

Ranim Tom  
C++ Language  
Introduction to Programming with C

## Contents

Visual Studio IDE for C++.....	5
C++  Debugging in Visual Studio.....	5
C++ Compiler Working .....	8
C++Linker Working .....	8
Visual Studio Tips: .....	9
Making the Background Dark: .....	9
For Spacing, Indentation and Settings:.....	9
Set up for Visual Studio.....	9
C++ Input/Output.....	14
C++ Explanation Code 2: Input from a User .....	15
Chapter 4: Mathematical Functions, Characters, and Strings .....	16
4.10 Formatting Console Output .....	16
Setw (w for width) .....	16
Chapter 6: Functions .....	17
Multiple Functions Syntax in One Source File.....	17
Code Version 1: .....	17
Code Version 2: .....	18
6.10: Inline Function.....	20
6.12 Passing Arguments by Reference .....	22
Chapter 7: Single Dimensional Arrays and C-Strings .....	25
❖ <b>Note: Constant Size</b> .....	25
❖ <b>Array Initializers</b> .....	26
❖    Caution.....	26
❖ <b>Note</b> Implicit Size .....	26
❖ <b>Note</b> Partial Initialization.....	26
7.7 Returning Arrays from Functions .....	27
Characters and Strings .....	29
Character Data Type and Operations.....	29
ASCII Code .....	29
char increment and decrement .....	29
Reading a Character from the Keyboard .....	29
Escape Sequences for Special Characters.....	30

Whitespace Character.....	30
Casting between char and Numeric Types .....	30
Numeric Operators on Characters.....	31
Comparing and Testing Characters.....	31
Character Functions .....	32
The string Type.....	33
String Index and Subscript Operator .....	34
subscript operator .....	34
String index range .....	34
Concatenating Strings .....	34
Comparing Strings.....	35
Reading Strings .....	36
Constructing a String.....	38
Appending to a String .....	38
Assigning a String .....	39
Functions at, clear, erase, and empty .....	40
Functions length, size, capacity, and c_str() .....	40
Comparing Strings.....	41
Obtaining Substrings.....	42
Searching in a String.....	42
Inserting and Replacing Strings.....	43
String Operators.....	44
Converting Numbers to Strings.....	45
Splitting Strings .....	45
Chapter 10: Object Oriented Design.....	46
Listing 9.10 CircleWithPrivateDataFields.cpp .....	48
Listing 9.11 TestCircleWithPrivateDataFields.cpp .....	49
Scope of Variables for Class Case .....	50
Points on Classes and Objects .....	51
10.5 Instance and Static Members.....	51
Chapter 11: Pointers and Dynamic Memory Allocation .....	52
11.2: Pointer Basics .....	52
11.3 Defining Synonymous Types Using the typedef Keyword.....	54

11.5 Arrays and Pointers .....	55
Pointers Arithmetic: .....	55
Pointer Based C-Strings: .....	57
11.6 Passing Pointer Arguments in a Function Call .....	58
Array Parameter or Pointer Parameter .....	60
11.9: Dynamic Persistent Memory Allocation .....	62
WrongReverse.Cpp .....	62
Fixing the Problem: the new Operator .....	62
Create Array Dynamically .....	63
Memory Structure: Heap and Stack .....	64
Delete Operator .....	67
Dangling Pointers .....	67
Memory Leak .....	68
11.10 Creating and Accessing Dynamic Objects .....	68
11.11: The this Pointer .....	69
11.12: Pointers in 2D Array .....	70
11.13: Pointers in Higher Dimensional Arrays .....	73
Chapter 15: Inheritance and Polymorphism .....	75
15.9 Abstract Classes and Pure Virtual Functions .....	77

## Visual Studio IDE for C++

When making a new project:

File, new project ; then choose empty project.

After that, select with left click on “Add” new Cpp file.

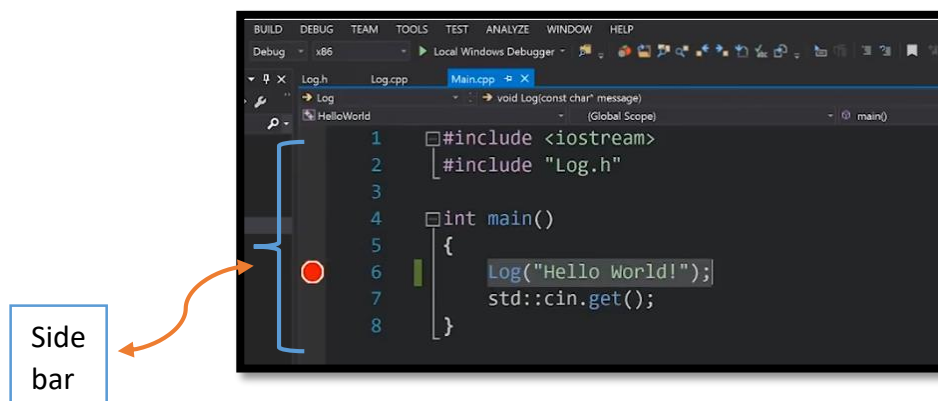
When you are doing OOP implementation, you can directly select in the source file, add new item, and choose a class file, and it will automatically open its associated header file.

### C++ Debugging in Visual Studio

Debugging is composed from (de)and (bug), so remove the bugs. The debugging process composed from making break points, and read memory. We will use them both, putting break points in order to read memory.

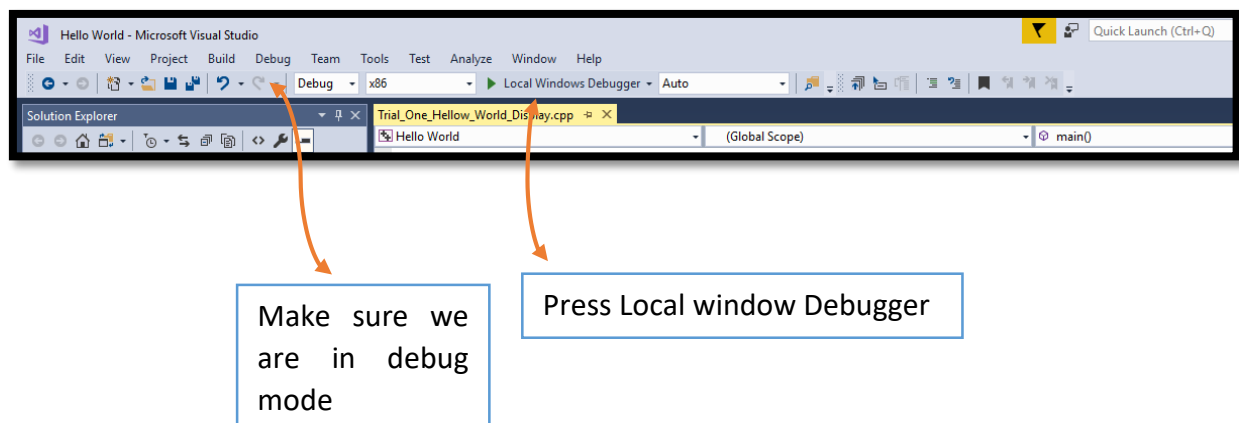
A break point is a point in our program at which the debugger break, and by break we mean pause.

We can set break point by putting the cursor at a random line of code, and then press f9. Or we can click on the side bar as shown in the figure below.

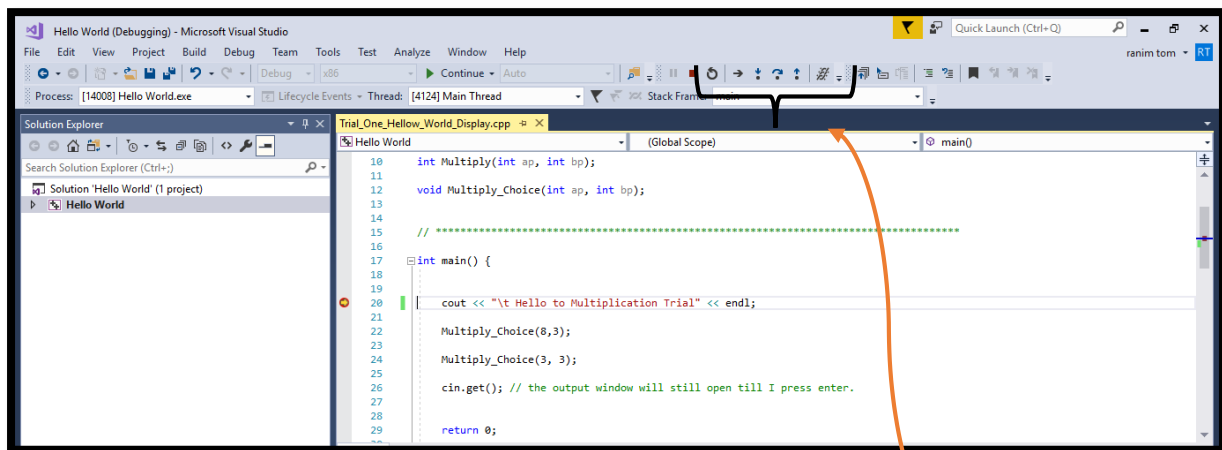


After putting a break point, we have to check if we are in debug mode, and not in release mode.

After that, we go to “Local Windows Debugger” and press it, and it will give a “Continue”.



After we press the “Local Windows Debugger”, we will get the following screen.



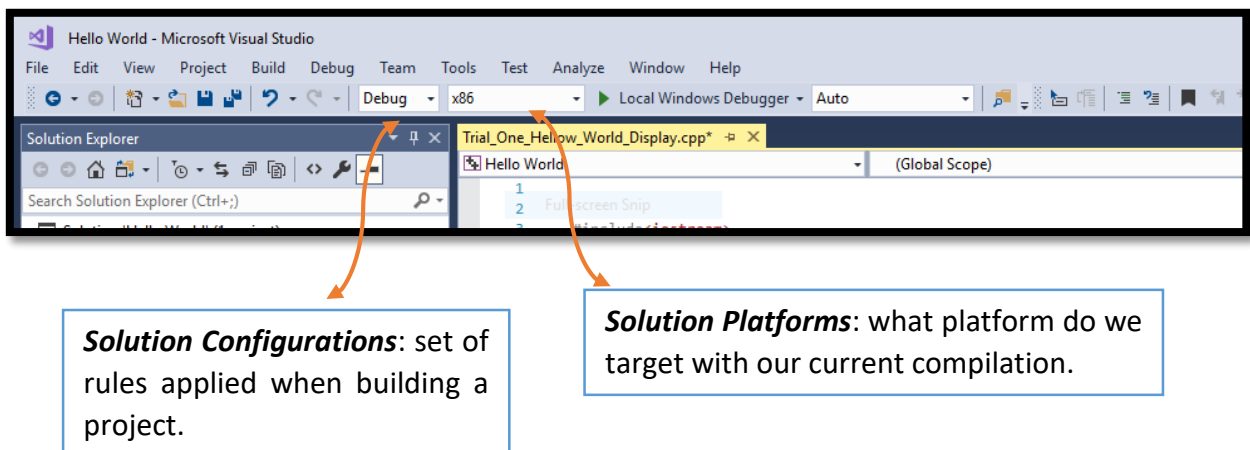
We will use these 3 arrows.

