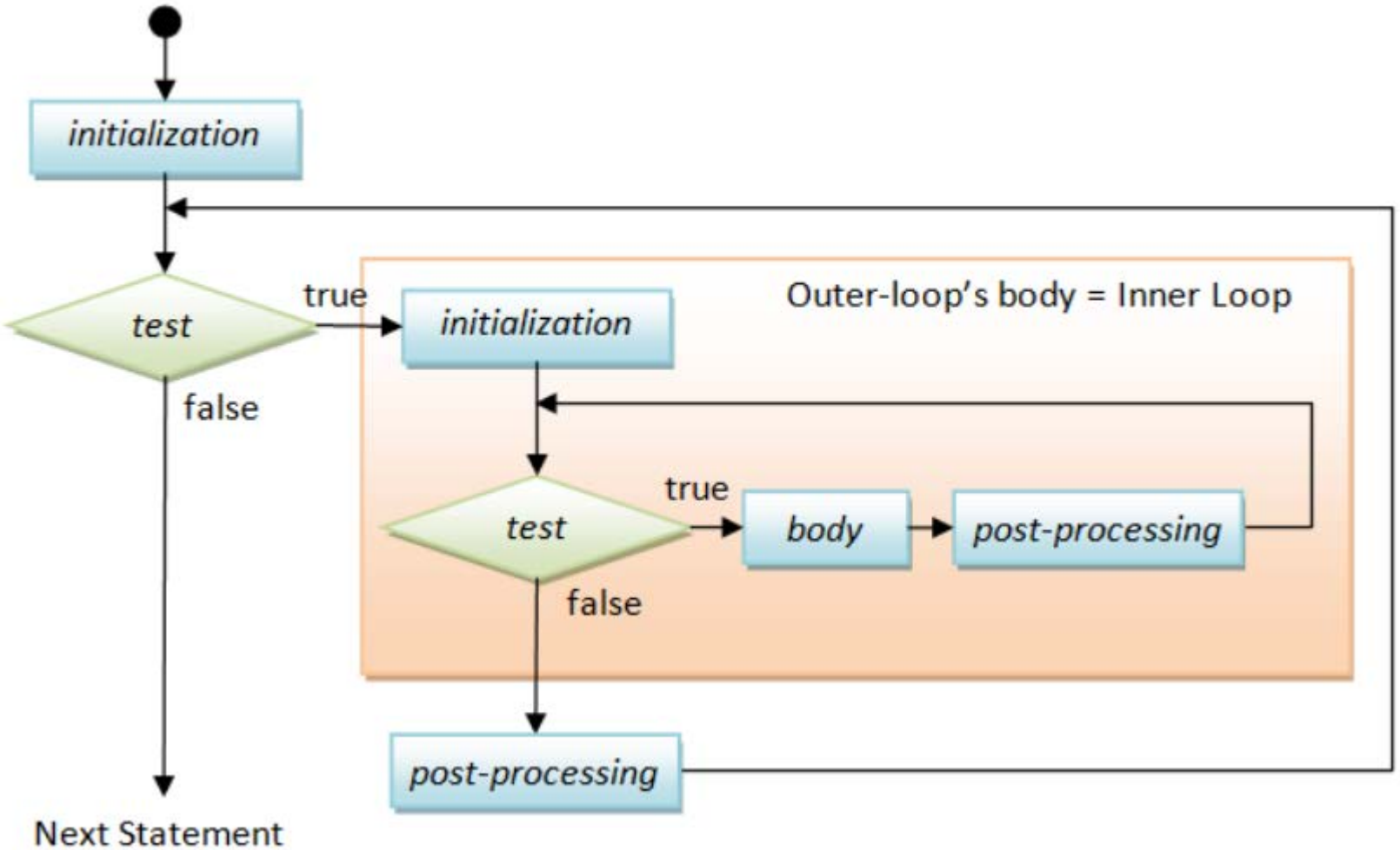
```
if (errorCount > 10) {
  cout << "too many errors" << endl;
  exit(-1);  // Terminate the program// OR abort();
}
```

