

C++ Programming Language

Basics

This chapter explains the features, technical details and syntaxes of the C++ programming language. I assume that you could write some simple programs. Otherwise, read "[Introduction To C++ Programming for Novices and First-time Programmers](#)".

To be a proficient programmer, you need to master two things: (1) the syntax of the programming language, and (2) the core libraries (i.e., [API](#)) associated with the language.

1. Introduction to C++

C++ Standards

C++ is standardized as ISO/IEC 14882. Currently, there are two versions:

1. C++98 (ISO/IEC 14882:1998): First standard version of C++.
2. C++03 (ISO/IEC 14882:2003): minor "bug-fix" to C++98 with no change to the language. Commonly refer to as C++98/C++03 or First C++ standard.
3. C++11 (ISO/IEC 14882:2011): Second standard version of C++. Informally called C++0x, as it was expected to finalize in 200x but was not released until 2011. It adds some new features to the language; more significantly, it greatly extends the C++ standard library and standard template library (STL).
4. C++14: Informally called C++1y, is a small extension to C++11, with bug fixes and small improvement.
5. C++17: informally called C++1z.
6. C++2a: the next planned standard in 2020.

C++ Features

TABLE OF CONTENTS (HIDE)

1. Introduction to C++
2. Basic Syntaxes
 - 2.1 Revision
 - 2.2 Comments
 - 2.3 Statements and Blocks
 - 2.4 White Spaces and Formatting
 - 2.5 Preprocessor Directives
3. Variables and Types
 - 3.1 Variables
 - 3.2 Identifiers
 - 3.3 Variable Declaration
 - 3.4 Constants (const)
 - 3.5 Expressions
 - 3.6 Assignment (=)
 - 3.7 Fundamental Types
 - 3.8 Literals for Fundamental Types
4. Operations
 - 4.1 Arithmetic Operators
 - 4.2 Arithmetic Expressions
 - 4.3 Mixed-Type Operations
 - 4.4 Overflow/UnderFlow
 - 4.5 Compound Assignment Operators
 - 4.6 Increment/Decrement Operators
 - 4.7 Implicit Type-Conversion vs. Explicit
 - 4.8 Relational and Logical Operators
5. Flow Control
 - 5.1 Sequential Flow Control
 - 5.2 Conditional (Decision) Flow Control
 - 5.3 Loop Flow Control
 - 5.4 Interrupting Loop Flow - "break"
 - 5.5 Terminating Program
 - 5.6 Nested Loops
 - 5.7 Some Issues in Flow Control

1. *C++ is C.* C++ supports (almost) all the features of C. Like C, C++ allows programmers to manage the memory directly, so as to develop efficient programs.
2. *C++ is OO.* C++ enhances the procedural-oriented C language with the object-oriented extension. The OO extension facilitates design, reuse and maintenance for complex software.
3. *Template C++.* C++ introduces generic programming, via the so-called template. You can apply the same algorithm to different data types.
4. *STL.* C++ provides a huge set of reusable standard libraries, in particular, the Standard Template Library (STL).

C++ Strength and Pitfall

C++ is a powerful language for high-performance applications, including writing operating systems and their subsystems, games and animation. C++ is also a complex and difficult programming language, which is really not meant for dummies. For example, to effectively use the C++ Standard Template Library (STL), you need to understand these difficult concepts: pointers, references, operator overloading and template, on top of the object-oriented programming concepts such as classes and objects, inheritance and polymorphism; and the traditional constructs such as decision and loop. C++ is performance centric. The C++ compiler does not issue warning/error message for many obvious programming mistakes, undefined and unspecified behaviors, such as array index out of range, using an uninitialized variable, etc, due to the focus on performance and efficiency rather than the ease of use - it assumes that those who choose to program in C++ are not dummies.

2. Basic Syntaxes

2.1 Revision

Below is a simple C++ program that illustrates the important programming constructs (*sequential flow*, *while-loop*, and *if-else*) and input/output. Read "[Introduction To C++ Programming for Novices and First-time Programmers](#)" if you need help in understanding this program.

```

1  /*
2   * Sum the odd and even numbers, respectively, from 1 to a given upperbound.
3   * Also compute the absolute difference.
4   * (SumOddEven.cpp)

```

5.8 Exercises

6. Writing Correct and Good Programs
7. Strings
 - 7.1 String Declaration and Initialization
 - 7.2 String Input/Output
 - 7.3 String Operations
 - 7.4 Exercises
8. Formatting Input/Output using
 - 8.1 Output Formatting
 - 8.2 Input Formatting
 - 8.3 Exercises
9. Arrays
 - 9.1 Array Declaration and Usage
 - 9.2 Array and Loop
 - 9.3 Range-based for loop (C++11)
 - 9.4 Multi-Dimensional Array
 - 9.5 Array of Characters - C-String
 - 9.6 Exercises
10. Functions
 - 10.1 Why Functions?
 - 10.2 Using Functions
 - 10.3 Default Arguments
 - 10.4 Function Overloading
 - 10.5 Functions and Arrays
 - 10.6 Pass-by-Value vs. Pass-by-Reference
 - 10.7 const Function Parameters
 - 10.8 Pass-by-Reference via "Reference"
 - 10.9 Mathematical Functions (Header <math.h>)
 - 10.10 Generating Random Numbers
 - 10.11 Exercises
11. File Input/Output (Header <fstream.h>)
 - 11.1 Example: File IO
 - 11.2 Exercises
12. Namespace

```
5  */
6  #include <iostream>    // Needed to use IO functions
7  using namespace std;
8
9  int main() {
10     int sumOdd  = 0; // For accumulating odd numbers, init to 0
11     int sumEven = 0; // For accumulating even numbers, init to 0
12     int upperbound; // Sum from 1 to this upperbound
13     int absDiff;    // The absolute difference between the two sums
14
15     // Prompt user for an upperbound
16     cout << "Enter the upperbound: ";
17     cin >> upperbound;
18
19     // Use a while-loop to repeatedly add 1, 2, 3,..., to the upperbound
20     int number = 1;
21     while (number <= upperbound) {
22         if (number % 2 == 0) { // Even number
23             sumEven += number; // Add number into sumEven
24         } else { // Odd number
25             sumOdd += number; // Add number into sumOdd
26         }
27         ++number; // increment number by 1
28     }
29
30     // Compute the absolute difference between the two sums
31     if (sumOdd > sumEven) {
32         absDiff = sumOdd - sumEven;
33     } else {
34         absDiff = sumEven - sumOdd;
35     }
36
37     // Print the results
38     cout << "The sum of odd numbers is " << sumOdd << endl;
39     cout << "The sum of even numbers is " << sumEven << endl;
40     cout << "The absolute difference is " << absDiff << endl;
41
42     return 0;
43 }
```

```
Enter the upperbound: 1000
The sum of odd numbers is 250000
The sum of even numbers is 250500
The absolute difference is 500
```

Program Notes

```
using namespace std;
```

The names `cout` and `endl` belong to the `std` namespace. They can be referenced via *fully qualified name* `std::cout` and `std::endl`, or simply as `cout` and `endl` with a `"using namespace std;"` statement. For simplicity, I shall use the latter approach in this section. I will discuss the significance later.

```
return 0;
```

The return value of 0 indicates normal termination; while non-zero (typically 1) indicates abnormal termination. C++ compiler will automatically insert a `"return 0;"` at the end of the `main()` function, thus, it statement can be omitted.

Instead of using numeric value of zero and non-zero, you can also use `EXIT_SUCCESS` or `EXIT_FAILURE`, which is defined in the `cstdlib` header (i.e., you need to `#include <cstdlib>`).

2.2 Comments

Comments are used to document and explain your codes and program logic. Comments are not programming statements and are *ignored* by the compiler, but they VERY IMPORTANT for providing documentation and explanation for others to understand your program (and also for yourself three days later).

There are two kinds of comments in C/C++:

1. *Multi-line Comment*: begins with a `/*` and ends with a `*/`, and can span several lines.
2. *End-of-line Comment*: begins with `//` and lasts till the end of the current line.

You should use comments *liberally* to explain and document your codes. During program development, instead of deleting a chunk of statements permanently, you could *comment-out* these statements so that you could get them back later, if needed.

2.3 Statements and Blocks

Statement: A programming *statement* is the smallest independent unit in a program, just like a sentence in the English language. It performs *a piece of programming action*. A programming statement must be terminated by a semi-colon (`;`), just like an English sentence ends with a period. (Why not ends with a period like an english sentence? This is because period crashes with decimal point - it is hard for the dumb computer to differentiate between period and decimal point!)

For examples,

```
// Each of the following lines is a programming statement, which ends with a semi-colon (;)
int number1 = 10;
int number2, number3 = 99;
int product;
product = number1 * number2 * number3;
cout << "Hello" << endl;
```

Block: A *block* (or a *compound statement*) is a group of statements surrounded by braces `{ }`. All the statements inside the block is treated as one unit. Blocks are used as the *body* in constructs like function, if-else and loop, which may contain multiple statements but are treated as one unit. There is no need to put a semi-colon after the closing brace to end a complex statement. Empty block (without any statement) is permitted. For examples,

```
// Each of the followings is a "complex" statement comprising one or more blocks of statements.
// No terminating semi-colon needed after the closing brace to end the "complex" statement.
// Take note that a "complex" statement is usually written over a few lines for readability.
```

```
if (mark >= 50) {
    cout << "PASS" << endl;
    cout << "Well Done!" << endl;
    cout << "Keep it Up!" << endl;
}

if (number == 88) {
    cout << "Got it" << endl;
} else {
    cout << "Try Again" << endl;
}

i = 1;
while (i < 8) {
    cout << i << endl;
    ++i;
}

int main() {
    ...statements...
}
```

2.4 White Spaces and Formatting Source Codes

White Spaces: *Blank*, *tab* and *new-line* are collectively called *white spaces*. C++, like most of the computing languages, ignores *extra* white spaces. That is, multiple contiguous white spaces are treated as a single white space.

You need to use a white space to separate two keywords or tokens, e.g.,

```
int sum=0;           // Need a white space between int and sum
double average;      // Need a white space between double and average
average=sum/100.0;
```

Additional white spaces and extra lines are, however, ignored, e.g.,

```
// same as above
int  sum
    = 0 ;

double average ;
average = sum / 100.0;
```

Formatting Source Codes: As mentioned, extra white spaces are ignored and have no computational significance. However, proper indentation (with tabs and blanks) and extra empty lines greatly improves the readability of the program, which is extremely important for others (and yourself three days later) to understand your programs. For example, the following hello-world works, but can you understand the program?

```
#include <iostream>
using namespace std;int main(){cout<<"Hello, world!"<<endl;return 0;}
```

Braces: Place the beginning brace at the end of the line, and align the ending brace with the start of the statement.

Indentation: Indent the body of a block by an extra 3 (or 4 spaces), according to its *level*.

For example,

```
/*
 * Recommended Programming style.
 */
#include <iostream>
using namespace std;

// blank line to separate sections of codes
int main() { // Place the beginning brace at the end of the current line
    // Indent the body by an extra 3 or 4 spaces for each level

    int mark = 70;
    if (mark >= 50) { // in level-1 block, indent once
        cout << "You Pass!" << endl; // in level-2 block, indent twice
    } else {
        cout << "You Fail!" << endl;
    }

    return 0;
} // ending brace aligned with the start of the statement
```

Most IDEs (such as CodeBlocks, Eclipse and NetBeans) have a command to *re-format* your source code automatically.

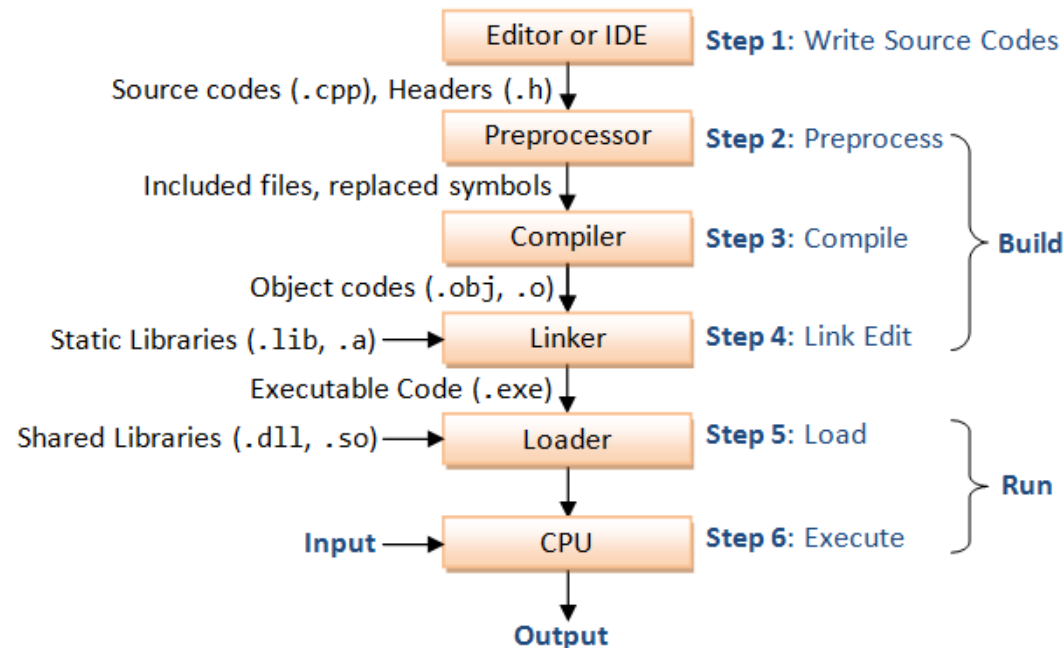
Note: Traditional C-style formatting places the beginning and ending braces on the same column. For example,

```
/*
 * Traditional C-style.
 */
#include <iostream>
using namespace std;

int main()
{
    if (mark >= 50) // in level-1 block, indent once
    {
        cout << "You Pass!" << endl; // in level-2 block, indent twice
    }
    else
    {
        cout << "You Fail!" << endl;
    }
}
```

2.5 Preprocessor Directives

C++ source code is *pre-processed* before it is *compiled* into object code (as illustrated).



A *preprocessor directive*, which begins with a # sign (such as `#include`, `#define`), tells the preprocessor to perform a certain action (such as including a header file, or performing text replacement), before compiling the source code into object code. Preprocessor directives are not programming statements, and therefore should NOT be terminated with a semi-colon. For example,

```
#include <iostream> // To include the IO library header
#include <cmath>    // To include the Math library header
#define PI 3.14159265 // To substitute the term PI with 3.14159265 in this file
// DO NOT terminate preprocessor directive with a semi-colon
```

In almost all of the C++ programs, we use `#include <iostream>` to include the input/output stream library header into our program, so as to use the IO library function to carry out input/output operations (such as `cin` and `cout`).

More on preprocessor directives later.

3. Variables and Types

3.1 Variables

Computer programs manipulate (or process) data. A *variable* is used to *store a piece of data* for processing. It is called *variable* because you can change the value stored.

More precisely, a *variable* is a *named* storage location, that stores a *value* of a particular data *type*. In other words, a *variable* has a *name*, a *type* and stores a *value*.

- A variable has a *name* (or *identifier*), e.g., radius, area, age, height. The name is needed to uniquely identify each variable, so as to assign a value to the variable (e.g., radius=1.2), and retrieve the value stored (e.g., area = radius*radius*3.1416).
- A variable has a *type*. Examples of *type* are,
 - int: for integers (whole numbers) such as 123 and -456;
 - double: for floating-point or real numbers such as 3.1416, -55.66, having a decimal point and fractional part.
- A variable can store a *value* of that particular *type*. It is important to take note that a variable in most programming languages is associated with a type, and can only store value of the particular type. For example, a int variable can store an integer value such as 123, but NOT real number such as 12.34, nor texts such as "Hello".
- The concept of *type* was introduced into the early programming languages to simplify interpretation of data made up of 0s and 1s. The type determines the size and layout of the data, the range of its values, and the set of operations that can be applied.

The following diagram illustrates two types of variables: int and double. An int variable stores an integer (whole number). A double variable stores a real number.