StepOver: Executes one step in the program, at the current level. If the program calls a function, that function is executed all at once, rather than by stepping through it. Use F10 to do this

StepInto: Stops at the first line of the first function called on this line. Use F11 here

StepOut: Runs the program until the current function returns. Use Shift-F11.

How do we get the executable file (.exe)? In any project we have 2 important things:



Note: x86 is the same as win32.

**Header files do not get compiled**, only when they are included into the cpp file, at this moment they will be compiled.

So in the program, we have the cpp file, plus the included header files. Each file get compiled individually, and the output of this compilation by the compiler is called an object file with extension (.o or .obj).

Once we have our different compiled object files, we need something in order to stitch them together, because for now they are separated. This is the job of the linker, which takes all the object files and stitch them together into one (.exe) file.

Now to **compile a file individually**, we can press the **ctr+F7** (we have a customize button for it.).

When we **compile a file individually, no linking happens**. So when we compile a file individually, we will obtain an object file as said before. To see the object file of the compiled source file, right click at the project name, press open folder in file explorer, then enter the debug folder, and you will see the object file associated with the compiled source file. **So for every cpp file we have, we will have an object file.**

### C++ Compiler Working

So when we write our code, which is a normal text file, we need some workers which transform this text file into an executable file (binary code). To do this, we have 2 steps as it has been seen earlier: compiling and linking. Now we will talk only about compiling.

Compiler take the source file (.cpp) and transform it into an object file. The compiler does several things in order to produce these object files.

1<sup>st</sup>, it will do the preprocessing of all the included files. Another preprocessing operations are (#define , pragma, #if end if, ...) in which we will discuss them later.

The most well know preprocessor operation is the #include, which means that the compiler will search the include file and include it (literally) in the source file.

After doing this operation, the compiler will make the transformation of the source file into object file, which are binary files.

### C++Linker Working

Linking is the process that comes after compiling. Linking will link all the multiple source file that composed our project, which may contains functions we have implement, some classes... and link all the files together into to make an executable code. Even if we don't have multiple files for our project, like we have written all of our functions together in a single source file, the program also need to be linked, because the C++ real time library needs to find where is the main function is (the entry point of the program).

Usually when we compile a source file, no linking happens. But when we build our project, compiling and linking will happen.



## Visual Studio Tips:

### Making the Background Dark:

Click Tool tab, then option.

In environment (left side), select General, then go to “color them” (right side) and choose dark.

#### ➤ Another method:

Click Tools tab, then “Extensions and Update”, then select “Online”, and type “color themes” in the search field (upper right).

### For Spacing, Indentation and Settings:

Click Tool tab, then option, Formatting and you will find your options

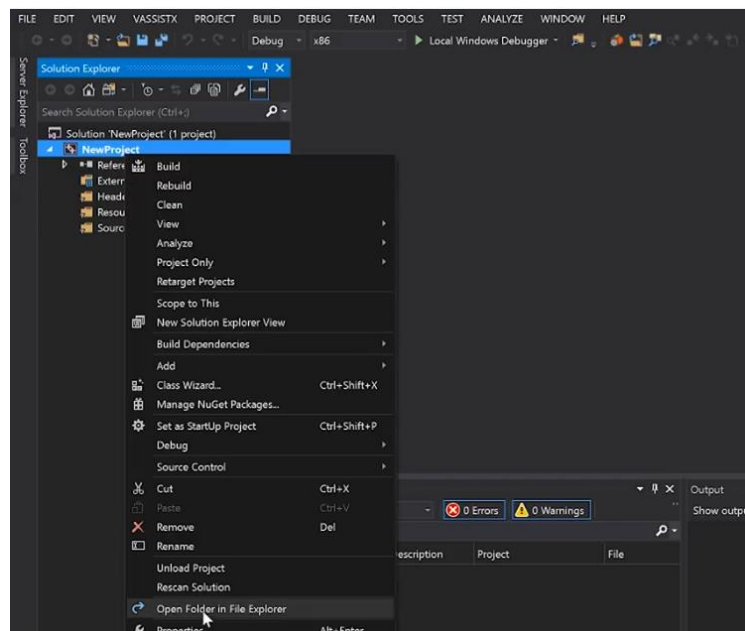
#### ➤ Note for formatting style:

When we choose a specific way of formatting the code style, it will take effect on the new lines of code we write, but the old ones stay the same. To ***apply the new format style to the old code lines***, press the **ctr K D**, and it will update the format.

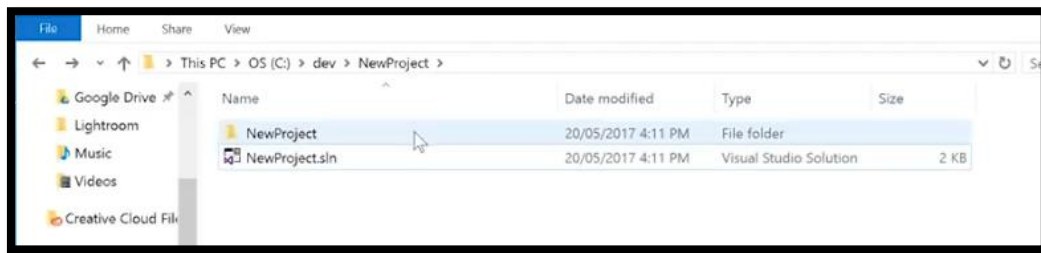
### Set up for Visual Studio

1<sup>st</sup> we will create an empty project called “New Project”.

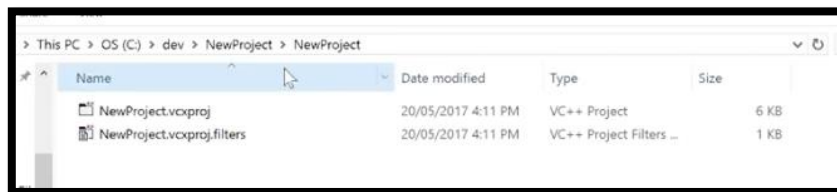
If we hit right click on the “New Project”, in the solution explorer tab, and press “Open Folder in File Explorer”, it will open the path of the project and its structure.



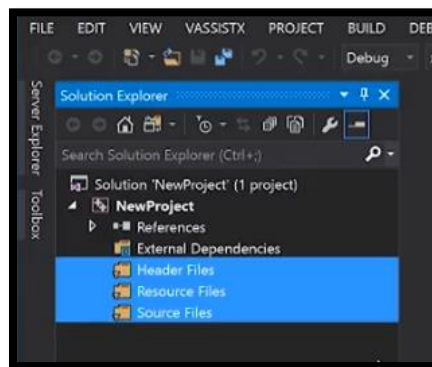
In the picture in the next page, are the directory visual studio create it for us.



Visual studio create a solution file, with the same name of the project. If we click to our project, we will see the files in the picture below.



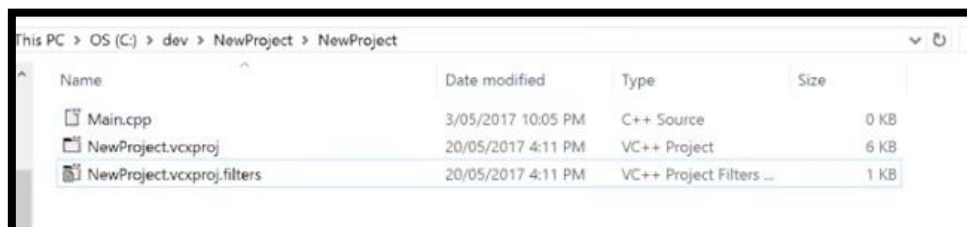
Now back to the solution explorer.



These highlighted folders, are not actually folders, they are called filters, which organize our source codes. So if we add a new filter (notice that is not named new folder), no folder will be created in the project structure we have.

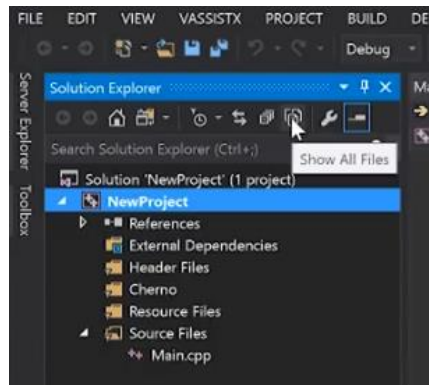
These highlighted filters are contained in the filter file, in 2<sup>nd</sup> figure of the structure we have.

Now if we go to the "source code", and we add new item, the main.cpp (or any other source file) will be putted next to our project file.

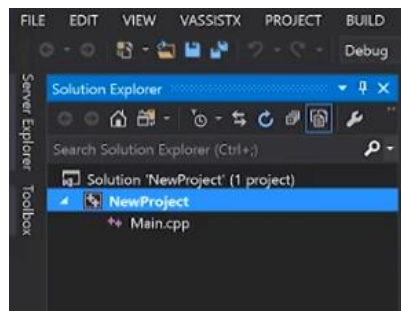


So we will create a separate folder in which it contains all our source files (code, header...)

Now to do this, we will click an option called “Show all Files” in the solution explorer.