- 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()
  }
  ...}
```

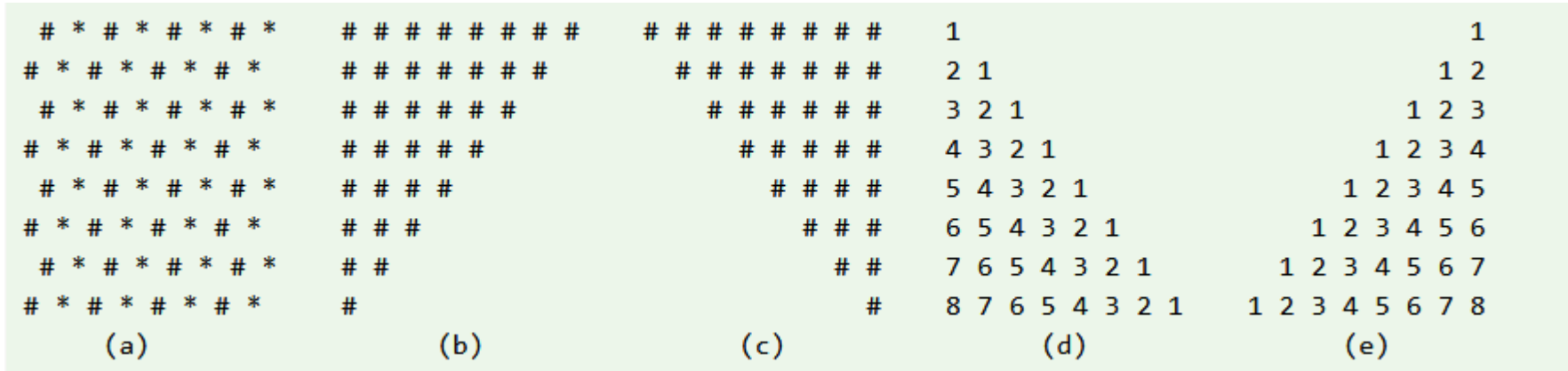
Terminating Program

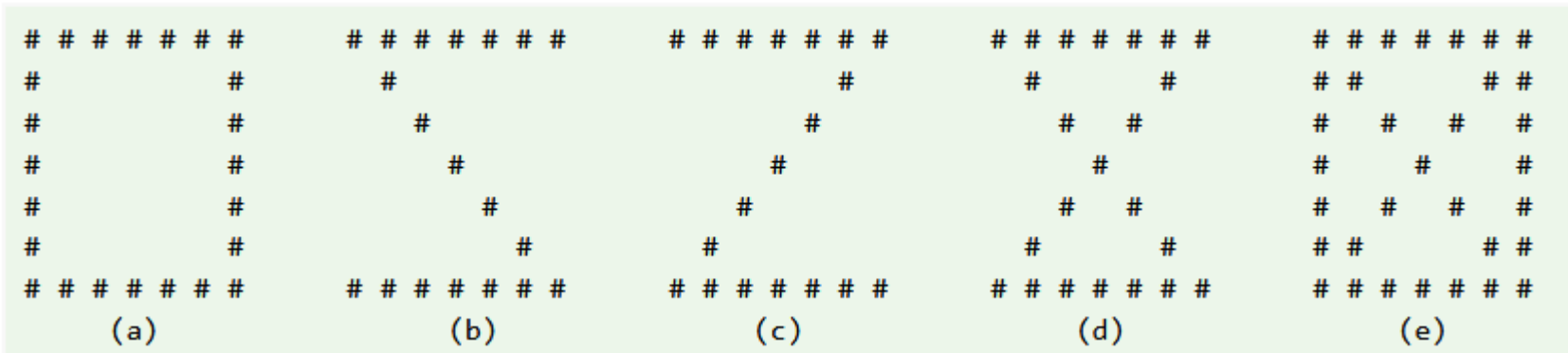
Nested Loops