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

3.2 Identifiers

An *identifier* is needed to *name* a variable (or any other entity such as a function or a class). C++ imposes the following *rules on identifiers*:

- An identifier is a sequence of characters, of up to a certain length (compiler-dependent, typically 255 characters), comprising uppercase and lowercase letters (a-z, A-Z), digits (0-9), and underscore "_".
- White space (blank, tab, new-line) and other special characters (such as +, -, *, /, @, &, commas, etc.) are not allowed.
- An identifier must begin with a letter or underscore. It cannot begin with a digit. Identifiers beginning with an underscore are typically reserved for system use.
- An identifier cannot be a reserved keyword or a reserved literal (e.g., int, double, if, else, for).
- Identifiers are case-sensitive. A rose is NOT a Rose, and is NOT a ROSE.

Caution: Programmers don't use *blank* character in names. It is either not supported, or will pose you more challenges.

Variable Naming Convention

A variable name is a noun, or a noun phrase made up of several words. The first word is in lowercase, while the remaining words are initial-capitalized, with no spaces between words. For example, `theFontSize`, `roomNumber`, `xMax`, `yMin`, `xTopLeft` and `thisIsAVeryLongVariableName`. This convention is also known as *camel-case*.

Recommendations

1. It is important to choose a name that is *self-descriptive* and closely reflects the meaning of the variable, e.g., `numberOfStudents` or `numStudents`.
2. Do not use meaningless names like `a`, `b`, `c`, `d`, `i`, `j`, `k`, `i1`, `j99`.
3. Avoid single-alphabet names, which is easier to type but often meaningless, unless they are common names like `x`, `y`, `z` for coordinates, `i` for index.
4. It is perfectly okay to use long names of says 30 characters to make sure that the name accurately reflects its meaning!
5. Use singular and plural nouns prudently to differentiate between singular and plural variables. For example, you may use the variable `row` to refer to a single row number and the variable `rows` to refer to many rows (such as an array of rows - to be discussed later).

3.3 Variable Declaration

To use a variable in your program, you need to first "introduce" it by *declaring* its *name* and *type*, in one of the following syntaxes:

Syntax

Example

```
// Declare a variable of a specified type
type identifier;
// Declare multiple variables of the same type, separated by commas
type identifier-1, identifier-2, ..., identifier-n;
// Declare a variable and assign an initial value
type identifier = value;
// Declare multiple variables with initial values
type identifier-1 = value-1, ..., identifier-n = value-n;
```

```
int option;

double sum, difference, product, quotient;

int magicNumber = 88;

double sum = 0.0, product = 1.0;
```

Example,

```
int mark1;           // Declare an int variable called mark1
mark1 = 76;          // Use mark1
int mark2;           // Declare int variable mark2
mark2 = mark1 + 10;   // Use mark2 and mark1
double average;      // Declare double variable average
average = (mark1 + mark2) / 2.0; // Use average, mark1 and mark2
int mark1;           // Error: Declare twice
mark2 = "Hello";     // Error: Assign value of a different type
```

Take note that:

- In C++, you need to declare the name of a variable before it can be used.
- C++ is a "strongly-type" language. A variable takes on a type. Once the *type* of a variable is declared, it can only store a value belonging to this particular type. For example, an `int` variable can hold only integer such as 123, and NOT floating-point number such as -2.17 or text string such as "Hello". The concept of *type* was introduced into the early programming languages to simplify interpretation of data made up of 0s and 1s. Knowing the *type* of a piece of data greatly simplifies its interpretation and processing.
- Each variable can only be declared once.
- In C++, you can declare a variable anywhere inside the program, as long as it is declared before used. (In C prior to C99, all the variables must be declared at the beginning of functions.) It is recommended that you declare a variable just before it is first used.
- The type of a variable cannot be changed inside the program.

CAUTION: Uninitialized Variables

When a variable is declared, it contains garbage until you assign an initial value. It is important to take note that C/C++ does not issue any warning/error if you use a variable before initialize it - which certainly leads to some unexpected results. For example,

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int number;           // Declared but not initialized
6      cout << number << endl; // Used before initialized
7                           // No warning/error, BUT unexpected result
8      return 0;
9  }
```

3.4 Constants (const)

Constants are *non-modifiable* variables, declared with keyword `const`. Their values cannot be changed during program execution. Also, `const` must be initialized during declaration. For examples:

```
const double PI = 3.1415926; // Need to initialize
```

Constant Naming Convention: Use uppercase words, joined with underscore. For example, `MIN_VALUE`, `MAX_SIZE`.

3.5 Expressions

An *expression* is a combination of *operators* (such as addition '+', subtraction '-', multiplication '*', division '/') and *operands* (variables or literal values), that can be *evaluated to yield a single value of a certain type*. For example,

```
1 + 2 * 3           // give int 7

int sum, number;
sum + number        // evaluated to an int value
```

```
double principal, interestRate;
principal * (1 + interestRate) // evaluated to a double value
```

3.6 Assignment (=)

An *assignment statement*:

1. assigns a literal value (of the RHS) to a variable (of the LHS); or
2. evaluates an expression (of the RHS) and assign the resultant value to a variable (of the LHS).

The RHS shall be a value; and the LHS shall be a variable (or memory address).

The syntax for assignment statement is:

Syntax

```
// Assign the literal value (of the RHS) to the variable (of the LHS)
variable = literal-value;
// Evaluate the expression (RHS) and assign the result to the variable (LHS)
variable = expression;
```

Example

```
number = 88;

sum = sum + number;
```

The assignment statement should be interpreted this way: The *expression* on the right-hand-side (RHS) is first evaluated to produce a resultant value (called *rvalue* or right-value). The *rvalue* is then assigned to the variable on the left-hand-side (LHS) (or *lvalue*, which is a location that can hold a *rvalue*). Take note that you have to first evaluate the RHS, before assigning the resultant value to the LHS. For examples,

```
number = 8;           // Assign literal value of 8 to the variable number
number = number + 1;  // Evaluate the expression of number + 1,
                      // and assign the resultant value back to the variable number
```

The symbol "=" is known as the *assignment operator*. The meaning of "=" in programming is different from Mathematics. It denotes *assignment* instead of *equality*. The RHS is a literal value; or an expression that evaluates to a value; while the LHS must be a variable. Note that $x = x + 1$ is valid (and often used) in programming. It evaluates $x + 1$ and assign the resultant value to the variable x . $x = x + 1$ illegal in Mathematics. While $x + y = 1$ is allowed in Mathematics, it is invalid in programming (because the LHS of an assignment statement must be a variable). Some programming languages use symbol ":", " \leftarrow ", " \rightarrow ", or " \mapsto " as the assignment operator to avoid confusion with equality.

3.7 Fundamental Types

Integers: C++ supports these integer types: `char`, `short`, `int`, `long`, `long long` (in C++11) in a non-decreasing order of size. The actual size depends on the implementation. The integers (except `char`) are *signed* number (which can hold zero, positive and negative numbers). You could use the keyword `unsigned` [`char|short|int|long|long long`] to declare an *unsigned* integers (which can hold zero and positive numbers). There are a total 10 types of integers - `signed|unsigned` combined with `char|short|int|long|long long`.

Characters: Characters (e.g., `'a'`, `'Z'`, `'0'`, `'9'`) are encoded in ASCII into integers, and kept in type `char`. For example, character `'0'` is 48 (decimal) or 30H (hexadecimal); character `'A'` is 65 (decimal) or 41H (hexadecimal); character `'a'` is 97 (decimal) or 61H (hexadecimal). Take note that the type `char` can be

interpreted as character in ASCII code, or an 8-bit integer. Unlike `int` or `long`, which is signed, `char` could be signed or unsigned, depending on the implementation. You can use `signed char` or `unsigned char` to explicitly declare signed or unsigned `char`.

Floating-point Numbers: There are 3 floating point types: `float`, `double` and `long double`, for single, double and long double precision floating point numbers. `float` and `double` are represented as specified by IEEE 754 standard. A `float` can represent a number between $\pm 1.40239846 \times 10^{-45}$ and $\pm 3.40282347 \times 10^{38}$, approximated. A `double` can represent a number between $\pm 4.94065645841246544 \times 10^{-324}$ and $\pm 1.79769313486231570 \times 10^{308}$, approximated. Take note that not all real numbers can be represented by `float` and `double`, because there are infinite real numbers. Most of the values are approximated.

Boolean Numbers: A special type called `bool` (for boolean), which takes a value of either `true` or `false`.

The table below shows the *typical* size, minimum, maximum for the primitive types. Again, take note that the sizes are implementation dependent.

Category	Type	Description	Bytes (Typical)	Minimum (Typical)	Maximum (Typical)
Integers	<code>int</code> (or signed <code>int</code>)	Signed integer (of at least 16 bits)	4 (2)	-2147483648	2147483647
	<code>unsigned int</code>	Unsigned integer (of at least 16 bits)	4 (2)	0	4294967295
	<code>char</code>	Character (can be either signed or unsigned depends on implementation)	1		
	<code>signed char</code>	Character or signed tiny integer (guarantee to be signed)	1	-128	127
	<code>unsigned char</code>	Character or unsigned tiny integer (guarantee to be unsigned)	1	0	255
	<code>short</code> (or short <code>int</code>) (or signed <code>short</code>) (or signed <code>short int</code>)	Short signed integer (of at least 16 bits)	2	-32768	32767
	<code>unsigned short</code> (or unsigned <code>shot int</code>)	Unsigned short integer (of at least 16 bits)	2	0	65535
	<code>long</code> (or long <code>int</code>) (or signed <code>long</code>) (or signed <code>long int</code>)	Long signed integer (of at least 32 bits)	4 (8)	-2147483648	2147483647
	<code>unsigned long</code> (or unsigned <code>long int</code>)	Unsigned long integer (of at least 32 bits)	4 (8)	0	same as above

Category	Type	Description	Bytes (Typical)	Minimum (Typical)	Maximum (Typical)
Real Numbers	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	$2^{64}-1$
	float	Floating-point number, ≈ 7 digits (IEEE 754 single-precision floating point format)	4	$3.4\text{e}38$	$3.4\text{e}-38$
	double	Double precision floating-point number, ≈ 15 digits (IEEE 754 double-precision floating point format)	8	$1.7\text{e}308$	$1.7\text{e}-308$
Boolean Numbers	long double	Long double precision floating-point number, ≈ 19 digits (IEEE 754 quadruple-precision floating point format)	12 (8)		
	bool	Boolean value of either true or false	1	false (0)	true (1 or non-zero)
	Wide Characters				
	wchar_t	Wide (double-byte) character	2 (4)		
	char16_t (C++11)				
	char32_t (C++11)				

In addition, many C++ library functions use a type called `size_t`, which is equivalent (typedef) to a unsigned `int`, meant for counting, size or length, with 0 and positive integers.

*The sizeof Operator

C/C++ provides an unary `sizeof` operator to get the size of the operand (in bytes). The following program uses `sizeof` operator to print the size of the fundamental types.

```

1  /*
2   * Print Size of Fundamental Types (SizeofTypes.cpp).
3   */
4  #include <iostream>
5  using namespace std;
6
7  int main() {
8      cout << "sizeof(char) is " << sizeof(char) << " bytes " << endl;
9      cout << "sizeof(short) is " << sizeof(short) << " bytes " << endl;

```

```
10     cout << "sizeof(int) is " << sizeof(int) << " bytes " << endl;
11     cout << "sizeof(long) is " << sizeof(long) << " bytes " << endl;
12     cout << "sizeof(long long) is " << sizeof(long long) << " bytes " << endl;
13     cout << "sizeof(float) is " << sizeof(float) << " bytes " << endl;
14     cout << "sizeof(double) is " << sizeof(double) << " bytes " << endl;
15     cout << "sizeof(long double) is " << sizeof(long double) << " bytes " << endl;
16     cout << "sizeof(bool) is " << sizeof(bool) << " bytes " << endl;
17     return 0;
18 }
```

```
sizeof(char) is 1 bytes
sizeof(short) is 2 bytes
sizeof(int) is 4 bytes
sizeof(long) is 4 bytes
sizeof(long long) is 8 bytes
sizeof(float) is 4 bytes
sizeof(double) is 8 bytes
sizeof(long double) is 12 bytes
sizeof(bool) is 1 bytes
```

The results may vary among different systems.

*Header <climits>

The `climits` header (ported to C++ from C's `limits.h`) contains information about limits of integer type. For example,

```
1  /* Test integer limits in <climits> header */
2  #include <iostream>
3  #include <climits>    // integer limits
4  using namespace std;
5
6  int main() {
7      cout << "int max = " << INT_MAX << endl;
8      cout << "int min = " << INT_MIN << endl;
9      cout << "unsigned int max = " << UINT_MAX << endl;
10     cout << "long long max = " << LLONG_MAX << endl;
11     cout << "long long min = " << LLONG_MIN << endl;
12     cout << "unsigned long long max = " << ULLONG_MAX << endl;
13     cout << "Bits in char = " << CHAR_BIT << endl;
14     cout << "char max = " << CHAR_MAX << endl;
15     cout << "char min = " << CHAR_MIN << endl;
16     cout << "signed char max = " << SCHAR_MAX << endl;
17     cout << "signed char min = " << SCHAR_MIN << endl;
18     cout << "unsigned char max = " << UCHAR_MAX << endl;
19     return 0;
20 }
```

```

int max = 2147483647
int min = -2147483648
unsigned int max = 4294967295
long long max = 9223372036854775807
long long min = -9223372036854775808
unsigned long long max = 18446744073709551615
Bits in char = 8
char max = 127
char min = -128
signed char max = 127
signed char min = -128
unsigned char max = 255

```

Again, the outputs depend on the system.

The minimum of unsigned integer is always 0. The other constants are SHRT_MAX, SHRT_MIN, USHRT_MAX, LONG_MIN, LONG_MAX, ULONG_MAX. Try inspecting this header (search for `climits` under your compiler).

*Header <float>

Similarly, the `cfloat` header (ported from C's `float.h`) contain information on limits for floating point numbers, such as minimum number of significant digits (FLT_DIG, DBL_DIG, LDBL_DIG for float, double and long double), number of bits for mantissa (FLT_MANT_DIG, DBL_MANT_DIG, LDBL_MANT_DIG), maximum and minimum exponent values, etc. Try inspecting this header (search for `cfloat` under your compiler).

*Header <limits>

The `climits` and `cfloat` headers are ported over from C's `limit.h` and `float.h`. C++ added a new header called `limits`.

[TODO]

Choosing Types

As a programmer, you need to choose variables and decide on the type of the variables to be used in your programs. Most of the times, the decision is intuitive. For example, use an integer type for counting and whole number; a floating-point type for number with fractional part, `char` for a single character, and `boolean` for binary outcome.

Rule of Thumb

- Use `int` for integer and double for floating point numbers. Use `byte`, `short`, `long` and `float` only if you have a good reason to choose that specific precision.
- Use `int` (or `unsigned int`) for *counting* and *indexing*, NOT floating-point type (float or double). This is because integer type are precise and more efficient in operations.
- Use an integer type if possible. Use a floating-point type only if the number contains a fractional part.

Read my article on "[Data Representation](#)" if you wish to understand how the numbers and characters are represented inside the computer memory. In brief, It is important to take note that `char '1'` is different from `int 1`, `short 1`, `float 1.0`, `double 1.0`, and `String "1"`. They are represented differently in the computer memory, with different precision and interpretation. For example, `short 1` is "00000000 00000001", `int 1` is "00000000 00000000 00000000

00000001", long long 1 is "00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001", float 1.0 is "0 01111111 00000000 00000000 00000000", double 1.0 is "0 01111111111 0000 00000000 00000000 00000000 00000000 00000000", char '1' is "00110001".

There is a subtle difference between `int 0` and `double 0.0`.

Furthermore, you MUST know the type of a value before you can interpret a value. For example, this value "00000000 00000000 00000000 00000001" cannot be interpreted unless you know the type.

*The typedef Statement

Typing "unsigned int" many time can get annoying. The typedef statement can be used to create a new name for an existing type. For example, you can create a new type called "uint" for "unsigned int" as follow. You should place the typedef immediately after #include. Use typedef with care because it makes the program hard to read and understand.

```
typedef unsigned int uint;
```

Many C/C++ compilers define a type called `size_t`, which is a typedef of `unsigned int`.