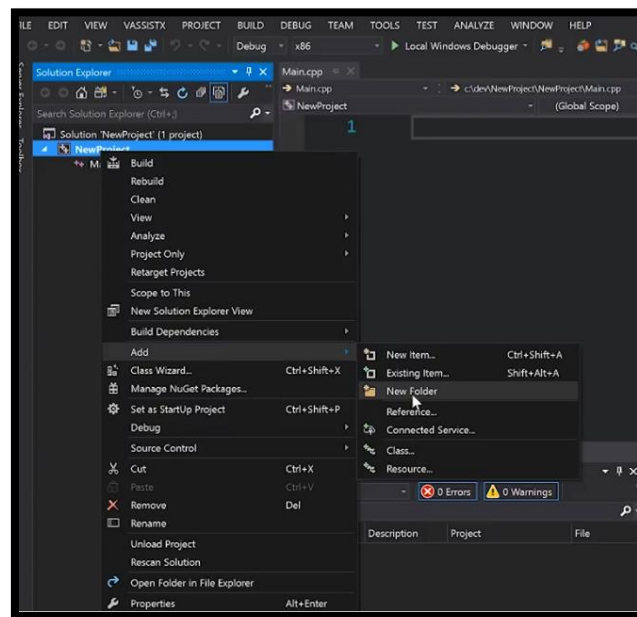


The result of this clicking is this.

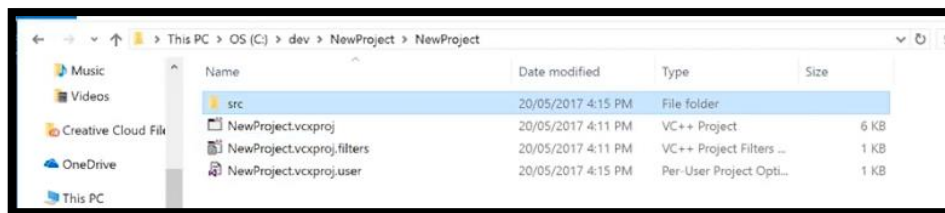


These are the actual files that are in our disk.

Now we can right click on the project name, and add a new folder. We will name it src.

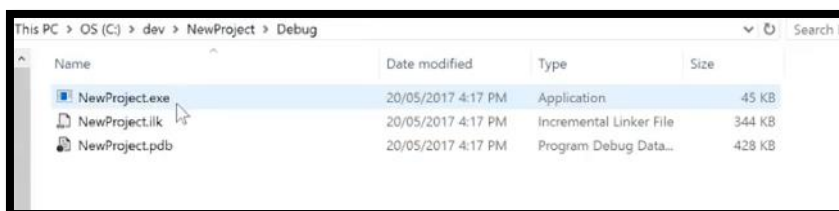
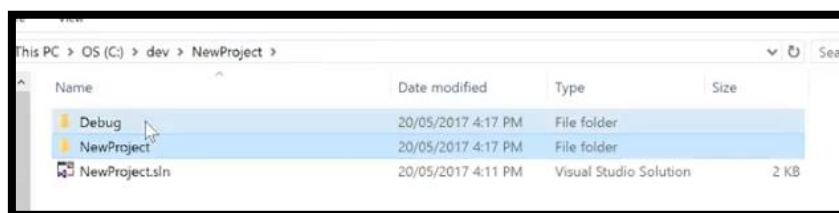


And then we will drag our main.cpp to this src folder.



The main.cpp will be inside the src folder.

Now if we want our .exe file, it will be in the Debug folder, in the same directory of the the solution file, and not the Debug folder of the same directory of the project file (where also the src folder exist)



Now we will see how to organize our Debug folders (both the intermediate and the one containing the .exe file).

## C++ Input/Output

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

**Preprocessor Statement.** When the compiler receives the source file, it begins preprocessing all the preprocessor statements we have, which have the (#) symbol. Now the #include always includes *files*, in this case the <iostream> file, allows the program to perform input and output.

the output window will still open till I press enter.

This **bit left shift operator** is an **overloaded operator**, which pushes the sentence to the output console. It is called **stream insertion operator**.

- **Line 2:** std stands for standard library, that means we are going to use all the built-in libraries in C++. So the 1<sup>st</sup> 2 lines are inclusive lines, 1) include a file and 2) include a library.
- **Line 5:** a) cout is called an output stream object, it is used when we want to print characters on the computer screen.
  - b) << is called stream insertion operator, it takes all the things to the right of it and type it on the screen.
  - c) endl means end line, and tell the computer to go to the next line.

### C++ Explanation Code 2: Input from a User

```
1) #include <iostream>
2) using namespace std;
3) int main()
4) {
5) int a,b,c ;
6) cout << "Enter a number: " << endl;
7) cin >> a ;
8) cout << "Enter a number: " << endl;
9) cin >> b ;
10) c = a*b ;
11) cout << " The product is " << c << endl;
12) return 0;
13) }
```

- **Line 7:** using cin which is called input stream object, used to get input from user. Also notice that we use cin>>, the opposite of cout<<. >> is called stream extraction operator.

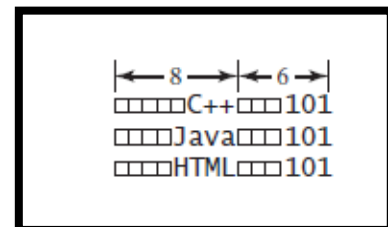
## Chapter 4: Mathematical Functions, Characters, and Strings

### 4.10 Formatting Console Output

TABLE 4.8 Frequently Used Stream Manipulators	
Operator	Description
<code>setprecision(n)</code>	sets the precision of a floating-point number
<code>fixed</code>	displays floating-point numbers in fixed-point notation
<code>showpoint</code>	causes a floating-point number to be displayed with a decimal point with trailing zeros even if it has no fractional part
<code>setw(width)</code>	specifies the width of a print field
<code>left</code>	justifies the output to the left
<code>right</code>	justifies the output to the right

#### Setw (w for width)

```
cout << setw(8) << "C++" << setw(6) << 101 << endl;  
cout << setw(8) << "Java" << setw(6) << 101 << endl;  
cout << setw(8) << "HTML" << setw(6) << 101 << endl;
```

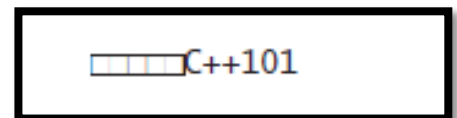


- The output is right-justified within the specified columns. In line 1, `setw(8)` specifies that "C++" is displayed in eight columns. So, there are five spaces before C++. `setw(6)` specifies that 101 is displayed in six columns. So, there are three spaces before 101.

- `setw` manipulator affects only the next output.

For example, `cout << setw(8) << "C++" << 101 << endl;`

The `setw(8)` manipulator affects only the next output "C++", not 101.



- If an item requires more spaces than the specified width, the width is automatically increased. For example, the following code

```
cout << setw(8) << "Programming" << "#" << setw(2) << 101;
```

displays

Programming#101

The specified width for `Programming` is 8, which is smaller than its actual size 11. The width is automatically increased to 11. The specified width for `101` is 2, which is smaller than its actual size 3. The width is automatically increased to 3.



## Chapter 6: Functions

### Multiple Functions Syntax in One Source File

#### Code Version 1:

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

void Log(const char* message) {
    cout << message << endl;
} // End Function Log

int Multiply(int a, int b) {
    Log("Multiply");
    return a * b;
} // End Function Multiply

//
*****
*****

int main() {
    cout << "\t Hello to Multiplication Trial" << endl;
    cout << Multiply(5,8)<<endl ;
    cin.get(); // the output window will still open till I press
enter.

    return 0;
} // End main function
```

Code Version 2:

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

// Functions Declaration

void Log(const char* message);

int Multiply(int ap, int bp);

void Multiply_Choice(int ap, int bp);

//
*****
*****

int main() {

    cout << "\t Hello to Multiplication Trial" << endl;

    Multiply_Choice(8,3);

    Multiply_Choice(3, 3);

    cin.get(); // the output window will still open till I press enter.

    return 0;

} // End main function

// *****

// Functions Definition

void Log(const char* message) {

    cout << "\t" << message ;

} // End Function Log
```

```
int Multiply(int a1, int b1) {  
  
    Log("Multiply"); // Calling Log Function  
  
    return a1 * b1;  
  
} // End Function Multiply  
  
void Multiply_Choice(int ap, int bp) {  
  
    int result;  
  
    result = Multiply( ap, bp); // Calling Multiply Function  
    cout << "\t" << result;  
  
    cout << endl;  
  
} // End Function Multiply_Choice
```

## 6.10: Inline Function

C++ provides inline functions to avoid function calls.

Inline functions are not called; rather, the compiler copies the function code in line at the point of each invocation.

To specify an inline function, precede the function declaration with the `inline` keyword, as shown in Listing 6.9

### Listing 6.9 InlineDemo.cpp

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

inline void f(int month, int year)
{
    cout << "month is " << month << endl;
    cout << "year is " << year << endl;
}

int main()
{
    int month = 10, year = 2008;
    f(month, year); // Invoke inline function
    f(9, 2010); // Invoke inline function
    return 0;
} // End main
```

```
month is 10
year is 2008
month is 9
year is 2010
```

As far as programming is concerned, inline functions are the same as regular functions, except they are preceded with the `inline` keyword.

However, behind the scenes, the C++ compiler expands the inline function call by copying the inline function code. So, Listing 6.9 is essentially equivalent to Listing 6.10.

### LISTING 6.10 InlineExpandedDemo.cpp

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int month = 10, year = 2008;
7     cout << "month is " << month << endl;
8     cout << "year is " << year << endl;
9     cout << "month is " << 9 << endl;
10    cout << "year is " << 2010 << endl;
11
12    return 0;
13 }
```

← Inline function expanded

```
month is 10
year is 2008
month is 9
year is 2010
```

### Note

Inline functions are desirable for short functions but not for long ones that are called in multiple places in a program, because making multiple copies will dramatically increase the executable code size.

For this reason, C++ allows the compilers to ignore the `inline` keyword if the function is too long.

So, the `inline` keyword is merely a request; it is up to the compiler to decide whether to honor or ignore it.

## 6.12 Passing Arguments by Reference

*Parameters can be passed by reference, which makes the formal parameter an alias of the actual argument. Thus, changes made to the parameters inside the function also made to the arguments.*

A reference variable is an alias for another variable. To declare a reference variable, place the ampersand (&) in front of the variable or after the data type for the variable. For example, the following code declares a reference variable `r` that refers to variable `count`.

`int &r = count;` or equivalently, `int& r = count;`

### ➤ Note

The following notations for declaring reference variables are equivalent:

`dataType &refVar;`

`dataType & refVar;`

`dataType& refVar;`

The last notation is more intuitive and it clearly states that the variable `refVar` is of the type `dataType&`. For this reason, the last notation will be used in this book.