```
/*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;
}
```

```
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
```


(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,

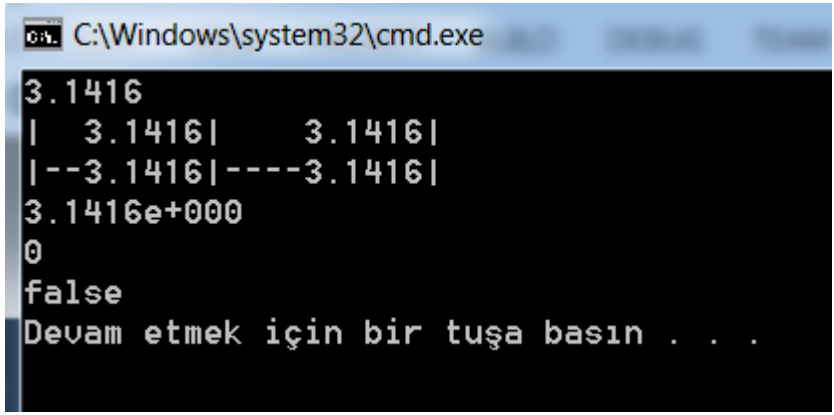
```
#include <algorithm>
......
string str("Hello, world");
replace(str.begin(), str.end(), 'l', '_');
cout << str << endl;      // "He__o, wor_d"
```

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-widht):** 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.

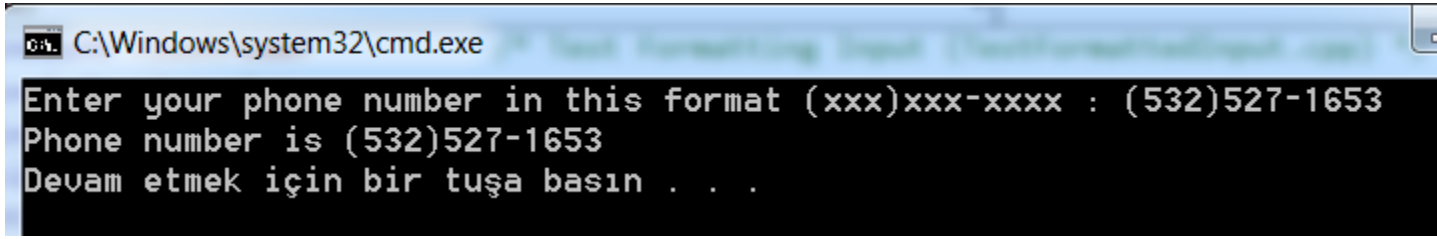
Output Formatting

```
#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;}
```

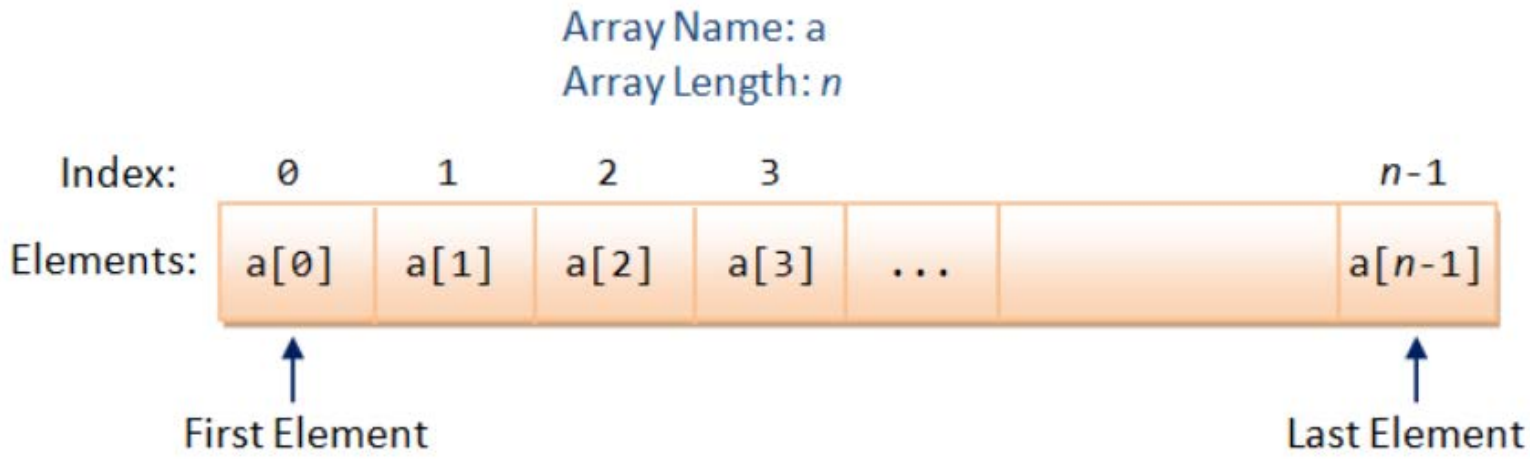