```
typedef unsigned int size_t;
```

3.8 Literals for Fundamental Types and String

A *literal* is a *specific constant value*, such as 123, -456, 3.14, 'a', "Hello", that can be assigned directly to a variable; or used as part of an expression. They are called *literals* because they literally and explicitly identify their values.

Integer Literals

A whole number, such as 123 and -456, is treated as an `int`, by default. For example,

```
int number = -123;
int sum = 4567;
int bigSum = 8234567890; // ERROR: this value is outside the range of int
```

An `int` literal may precede with a plus (+) or minus (-) sign, followed by digits. No commas or special symbols (e.g., \$ or space) is allowed (e.g., 1,234 and \$123 are invalid). No preceding 0 is allowed too (e.g., 007 is invalid).

Besides the default base 10 integers, you can use a prefix '0' (zero) to denote a value in octal, prefix '0x' for a value in hexadecimal, and prefix '0b' for binary value (in some compilers), e.g.,

```
int number1 = 1234;           // Decimal
int number2 = 01234;          // Octal 1234, Decimal 2322
int number3 = 0x1abc;         // hexadecimal 1ABC, decimal 15274
int number4 = 0b10001001;     // binary (may not work in some compilers)
```

A long literal is identified by a suffix 'L' or 'l' (avoid lowercase, which can be confused with the number one). A long long int is identified by a suffix 'LL'. You can also use suffix 'U' for unsigned int, 'UL' for unsigned long, and 'ULL' for unsigned long long int. For example,


```
long number = 12345678L;    // Suffix 'L' for long
long sum = 123;             // int 123 auto-casts to long 123L
long long bigNumber = 987654321LL; // Need suffix 'LL' for long long int
```

No suffix is needed for short literals. But you can only use integer values in the permitted range. For example,

```
short smallNumber = 1234567890; // ERROR: this value is outside the range of short.
short midSizeNumber = -12345;
```

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;
```

Use suffix 'L' (or 'l') for long double.

Character Literals and Escape Sequences

A printable char literal is written by enclosing the character with a pair of *single quotes*, e.g., 'z', '\$', and '9'. 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.

You can find the [ASCII code table](#) [HERE](#).

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
```

Non-printable and control characters can be represented by a so-called *escape sequence*, which begins with a back-slash (\). The commonly-used escape sequences are:

Escape Sequence	Description	Hex (Decimal)
\n	New-line (or Line-feed)	0AH (10D)

Escape Sequence	Description	Hex (Decimal)
\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)

Notes:

- New-line (0AH) and carriage return (0dH), represented by \n, and \r respectively, are used as *line delimiter* (or *end-of-line*, or *EOL*). However, take note that Unixes/Mac use \n as EOL, Windows use \r\n.
- Horizontal Tab (09H) is represented as \t.
- To resolve ambiguity, characters back-slash (\), single-quote (') and double-quote (") are represented using escape sequences \\, \' and \", respectively. This is because a single back-slash begins an escape sequence, while single-quotes and double-quotes are used to enclose character and string.
- Other less commonly-used escape sequences are: \? or ?, \a for alert or bell, \b for backspace, \f for form-feed, \v for vertical tab. These may not be supported in some consoles.

The <cctype> Header

The cctype header (ported from C's ctype.h) provides functions such as isalpha(), isdigit(), isspace(), ispunct(), isalnum(), isupper(), islower() to determine the type of character; and toupper(), tolower() for case conversion.

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\ta\tstring" << endl;
```

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

TRY: Write a program to print the following picture. Take note that you need to use escape sequences to print special characters.

```

      ' '
      (oo)
+=====\/
/  ||  %%  ||
*  ||  ----  ||
   ""      ""

```

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.

Example (Literals)

```

1  /* Testing Primitive Types (TestLiteral.cpp) */
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      char gender = 'm';           // char is single-quoted
7      bool isMarried = true;       // true(non-zero) or false(0)
8      unsigned short numChildren = 8; // [0, 255]
9      short yearOfBirth = 1945;    // [-32767, 32768]
10     unsigned int salary = 88000;   // [0, 4294967295]
11     double weight = 88.88;         // With fractional part
12     float gpa = 3.88f;             // Need suffix 'f' for float
13
14     // "cout <<" can be used to print value of any type
15     cout << "Gender is " << gender << endl;
16     cout << "Is married is " << isMarried << endl;
17     cout << "Number of children is " << numChildren << endl;
18     cout << "Year of birth is " << yearOfBirth << endl;
19     cout << "Salary is " << salary << endl;
20     cout << "Weight is " << weight << endl;
21     cout << "GPA is " << gpa << endl;
22     return 0;
23 }

```

```

Gender is m
Is married is 1    // true
Number of children is 8
Year of birth is 1945
Salary is 88000
Weight is 88.88
GPA is 3.88

```

4. Operations

4.1 Arithmetic Operators

C++ supports the following arithmetic operators for numbers: short, int, long, long long, char (treated as 8-bit signed integer), unsigned short, unsigned int, unsigned long, unsigned long long, unsigned char, float, double and long double.

Operator	Description	Usage	Examples
*	Multiplication	$expr1 * expr2$	$2 * 3 \rightarrow 6$; $3.3 * 1.0 \rightarrow 3.3$
/	Division	$expr1 / expr2$	$1 / 2 \rightarrow 0$; $1.0 / 2.0 \rightarrow 0.5$
%	Remainder (Modulus)	$expr1 \% expr2$	$5 \% 2 \rightarrow 1$; $-5 \% 2 \rightarrow -1$
+	Addition	$expr1 + expr2$	$1 + 2 \rightarrow 3$; $1.1 + 2.2 \rightarrow 3.3$
-	Subtraction	$expr1 - expr2$	$1 - 2 \rightarrow -1$; $1.1 - 2.2 \rightarrow -1.1$

All the above operators are *binary* operators, i.e., they take two operands. The multiplication, division and remainder take *precedence* over addition and subtraction. Within the same precedence level (e.g., addition and subtraction), the expression is evaluated from left to right. For example, $1+2+3-4$ is evaluated as $((1+2)+3)-4$.

It is important to take note that int/int produces an int, with the result *truncated*, e.g., $1/2 \rightarrow 0$ (instead of 0.5).

Take note that C/C++ does not have an exponent (power) operator ('^' is exclusive-or, not exponent).

4.2 Arithmetic Expressions

In programming, the following arithmetic expression:

$$\frac{1+2a}{3} + \frac{4(b+c)(5-d-e)}{f} - 6\left(\frac{7}{g} + h\right)$$

must be written as $(1+2*a)/3 + (4*(b+c)*(5-d-e))/f - 6*(7/g+h)$. You cannot omit the multiplication symbol '*' (as in Mathematics).

Like Mathematics, the multiplication '*' and division '/' take precedence over addition '+' and subtraction '-'. Parentheses () have higher precedence. The operators '+', '-', '*', and '/' are *left-associative*. That is, $1 + 2 + 3 + 4$ is treated as $((1+2) + 3) + 4$.

4.3 Mixed-Type Operations

If both the operands of an arithmetic operation belong to the *same type*, the operation is carried out in that type, and the result belongs to that type. For example, $\text{int}/\text{int} \rightarrow \text{int}$; $\text{double}/\text{double} \rightarrow \text{double}$.

However, if the two operands belong to *different types*, the compiler promotes the value of the *smaller* type to the *larger* type (known as *implicit type-casting*). The operation is then carried out in the *larger* type. For example, $\text{int}/\text{double} \rightarrow \text{double}/\text{double} \rightarrow \text{double}$. Hence, $1/2 \rightarrow 0$, $1.0/2.0 \rightarrow 0.5$, $1.0/2 \rightarrow 0.5$, $1/2.0 \rightarrow 0.5$.

For example,

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

Example

```

1  /* Testing mix-type arithmetic operations (TestMixTypeOp.cpp) */
2  #include <iostream>
3  #include <iomanip>    // needed for formatting floating-point numbers
4  using namespace std;
5
6  int main() {
7      int i1 = 2, i2 = 4;
8      double d1 = 2.5, d2 = 5.0;
9
10     // Print floating-points in fixed format with 1 decimal place
11     cout << fixed << setprecision(1);    // need <iomanip>
12
13     cout << i1 << " + " << i2 << " = " << i1+i2 << endl;    // 6
14     cout << d1 << " + " << d2 << " = " << d1+d2 << endl;    // 7.5
15     cout << i1 << " + " << d2 << " = " << i1+d2 << endl;    // 7.0 <==
16
17     cout << i1 << " - " << i2 << " = " << i1-i2 << endl;    // -2
18     cout << d1 << " - " << d2 << " = " << d1-d2 << endl;    // -2.5
19     cout << i1 << " - " << d2 << " = " << i1-d2 << endl;    // -3.0 <==
20
21     cout << i1 << " * " << i2 << " = " << i1*i2 << endl;    // 8
22     cout << d1 << " * " << d2 << " = " << d1*d2 << endl;    // 12.5

```

```

23     cout << i1 << " * " << d2 << " = " << i1*d2 << endl;    // 10.0 <==
24
25     cout << i1 << " / " << i2 << " = " << i1/i2 << endl;    // 0      <==
26     cout << d1 << " / " << d2 << " = " << d1/d2 << endl;    // 0.5
27     cout << i1 << " / " << d2 << " = " << i1/d2 << endl;    // 0.4 <==
28     return 0;
29 }

```

4.4 Overflow/UnderFlow

Study the output of the following program:

```

1  /* Test Arithmetic Overflow/Underflow (TestOverflow.cpp) */
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      // Range of int is [-2147483648, 2147483647]
7      int i1 = 2147483647;    // max int
8      cout << i1 + 1 << endl;  // -2147483648 (overflow)
9      cout << i1 + 2 << endl;  // -2147483647
10     cout << i1 * i1 << endl;  // 1
11
12     int i2 = -2147483648;    // min int
13     cout << i2 - 1 << endl;    // 2147483647 (underflow)
14     cout << i2 - 2 << endl;    // 2147483646
15     cout << i2 * i2 << endl;    // 0
16     return 0;
17 }

```

In arithmetic operations, the resultant value *wraps around* if it exceeds its range (i.e., overflow or underflow). C++ runtime does not issue an error/warning message but produces *incorrect* result.

It is important to take note that *checking of overflow/underflow is the programmer's responsibility*, i.e., your job!

This feature is an legacy design, where processors were slow. Checking for overflow/underflow consumes computation power and reduces performance.

To check for arithmetic overflow (known as *secure coding*) is tedious. Google for "INT32-C. Ensure that operations on signed integers do not result in overflow" @ www.securecoding.cert.org.

4.5 Compound Assignment Operators

Besides the usual simple assignment operator '=' described earlier, C++ also provides the so-called *compound assignment operators* as listed:

Operator	Usage	Description	Example
=	<code>var = expr</code>	Assign the value of the LHS to the variable at the RHS	<code>x = 5;</code>

Operator	Usage	Description	Example
+=	<i>var += expr</i>	same as <i>var = var + expr</i>	<i>x += 5; same as x = x + 5</i>
-=	<i>var -= expr</i>	same as <i>var = var - expr</i>	<i>x -= 5; same as x = x - 5</i>
*=	<i>var *= expr</i>	same as <i>var = var * expr</i>	<i>x *= 5; same as x = x * 5</i>
/=	<i>var /= expr</i>	same as <i>var = var / expr</i>	<i>x /= 5; same as x = x / 5</i>
%=	<i>var %= expr</i>	same as <i>var = var % expr</i>	<i>x %= 5; same as x = x % 5</i>

4.6 Increment/Decrement Operators

C++ supports these *unary* arithmetic operators: increment '++' and decrement '--'.

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

For example,

```

1  /* Test on increment (++) and decrement (--) Operator (TestIncDec.cpp) */
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      int mark = 76;          // declare & assign
7      cout << mark << endl; // 76
8
9      mark++;                 // increase by 1 (post-increment)
10     cout << mark << endl; // 77
11
12     ++mark;                  // increase by 1 (pre-increment)
13     cout << mark << endl; // 78
14
15     mark = mark + 1;         // also increase by 1 (or mark += 1)
16     cout << mark << endl; // 79
17
18     mark--;                  // decrease by 1 (post-decrement)
19     cout << mark << endl; // 78
20
21     --mark;                  // decrease by 1 (pre-decrement)
22     cout << mark << endl; // 77
23
24     mark = mark - 1;         // also decrease by 1 (or mark -= 1)
25     cout << mark << endl; // 76

```

```

26     return 0;
27 }

```

The increment/decrement unary operator can be placed before the operand (prefix operator), or after the operands (postfix operator). They takes on different meaning in operations.

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

If '++' or '--' involves another operation, then pre- or post-order is important to specify the order of the two operations. For examples,

```

x = 5;
cout << x++ << endl; // Save x (5); Increment x (=6); Print old x (5).
x = 5;
cout << ++x << endl; // Increment x (=6); Print x (6).
// This is confusing! Try to avoid! What is i=++i? What is i=i++?

```

Prefix operator (e.g, ++i) could be more efficient than postfix operator (e.g., i++) in some situations.

4.7 Implicit Type-Conversion vs. Explicit Type-Casting

Converting a value from one type to another type is called *type casting* (or *type conversion*). There are two kinds of type casting:

1. Implicit type-conversion performed by the compiler automatically, and
2. Explicit type-casting via an unary *type-casting operator* in the form of *(new-type) operand* or *new-type(operand)*.

Implicit (Automatic) Type Conversion

When you assign a value of a fundamental (built-in) type to a variable of another fundamental type, C++ automatically converts the value to the receiving type, if the two types are compatible. For examples,

- If you assign an `int` value to a `double` variable, the compiler automatically casts the `int` value to a `double` (e.g., from 1 to 1.0) and assigns it to the `double` variable.
- if you assign a `double` value to an `int` variable, the compiler automatically casts the `double` value to an `int` value (e.g., from 1.2 to 1) and assigns it to the `int` variable. The fractional part would be truncated and lost. Some compilers issue a warning/error "possible loss in precision"; others do not.

```

1  /*

```



```

2  * Test implicit type casting (TestImplicitTypeCast.cpp)
3  */
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  int main() {
9      int i;
10     double d;
11
12     // print floating point number in fixed format with 1 decimal place
13     cout << fixed << setprecision(1);
14
15     i = 3;
16     d = i;    // Assign an int value to double
17     cout << "d = " << d << endl;    // 3.0
18
19     d = 5.5;
20     i = d;    // Assign a double value to int
21     cout << "i = " << i << endl;    // 5 (truncated, no warning!)
22
23     i = 6.6;  // Assign a double literal to int
24     cout << "i = " << i << endl;    // 6 (truncated, no warning!)
25 }

```

C++ will not perform automatic type conversion, if the two types are not compatible.