### Listing 6.15 TestReferenceVariable.cpp

```
#include <iostream>

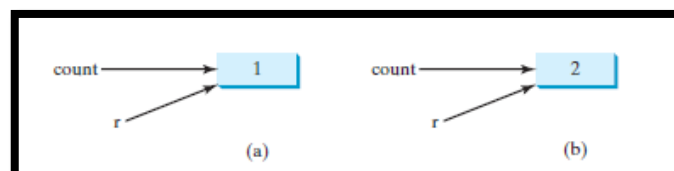
using namespace std;

int main()
{
    int count = 1;
    int& r = count; // declare reference variable

    cout << "count is " << count << endl;
    cout << "r is " << r << endl;
    r++; // use reference variable
    cout << "count is " << count << endl;
    cout << "r is " << r << endl;
    count = 10;
    cout << "count is " << count << endl;
    cout << "r is " << r << endl;
    return 0;
} // End main function
```

```
count is 1
r is 1
count is 2
r is 2
count is 10
r is 10
```

`r` and `count` reference the same value . You can access and modify the value stored in the original variable through the reference variable.



### Listing 6.16 IncrementWithPassByReference.cpp

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

void increment(int& n) ;

int main()
{
    int x = 1;
    cout << "Before the call, x is " << x << endl;
    increment(x);
    cout << "After the call, x is " << x << endl;

    return 0;
}

void increment(int& n) ;
{
    n++;
    cout << "n inside the function is " << n << endl;
}
```

Pass-by-value and pass-by-reference are two ways of passing arguments to the parameters of a function.

Pass-by-value passes the value to an independent variable and ***pass-by-reference shares the same variable***. Semantically ***pass-by-reference*** can be described as ***pass-by-sharing***.



## Chapter 7: Single Dimensional Arrays and C-Strings

### Listing 7.1 AnalyzeNumbers.cpp

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

int main()
{
    const int NUMBER_OF_ELEMENTS = 100;
    double numbers[NUMBER_OF_ELEMENTS];
    double sum = 0;

    for (int i = 0; i < NUMBER_OF_ELEMENTS; i++) .
    {
        cout << "Enter a new number: ";
        cin >> numbers[i];
        sum += numbers[i];
    } // End for

    double average = sum / NUMBER_OF_ELEMENTS;
    int count = 0; // The number of elements above average

    for (int i = 0; i < NUMBER_OF_ELEMENTS; i++)
    if (numbers[i] > average)
        count++; // End for

    cout << "Average is " << average << endl;
    cout << "Number of elements above the average " << count << endl;
    return 0;
} // End main
```

Declaring Arrays `double myList[10];`

#### ❖ Note: Constant Size

The **array size** used to declare an array must be a **constant expression in standard C++**. For example, the following code is illegal:

```
int size = 4;
double myList[size]; // Wrong
```

But it is all right if **SIZE** is a constant as follows:

```
const int SIZE = 4;
double myList[SIZE]; // Correct
```

### ❖ Array Initializers

C++ has a shorthand notation, known as the array initializer, which declares and initializes an array in a single statement.

For example: `double myList[4] = {1.9, 2.9, 3.4, 3.5};`

This statement declares and initializes the array `myList` with four elements, making it equivalent to the statements shown below:

```
double myList[4];
```

```
myList[0] = 1.9;
```

```
myList[1] = 2.9;
```

```
myList[2] = 3.4;
```

```
myList[3] = 3.5;
```

### ❖ Caution

Using an array initializer, you must declare and initialize the array in one statement. Splitting it would cause a syntax error. Thus, the next statement is wrong:

```
double myList[4];
```

```
myList = {1.9, 2.9, 3.4, 3.5};
```

### ❖ Note Implicit Size

C++ allows you to omit the array size when declaring and creating an array using an initializer. For example, the following declaration is fine:

```
double myList[] = {1.9, 2.9, 3.4, 3.5};
```

The compiler automatically figures out how many elements are in the array.

### ❖ Note Partial Initialization

C++ allows you to initialize a part of the array. For example, the following statement assigns values `1.9`, `2.9` to the first two elements of the array. The other two elements will be set to zero.

```
double myList[4] = {1.9, 2.9};
```

Note that if an array is declared, but not initialized, all its elements will contain

“

garbage,” like all other local variables.

## 7.7 Returning Arrays from Functions

*To return an array from a function, pass it as a parameter in a function.*

You can declare a function to return a primitive type value or an object. For example,

```
// Return the sum of the elements in the list
```

```
int sum(const int list[], int size)
```

**Can you return an array from a function** using a similar syntax? For example, you may attempt to declare a function that returns a new array that is a reversal of an array, as follows:

```
// Return the reversal of list
```

```
int[] reverse(const int list[], int size). This is not allowed in C++.
```

However, you can circumvent this restriction by passing two array arguments in the function:

```
// newList is the reversal of list
```

```
void reverse(const int list[], int newList[], int size)
```

`const int list[]` tells the compiler that to keep the list array unchanged, so any attempt to change it will be a compiler error.

### Listing 7.7 ReverseArray.cpp

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

// newList is the reversal of list

void reverse(const int list[], int newList[], int size)
{
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        newList[j] = list[i]; // reverse to new list
    } // End for
} // End reverse

void printArray(const int list[], int size)
{
    for (int i = 0; i < size; i++)
        cout << list[i] << " ";
}

int main()
{
    const int SIZE = 6;
    int list[] = {1, 2, 3, 4, 5, 6}; // declare original array
    int newList[SIZE]; // declare new array
    reverse(list, newList, SIZE);
    cout << "The original array: ";
    printArray(list, SIZE); // print original array
    cout << endl;
    cout << "The reversed array: ";
    printArray(newList, SIZE); // print reversed array
    cout << endl;
    return 0; } // End main
```

## Characters and Strings

### Character Data Type and Operations

```
char letter = 'A';
```

```
char numChar = '4';
```

A character literal is enclosed in single quotation marks

On most systems, the **size of the char type** is 1 byte.

### ASCII Code

*ASCII (American Standard Code for Information Interchange)*, an 8-bit encoding scheme for representing all uppercase and lowercase letters, digits, punctuation marks, and control characters.

**TABLE 4.4** ASCII Code for Commonly Used Characters

<i>Characters</i>	<i>ASCII Code</i>
'0' to '9'	48 to 57
'A' to 'Z'	65 to 90
'a' to 'z'	97 to 122

### char increment and decrement

The **increment and decrement operators** can also be used on char variables to get the **next or preceding ASCII code character**. For example, the following statements display character b.

```
char ch = 'a';
```

```
cout << ++ch;
```

### Reading a Character from the Keyboard

```
cout << "Enter a character: ";
```

```
char ch;
```

```
cin >> ch; // Read a character
```

```
cout << "The character read is " << ch << endl;
```

## Escape Sequences for Special Characters

Escape Sequence	Name	ASCII Code	Escape Sequence	Name	ASCII Code
<code>\b</code>	Backspace	8	<code>\r</code>	Carriage Return	13
<code>\t</code>	Tab	9	<code>\\</code>	Backslash	92
<code>\n</code>	Linefeed	10	<code>\"</code>	Double Quote	34
<code>\f</code>	Formfeed	12			

This special notation, called an **escape sequence**, consists of a backslash (\) followed by a character or a combination of digits.

### Examples:

```
cout << "He said \"Programming is fun\" << endl;
```

The output is: He said "Programming is fun".

Note that the symbols \ and " together represent one character.

The backslash \ is called an **escape character**. It is a special character. To display this character, you have to use an escape sequence \\.

For example, the following code: `cout << "\\t is a tab character" << endl;`

Output: `\t` is a tab character.

### Whitespace Character

The characters ' ', '\t', '\f', '\r', and '\n' are known as the **whitespace characters**.

### Casting between char and Numeric Types

A char can be cast into any numeric type, and vice versa. When an integer is cast into a char, only its lower 8 bits of data are used; the other part is ignored.

### Examples:

```
char c = 0xFF41; // The lower 8 bits hex code 41 is assigned to c
```

```
cout << c; // variable c is character A
```

When a floating-point value is cast into a char, the floating-point value is first cast into an int, which is then cast into a char.

```
char c = 65.25; // 65 is assigned to variable c
```

```
cout << c; // variable c is character A
```

When a char is cast into a numeric type, the character's ASCII is cast into the specified numeric type. For example:

```
int i = 'A'; // The ASCII code of character A is assigned to i  
cout << i; // variable i is 65
```

### Numeric Operators on Characters

The char type is treated as if it were an integer of the byte size. All numeric operators can be applied to char operands. A char operand is automatically cast into a number if the other operand is a number or a character.

```
// The ASCII code for '2' is 50 and for '3' is 51  
int i = '2' + '3';  
cout << "i is " << i << endl; // i is now 101  
int j = 2 + 'a'; // The ASCII code for 'a' is 97  
cout << "j is " << j << endl;  
cout << j << " is the ASCII code for character " <<  
static_cast<char>(j) << endl;
```

#### Output:

i is 101

j is 99

99 is the ASCII code for  
character c

Note that the `static_cast<char>(value)` operator explicitly casts a numeric value into a character.

### Comparing and Testing Characters

```
if (ch >= 'A' && ch <= 'Z')  
    cout << ch << " is an uppercase letter" << endl;  
else if (ch >= 'a' && ch <= 'z')  
    cout << ch << " is a lowercase letter" << endl;  
else if (ch >= '0' && ch <= '9')  
    cout << ch << " is a numeric character" << endl;
```

## Character Functions

C++ provides several functions for testing a character and for converting a character in the `<cctype>` header file, as shown in Table 4.6.

**TABLE 4.6** Character Functions

Function	Description
<code>isdigit(ch)</code>	Returns true if the specified character is a digit.
<code>isalpha(ch)</code>	Returns true if the specified character is a letter.
<code>isalnum(ch)</code>	Returns true if the specified character is a letter or digit.

(continued)

**TABLE 4.6** (continued)

Function	Description
<code>islower(ch)</code>	Returns true if the specified character is a lowercase letter.
<code>isupper(ch)</code>	Returns true if the specified character is an uppercase letter.
<code>isspace(ch)</code>	Returns true if the specified character is a whitespace character.
<code>tolower(ch)</code>	Returns the lowercase of the specified character.
<code>toupper(ch)</code>	Returns the uppercase of the specified character.

### CharacterFunctions.cpp

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

int main()
{
    cout << "Enter a character: ";
    char ch;
    cin >> ch;

    cout << "You entered " << ch << endl;

    if (islower(ch))
    {
        cout << "It is a lowercase letter " << endl;
        cout << "Its equivalent uppercase letter is " <<
            static_cast<char>(toupper(ch)) << endl;
    }
    else if (isupper(ch))
    {
        cout << "It is an uppercase letter " << endl;
        cout << "Its equivalent lowercase letter is " <<
            static_cast<char>(tolower(ch)) << endl;
    }
    else if (isdigit(ch))
    {
        cout << "It is a digit character " << endl;
    }

    return 0;
}
```



## The string Type

```
string message = "Programming is fun";
```

The **string type** is not a primitive type. It is known as an **object type**. Here **message** represents a **string object** with **contents Programming is fun**.