```
#include <iostream>        /* Test Formatting Input (TestFormattedInput.cpp) */
#include <iomanip>     #include <string>
using namespace std;
int main() {
  string areaCode, phoneCode;
  string inStr;
  cout << "Enter your phone number in this format (xxx)xxx-xxxx : ";
  cin.ignore();   // skip '('
  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];
```

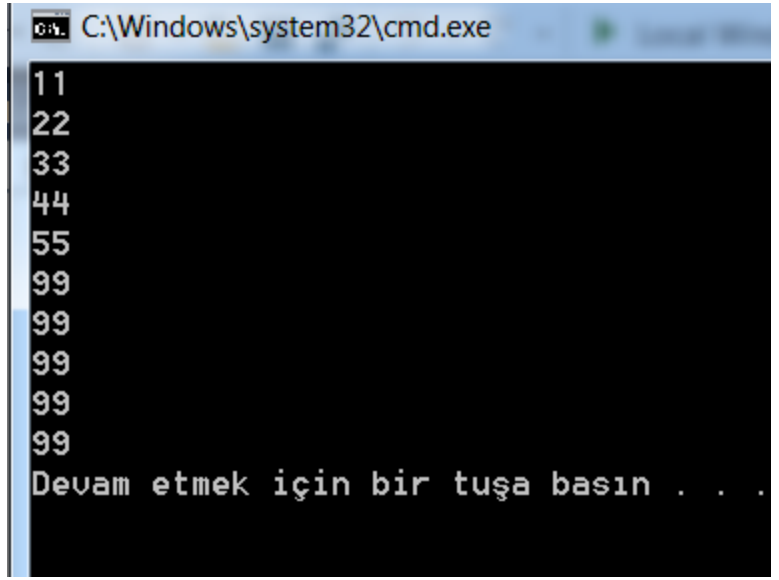
```
// Declare and initialize an int array of 3 elements
int numbers[3] = {11, 33, 44};
// If length is omitted, the compiler counts the elements
int numbers[] = {11, 33, 44};
// Number of elements in the initialization shall be equal to or less than length
int numbers[5] = {11, 33, 44};  // Remaining 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;
}
```

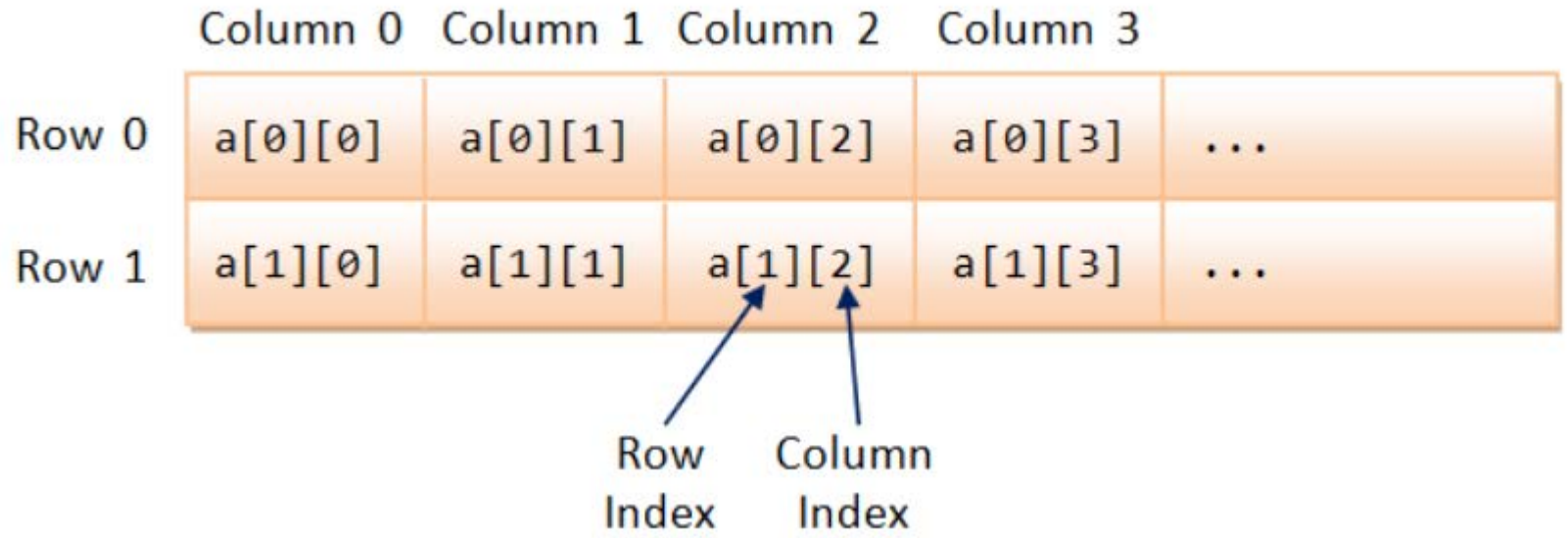
Array and Loop

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;
}
```