Explicit Type-Casting

You can explicitly perform type-casting via the so-called unary *type-casting operator* in the form of *(new-type)operand* or *new-type(operand)*. The type-casting operator takes one operand in the particular type, and returns an equivalent value in the new type. Take note that it is an operation that yields a resultant value, similar to an addition operation although addition involves two operands. For example,

```

// Print floating-point number in fixed format with 1 decimal point (need <iomanip>)
cout << fixed << setprecision(1);

cout << (double)5 << endl;    // int 5 → double 5.0
cout << (int)5.5 << endl;    // double 5.5 → int 5

double aDouble = 5.6;
int anInt = (int)aDouble; // return 5 and assign to anInt. aDouble does not change!

// C++ also supports function-style type cast.
cout << double(5) << endl;    // 5.0
cout << int(5.5) << endl;    // 5
cout << int(aDouble) << endl; // 5

```

Example: Suppose that you want to find the average (in double) of the integers between 1 and 100. Study the following codes:

```

1  /*
2   * Testing Explicit Type Cast (Average1to100.cpp).
3   */
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  int main() {
9      int sum = 0;
10     double average;
11     for (int number = 1; number <= 100; ++number) {
12         sum += number;      // Final sum is int 5050
13     }
14     average = sum / 100;    // Won't work (average = 50.0 instead of 50.5)
15     cout << fixed << setprecision(1);
16     cout << "Average is " << average << endl; // Average is 50.0
17     return 0;
18 }
```

You don't get the fractional part although the average is a double. This is because both the sum and 100 are int. The result of division is an int, which is then implicitly casted to double and assign to the double variable average. To get the correct answer, you can do either:

```

average = (double)sum / 100;    // Cast sum from int to double before division
average = sum / (double)100;   // Cast 100 from int to double before division
average = sum / 100.0;
average = (double)(sum / 100); // Won't work. why?

// C++ also support function-style type casting in the form of new-type(operand)
average = double(sum) / 100;   // Same as (double)sum / 100
```

Example:

```

1  /* Test Type Casting (TestTypeCast.cpp) */
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  int main() {
7      // Print floating-point number in fixed format with 1 decimal place
8      cout << fixed << setprecision(1);
9
10     // Test explicit type casting
11     int i1 = 4, i2 = 8;
12     cout << i1 / i2 << endl;      // 0
13     cout << (double)i1 / i2 << endl; // 0.5
```

```

14     cout << i1 / (double)i2 << endl;    // 0.5
15     cout << (double)(i1 / i2) << endl; // 0.0
16
17     double d1 = 5.5, d2 = 6.6;
18     cout << (int)d1 / i2 << endl;      // 0
19     cout << (int)(d1 / i2) << endl;    // 0
20
21     // Test implicit type casting
22     d1 = i1;           // int implicitly casts to double
23     cout << d1 << endl; // 4.0
24     i2 = d2;           // double truncates to int! (Warning?)
25     cout << i2 << endl; // 6
26 }

```

Example:

```

1  /*
2  * Converting between Celsius and Fahrenheit (ConvertTemperature.cpp)
3  *   Celsius = (5/9)(Fahrenheit-32)
4  *   Fahrenheit = (9/5)Celsius+32
5  */
6  #include <iostream>
7  #include <iomanip>    // needed for formatting floating-point numbers
8  using namespace std;
9
10 int main() {
11     double celsius, fahrenheit;
12
13     // Format floating-points in fixed with 2 decimal places
14     cout << fixed << setprecision(2);
15
16     cout << "Enter the temperature in celsius: ";
17     cin >> celsius;
18     fahrenheit = celsius*9/5 + 32;
19     // 9/5*celsius + 32 gives wrong answer! Why?
20     cout << celsius << "C is " << fahrenheit << "F" << endl;
21
22     cout << "Enter the temperature in fahrenheit: ";
23     cin >> fahrenheit;
24     celsius = (fahrenheit - 32)*5/9;
25     // 5/9*(fahrenheit - 32) gives wrong answer! Why?
26     cout << fahrenheit << "F is " << celsius << "C" << endl;
27     return 0;
28 }

```

***Operator static-cast<type>**

C++ introduces a new operator called `static_cast<type>` to perform type conversion (because the regular cast mentioned earlier is too lax and could produce expected results). `static_cast` signal an error if conversion fails. For example,

```
double d = 5.5;
int i = static_cast<int>(d);
float f = static_cast<float>(i);
long l = static_cast<long>(d);
```

4.8 Relational and Logical Operators

Very often, you need to compare two values before deciding on the action to be taken, e.g., if mark is more than or equal to 50, print "PASS".

C++ provides six *comparison operators* (or *relational operators*):

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

In C++, these comparison operations returns a `bool` value of either `false` (0) or `true` (1 or a non-zero value).

Each comparison operation involves two operands, e.g., `x <= 100`. It is invalid to write `1 < x < 100` in programming. Instead, you need to break out the two comparison operations `x > 1`, `x < 100`, and join with with a logical AND operator, i.e., `(x > 1) && (x < 100)`, where `&&` denotes AND operator.

C++ provides four logical operators (which operate on `boolean` operands only):

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>

The truth tables are as follows:

	AND (&&)	true	false
true		true	false
false		false	false

OR ()	true	false
true	true	true
false	true	false
NOT (!)	true	false
	false	true
XOR (^)	true	false
true	false	true
false	true	false

Example:

```
// Return true if x is between 0 and 100 (inclusive)
(x >= 0) && (x <= 100)
// wrong to use 0 <= x <= 100

// Return true if x is outside 0 and 100 (inclusive)
(x < 0) || (x > 100) //or
!((x >= 0) && (x <= 100))

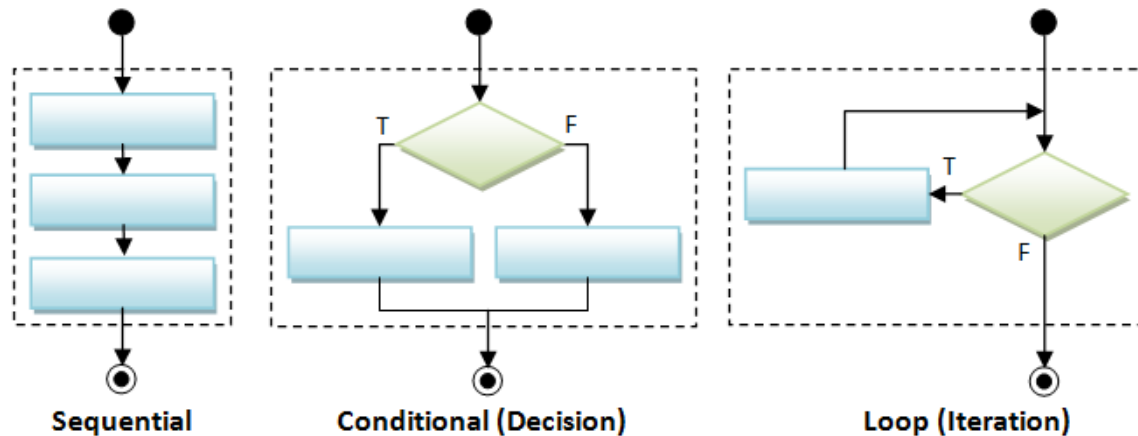
// Return true if year is a leap year
// A year is a leap year if it is divisible by 4 but not by 100, or it is divisible by 400.
((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)
```

Exercise: Given the year, month (1-12), and day (1-31), write a boolean expression which returns true for dates before October 15, 1582 (Gregorian calendar cut over date).

Ans: (year < 1582) || (year == 1582 && month < 10) || (year == 1582 && month == 10 && day < 15)

5. Flow Control

There are three basic flow control constructs - *sequential*, *conditional* (or *decision*), and *loop* (or *iteration*), as illustrated below.



5.1 Sequential Flow Control

A program is a sequence of instructions. *Sequential* flow is the most common and straight-forward, where programming statements are executed in the order that they are written - from top to bottom in a sequential manner.

5.2 Conditional (Decision) Flow Control

There are a few types of conditionals, *if-then*, *if-then-else*, *nested-if (if-elseif-elseif...-else)*, *switch-case*, and *conditional expression*.

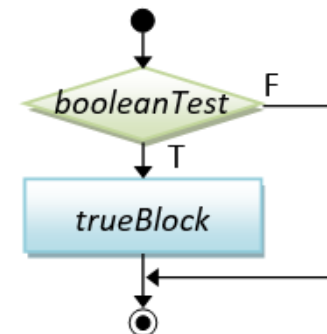
Syntax

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

Example

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

Flowchart



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

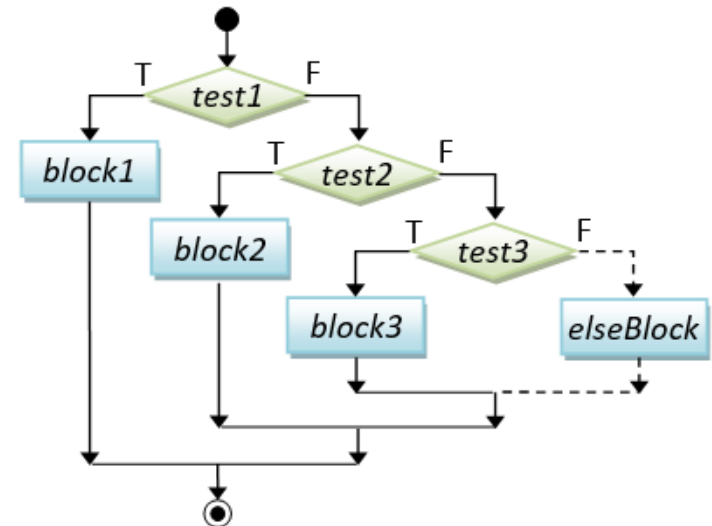
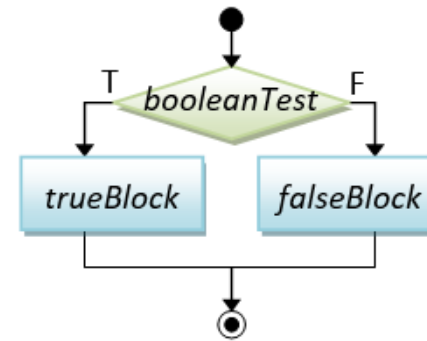
```
if (mark >= 50) {
    cout << "Congratulation!" << endl;
    cout << "Keep it up!" << endl;
} else {
    cout << "Try Harder!" << endl;
}
```

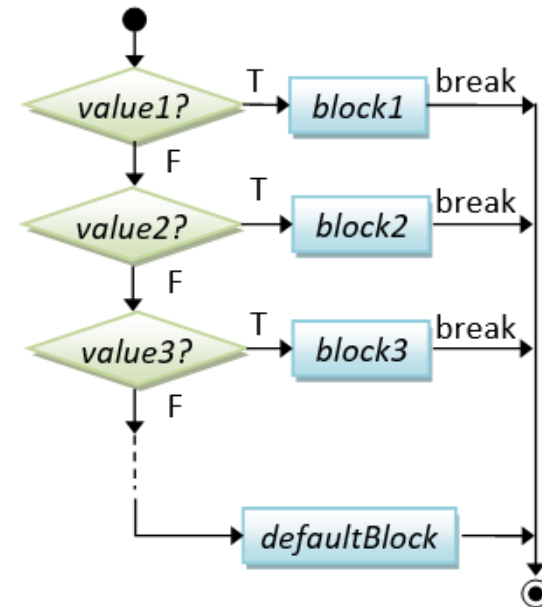
```
// nested-if
if ( booleanExpr-1 ) {
    block-1 ;
} else if ( booleanExpr-2 ) {
    block-2 ;
} else if ( booleanExpr-3 ) {
    block-3 ;
} else if ( booleanExpr-4 ) {
    .....
} else {
    elseBlock ;
}
```

```
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;
}
```





"switch-case" is an alternative to the "nested-if". In a *switch-case* statement, a *break* statement is needed for each of the cases. If *break* is missing, execution will flow through the following case. You can use either an *int* or *char* variable as the *case-selector*.

Conditional Operator: A conditional operator is a ternary (3-operand) operator, in the form of *booleanExpr* ? *trueExpr* : *falseExpr*. Depending on the *booleanExpr*, it evaluates and returns the value of *trueExpr* or *falseExpr*.

Syntax

booleanExpr ? *trueExpr* : *falseExpr*

Example

```

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
  
```

Braces: You could omit the braces { }, if there is only one statement inside the block. For example,

```

if (mark >= 50)
    cout << "PASS" << endl;    // Only one statement, can omit { } but not recommended
else {
    cout << "FAIL" << endl;    // more than one statements, need { }
    cout << "Try Harder!" << endl;
}
  
```

However, I recommend that you keep the braces, even though there is only one statement in the block, to improve the readability of your program.

Exercises

[TODO]

5.3 Loop Flow Control

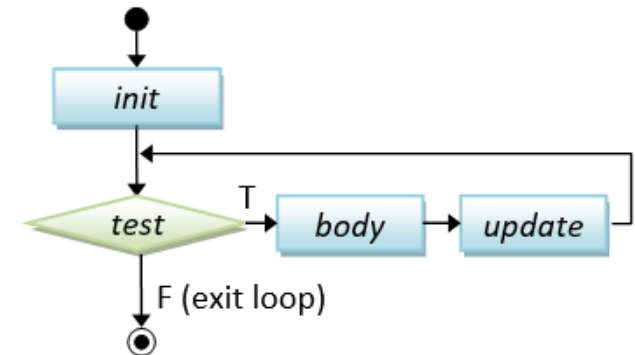
Again, there are a few types of loops: *for-loop*, *while-do*, and *do-while*.

Syntax

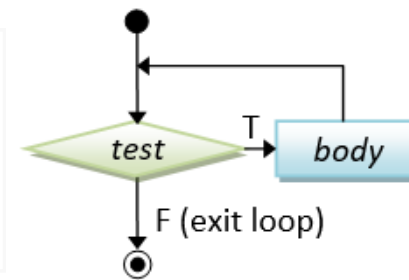
Example

Flowchart

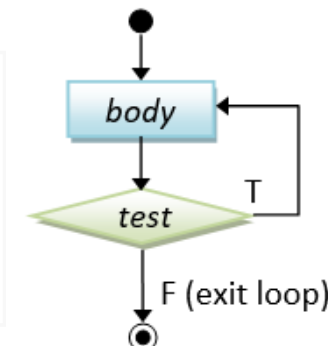
<pre>// for-loop for (init; test; post-proc) { body ; }</pre>	<pre>// Sum from 1 to 1000 int sum = 0; for (int number = 1; number <= 1000; ++number) { sum += number; }</pre>
---	--



<pre>// while-do while (condition) { body ; }</pre>	<pre>int sum = 0, number = 1; while (number <= 1000) { sum += number; ++number; }</pre>
---	--



<pre>// do-while do { body ; } while (condition);</pre>	<pre>int sum = 0, number = 1; do { sum += number; ++number; } while (number <= 1000);</pre>
---	--



The difference between *while-do* and *do-while* lies in the order of the *body* and *condition*. In *while-do*, the *condition* is tested first. The body will be executed if the *condition* is true and the process repeats. In *do-while*, the *body* is executed and then the *condition* is tested. Take note that the *body* of *do-while* will be executed at least once (vs. possibly zero for *while-do*).

Suppose that your program prompts user for a number between 1 to 10, and checks for valid input, do-while with a boolean flag could be more appropriate.

```
// Input with validity check
bool valid = false;
int number;
do {
    // prompt user to enter an int between 1 and 10
    .....
    // if the number entered is valid, set done to exit the loop
    if (number >= 1 && number <= 10) {
        valid = true;
    }
} while (!valid); // Need a semi-colon to terminate do-while
```

Below is an example of using while-do:

```
// Game loop
bool gameOver = false;
while (!gameOver) {
    // play the game
    .....
    // Update the game state
    // Set gameOver to true if appropriate to exit the game loop
    .....
}
```

Example (Counter-Controlled Loop): Prompt user for an upperbound. Sum the integers from 1 to a given upperbound and compute its average.

```
1  /*
2   * Sum from 1 to a given upperbound and compute their average (SumNumbers.cpp)
3   */
4  #include <iostream>
5  using namespace std;
6
7  int main() {
8      int sum = 0;    // Store the accumulated sum
9      int upperbound;
10     cout << "Enter the upperbound: ";
11     cin >> upperbound;
12
13     // Sum from 1 to the upperbound
14     for (int number = 1; number <= upperbound; ++number) {
15         sum += number;
16     }
17     cout << "Sum is " << sum << endl;
18     cout << "Average is " << (double)sum / upperbound << endl;
19
20     // Sum only the odd numbers
```

```

21     int count = 0;    // counts of odd numbers
22     sum = 0;         // reset sum
23     for (int number=1; number <= upperbound; number=number+2) {
24         ++count;
25         sum += number;
26     }
27     cout << "Sum of odd numbers is " << sum << endl;
28     cout << "Average is " << (double)sum / count << endl;
29 }

```

Example (Sentinel-Controlled Loop): Prompt user for positive integers, and display the count, maximum, minimum and average. Terminate when user enters -1.

```

1  /* Prompt user for positive integers and display the count, maximum,
2     minimum and average. Terminate the input with -1 (StatNumbers.cpp) */
3  #include <iostream>
4  #include <climits> // for INT_MAX
5  #include <iomanip>  // for setprecision(n)
6  using namespace std;
7
8  int main() {
9      int numberIn;    // input number (positive integer)
10     int count = 0;    // count of inputs, init to 0
11     int sum = 0;      // sum of inputs, init to 0
12     int max = 0;      // max of inputs, init to minimum
13     int min = INT_MAX; // min of inputs, init to maximum (need <climits>)
14     int sentinel = -1; // Input terminating value
15
16     // Read Inputs until sentinel encountered
17     cout << "Enter a positive integer or " << sentinel << " to exit: ";
18     while (cin >> numberIn && numberIn != sentinel) {
19         // Check input for positive integer
20         if (numberIn > 0) {
21             ++count;
22             sum += numberIn;
23             if (max < numberIn) max = numberIn;
24             if (min > numberIn) min = numberIn;
25         } else {
26             cout << "error: input must be positive! try again..." << endl;
27         }
28         cout << "Enter a positive integer or " << sentinel << " to exit: ";
29     }
30
31     // Print result
32     cout << endl;
33     cout << "Count is " << count << endl;
34     if (count > 0) {

```

```

35     cout << "Maximum is " << max << endl;
36     cout << "Minimum is " << min << endl;
37     cout << fixed << setprecision(2);
38     // print floating point to 2 decimal places (need <iomanip>)
39     cout << "Average is " << (double)sum / count << endl;
40 }
41 }

```

Program Notes

- In computing, a *sentinel value* is a special value that indicates the end of data (e.g., a negative value to end a sequence of positive value, end-of-file, null character in the null-terminated string). In this example, we use -1 as the sentinel value to indicate the end of inputs, which is a sequence of positive integers. Instead of hardcoding the value of -1, we use a variable called `sentinel` for flexibility and ease-of-maintenance.
- Take note of the *while-loop pattern* in reading the inputs. In this pattern, you need to *repeat* the prompting statement.
- To control the precision of floating point numbers, use:

```
cout << fixed << setprecision(n);
```

where `n` is the number of decimal places (after the decimal point). You need to include `<iomanip>` header. The `setprecision()` is *sticky*. That is, it will remain in effect until another value is set.