Objects are defined using classes. string is a predefined class in the <string> header file. An object is also known as an instance of a class.

**TABLE 4.7** Simple Functions for string Objects

Function	Description
length()	Returns the number of characters in this string.
size()	Same as length().
at(index)	Returns the character at the specified index from this string.

The **functions in the string class** can only be **invoked from a specific string instance**. For this reason, these functions are called **instance functions**.

### Example:

```
string message = "ABCD";  
cout << message.length() << endl;  
cout << message.at(0) << endl;  
string s = "Bottom";  
cout << s.length() << endl;  
cout << s.at(1) << endl;
```

- message.length() returns 4
- message.at(0) returns character A.
- s.length() returns 6
- s.at(1) returns character o.

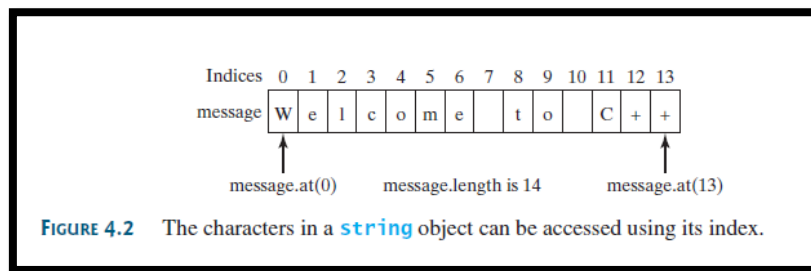
By default, a string is initialized to an **empty string**, i.e., a string containing no characters.

An empty string literal can be written as "".

```
string s;  
string s = "";
```

2 statements have the same effect

## String Index and Subscript Operator



### subscript operator

For convenience, C++ provides the subscript operator for accessing the character at a specified index in a string using the syntax `stringName[index]`. You can use this syntax to retrieve and modify the character in a string.

```
string s = "ABCD";  
s[0] = 'P';  
cout << s[0] << endl;
```

Sets a new character P at index 0  
using `s[0] = 'P'` and displays it

### String index range

Attempting to access characters in a string `s` out of bounds is a common programming error. To avoid it, make sure that you do not use an index beyond `s.length() - 1`.

For example, `s.at(s.length())` or `s[s.length()]` would cause an error.

### Concatenating Strings

C++ provides the `+` operator for concatenating two strings. The statement shown below, for example, concatenates strings `s1` and `s2` into `s3`:

```
string s3 = s1 + s2;
```

The augmented `+=` operator can also be used for string concatenation. For example, the following code appends the string "and programming is fun" with the string "Welcome to C++" in `message`.

```
message += " and programming is fun";
```

Therefore, the new message is "Welcome to C++ and programming is fun".

You can also **concatenate a character with a string**. For example,

```
string s = "ABC";
```

```
s += 'D';
```

Therefore, the new `s` is "ABCD".

### Caution

It is *illegal to concatenate two string literals*. For example, the following code is incorrect:

```
string cites = "London" + "Paris";
```

### Correction

However, the following code is correct, because it first concatenates string `s` with "London" and then the new string is concatenated with "Paris".

```
string s = "New York";  
string cites = s + "London" + "Paris";
```

### Comparing Strings

You can use the relational operators `==`, `!=`, `<`, `<=`, `>`, `>=` to compare two strings. This is done by comparing their corresponding characters one by one from left to right.

```
string s1 = "ABC";  
string s2 = "ABE";  
  
cout << (s1 == s2) << endl; // Displays 0 (means false)  
cout << (s1 != s2) << endl; // Displays 1 (means true)  
cout << (s1 > s2) << endl; // Displays 0 (means false)  
cout << (s1 >= s2) << endl; // Displays 0 (means false)  
cout << (s1 < s2) << endl; // Displays 1 (means true)  
cout << (s1 <= s2) << endl; // Displays 1 (means true)
```

Consider evaluating `s1 > s2`. The first two characters (A versus A) from `s1` and `s2` are compared. Because they are equal, the second two characters (B versus B) are compared.

Because they are also equal, the third two characters (C versus E) are compared. Since the character C is less than E, the comparison returns 0.

## Reading Strings

```
1 string city;  
2 cout << "Enter a city: ";  
3 cin >> city; // Read to string city  
4 cout << "You entered " << city << endl;
```

Line 3 reads a string to city. This approach to reading a string is simple, but there is a problem.

The input ends with a whitespace character. If you want to enter New York, you have to use an alternative approach.

C++ provides the `getline` function in the `string` header file, which reads a string from the keyboard using the following syntax:

```
getline(cin, s, delimiterCharacter)
```

The function stops reading characters when the delimiter character is encountered. The delimiter is read but not stored into the string.

The third argument `delimiterCharacter` has a default value (`'\n'`).

The following code uses the `getline` function to read a string.

```
1 string city;  
2 cout << "Enter a city: ";  
3 getline(cin, city, '\n'); // Same as getline(cin, city)  
4 cout << "You entered " << city << endl;
```

Since the default value for the third argument in the `getline` function is `'\n'`, line 3 can be replaced by `getline(cin, city); // Read a string`

## OrderTwoCities.cpp

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

Don't forget to include the string class

```
int main()
{
    string city1, city2;
    cout << "Enter the first city: ";
    getline(cin, city1);
    cout << "Enter the second city: ";
    getline(cin, city2);

    cout << "The cities in alphabetical order are ";
    if (city1 < city2)
        cout << city1 << " " << city2 << endl;
    else
        cout << city2 << " " << city1 << endl;

    return 0;
}
```

If we use `cin >> city1`, you cannot enter a string with spaces for `city1`. Since a city name may contain multiple words separated by spaces, the program uses the `getline` function to read a string



```
Enter the first city: New York
Enter the second city: Boston
The cities in alphabetical order are Boston New York
```

In C++ there are two ways to process strings. One way is to treat them as arrays of characters ending with the null terminator ('\0'), as discussed in Section 7.11, "C-Strings." These are known as *C-strings*. The null terminator indicates the end of the string, which is important for the C-string functions to work.

The other way is to process strings using the string class.

You can use the C-string functions to manipulate and process strings, but the string class is easier. Processing C-strings requires the programmer to know how characters are stored in the array.

The string class hides the low-level storage from the programmer. The programmer is freed from implementation details.

### Constructing a String

You created a string using a syntax like this:

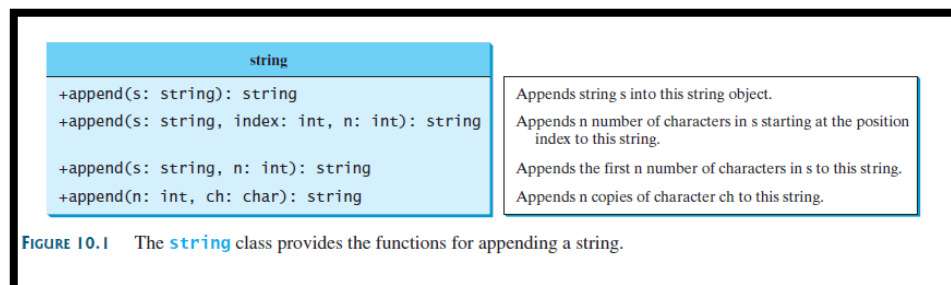
```
string s = "Welcome to C++";
```

This statement is not efficient because it takes two steps. It first creates a string object using a string literal and then copies the object to s.

A **better way to create a string** is to **use the string constructor** like this:

```
string s("Welcome to C++");
```

### Appending to a String

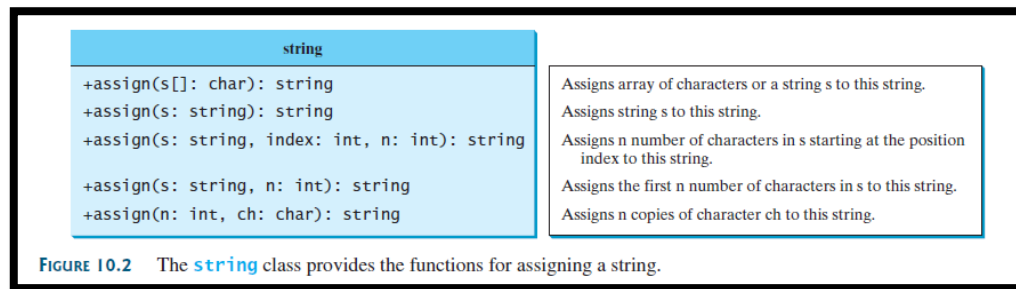


For example:

```
string s1("Welcome");  
  
s1.append(" to C++"); // Appends " to C++" to s1  
  
cout << s1 << endl; // s1 now becomes Welcome to C++  
  
string s2("Welcome");  
  
s2.append(" to C and C++", 0, 5); // Appends " to C" to s2  
  
cout << s2 << endl; // s2 now becomes Welcome to C  
  
string s3("Welcome");
```

```
s3.append(" to C and C++", 5); // Appends " to C" to s3
cout << s3 << endl; // s3 now becomes Welcome to C
string s4("Welcome");
s4.append(4, 'G'); // Appends "GGGG" to s4
cout << s4 << endl; // s4 now becomes WelcomeGGGG
```

### Assigning a String



For example:

```
string s1("Welcome");
s1.assign("Dallas"); // Assigns "Dallas" to s1
cout << s1 << endl; // s1 now becomes Dallas
string s2("Welcome");
s2.assign("Dallas, Texas", 0, 5); // Assigns "Dalla" to s2
cout << s2 << endl; // s2 now becomes Dalla
string s3("Welcome");
s3.assign("Dallas, Texas", 5); // Assigns "Dalla" to s3
cout << s3 << endl; // s3 now becomes Dalla
string s4("Welcome");
s4.assign(4, 'G'); // Assigns "GGGG" to s4
cout << s4 << endl; // s4 now becomes GGGG
```

### Functions at, clear, erase, and empty

You can use the `at(index)` function to retrieve a character at a specified index, `clear()` to clear the string, `erase(index, n)` to delete part of the string, and `empty()` to test whether a string is empty .

string	
+at(index: int): char	Returns the character at the position index from this string.
+clear(): void	Removes all characters in this string.
+erase(index: int, n: int): string	Removes n characters from this string starting at position index.
+empty(): bool	Returns true if this string is empty.