Multi-Dimensional Array



```
/* 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;
   }
}
```

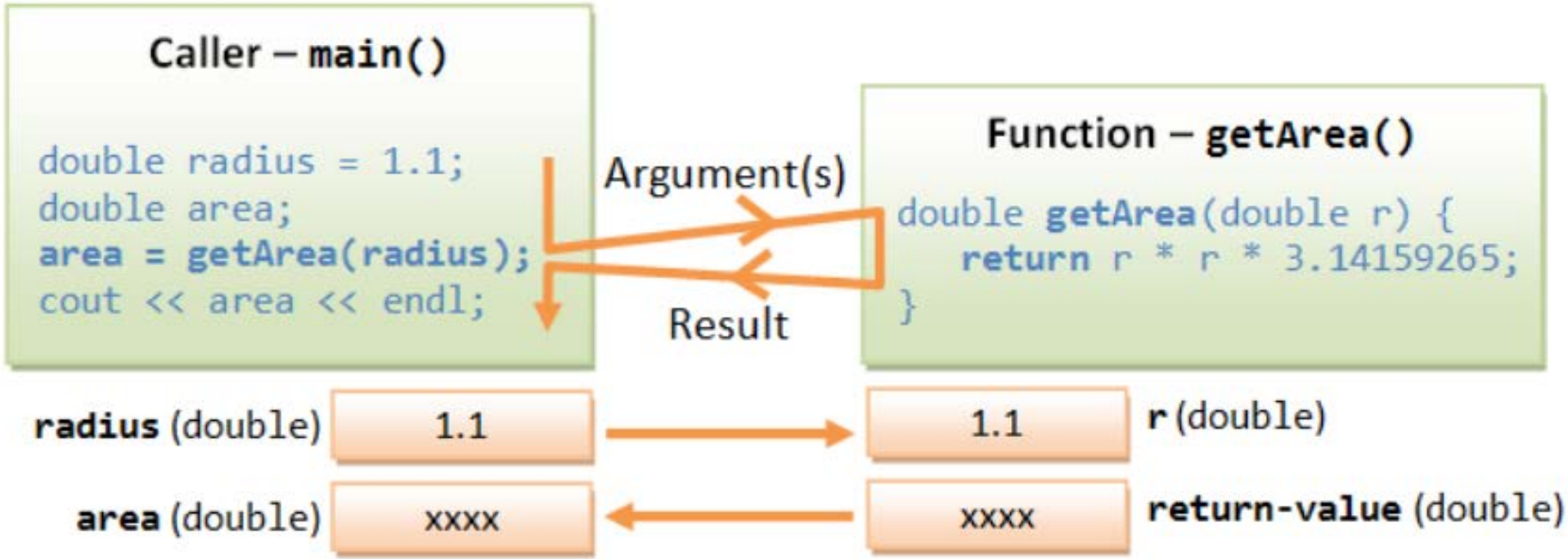
Array of Characters - C-String

- 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)
Result
Function – getArea()
double getArea(double r) {
return r * r * 3.14159265;
}
radius (double)
1.1
1.1
r (double)
area (double)
xxxx
xxxx
return-value (double)