Exercises

[TODO]

5.4 Interrupting Loop Flow - "break" and "continue"

The `break` statement breaks out and exits the current (innermost) loop.

The `continue` statement aborts the current iteration and continue to the next iteration of the current (innermost) loop.

`break` and `continue` are poor structures as they are hard to read and hard to follow. Use them only if absolutely necessary. You can always write the same program without using `break` and `continue`.

Example (break): The following program lists the non-prime numbers between 2 and an upperbound.

```

1  /*
2   * List non-prime from 1 to an upperbound (NonPrimeList.cpp).
3   */
4  #include <iostream>
5  #include <cmath>
6  using namespace std;
7
8  int main() {
9      int upperbound;
10     cout << "Enter the upperbound: ";
11     cin >> upperbound;

```

```

12     for (int number = 2; number <= upperbound; ++number) {
13         // Not a prime, if there is a factor between 2 and sqrt(number)
14         int maxFactor = (int)sqrt(number);
15         for (int factor = 2; factor <= maxFactor; ++factor) {
16             if (number % factor == 0) { // Factor?
17                 cout << number << " ";
18                 break; // A factor found, no need to search for more factors
19             }
20         }
21     }
22     cout << endl;
23     return 0;
24 }

```

Let's rewrite the above program to list all the primes instead. A boolean flag called `isPrime` is used to indicate whether the current number is a prime. It is then used to control the printing.

```

1  /*
2  * List primes from 1 to an upperbound (PrimeListWithBreak.cpp).
3  */
4  #include <iostream>
5  #include <cmath>
6  using namespace std;
7
8  int main() {
9      int upperbound;
10     cout << "Enter the upperbound: ";
11     cin >> upperbound;
12     for (int number = 2; number <= upperbound; ++number) {
13         // Not a prime, if there is a factor between 2 and sqrt(number)
14         int maxFactor = (int)sqrt(number);
15         bool isPrime = true; // boolean flag to indicate whether number is a prime
16         for (int factor = 2; factor <= maxFactor; ++factor) {
17             if (number % factor == 0) { // Factor?
18                 isPrime = false; // number is not a prime
19                 break; // A factor found, no need to search for more factors
20             }
21         }
22         if (isPrime) cout << number << " ";
23     }
24     cout << endl;
25     return 0;
26 }

```

Let's rewrite the above program without using break statement. A while loop is used (which is controlled by the boolean flag) instead of for loop with break.

```

1  /*

```

```
2  * List primes from 1 to an upperbound (PrimeList.cpp).
3  */
4  #include <iostream>
5  #include <cmath>
6  using namespace std;
7
8  int main() {
9      int upperbound;
10     cout << "Enter the upperbound: ";
11     cin >> upperbound;
12
13     for (int number = 2; number <= upperbound; ++number) {
14         // Not prime, if there is a factor between 2 and sqrt of number
15         int maxFactor = (int)sqrt(number);
16         bool isPrime = true;
17         int factor = 2;
18         while (isPrime && factor <= maxFactor) {
19             if (number % factor == 0) { // Factor of number?
20                 isPrime = false;
21             }
22             ++factor;
23         }
24         if (isPrime) cout << number << " ";
25     }
26     cout << endl;
27     return 0;
28 }
```

Example (continue):

```
// Sum 1 to upperbound, exclude 11, 22, 33,...
int upperbound = 100;
int sum = 0;
for (int number = 1; number <= upperbound; ++number) {
    if (number % 11 == 0) continue; // Skip the rest of the loop body, continue to the next iteration
    sum += number;
}
// It is better to re-write the loop as:
for (int number = 1; number <= upperbound; ++number) {
    if (number % 11 != 0) sum += number;
}
```

Example (break and continue): Study the following program.

```
1  /* A mystery series (Mystery.cpp) */
2  #include <iostream>
3  using namespace std;
4
```

```
5  int main() {
6      int number = 1;
7      while (true) {
8          ++number;
9          if ((number % 3) == 0) continue;
10         if (number == 133) break;
11         if ((number % 2) == 0) {
12             number += 3;
13         } else {
14             number -= 3;
15         }
16         cout << number << " ";
17     }
18     cout << endl;
19     return 0;
20 }
```

5.5 Terminating Program

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*.

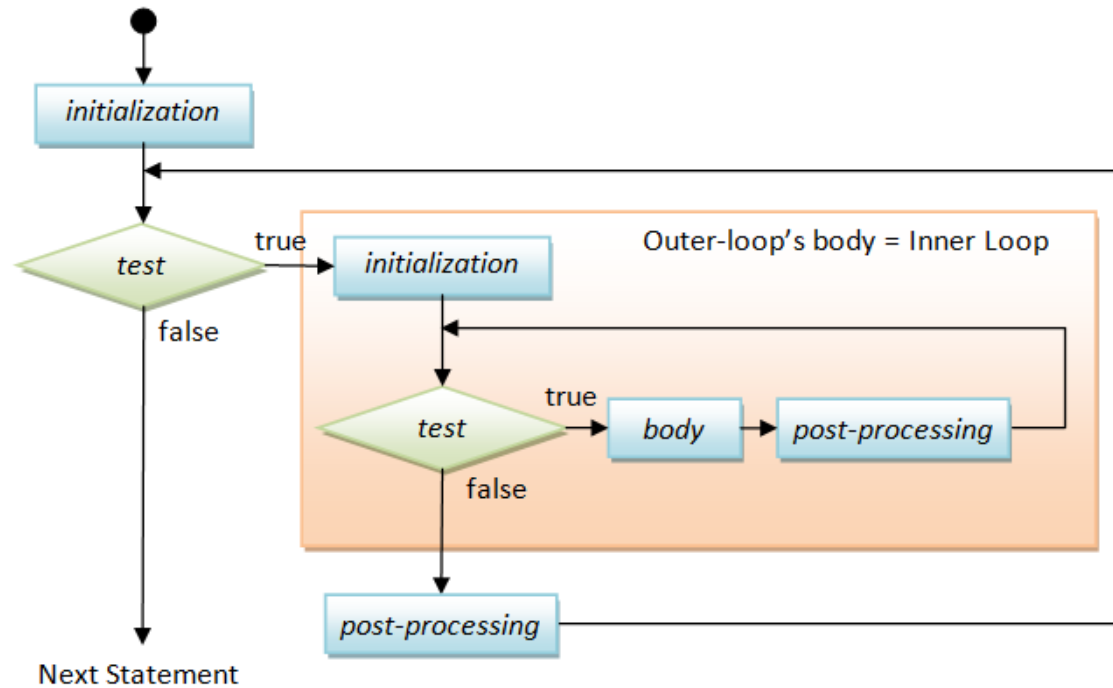
```
if (errorCount > 10) {
    cout << "too many errors" << endl;
    exit(-1); // Terminate the program
             // OR abort();
}
```

The "return" Statement: You could also use a "return *returnValue*" statement in the `main()` function to terminate the program and return control back to the Operating System. For example,

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

5.6 Nested Loops

The following diagram illustrates a nested for-loop, i.e., an inner for-loop within an outer for-loop.



Try out the following program, which prints a 8-by-8 checker box pattern using *nested loops*, as follows:

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

```

1  /*
2  *   Print square pattern (PrintSquarePattern.cpp).
3  */
4  #include <iostream>
5  using namespace std;
6
7  int main() {
8      int size = 8;
9      for (int row = 1; row <= size; ++row) {    // Outer loop to print all the rows
10         for (int col = 1; col <= size; ++col) { // Inner loop to print all the columns of each row
11             cout << "# ";

```



```

12     }
13     cout << endl;    // A row ended, bring the cursor to the next line
14 }
15
16     return 0;
17 }

```

This program contains two *nested* for-loops. The inner loop is used to print a row of eight "# ", which is followed by printing a newline. The outer loop repeats the inner loop to print all the rows.

Suppose that you want to print this pattern instead (in program called PrintCheckerPattern.cpp):

```

# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #
# # # # # # # #

```

You need to print an additional space for even-number rows. You could do so by adding the following statement before Line 8.

```

if ((row % 2) == 0) {    // print a leading space for even-numbered rows
    cout << " ";
}

```

Exercises

1. Print these patterns using nested loop (in a program called PrintPattern1x). Use a variable called size for the size of the pattern and try out various sizes. You should use as few printing statements as possible.

# * # * # * # *	# # # # # # # #	# # # # # # # #	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)

Hints:

The equations for major and opposite diagonals are $\text{row} = \text{col}$ and $\text{row} + \text{col} = \text{size} + 1$. Decide on what to print above and below the diagonal.

2. Print the timetable of 1 to 9, as follows, using nested loop. (Hints: you need to use an *if-else* statement to check whether the product is single-digit or double-digit, and print an additional space if needed.)

```

1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
.....

```

3. Print these patterns using nested loop.

```

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

```

5.7 Some Issues in Flow Control

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
}

```

Endless Loop: The following constructs:

```
while (true) { ..... }
```

is commonly used. It seems to be an endless loop (or infinite loop), but it is usually terminated via a break or return statement inside the loop body. This kind of code is hard to read - avoid if possible by re-writing the condition.

5.8 Exercises

[TODO]

6. Writing Correct and Good Programs

It is important to write programs that produce the correct results. It is also important to write programs that others (and you yourself three days later) can understand, so that the programs can be maintained - I call these programs good programs.

Here are the suggestions:

- Follow established convention so that everyone has the same basis of understanding.
- Format and layout of the source code with appropriate indents, white spaces and white lines. Use 3 or 4 spaces for indent, and blank lines to separate sections of codes.
- Choose good names that are self-descriptive and meaningful, e.g., `row`, `col`, `size`, `xMax`, `numStudents`. Do not use meaningless names, such as `a`, `b`, `c`, `d`. Avoid single-alphabet names (easier to type but often meaningless), except common names like `x`, `y`, `z` for co-ordinates and `i` for index.
- Provide comments to explain the important as well as salient concepts. Comment your codes liberally.
- Write your program documentation while writing your programs.
- Avoid *un-structured* constructs, such as `break` and `continue`, which are hard to follow.
- Use "mono-space" fonts (such as Consola, Courier New, Courier) for writing/displaying your program.

Programming Errors

There are generally three classes of programming errors:

1. *Compilation Error* (or *Syntax Error*): can be fixed easily.
2. *Runtime Error*: program halts pre-maturely without producing the results - can also be fixed easily.
3. *Logical Error*: program completes but produces incorrect results. It is easy to detect if the program always produces wrong result. It is extremely hard to fix if the program produces the correct result most of the times, but incorrect result sometimes. For example,

```
// Can compile and execute, but give wrong result – sometimes!  
if (mark > 50) {  
    cout << "PASS" << endl;  
} else {  
    cout << "FAIL" << endl;  
}
```

This kind of errors is very serious if it is not caught before production. Writing good programs helps in minimizing and detecting these errors. A good *testing strategy* is needed to ascertain the correctness of the program. *Software testing* is an advanced topics which is beyond our current scope.

Debugging Programs

Here are the common debugging techniques:

1. Stare at the screen! Unfortunately, errors usually won't pop-up even if you stare at it extremely hard.
2. Study the error messages! Do not close the console when error occurs and pretending that everything is fine. This helps most of the times.

3. Insert print statements at appropriate locations to display the intermediate results. It works for simple toy program, but it is neither effective nor efficient for complex program.
4. Use a graphic debugger. This is the most effective means. Trace program execution step-by-step and watch the value of variables and outputs.
5. Advanced tools such as profiler (needed for checking memory leak and function usage).
6. Proper program testing to wipe out the logical errors.

Testing Your Program for Correctness

How to ensure that your program always produces correct result, 100% of the times? It is impossible to try out all the possible outcomes, even for a simple program. Program testing usually involves a set of representative test cases, which are designed to catch the major classes of errors. Program testing is beyond the scope of this writing.

7. Strings

C++ supports two types of strings:

1. the original C-style string: A string is a `char` array, terminated with a NULL character `'\0'` (Hex 0). It is also called Character-String or C-style string. C-string will be discussed [later](#).
2. the new `string` class introduced in C++98.

The "high-level" `string` class is recommended, because it is much easier to use and understood. However, many legacy programs used C-strings; many programmers also use "low-level" C-strings for full control and efficiency; furthermore, in some situation such as command-line arguments, only C-strings are supported. Hence, you may have to understand both sets of strings. However, avoid C-string unless it is absolutely necessary.

We shall describe `string` class here, and C-string later.

7.1 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
```

7.2 String Input/Output

For example,

```

1  /* Testing string class input and output (TestStringIO.cpp) */
2  #include <iostream>
3  #include <string>      // Need this header to use string class
4  #include <limits>
5  using namespace std; // Also needed for <string>
6
7  int main() {
8      string message("Hello");
9      cout << message << endl;
10
11     // Input a word (delimited by space) into a string
12     cout << "Enter a message (no space): ";
13     cin >> message;
14     cout << message << endl;
15
16     cin.ignore(numeric_limits<streamsize>::max(), '\n');
17     // flush cin up to newline (need <limits> header)
18
19     // Input a line into a string
20     cout << "Enter a message (with spaces): ";
21     getline(cin, message); // Read input from cin into message
22     cout << message << endl;
23     return 0;
24 }
```

NOTES:

- We need to "#include <string>" to use the string class, and "using namespace std" as string is defined under std namespace.
- "cin >> aStr" reads a word (delimited by space) from cin (keyboard), and assigns to string variable aStr.
- getline(cin, aStr) reads the entire line (up to '\n') from cin, and assigns to aStr. The '\n' character is discarded.
- To flush cin, you could use ignore(numeric_limits<streamsize>::max(), '\n') function to discard all the characters up to '\n'. numeric_limits is in the <limits> header.