**FIGURE 10.3** The `string` class provides the functions for retrieving a character, clearing and erasing a string, and checking whether a string is empty.

For example:

```
string s1("Welcome");  
  
cout << s1.at(3) << endl; // s1.at(3) returns c  
  
cout << s1.erase(2, 3) << endl; // s1 is now Weme  
  
s1.clear(); // s1 is now empty  
  
cout << s1.empty() << endl; // s1.empty returns 1 (means true)
```

### Functions length, size, capacity, and c\_str()

You can use the functions `length()`, `size()`, and `capacity()` to obtain a string's length, size, and capacity and `c_str()` to return a C-string, as shown in Figure 10.4.

The functions `length()` and `size()` are aliases.

The functions `c_str()` and `data()` are the same in the new C++11. The `capacity()` function returns the internal buffer size which is always greater than or equal to the actual string size.

string	
+length(): int	Returns the number of characters in this string.
+size(): int	Same as length().
+capacity(): int	Returns the size of the storage allocated for this string.
+c_str(): char[]	Returns a C-string for this string.
+data(): char[]	Same as c_str().

**FIGURE 10.4** The `string` class provides the functions for getting the length, capacity, and C-string of the string.



For example, see the following code:

```
1 string s1("Welcome");
2 cout << s1.length() << endl; // Length is 7
3 cout << s1.size() << endl; // Size is 7
4 cout << s1.capacity() << endl; // Capacity is 15
5
6 s1.erase(1, 2); // erase two characters
7 cout << s1.length() << endl; // Length is now 5
8 cout << s1.size() << endl; // Size is now 5
9 cout << s1.capacity() << endl; // Capacity is still 15
```

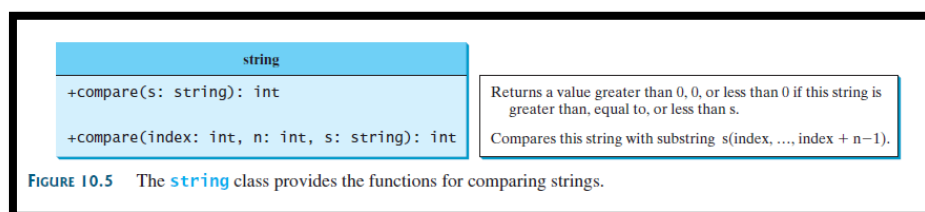
### **Note**

The *capacity* is set to 15 when string *s1* is created in line 1. After two characters are erased in line 6, the capacity is still 15, but the length and size become 5.

### Comparing Strings

Often, in a program, you need to compare the contents of two strings. You can use the compare function.

This function returns an int value greater than 0, 0, or less than 0 if this string is greater than, equal to, or less than the other string, as shown in Figure 10.5.

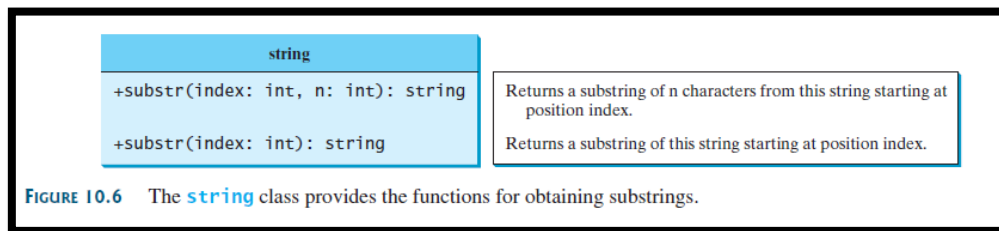


For example:

```
string s1("Welcome");
string s2("Welcomg");
cout << s1.compare(s2) << endl; // Returns -1
cout << s2.compare(s1) << endl; // Returns 1
cout << s1.compare("Welcome") << endl; // Returns 0
```

## Obtaining Substrings

You can obtain a single character from a string using the `at` function. You can also obtain a substring from a string using the `substr` function, as shown in Figure 10.6.



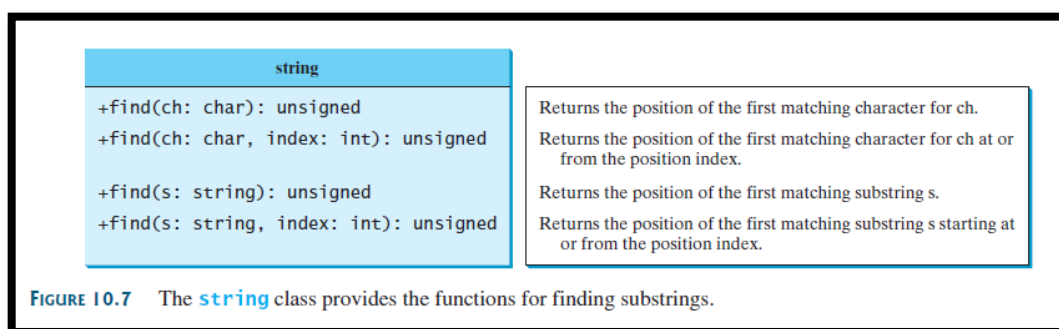
For example:

```
string s1("Welcome");  
cout << s1.substr(0, 1) << endl; // Returns W  
cout << s1.substr(3) << endl; // Returns come  
cout << s1.substr(3, 3) << endl; // Returns com
```

## Searching in a String

You can use the `find` function to search for a substring or a character in a string, as shown in Figure 10.7.

The function returns `string::npos` (not a position) if no match is found. `Npos` is a constant defined in the `string` class.

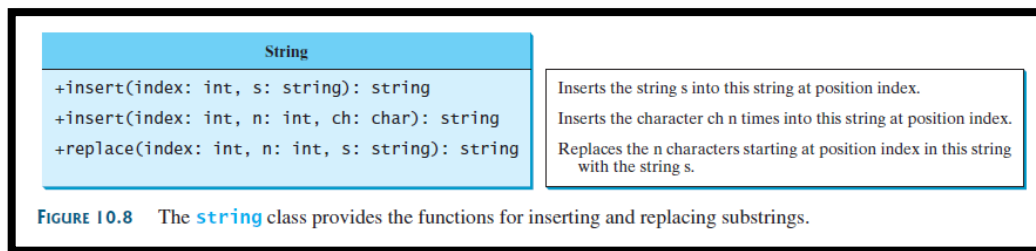


For example:

```
string s1("Welcome to HTML");  
cout << s1.find("co") << endl; // Returns 3  
cout << s1.find("co", 6) << endl; // Returns string::npos  
cout << s1.find('o') << endl; // Returns 4  
cout << s1.find('o', 6) << endl; // Returns 9
```

## Inserting and Replacing Strings

You can use the insert and replace functions to insert a substring and replace a substring in a string, as shown in Figure 10.8.



Here are examples of using the insert and replace functions:

```
string s1("Welcome to HTML");  
s1.insert(11, "C++ and ");  
cout << s1 << endl; // s1 becomes Welcome to C++ and HTML  
  
string s2("AA");  
s2.insert(1, 4, 'B');  
cout << s2 << endl; // s2 becomes to ABBBBBA  
  
string s3("Welcome to HTML");  
s3.replace(11, 4, "C++");  
cout << s3 << endl; // s3 becomes Welcome to C++
```

### Note

- 1) A string object invokes the append, assign, erase, replace, and insert functions to change the contents of the string object.

These functions also return the new string. For example, in the following code, s1 invokes the insert function to insert "C++ and " into s1, and the new string is returned and assigned to s2.

```
string s1("Welcome to HTML");  
string s2 = s1.insert(11, "C++ and ");  
cout << s1 << endl; // s1 becomes Welcome to C++ and HTML  
cout << s2 << endl; // s2 becomes Welcome to C++ and HTML
```

- 2) On most compilers, the capacity is automatically increased to accommodate more characters for the functions append, assign, insert, and replace. If the capacity is fixed and is too small, the function will copy as many characters as possible.

## String Operators

C++ supports operators to simplify string operations. Table 10.1 lists the string operators.

TABLE 10.1 String Operators	
Operator	Description
<code>[]</code>	Accesses characters using the array subscript operator.
<code>=</code>	Copies the contents of one string to the other.
<code>+</code>	Concatenates two strings into a new string.
<code>+=</code>	Appends the contents of one string to the other.
<code>&lt;&lt;</code>	Inserts a string to a stream
<code>&gt;&gt;</code>	Extracts characters from a stream to a string delimited by a whitespace or the null terminator character.
<code>==, !=, &lt;, &gt;, &lt;=, &gt;=</code>	Six relational operators for comparing strings.

Here are the examples to use these operators:

```
string s1 = "ABC"; // The = operator
string s2 = s1; // The = operator
for (int i = s2.size() - 1; i >= 0; i--)
cout << s2[i]; // The [] operator
string s3 = s1 + "DEFG"; // The + operator
cout << s3 << endl; // s3 becomes ABCDEFG
s1 += "ABC";
cout << s1 << endl; // s1 becomes ABCABC
s1 = "ABC";
s2 = "ABE";
cout << (s1 == s2) << endl; // Displays 0 (means false)
cout << (s1 != s2) << endl; // Displays 1 (means true)
cout << (s1 > s2) << endl; // Displays 0 (means false)
cout << (s1 >= s2) << endl; // Displays 0 (means false)
cout << (s1 < s2) << endl; // Displays 1 (means true)
cout << (s1 <= s2) << endl; // Displays 1 (means true)
```

## Converting Numbers to Strings

Section 7.11.6, “Conversion between Strings and Numbers,” introduced how to convert a string to an integer and a floating-point number using the functions `atoi` and `atof`.