Function Definition

The syntax for function definition is as follows:

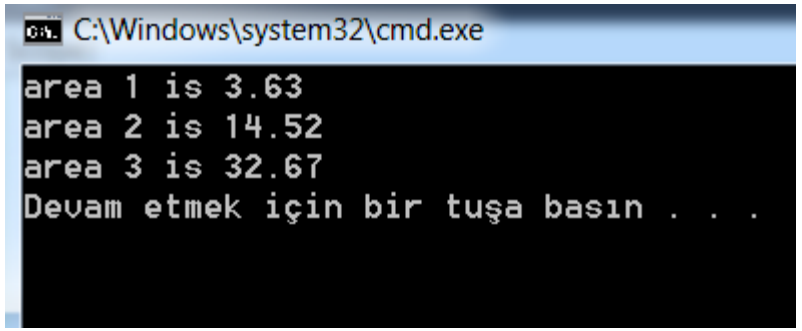
```
returnValueType functionName ( parameterList ) {
   functionBody ;
}
```

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 <iostreamusing 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.**

const Function Parameters

```
#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;
}
```

Pass-by-Reference via "Reference" 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;
}
```

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;
}
```

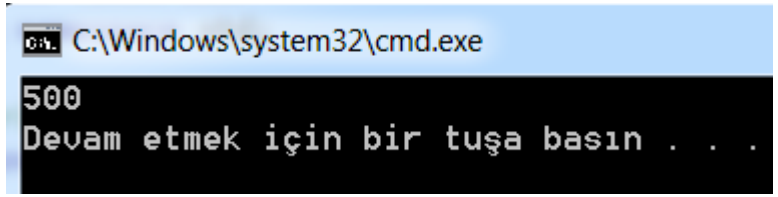
- **When you use different library modules, there is always a potential for name crashes, as different library may use the same name for different purposes**.
- This problem can be resolved via the use of **namespace in C++.** A namespace is a collection for identifiers under the same naming scope. (It 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
}

// To reference the entities, use
myNameSpace::foo
myNameSpace::f()
myNameSpace::Bar
```

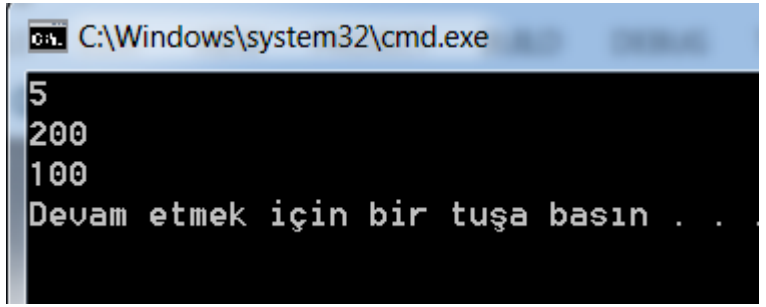
Namespace

- 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 crashes 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,

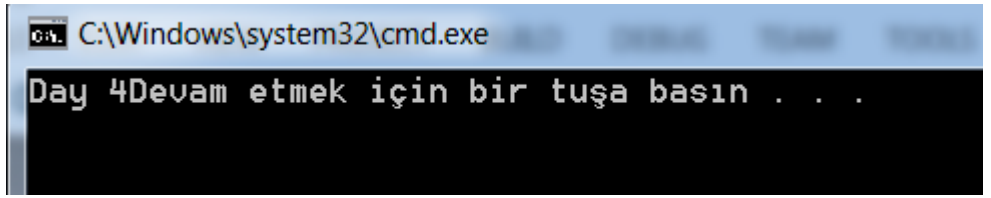
```
enum Color {
   RED, GREEN, BLUE
} myColor;       // Define an enum and declare a variable of the enum
......
myColor = RED;    // Assign a value to an enum
Color yourColor;
yourColor = GREEN;
```

- 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 assigned 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;
   }
```