7.3 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"
```

- Many others.

Example 1:

```
1  /* Example on C++ string function (TestStringOp.cpp) */
2  #include <iostream>
3  #include <string>    // use string class
4  using namespace std;
```

```
5
6 int main() {
7     string msg = "hello, world!";
8     cout << msg << endl;
9     cout << msg.length() << endl; // length of string
10    cout << msg.at(1) << endl;    // char at index 1
11    cout << msg[1] << endl;       // same as above
12    cout << msg.empty() << endl;  // test for empty string
13    cout << msg.substr(3, 3) << endl; // sub-string begins at
14                                   // pos 3 of size 3
15    cout << msg.replace(3, 3, "why") << endl; // replace sub-string
16    cout << msg.append("end") << endl;      // append behind
17    cout << msg + "end" << endl;           // same as above
18    cout << msg.insert(3, "insert") << endl; // insert after pos 3
19
20    string msg1;
21    msg1 = msg; // copy
22    cout << msg1 << endl;
23
24    cout << "Enter a line: ";
25    getline(cin, msg); // read a line of input
26    cout << msg << endl;
27 }
```

Example 2:

[TODO]

7.4 Exercises

[TODO]

8. 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 (with just enough width to accommodate the field).
- `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.

8.1 Output Formatting

Example

```
1  /* Test Formatting Output (TestFormattedOutput.cpp) */
2  #include <iostream>
3  #include <iomanip>    // Needed to do formatted I/O
4  using namespace std;
5
6  int main() {
7      // Floating point numbers
8      double pi = 3.14159265;
9      cout << fixed << setprecision(4); // fixed format with 4 decimal places
10     cout << pi << endl;
11     cout << "|" << setw(8) << pi << "|" << setw(10) << pi << "|" << endl;
12     // setw() is not sticky, only apply to the next operation.
13     cout << setfill('-');
14     cout << "|" << setw(8) << pi << "|" << setw(10) << pi << "|" << endl;
15     cout << scientific; // in scientific format with exponent
16     cout << pi << endl;
17
18     // booleans
19     bool done = false;
20     cout << done << endl; // print 0 (for false) or 1 (for true)
21     cout << boolalpha;    // print true or false
22     cout << done << endl;
23     return 0;
24 }
```

8.2 Input Formatting

Example

```
1  /* Test Formatting Input (TestFormattedInput.cpp) */
2  #include <iostream>
3  #include <iomanip>
4  #include <string>
5  using namespace std;
6
7  int main() {
8      string areaCode, phoneCode;
9      string inStr;
10 }
```

```

11     cout << "Enter your phone number in this format (xxx)xxx-xxxx : ";
12     cin.ignore();    // skip '('
13     cin >> setw(3) >> areaCode;
14     cin.ignore();    // skip ')'
15     cin >> setw(3) >> phoneCode;
16     cin.ignore();    // skip '-'
17     cin >> setw(4) >> inStr;
18     phoneCode += inStr;
19
20     cout << "Phone number is (" << areaCode << ")"
21         << phoneCode.substr(0, 3) << "-"
22         << phoneCode.substr(3, 4) << endl;
23     return 0;
24 }

```

8.3 Exercises

[TODO]

9. Arrays

9.1 Array Declaration and Usage

Suppose that you want to find the average of the marks for a class of 30 students, you certainly do not want to create 30 variables: mark1, mark2, ..., mark30. Instead, You could use a single variable, called an *array*, with 30 elements.

An array is *a list of elements of the same type*, identified by a pair of square brackets []. To use an array, you need to *declare* the array with 3 things: a *name*, a *type* and a *dimension* (or *size*, or *length*). The syntax is:

```
type arrayName[arraylength]; // arraylength can be a literal or a variable
```

I recommend using a plural name for array, e.g., marks, rows, numbers. For example,

```

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

// Some compilers support an variable as array length, e.g.,
int size;
cout << "Enter the length of the array: ";
cin >> size;
float values[size];

```

Take note that, in C++, the value of the elements are undefined after declaration.

You can also initialize the array during declaration with a comma-separated list of values, as follows:

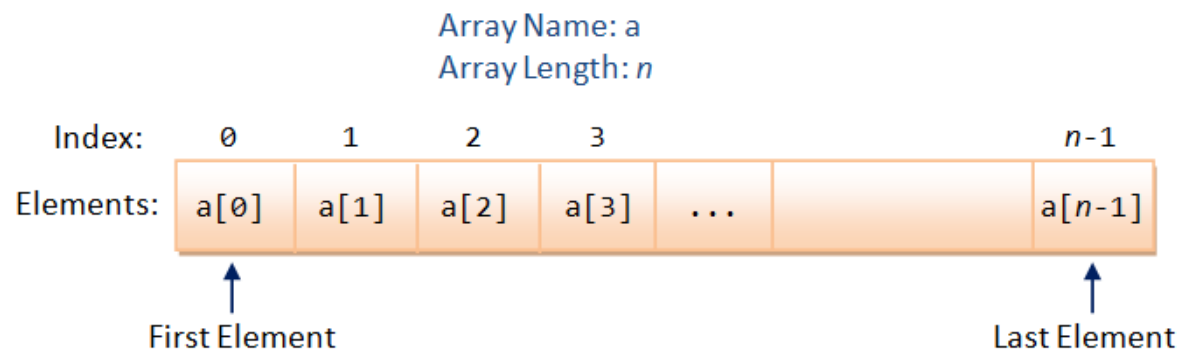
```
// Declare and initialize an int array of 3 elements
int numbers[3] = {11, 33, 44};
// If length is omitted, the compiler counts the elements
int numbers[] = {11, 33, 44};
// Number of elements in the initialization shall be equal to or less than length
int numbers[5] = {11, 33, 44}; // Remaining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

```
1  /* Test local array initialization (TestArrayInit.cpp) */
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      int const SIZE = 5;
7
8      int a1[SIZE]; // Uninitialized
9      for (int i = 0; i < SIZE; ++i) cout << a1[i] << " ";
10     cout << endl; // ? ? ? ? ?
11
12     int a2[SIZE] = {21, 22, 23, 24, 25}; // All elements initialized
13     for (int i = 0; i < SIZE; ++i) cout << a2[i] << " ";
14     cout << endl; // 21 22 23 24 25
15
16     int a3[] = {31, 32, 33, 34, 35}; // Size deduced from init values
17     int a3Size = sizeof(a3)/sizeof(int);
18     cout << "Size is " << a3Size << endl; // 5
19     for (int i = 0; i < a3Size; ++i) cout << a3[i] << " ";
20     cout << endl; // 31 32 33 34 35
21
22     int a4[SIZE] = {41, 42}; // Leading elements initialized, the rests to 0
23     for (int i = 0; i < SIZE; ++i) cout << a4[i] << " ";
24     cout << endl; // 41 42 0 0 0
25
26     int a5[SIZE] = {0}; // First elements to 0, the rests to 0 too
27     for (int i = 0; i < SIZE; ++i) cout << a5[i] << " ";
28     cout << endl; // 0 0 0 0 0
29
30     int a6[SIZE] = {}; // All elements to 0 too
31     for (int i = 0; i < SIZE; ++i) cout << a6[i] << " ";
32     cout << endl; // 0 0 0 0 0
33 }
```

You can refer to an element of an array via an index (or subscript) enclosed within the square bracket `[]`. C++'s array index begins with zero. For example, suppose that `marks` is an `int` array of 5 elements, then the 5 elements are: `marks[0]`, `marks[1]`, `marks[2]`, `marks[3]`, and `marks[4]`.

```
// Declare & allocate a 5-element int array
int marks[5];
// Assign values to the elements
marks[0] = 95;
marks[1] = 85;
marks[2] = 77;
marks[3] = 69;
marks[4] = 66;
cout << marks[0] << endl;
cout << marks[3] + marks[4] << endl;
```



To create an array, you need to know the length (or size) of the array in advance, and allocate accordingly. Once an array is created, its length is fixed and cannot be changed. At times, it is hard to ascertain the length of an array (e.g., how many students in a class?). Nonetheless, you need to estimate the length and allocate an upper bound. This is probably the major drawback of using an array. C++ has a vector template class (and C++11 added an array template class), which supports dynamic resizable array.

You can find the array length using expression `sizeof(arrayName)/sizeof(arrayName[0])`, where `sizeof(arrayName)` returns the total bytes of the array and `sizeof(arrayName[0])` returns the bytes of first element.

C/C++ does not perform array *index-bound check*. In other words, if the index is beyond the array's bounds, it does not issue a warning/error. For example,

```
const int size = 5;
int numbers[size]; // array index from 0 to 4

// Index out of bound!
// Can compile and run, but could pose very serious side effect!
numbers[88] = 999;
cout << numbers[77] << endl;
```

This is another pitfall of C/C++. Checking the index bound consumes computation power and depicts the performance. However, it is better to be safe than fast. Newer programming languages such as Java/C# performs array index bound check.

9.2 Array and Loop

Arrays works hand-in-hand with loops. You can process all the elements of an array via a loop, for example,

```
1  /*
2  *   Find the mean and standard deviation of numbers kept in an array (MeanStdArray.cpp).
3  */
4  #include <iostream>
5  #include <iomanip>
6  #include <cmath>
7  #define SIZE 7
8  using namespace std;
9
10 int main() {
11     int marks[] = {74, 43, 58, 60, 90, 64, 70};
12     int sum = 0;
13     int sumSq = 0;
14     double mean, stdDev;
15     for (int i = 0; i < SIZE; ++i) {
16         sum += marks[i];
17         sumSq += marks[i]*marks[i];
18     }
19     mean = (double)sum/SIZE;
20     cout << fixed << "Mean is " << setprecision(2) << mean << endl;
21
22     stdDev = sqrt((double)sumSq/SIZE - mean*mean);
23     cout << fixed << "Std dev is " << setprecision(2) << stdDev << endl;
24
25     return 0;
26 }
```

Exercises

[TODO]

9.3 Range-based for loop (C++11)

C++11 introduces a range-based for loop (or for-each loop) to iterate thru an array, as illustrated in the following example:

```
1  /* Testing For-each loop (TestForEach.cpp) */
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      int numbers[] = {11, 22, 33, 44, 55};
```

```

7
8 // For each member called number of array numbers - read only
9 for (int number : numbers) {
10     cout << number << endl;
11 }
12
13 // To modify members, need to use reference (&)
14 for (int &number : numbers) {
15     number = 99;
16 }
17
18 for (int number : numbers) {
19     cout << number << endl;
20 }
21 return 0;
22 }

```

To compile the program under GNU GCC (g++), you may need to specify option `-std=c++0x` or `-std=c++11`:

```

g++ -std=c++0x -o TestForEach.exe TestForEach.cpp
// or
g++ -std=c++11 -o TestForEach.exe TestForEach.cpp

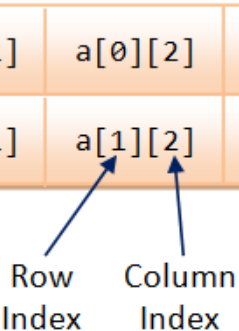
```

9.4 Multi-Dimensional Array

For example,

```
int[2][3] = { {11, 22, 33}, {44, 55, 66} };
```

	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]	...



For 2D array (table), the first index is the row number, second index is the column number. The elements are stored in a so-called *row-major* manner, where the column index runs out first.

Example

```

1  /* Test Multi-dimensional Array (Test2DArray.cpp) */
2  #include <iostream>
3  using namespace std;
4  void printArray(const int[][3], int);
5
6  int main() {
7      int myArray[][3] = {{8, 2, 4}, {7, 5, 2}}; // 2x3 initialized
8          // Only the first index can be omitted and implied
9      printArray(myArray, 2);
10     return 0;
11 }
12
13 // Print the contents of rows-by-3 array (columns is fixed)
14 void printArray(const int array[][3], int rows) {
15     for (int i = 0; i < rows; ++i) {
16         for (int j = 0; j < 3; ++j) {
17             cout << array[i][j] << " ";
18         }
19         cout << endl;
20     }
21 }

```

9.5 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.

```

For novices, avoid C-string. Use C++ string (in header <string>) discussed earlier.

Example

You can use cin and cout to handle C-strings.

- cin << reads a string delimited by whitespace;
- cin.getline(var, size) reads a string of into var till newline of length up to size-1, discarding the newline (replaced by '\0'). The size typically corresponds to the length of the C-string array.

- `cin.get(var, size)` reads a string till newline, but leaves the newline in the input buffer.
- `cin.get()`, without argument, reads the next character.

```
1  /* Test C-string (TestCString.cpp) */
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      char msg[256]; // Hold a string of up to 255 characters (terminated by '\0')
7
8      cout << "Enter a message (with space)" << endl;
9      cin.getline(msg, 256); // Read up to 255 characters into msg
10     cout << msg << endl;
11
12     // Access via null-terminated character array
13     for (int i = 0; msg[i] != '\0'; ++i) {
14         cout << msg[i];
15     }
16     cout << endl;
17
18     cout << "Enter a word (without space)" << endl;
19     cin >> msg;
20     cout << msg << endl;
21
22     // Access via null-terminated character array
23     for (int i = 0; msg[i] != '\0'; ++i) {
24         cout << msg[i];
25     }
26     cout << endl;
27     return 0;
28 }
```

9.6 Exercises

[TODO]

10. Functions

10.1 Why 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:

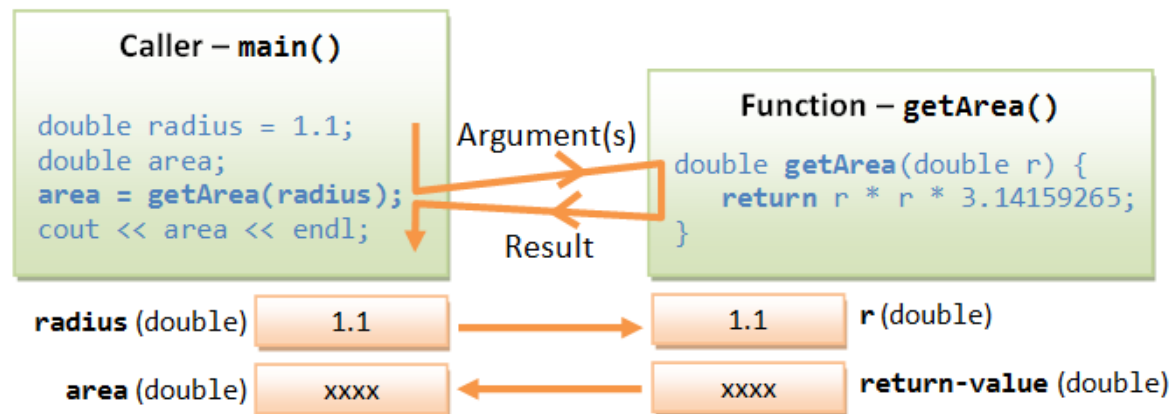
1. *Divide and conquer*: construct the program from simple, small pieces or components. Modularize the program into self-contained tasks.
2. *Avoid repeating codes*: It is easy to copy and paste, but hard to maintain and synchronize all the copies.
3. *Software Reuse*: you can reuse the functions in 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.

10.2 Using Functions

Get Started with an Example

Suppose that we need to evaluate the area of a circle many times, it is better to write a function called `getArea()`, and re-use it when needed.



```
1  /* Test Function (TestFunction.cpp) */
2  #include <iostream>
3  using namespace std;
4  const int PI = 3.14159265;
5
6  // Function Prototype (Function Declaration)
7  double getArea(double radius);
8
9  int main() {
10     double radius1 = 1.1, area1, area2;
11     // call function getArea()
12     area1 = getArea(radius1);
13     cout << "area 1 is " << area1 << endl;
14     // call function getArea()
15     area2 = getArea(2.2);
16     cout << "area 2 is " << area2 << endl;
```

```
17 // call function getArea()
18 cout << "area 3 is " << getArea(3.3) << endl;
19 }
20
21 // Function Definition
22 // Return the area of a circle given its radius
23 double getArea(double radius) {
24     return radius * radius * PI;
25 }
```

```
area 1 is 3.63
area 2 is 14.52
area 3 is 32.67
```

In the above example, a reusable function called `getArea()` is defined, which receives a parameter (in `double`) from the caller, performs the calculation, and return a piece of result (in `double`) to the caller. In the `main()`, we invoke `getArea()` functions thrice, each time with a different parameter.

In C++, you need to declare a *function prototype* (before the function is used), and provide a *function definition*, with a body containing the programmed operations.

Function Definition

The syntax for function definition is as follows:

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

- The *parameterList* consists of comma-separated *parameter-type* and *parameter-name*, i.e., *param-1-type param-1-name, param-2-type param-2-name, ...*
- The *returnValueType* specifies the type of the return value, such as `int` or `double`. An special return type called `void` can be used to denote that the function returns no value. In C++, a function is allowed to return one value or no value (`void`). It cannot return multiple values. [C++ does not allow you to return an array!]

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
```

Take note that invoking a function (by the caller) transfers the control to the function. The return statement in the function transfers the control back to the caller.