You can also use the `itoa` function to convert an integer to a string. Sometimes you need to convert a floating-point number to a string. You can write a function to perform the conversion.

However, a simple approach is to use the `stringstream` class in the `<sstream>` header. `Stringstream` provides an interface to manipulate strings as if they were input/output streams.

One application of `stringstream` is for converting numbers to strings. Here is an example:

```
1 stringstream ss;
2 ss << 3.1415; //number to stringstream
3 string s = ss.str(); //stringstream to string
```

## Splitting Strings

Often you need to extract the words from a string. Assume that the words are separated by whitespaces. You can use the `stringstream` class discussed in the preceding section to accomplish this task. Listing 10.1 gives an example that extracts the words from a string and displays the words in separate lines.

### **Listing 10.1** ExtractWords.cpp

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

```
int main()
{
    string text("Programming is fun");
    stringstream ss(text);

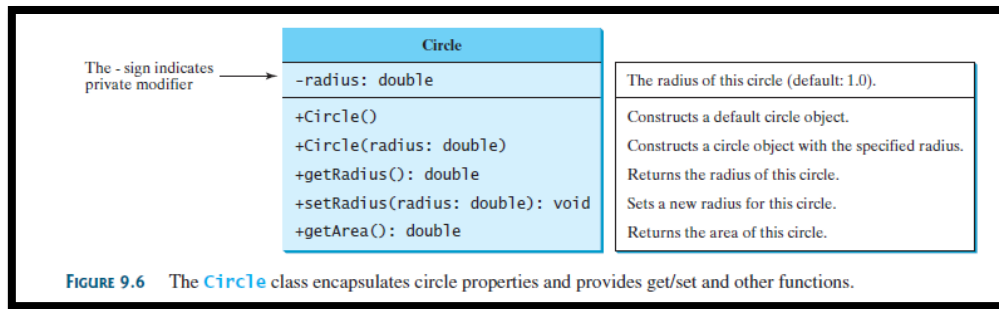
    cout << "The words in the text are " << endl;
    string word;
    while (!ss.eof())
    {
        ss >> word;
        cout << word << endl;
    }

    return 0;
}
```

Creates a stringstream object for the text string (line 9) and this object can be used just like an input stream for reading data from the console.

ss sends data from the string stream to a string object word

## Chapter 10: Object Oriented Design



### Listing 9.8 CircleWithInclusionGuard.h

```
#ifndef CIRCLE_H
#define CIRCLE_H

class Circle
{
public:
// Construct a default circle object
Circle();

// Construct a circle object
Circle(double);

// Return the area of this circle
double getArea();

double getRadius();
void setRadius(double);

private :
double radius ;
};
#endif
```

The data field radius is referred to as an *instance member variable* or simply *instance variable*, because it is dependent on a specific instance.

**Constructors are for initializing data fields.** The data field radius does not have an initial value, so it must be initialized in the constructor.

Note that a variable (local or global) can be declared and initialized in one statement, but as a class member, a data field cannot be initialized when it is declared. For example, it would be wrong to replace

`double radius = 5; // Wrong for data field declaration`

Don't forget the semicolon

The picture above is the header file (.h) for the class circle. ***This header file represents the class definition.***

Then we will have the implementation file of this class (.cpp)

The class definition and implementation may be in two separate files. Both files should have the same name but different extension names. The class definition file has an extension name .h (h means header) and the class implementation file an extension name .cpp.

In C++, when you program for example a list of functions, the compiler need to see the declaration then the implementation of the function.

For a cleaner code, we could put the ***implemented functions*** in other cpp file, and let the declaration alone.

However, if we want to use these function into another main file, so we need to copy and paste each time the declaration file into our project. To avoid this, we can use the header file methods introduced above, in which in the header file we put all our declarations, and then include this file into our main cpp file. So instead of having so many declarations in our main program before the main, we will have only one simple line (# include "Arithmetics Methods" for example).

### Listing 9.10 CircleWithPrivateDataFields.cpp

```
#include "CircleWithPrivateDataFields.h"
```

```
// Construct a default circle object
```

```
Circle::Circle()
```

```
{
```

```
radius = 1;
```

```
}
```

```
// Construct a circle object
```

```
Circle::Circle(double newRadius)
```

```
{
```

```
radius = newRadius;
```

```
}
```

```
// Return the area of this circle
```

```
double Circle::getArea()
```

```
{
```

```
return radius * radius * 3.14159;
```

```
}
```

```
// Return the radius of this circle
```

```
double Circle::getRadius()
```

```
{
```

```
return radius;
```

```
}
```

```
// Set a new radius
```

```
void Circle::setRadius(double newRadius)
```

```
{
```

```
radius = (newRadius >= 0) ? newRadius : 0;
```

```
}
```

The (::) is called **binary scope resolution operator**.

Here, Circle:: **preceding each constructor and function** in the Circle class tells the compiler that these constructors and functions are defined in the Circle class

**Constructors** are a special kind of **function**, with three peculiarities:

- Constructors must have the **same name as the class itself**.
- Constructors **do not have a return type**—not even void.
- Constructors are **invoked when an object is created**.

Constructors play the role of initializing objects.

Like regular functions, constructors can be overloaded (i.e., multiple constructors with the same name but different signatures), making it easy to construct objects with different sets of data values.

These functions are the only ways to read and modify radius, you have total control over how the radius property is accessed. If you have to change the functions' implementation, you need not change the client programs. This makes the class easy to maintain.

Now the program which use this class is called the **client class**.



[Listing 9.11 TestCircleWithPrivateDataFields.cpp](#)

```
#include <iostream>

#include "CircleWithPrivateDataFields.h"

using namespace std;

int main()
{
    Circle circle1;
    Circle circle2(5.0);

    cout << "The area of the circle of radius "
    << circle1.getRadius() << " is " << circle1.getArea() << endl;
    cout << "The area of the circle of radius "
    << circle2.getRadius() << " is " << circle2.getArea() << endl;
    // Modify circle radius
    circle2.setRadius(100);
    cout << "The area of the circle of radius "
    << circle2.getRadius() << " is " << circle2.getArea() << endl;
    return 0;
}
```

➤ A note about class implementation:

*Making **data fields private** protects data and makes the class easy to maintain.*

The data fields `radius` in the `Circle` class in Listing 9.1 can be modified directly (e.g., `circle1.radius = 5`). This is not a good practice—for two reasons:

- First, data may be tampered with.
- Second, it makes the class difficult to maintain and vulnerable to bugs. Suppose you want to modify the `Circle` class to ensure that the radius is nonnegative after other programs have already used the class. You have to change not only the `Circle` class, but also the programs that use the `Circle` class. This is because the clients may have modified the radius directly (e.g., `myCircle.radius = -5`). To prevent direct modifications of properties, you should declare the data field private, using the private keyword. This is known as **data field encapsulation**.

## Scope of Variables for Class Case

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

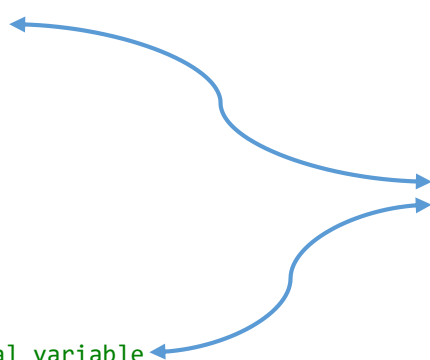
class Foo
{
public:
    int x; // Data field
    int y; // Data field

    Foo()
    {
        x = 10;
        y = 10;
    }

    void p()
    {
        int x = 20; // Local variable
        cout << "x is " << x << endl;
        cout << "y is " << y << endl;
    }
};

int main()
{
    Foo foo;
    foo.p(); // x is 20 and y is 10

    return 0;
}
```



The variable x is used as a data field and local variable for the method. The precedence is for the functions

## Points on Classes and Objects

- You can use primitive data types to define variables. You can also use class names to declare object names. In this sense, a class is also a data type.
- In C++, you can use the assignment operator = to copy the contents from one object to the other. By default, each data field of one object is copied to its counterpart in the other object. For example,
- `circle2 = circle1;`
- copies the radius in circle1 to circle2. After the copy, circle1 and circle2 are still two different objects but have the same radius.
- Object names are like array names. Once an object name is declared, it represents an object. It cannot be reassigned to represent another object. In this sense, an object name is a constant, though the contents of the object may change. Member wise copy can change an object's contents but not its name.
- An object contains data and may invoke functions. This may lead you to think that an object is quite large. It isn't, though. Data are physically stored in an object, but functions are not. Since functions are shared by all objects of the same class, the compiler creates just one copy for sharing. You can find out the actual size of an object using the `sizeof` function. For example, the following code displays the size of objects circle1 and circle2. Their size is 8, since the data field radius is double, which takes 8 bytes.