Function Naming Convention

A function's name shall be a verb or verb phrase (action), comprising one or more words. The first word is in lowercase, while the rest are initial-capitalized (known as *camel-case*). For example, `getArea()`, `setRadius()`, `moveDown()`, `isPrime()`, etc.

Function Prototype

In C++, a function must be declared before it can be called. It can be achieved by either placing the *function definition* before it is being used, or declare a so-called *function prototype*.

A function prototype tells the compiler the function's interface, i.e., the return-type, function name, and the parameter type list (the number and type of parameters). The function can now be defined anywhere in the file. For example,

```
// Function prototype - placed before the function is used.
double getArea(double); // without the parameter name
int max(int, int);
```

You could optionally include the parameter names in the function prototype. The names will be ignored by the compiler, but serve as documentation. For example,

```
// Function Prototype
double getArea(double radius); // parameter names are ignored, but serve as documentation
int max(int number1, int number2);
```

Function prototypes are usually grouped together and placed in a so-called *header file*. The header file can be included in many programs. We will discuss header file later.

Another Example

We have a function called `max(int, int)`, which takes two `int` and return their maximum. We invoke the `max()` function from the `main()`.

```
1  /* Testing max function (TestMaxFunction.cpp) */
2  #include <iostream>
3  using namespace std;
4
5  int maximum(int, int); // Function prototype (declaration)
6
7  int main() {
8      cout << maximum(5, 8) << endl; // Call maximum() with literals
9
10     int a = 6, b = 9, c;
11     c = maximum(a, b);           // Call maximum() with variables
12     cout << c << endl;
13
14     cout << maximum(c, 99) << endl; // Call maximum()
15 }
16
17 // Function definition
18 // A function that returns the maximum of two given int
19 int maximum(int num1, int num2) {
20     return (num1 > num2) ? num1 : num2;
21 }
```

The "void" Return Type

Suppose that you need a function to perform certain actions (e.g., printing) without a need to return a value to the caller, you can declare its return-value type as `void`. In the function's body, you could use a `"return;"` statement without a return value to return control to the caller. In this case, the `return` statement is optional. If there is no `return` statement, the entire body will be executed, and control returns to the caller at the end of the body.

Actual Parameters vs. Formal Parameters

Recall that a function receives *arguments* from its caller, performs the actions defined in the function's body, and return a value (or nothing) to the caller.

In the above example, the variable (`double radius`) declared in the signature of `getArea(double radius)` is known as *formal parameter*. Its scope is within the function's body. When the function is invoked by a caller, the caller must supply so-called *actual parameters* (or *arguments*), whose value is then used for the actual computation. For example, when the function is invoked via `"area1 = getArea(radius1)"`, `radius1` is the actual parameter, with a value of 1.1.

Scope of Function's Local Variables and Parameters

All variables, including function's parameters, declared inside a function are available only to the function. They are created when the function is called, and freed (destroyed) after the function returns. They are called *local variables* because they are local to the function and not available outside the function. They are also called *automatic variables*, because they are created and destroyed automatically - no programmer's explicit action needed to allocate and deallocate them.

Boolean Functions

A boolean function returns a `bool` value (of either `true` or `false`) to the caller.

Suppose that we wish to write a function called `isOdd()` to check if a given number is odd.

```

1  /*
2   * Test Boolean function (BooleanfunctionTest.cpp).
3   */
4  #include <iostream>
5  using namespace std;
6
7  // Function Prototype
8  bool isOdd(int);
9
10 int main() {
11     cout << boolalpha;    // print bool as true or false
12     cout << isOdd(5) << endl; // true
13     cout << isOdd(6) << endl; // false
14     cout << isOdd(-5) << endl; // false
15 }
16
17 bool isOdd(int number) {
18     if (number % 2 == 1) {
19         return true;
20     } else {

```

```
21     return false;
22 }
23 }
```

This seemingly correct code produces false for -5, because $-5\%2$ is -1 instead of 1. You may rewrite the condition:

```
bool isOdd(int number) {
    if (number % 2 == 0) {
        return false;
    } else {
        return true;
    }
}
```

The above code produces the correct answer, but is poor. For boolean function, you should simply return the resultant bool value of the comparison, instead of using a conditional statement, as follow:

```
bool isEven(int number) {
    return (number % 2 == 0);
}

bool isOdd(int number) {
    return !(number % 2 == 0); // OR return !isEven(number);
}

int main() {
    int number = -9;
    if (isEven(number)) { // Don't write (isEven(number) == true)
        cout << "Even" << endl;
    }
}
```

10.3 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. For example,

```
1  /* Test Function default arguments (functionDefaultArgument.cpp) */
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype - Specify the default arguments here
6  int fun1(int = 1, int = 2, int = 3);
7  int fun2(int, int, int = 3);
8
9  int main() {
```

```

10     cout << fun1(4, 5, 6) << endl; // No default
11     cout << fun1(4, 5) << endl;    // 4, 5, 3(default)
12     cout << fun1(4) << endl;      // 4, 2(default), 3(default)
13     cout << fun1() << endl;       // 1(default), 2(default), 3(default)
14
15     cout << fun2(4, 5, 6) << endl; // No default
16     cout << fun2(4, 5) << endl;    // 4, 5, 3(default)
17     // cout << fun2(4) << endl;
18     // error: too few arguments to function 'int fun2(int, int, int)'
19 }
20
21 int fun1(int n1, int n2, int n3) {
22     // cannot repeat default arguments in function definition
23     return n1 + n2 + n3;
24 }
25
26 int fun2(int n1, int n2, int n3) {
27     return n1 + n2 + n3;
28 }

```

You should specify the default arguments in the function prototype (declaration). They can only be defined once (one-definition rule), and cannot be repeated in the function definition.

Default argument is not absolutely necessary. The codes could be hard to maintain.

10.4 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. For example,

```

1  /* Test Function Overloading (FunctionOverloading.cpp) */
2  #include <iostream>
3  using namespace std;
4
5  void fun(int, int, int); // Version 1
6  void fun(double, int);   // Version 2
7  void fun(int, double);   // Version 3
8
9  int main() {
10     fun(1, 2, 3); // version 1
11     fun(1.0, 2);  // version 2
12     fun(1, 2.0);  // version 3
13     fun(1.1, 2, 3); // version 1 - double 1.1 casted to int 1 (without warning)
14
15     // fun(1, 2, 3, 4);
16     // error: no matching function for call to 'fun(int, int, int, int)'

```

```

17 // fun(1, 2);
18 // error: call of overloaded 'fun(int, int)' is ambiguous
19 // note: candidates are:
20 //     void fun(double, int)
21 //     void fun(int, double)
22 // fun(1.0, 2.0);
23 // error: call of overloaded 'fun(double, double)' is ambiguous
24 }
25
26 void fun(int n1, int n2, int n3) { // version 1
27     cout << "version 1" << endl;
28 }
29
30 void fun(double n1, int n2) { // version 2
31     cout << "version 2" << endl;
32 }
33
34 void fun(int n1, double n2) { // version 3
35     cout << "version 3" << endl;
36 }

```

Overloaded functions cannot be differentiated by the return-type (compilation error).

*Name Mangling

To differentiate between different versions of an overloaded function, many compilers (such as GNU GCC) adopt a *name mangling* or *name decoration* scheme for naming functions.

```

// Compile source into object code FunctionOverloading.o
> g++ -c FunctionOverloading.cpp

// List the symbol table
> nm FunctionOverloading.o
.....
000000b5 T __Z3fundi
000000ed T __Z3funid
00000089 T __Z3funiii
.....

```

Each of the function is identified via a prefix `__Z`, followed by an integer containing the number of characters of the function name (3 in this case for "fun"), followed by the parameter type list (where i for int and d for double). For example, di means a double followed by an int; id for an int followed by a double; iii for 3 ints.

You can choose to use the C's naming protocol by appending the keyword `extern "C"` to the function prototype. C does not support function overloading. Thus, it does not need name mangling. It simply append an underscore in front of the function name. For example,

```
extern "C" void fun(int, int, int, int); // Compiled as _fun
```

10.5 Functions and Arrays

You can also pass arrays into function. However, you also need to pass the size of the array into the function. This is because there is no way to tell the size of the array from the array argument inside the called function.

For example,

Example: Computing the Sum of an Array and Print Array's Contents

```
1  /* Function to compute the sum of an array (SumArray.cpp) */
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype
6  int sum(int array[], int size);    // Need to pass the array size too
7  void print(int array[], int size);
8
9  // Test Driver
10 int main() {
11     int a1[] = {8, 4, 5, 3, 2};
12     print(a1, 5);    // {8,4,5,3,2}
13     cout << "sum is " << sum(a1, 5) << endl;    // sum is 22
14 }
15
16 // Function definition
17 // Return the sum of the given array
18 int sum(int array[], int size) {
19     int sum = 0;
20     for (int i = 0; i < size; ++i) {
21         sum += array[i];
22     }
23     return sum;
24 }
25
26 // Print the contents of the given array
27 void print(int array[], int size) {
28     cout << "{";
29     for (int i = 0; i < size; ++i) {
30         cout << array[i];
31         if (i < size - 1) {
32             cout << ",";
33         }
34     }
35     cout << "}" << endl;
36 }
```


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

There are two ways that a parameter can be passed into a function: *pass by value* vs. *pass by reference*.

Pass-by-Value

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. In C/C++, fundamental types (such as `int` and `double`) are passed by value. That is, you cannot modify caller's value inside the function - there is no *side effect*.

Example (Fundamental Types are Passed by Value)

```
1  /* Fundamental types are passed by value into Function (TestPassByValue.cpp) */
2  #include <iostream>
3  using namespace std;
4
5  // Function prototypes
6  int inc(int number);
7
8  // Test Driver
9  int main() {
10     int n = 8;
11     cout << "Before calling function, n is " << n << endl; // 8
12     int result = inc(n);
13     cout << "After calling function, n is " << n << endl;   // 8
14     cout << "result is " << result << endl;                 // 9
15 }
16
17 // Function definitions
18 // Return number+1
19 int inc(int number) {
20     ++number; // Modify parameter, no effect to caller
21     return number;
22 }
```

Pass-by-Reference

On the other hand, 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 (e.g., sorting the elements 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.

Example (Array is passed by Reference): Increment Each Element of an Array

```

1  /* Function to increment each element of an array (IncrementArray.cpp) */
2  #include <iostream>
3  using namespace std;
4
5  // Function prototypes
6  void inc(int array[], int size);
7  void print(int array[], int size);
8
9  // Test Driver
10 int main() {
11     int a1[] = {8, 4, 5, 3, 2};
12
13     // Before increment
14     print(a1, 5);    // {8,4,5,3,2}
15     // Do increment
16     inc(a1, 5);      // Array is passed by reference (having side effect)
17     // After increment
18     print(a1, 5);    // {9,5,6,4,3}
19 }
20
21 // Function definitions
22
23 // Increment each element of the given array
24 void inc(int array[], int size) { // array[] is not const
25     for (int i = 0; i < size; ++i) {
26         array[i]++; // side-effect
27     }
28 }
29
30 // Print the contents of the given array
31 void print(int array[], int size) {
32     cout << "{";
33     for (int i = 0; i < size; ++i) {
34         cout << array[i];
35         if (i < size - 1) {
36             cout << ",";
37         }
38     }
39     cout << "}" << endl;
40 }

```

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 (i.e., side effect).

Why Arrays are Pass-by-Reference?

Array is designed to be passed by reference, instead of by value using a cloned copy. This is because passing huge array by value is inefficient - the huge array needs to be cloned.

10.7 const Function Parameters

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.

Use `const` whenever possible for passing references as it prevent you from inadvertently modifying the parameters and protects you against many programming errors.

Example: Search an Array using Linear Search

In a linear search, the search key is compared with each element of the array linearly. If there is a match, it returns the index of the array between `[0, size-1]`; otherwise, it returns -1 or the size of of the array (some implementations deal with only positive indexes). Linear search has complexity of $O(n)$.

```
1  /* Search an array for the given key using Linear Search (LinearSearch.cpp) */
2  #include <iostream>
3  using namespace std;
4
5  int linearSearch(const int a[], int size, int key);
6
7  int main() {
8      const int SIZE = 8;
9      int a1[SIZE] = {8, 4, 5, 3, 2, 9, 4, 1};
10
11      cout << linearSearch(a1, SIZE, 8) << endl; // 0
12      cout << linearSearch(a1, SIZE, 4) << endl; // 1
13      cout << linearSearch(a1, SIZE, 99) << endl; // 8 (not found)
14  }
15
16  // Search the array for the given key
17  // If found, return array index [0, size-1]; otherwise, return size
18  int linearSearch(const int a[], int size, int key) {
19      for (int i = 0; i < size; ++i) {
20          if (a[i] == key) return i;
21      }
22      return size;
23  }
```

Program Notes:

- [TODO]

Example: Sorting an Array using Bubble Sort

Wiki "[Bubble Sort](#)" for the detailed algorithm and illustration. In brief, we pass thru the list, compare two adjacent items and swap them if they are in the wrong order. Repeat the pass until no swaps are needed. For example,

```
{8,4,5,3,2,9,4,1}
PASS 1 ...
{8,4,5,3,2,9,4,1} => {4,8,5,3,2,9,4,1}
{4,8,5,3,2,9,4,1} => {4,5,8,3,2,9,4,1}
{4,5,8,3,2,9,4,1} => {4,5,3,8,2,9,4,1}
{4,5,3,8,2,9,4,1} => {4,5,3,2,8,9,4,1}
{4,5,3,2,8,9,4,1} => {4,5,3,2,8,4,9,1}
{4,5,3,2,8,4,9,1} => {4,5,3,2,8,4,1,9}
PASS 2 ...
{4,5,3,2,8,4,1,9} => {4,3,5,2,8,4,1,9}
{4,3,5,2,8,4,1,9} => {4,3,2,5,8,4,1,9}
{4,3,2,5,8,4,1,9} => {4,3,2,5,4,8,1,9}
{4,3,2,5,4,8,1,9} => {4,3,2,5,4,1,8,9}
PASS 3 ...
{4,3,2,5,4,1,8,9} => {3,4,2,5,4,1,8,9}
{3,4,2,5,4,1,8,9} => {3,2,4,5,4,1,8,9}
{3,2,4,5,4,1,8,9} => {3,2,4,4,5,1,8,9}
{3,2,4,4,5,1,8,9} => {3,2,4,4,1,5,8,9}
PASS 4 ...
{3,2,4,4,1,5,8,9} => {2,3,4,4,1,5,8,9}
{2,3,4,4,1,5,8,9} => {2,3,4,1,4,5,8,9}
PASS 5 ...
{2,3,4,1,4,5,8,9} => {2,3,1,4,4,5,8,9}
PASS 6 ...
{2,3,1,4,4,5,8,9} => {2,1,3,4,4,5,8,9}
PASS 7 ...
{2,1,3,4,4,5,8,9} => {1,2,3,4,4,5,8,9}
PASS 8 ...
{1,2,3,4,4,5,8,9}
```

Bubble sort is not efficient, with complexity of $O(n^2)$.

```
1  /* Sorting an array using Bubble Sort (BubbleSort.cpp) */
2  #include <iostream>
3  using namespace std;
4
5  void bubbleSort(int a[], int size);
6  void print(const int a[], int size);
7
8  int main() {
9      const int SIZE = 8;
10     int a[] = {8, 4, 5, 3, 2, 9, 4, 1};
11
```

```
12     print(a, SIZE);
13     cout << endl;
14     bubbleSort(a, SIZE);
15     print(a, SIZE);
16     cout << endl;
17 }
18
19 // Sort the given array of size
20 void bubbleSort(int a[], int size) {
21     bool done = false; // terminate if no more swap thru a pass
22     int pass = 0;       // pass number, for tracing
23     int temp;           // use for swapping
24
25     while (!done) {
26         cout << "PASS " << ++pass << " ..." << endl; // for tracing
27         done = true;
28         // Pass thru the list, compare adjacent items and swap
29         // them if they are in wrong order
30         for (int i = 0; i < size - 1; ++i) {
31             if (a[i] > a[i+1]) {
32                 print(a, size); // for tracing
33                 temp = a[i];
34                 a[i] = a[i+1];
35                 a[i+1] = temp;
36                 done = false; // swap detected, one more pass
37                 cout << "=> "; // for tracing
38                 print(a, size);
39                 cout << endl;
40             }
41         }
42     }
43 }
44
45 // Print the contents of the given array of size
46 void print(const int a[], int size) {
47     cout << "{";
48     for (int i = 0; i < size; ++i) {
49         cout << a[i];
50         if (i < size - 1) cout << ",";
51     }
52     cout << "} ";
53 }
```

Program Notes:

- [TODO]

Example: Sorting an Array using Insertion Sort

Wiki "[Insertion Sort](#)" for the algorithm and illustration. In brief, pass thru the list. For each element, compare with all previous elements and insert it at the correct position by shifting the other elements. For example,

```
{8,4,5,3,2,9,4,1}
{8} {4,5,3,2,9,4,1}
{4,8} {5,3,2,9,4,1}
{4,5,8} {3,2,9,4,1}
{3,4,5,8} {2,9,4,1}
{2,3,4,5,8} {9,4,1}
{2,3,4,5,8,9} {4,1}
{2,3,4,4,5,8,9} {1}
{1,2,3,4,4,5,8,9}
```

Insertion sort is also not efficient, with complexity of $O(n^2)$.

```
1  /* Sorting an array using Insertion Sort (InsertionSort.cpp) */
2  #include <iostream>
3  using namespace std;
4
5  void insertionSort(int a[], int size);
6  void print(const int a[], int iMin, int iMax);
7
8  int main() {
9      const int SIZE = 8;
10     int a[SIZE] = {8, 4, 5, 3, 2, 9, 4, 1};
11
12     print(a, 0, SIZE - 1);
13     cout << endl;
14     insertionSort(a, SIZE);
15     print(a, 0, SIZE - 1);
16     cout << endl;
17 }
18
19 // Sort the given array of size using insertion sort
20 void insertionSort(int a[], int size) {
21     int temp;    // for shifting elements
22     for (int i = 1; i < size; ++i) {
23         // for tracing
24         print(a, 0, i - 1);    // already sorted
25         print(a, i, size - 1); // to be sorted
26         cout << endl;
27
28         // For element at i, insert into proper position in [0, i-1]
29         // which is already sorted.
30         // Shift down the other elements
```

```

31     for (int prev = 0; prev < i; ++prev) {
32         if (a[i] < a[prev]) {
33             // insert a[i] at prev, shift the elements down
34             temp = a[i];
35             for (int shift = i; shift > prev; --shift) {
36                 a[shift] = a[shift-1];
37             }
38             a[prev] = temp;
39             break;
40         }
41     }
42 }
43 }
44
45 // Print the contents of the array in [iMin, iMax]
46 void print(const int a[], int iMin, int iMax) {
47     cout << "{";
48     for (int i = iMin; i <= iMax; ++i) {
49         cout << a[i];
50         if (i < iMax) cout << ",";
51     }
52     cout << "} ";
53 }

```

Program Notes:

- [TODO]

Example: Sorting an Array using Selection Sort

Wiki "[Selection Sort](#)" for the algorithm and illustration. In brief, Pass thru the list. Select the smallest element and swap with the head of the list. For example,

```

{8,4,5,3,2,9,4,1}
{} {8,4,5,3,2,9,4,1} => {} {1,4,5,3,2,9,4,8}
{1} {4,5,3,2,9,4,8} => {1} {2,5,3,4,9,4,8}
{1,2} {5,3,4,9,4,8} => {1,2} {3,5,4,9,4,8}
{1,2,3} {5,4,9,4,8} => {1,2,3} {4,5,9,4,8}
{1,2,3,4} {5,9,4,8} => {1,2,3,4} {4,9,5,8}
{1,2,3,4,4} {9,5,8} => {1,2,3,4,4} {5,9,8}
{1,2,3,4,4,5} {9,8} => {1,2,3,4,4,5} {8,9}
{1,2,3,4,4,5,8,9}

```

Selection sort is also not efficient, with complexity of $O(n^2)$.

```

1  /* Sorting an array using Selection Sort (SelectionSort.cpp) */
2  #include <iostream>
3  using namespace std;
4

```

```
5 void selectionSort(int a[], int size);
6 void print(const int a[], int iMin, int iMax);
7
8 int main() {
9     const int SIZE = 8;
10    int a[SIZE] = {8, 4, 5, 3, 2, 9, 4, 1};
11
12    print(a, 0, SIZE - 1);
13    cout << endl;
14    selectionSort(a, SIZE);
15    print(a, 0, SIZE - 1);
16    cout << endl;
17 }
18
19 // Sort the given array of size using selection sort
20 void selectionSort(int a[], int size) {
21     int temp; // for swapping
22     for (int i = 0; i < size - 1; ++i) {
23         // for tracing
24         print(a, 0, i - 1);
25         print(a, i, size - 1);
26
27         // [0, i-1] already sort
28         // Search for the smallest element in [i, size-1]
29         // and swap with a[i]
30         int minIndex = i; // assume first element is the smallest
31         for (int j = i + 1; j < size; ++j) {
32             if (a[j] < a[minIndex]) minIndex = j;
33         }
34         if (minIndex != i) { // swap
35             temp = a[i];
36             a[i] = a[minIndex];
37             a[minIndex] = temp;
38         }
39
40         // for tracing
41         cout << "=> ";
42         print(a, 0, i - 1);
43         print(a, i, size - 1);
44         cout << endl;
45     }
46 }
47
48 // Print the contents of the array in [iMin, iMax]
49 void print(const int a[], int iMin, int iMax) {
50     cout << "{";
51     for (int i = iMin; i <= iMax; ++i) {
```



```
52     cout << a[i];
53     if (i < iMax) cout << ",";
54 }
55 cout << "} ";
56 }
```

Program Notes:

- [TODO]

"const" Fundamental-Type Function Parameters?

You could also use `const` for fundamental-type function parameters (such as `int`, `double`) to prevent the parameters from being modified inside the function. However, as fundamental-type parameters are passed by value (with a cloned copy), there will never be side effect on the caller. We typically do not use the `const` keyword for fundamental types. In other words, `const` is used to indicate that there shall NOT be side-effect.

10.8 Pass-by-Reference via "Reference" Parameters

As mentioned, parameters of fundamental types (such as `int`, `double`) are passed-by-value. That is, a clone copy is used inside the function. Change to the cloned copy inside the function has no side-effect to the caller's copy.

Nonetheless, you can pass a fundamental type parameter by reference via the so-called *reference parameter* denoted by `&`. For example,

```
1  /* Test Pass-by-reference for fundamental-type parameter
2     via reference declaration (TestPassByReference.cpp) */
3  #include <iostream>
4  using namespace std;
5
6  int squareByValue (int number);          // Pass-by-value
7  void squareByReference (int & number);    // Pass-by-reference
8
9  int main() {
10     int n1 = 8;
11     cout << "Before call, value is " << n1 << endl; // 8
12     cout << squareByValue(n1) << endl; // no side-effect
13     cout << "After call, value is " << n1 << endl; // 8
14
15     int n2 = 9;
16     cout << "Before call, value is " << n2 << endl; // 9
17     squareByReference(n2); // side-effect
18     cout << "After call, value is " << n2 << endl; // 81
19 }
20
21 // Pass parameter by value - no side effect
22 int squareByValue (int number) {
23     return number * number;
24 }
```

```
25
26 // Pass parameter by reference by declaring as reference (&)
27 // - with side effect to the caller
28 void squareByReference (int & number) {
29     number = number * number;
30 }
```

In function `squareByReference()`, we declare the parameter `number` is passed by reference by declaring its type as `int &` (reference of `int`). In this way, the caller's copy is used inside the function (instead of a cloned copy in pass-by-value). Changes inside the function has side-effect.

Pass-by-reference is NOT commonly used for fundamental types (such as `int`, `double`) - the above example is purely meant for academic illustration. But it is used extensively for compound types (such as arrays and objects) to avoid cloning huge data for better performance. We shall revisit pass-by-reference in Object-Oriented Programming (OOP) chapters.

10.9 Mathematical Functions (Header `<cmath>`)

C++ provides many common-used Mathematical functions in library `<cmath>` (ported over from C's `"math.h"`). The signatures of some of these functions are:

`sin(x)`, `cos(x)`, `tan(x)`, `asin(x)`, `acos(x)`, `atan(x)`:
Take argument-type and return-type of `float`, `double`, `long double`.

`atan2(y, x)`:
Return arc-tan of `y/x`. Better than `atan(x)` for handling 90 degree.

`sinh(x)`, `cosh(x)`, `tanh(x)`:
hyper-trigonometric functions.

`pow(x, y)`, `sqrt(x)`:
power and square root.

`ceil(x)`, `floor(x)`:
returns the ceiling and floor integer of floating point number.

`fabs(x)`, `fmod(x, y)`:
floating-point absolute and modulus.

`exp(x)`, `log(x)`, `log10(x)`:
exponent and logarithm functions.

10.10 Generating Random Numbers

The `cstdlib` header (ported from C's `stdlib.h`) provides a function `rand()`, which generates a pseudo-random integral number between 0 and `RAND_MAX` (inclusive). `RAND_MAX` is a constant defined in `cstdlib` (typically the maximum value of 16-/32-bit signed integer, such as 32767). You can generate a random number between `[0, n)` via `rand() % n`.

`rand()` generates the same sequence of pseudo-random numbers on different invocations. The `cstdlib` also provides a `srand()` function to *seed* or initialize the random number generator. We typically seed it with the current time obtained via `time(0)` function (in `<ctime>` header), which returns the number of seconds since January 1st, 1970.

Example 1: Test `rand()` and `srand(time(0))`

```
1  /* Test Random Number Generation (TestRand.cpp) */
2  #include <iostream>
3  #include <cstdlib> // for rand(), srand()
4  #include <ctime>   // for time()
5  using namespace std;
6
7  int main() {
8      // rand() generate a random number in [0, RAND_MAX]
9      cout << "RAND_MAX is " << RAND_MAX << endl; // 32767
10
11     // Generate 10 pseudo-random numbers between 0 and 99
12     // without seeding the generator.
13     // You will get the same sequence, every time you run this program
14     for (int i = 0; i < 10; ++i) {
15         cout << rand() % 100 << " "; // need <cstdlib> header
16     }
17     cout << endl;
18
19     // Seed the random number generator with current time
20     srand(time(0)); // need <cstdlib> and <ctime> header
21     // Generate 10 pseudo-random numbers
22     // You will get different sequence on different run,
23     // because the current time is different
24     for (int i = 0; i < 10; ++i) {
25         cout << rand() % 100 << " "; // need <cstdlib> header
26     }
27     cout << endl;
28 }
```

Example 2: Test `rand()`'s Distribution

We shall test the `rand()`'s distribution by repeatedly throwing a 6-sided die and count the occurrences.

```
1  /* Test rand() distribution by throwing a die repeatedly (TestRandomDie.cpp) */
2  #include <iostream>
3  #include <iomanip>
4  #include <cstdlib> // for rand(), srand()
5  #include <ctime>   // for time()
6  using namespace std;
7
8  const int TOTAL_COUNT = 2000000; // Close to INT_MAX
```

```

9  const int NUM_FACES = 6;
10 int frequencies[6] = {0}; // frequencies of 0 to 5, init to zero
11
12 int main() {
13     srand(time(0)); // seed random number generator with current time
14     // Throw the die and count the frequencies
15     for (int i = 0; i < TOTAL_COUNT; ++i) {
16         ++frequencies[rand() % 6];
17     }
18
19     // Print statistics
20     cout << fixed << setprecision(2);
21     for (int i = 0; i < NUM_FACES; i++) {
22         cout << i+1 << ": " << frequencies[i]
23             << " (" << 100.0 * frequencies[i] / TOTAL_COUNT << "%)" << endl;
24     }
25 }

```

```

1: 333109 (16.66%)
2: 333113 (16.66%)
3: 333181 (16.66%)
4: 333562 (16.68%)
5: 333601 (16.68%)
6: 333434 (16.67%)

```

As seen from the output, `rand()` is fairly *uniformly-distributed* over `[0, RAND_MAX]`.

10.11 Exercises

[TODO]

11. 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:

1. Create a `ifstream` for input, or `ofstream` for output.
2. Connect the stream to an input or output file via `open(filename)`.
3. Perform formatted output via stream insertion operator `<<`, or input via stream extraction operator `>>`, similar to `cout <<` and `cin >>`.
4. Close the file and free the stream.

11.1 Example: File IO

```

1  /* Test File I/O (TestFileIO.cpp)

```

```
2  Read all the integers from an input file and
3  write the average to an output file      */
4  #include <iostream>
5  #include <fstream>    // file stream
6  #include <cstdlib>
7  using namespace std;
8
9  int main() {
10     ifstream fin;    // Input stream
11     ofstream fout;   // Output stream
12
13     // Try opening the input file
14     fin.open("in.txt");
15     if (!fin.is_open()) {
16         cerr << "error: open input file failed" << endl;
17         abort();    // Abnormally terminate the program (in <stdlib>)
18     }
19
20     int sum = 0, number, count = 0;
21     while (!(fin.eof())) {
22         // Use >> to read
23         fin >> number;
24         sum += number;
25         ++count;
26     }
27     double average = double(sum) / count;
28     cout << "Count = " << count << " average = " << average << endl;
29     fin.close();
30
31     // Try opening the output file
32     fout.open("out.txt");
33     if (!fout.is_open()) {
34         cerr << "error: open output file failed" << endl;
35         abort();
36     }
37     // Write the average to the output file using <<
38     fout << average;
39     fout.close();
40     return 0;
41 }
```

Input File: in.txt

12 15 35 26 68

Output File: out.txt

31.20

Program Notes:

- Once the file is opened, you can use >> and << for input and output, similar to `cin >>` and `cout <<`. (Advanced note: `ifstream` is a subclass of `istream`, where `cin` belongs. `ofstream` is a subclass of `ostream`, where `cout` belongs.)
- Similarly, IO manipulators, such as `fixed`, `setprecision()` and `setw()`, work on the file streams.

11.2 Exercises

[TODO]

12. Namespace

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`.

To place an entity under a namespace, use keyword `namespace` as follow:

```
// 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
```

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

Namespace Example

```
1  #include <iostream>
2
3  namespace a {    // contains variables
4      int i1 = 8;
5      int i2 = 9;
6  }
7
8  namespace b {    // contains function
```

```

9      int max(int n1, int n2) {
10         return (n1 > n2) ? n1 : n2;
11     }
12 }
13
14 int main() {
15     std::cout << a::i1 << std::endl;           // 8
16     std::cout << b::max(a::i1, a::i2) << std::endl; // 9
17 }

```

Namespaces are opened. In other words, you can add more names into an existing namespace using additional namespace definition.

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:

1. 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".

2. 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`.

3. 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.

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

```
namespace shorthand = namespace-name;
```

You can now refer to your class as `shorthand::entityName`.

As mentioned, all the standard C++ library components are packaged inside a namespace called `std`. They include `cin`, `cout`, `endl`, `setw()`, `setprecision()`, and `string`. Hence, we always included a "`using namespace std;`" directive after the "`#include <iostream>`" and "`#include <string>`". You could also use the

qualified name such as `std::cin`, `std::cout`, `std::endl`, `std::string`, instead of the `using` directive. You could also use selective using such as `"using std::cin;"`, `"using std::cout;"`, `"using std::endl;"` and `"using std::string;"`.

Compare the following two code samples:

```
#include <iostream>
using namespace std; // std namespace is available to ALL

int main() { ..... }
```

```
#include <iostream>

int main() {
    using namespace std; // std namespace is available to main() only
    .....
}
```

Note: you could use the C-style header `"iostream.h"` which does not use namespace to replace the first two lines in the first sample.

Global Namespace

In C++, an entity (variable, function, or class) belongs to the *global namespace* (identified by `::` with no namespace name), if it is not enclosed within a namespace declaration. For example, `main()` can be identified as `::main()`.

Link to "C++ References and Resources"

Latest version tested: Cygwin/MinGW GCC g++ 4.6.2

Last modified: June, 2013

Feedback, comments, corrections, and errata can be sent to [Chua Hock-Chuan \(ehchua@ntu.edu.sg\)](mailto:ehchua@ntu.edu.sg) | [HOME](#)