```
Circle circle1;
```

```
Circle circle2(5.0);
```

```
cout << sizeof(circle1) << endl;
```

```
cout << sizeof(circle2) << endl;
```

## 10.5 Instance and Static Members

If you want ***all the instances of a class to share data***, use ***static variables***, also known as ***class variables***.

Static variables store values for the variables in a common memory location.

Accordingly, all objects of the same class are affected if one object changes the value of a static variable. C++ supports static functions as well as static variables.

***Static functions*** can be ***called without creating an instance of the class***. Recall that ***instance functions*** can only be called from a specific instance.

## Chapter 11: Pointers and Dynamic Memory Allocation

### 11.2: Pointer Basics

#### **Listing 11.1** TestPointer.cpp

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

int main()
{
    int count = 5;
    int* pCount = &count; // Declaring and Initializing directly the value for the
    pointer

    // We could also do it separately

    /*
        int* pCount;

        pcount = &count ; // this is correct

        *pcount = &count ; // this is wrong
    */

    cout << "The value of count is " << count << endl;

    cout << "The address of count is " << &count << endl;
    cout << "The address of count is " << pCount << endl;

    /* We have use the dereference operator (*).
        *pcount mean to access to the value pointed by the pointer pcount, which
        is count
    */

    cout << "The value of count is " << *pCount << endl;

    cin.get(); // the output window will still open till I press enter.

    return 0;
} // End Main function
```

### Note for Declaration:

You can declare an int pointer using the syntax

`int* p;` or `int *p;` or `int * p;`

All these are equivalent . we uses the style `int* p` for declaring a pointer for two reasons:

1. `int* p` clearly separates the type `int*` from the identifier `p`. `p` is of the type `int*`, not of the type `int`.
2. Later in the book, you will see that a function may return a pointer. It is more intuitive to write the function header as:

`typeName* functionName(parameterList);`

rather than

`typeName *functionName(parameterList);`

One drawback of using the `int* p` style syntax is that it may lead to a mistake like this:

`int* p1, p2;`

This line seems as if it declares two pointers, but it is actually same as

`int *p1, p2;`

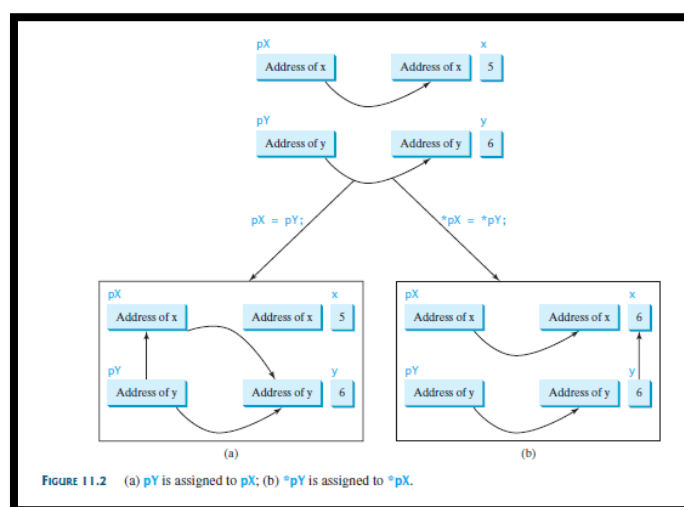
We recommend that you always declare a pointer variable in a single line like this:

`int* p1;`

`int* p2;`

### ➤ Relationship between Pointers

Suppose `pX` and `pY` are two pointer variables for variables `x` and `y`, as shown in Figure 11.2. To understand the relationships between the variables and their pointers, let us examine the effect of assigning `pY` to `pX` and `*pY` to `*pX`.



### 11.3 Defining Synonymous Types Using the typedef Keyword

*A synonymous type can be defined using the typedef keyword.*

Recall that the `unsigned` type is synonymous to `unsigned int`. C++ enables you to define custom synonymous types using the `typedef` keyword. Synonymous types can be used to simplify coding and avoid potential errors.

For example, the following statement defines `integer` as a synonym for `int`:  
`typedef int integer;`

So, now you can declare an `int` variable using  
`integer value = 40;`

The `typedef` declaration does not create new data types. It merely creates a synonym for a data type. This feature is useful to define a pointer type name to make the program easy to read.

For example, you can define a type named `intPointer` for `int*` as follows:  
`typedef int* intPointer;`

An integer pointer variable can now be declared as follows:  
`intPointer p;`  
which is the same as  
`int* p;`

One advantage of using a pointer type name is to avoid the errors involving missing asterisks.

If you intend to declare two pointer variables, the following declaration is wrong:  
`int* p1, p2;`

A good way to avoid this error is to use the synonymous type `intPointer` as follows:  
`intPointer p1, p2;`

## 11.5 Arrays and Pointers

A C++ array name is actually a constant pointer to the first element in the array.

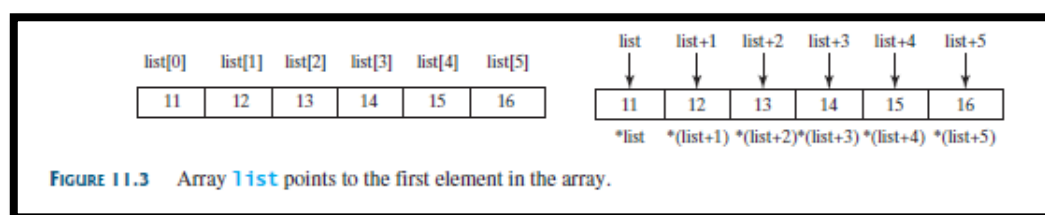
An **array without a bracket and a subscript** actually represents the **starting address of the array**.

In this sense, **an array is essentially a pointer**. Suppose you declare an array of int values as follows:

```
int list[6] = {11, 12, 13, 14, 15, 16};
```

The following statement displays the starting address of the array:

```
cout << "The starting address of the array is " << list << endl;
```



To **access the first element**, use `*list`. Other elements can be accessed using `*(list + 1)`, `*(list + 2)`, `*(list + 3)`, `*(list + 4)`, and `*(list + 5)`.

### Pointers Arithmetic:

An integer may be added to or subtracted from a pointer. The pointer is incremented or decremented by that integer times the size of the element to which the pointer points.

Array `list` points to the starting address of the array. Suppose this address is 1000.

Will `list + 1` be 1001? No. It is `1000 + sizeof(int)`. Why?

Since `list` is declared as an array of `int` elements, C++ automatically calculates the address for the next element by adding `sizeof(int)`.

Recall that `sizeof(type)` function returns the size of a data type (see Section 2.8, "Numeric Data Types and Operations").

The **size of each data type is machine dependent**. On Windows, the size of the `int` type is usually 4. So, no matter how big each element of the list is, `list + 1` points to the second element of the list, and `list + 3` points to the third, and so on.

➤ Note

Pointers can be compared using relational operators (==, !=, <, <=, >, >=) to determine their order.

Arrays and pointers form a close relationship. A pointer for an array can be used just like an array. You can even use pointer with index. Listing 11.3 gives such an example.

Listing 11.3 PointerWithIndex.cpp

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

int main()
{
    int list[6] = {11, 12, 13, 14, 15, 16}; // Declaring Array
    int* p = list; // Declaring Pointer

    // or we can declar it as this:
    // int* p = &list[0] ;

    for (int i = 0; i < 6; i++)
        cout << "address: " << (list + i) <<
            " value: " << *(list + i) << " " <<
            " value: " << list[i] << " " <<
            " value: " << *(p + i) << " " <<
            " value: " << p[i] << endl;

    return 0;
}
```

address: 0013FF4C	value: 11	value: 11	value: 11	value: 11
address: 0013FF50	value: 12	value: 12	value: 12	value: 12
address: 0013FF54	value: 13	value: 13	value: 13	value: 13
address: 0013FF58	value: 14	value: 14	value: 14	value: 14
address: 0013FF5C	value: 15	value: 15	value: 15	value: 15
address: 0013FF60	value: 16	value: 16	value: 16	value: 16

Pointer with index: since the array name is a pointer, and arrayname[index] = value , so in this case p[index] = also a value, because int\*p = list ;

➤ Note:

There is **one difference between arrays and pointers**. Once an array is declared, you cannot change its address. For example, the following statement is illegal:

```
int list1[10], list2[10];
list1 = list2; // Wrong
```

An **array name** is actually treated as a **constant pointer in C++**.



### Pointer Based C-Strings:

C-strings are often referred to as *pointer-based strings*, because they can be conveniently accessed using pointers. For example, the following two declarations are both fine:

```
char city[7] = "Dallas"; // Option 1
```

```
char* pCity = "Dallas"; // Option 2
```

Each declaration creates a sequence that contains characters 'D', 'a', 'l', 'l', 'a', 's', and '\0'.

You can access city or pCity using the array syntax or pointer syntax. For example, each of the following :

```
cout << city[1] << endl;
```

```
cout << *(city + 1) << endl;
```

```
cout << pCity[1] << endl;
```

```
cout << *(pCity + 1) << endl;
```

displays character a (the second element in the string).

## 11.6 Passing Pointer Arguments in a Function Call

### Listing 11.4 TestPointerArgument.cpp

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

// Swap two variables using pass-by-value
void swap1(int n1, int n2)
{
    int temp = n1;
    n1 = n2;
    n2 = temp;
}

// Swap two variables using pass-by-reference
void swap2(int& n1, int& n2)
{
    int temp = n1;
    n1 = n2;
    n2 = temp;
}

// Pass two pointers by value
void swap3(int* p1, int* p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

// Pass two pointers by reference
void swap4(int* &p1, int* &p2)
{
    int* temp = p1;
    p1 = p2;
    p2 = temp;
}

int main()
{
    // Declare and initialize variables
    int num1 = 1;
    int num2 = 2;

    cout << "Before invoking the swap function, num1 is "
         << num1 << " and num2 is " << num2 << endl;

    // Invoke the swap function to attempt to swap two variables
    swap1(num1, num2);

    cout << "After invoking the swap function, num1 is " << num1 <<
         " and num2 is " << num2 << endl;

    cout << "Before invoking the swap function, num1 is "
         << num1 << " and num2 is " << num2 << endl;

    // Invoke the swap function to attempt to swap two variables
    swap2(num1, num2);

    cout << "After invoking the swap function, num1 is " << num1 <<
         " and num2 is " << num2 << endl;
```

```

cout << "Before invoking the swap function, num1 is "
    << num1 << " and num2 is " << num2 << endl;

// Invoke the swap function to attempt to swap two variables
swap3(&num1, &num2);

cout << "After invoking the swap function, num1 is " << num1 <<
    " and num2 is " << num2 << endl;

int* p1 = &num1;
int* p2 = &num2;
cout << "Before invoking the swap function, p1 is "
    << p1 << " and p2 is " << p2 << endl;

// Invoke the swap function to attempt to swap two variables
swap4(p1, p2);

cout << "After invoking the swap function, p1 is " << p1 <<
    " and p2 is " << p2 << endl;

return 0;
}

```

```

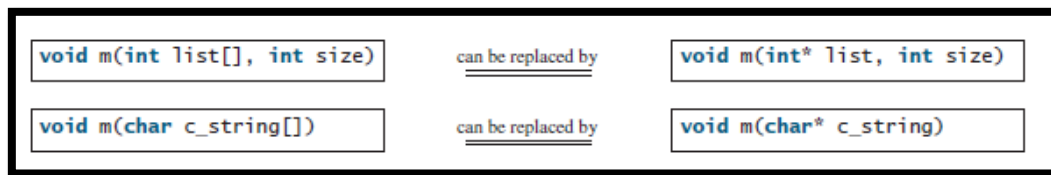
Before invoking the swap function, num1 is 1 and num2 is 2
After invoking the swap function, num1 is 1 and num2 is 2
Before invoking the swap function, num1 is 1 and num2 is 2
After invoking the swap function, num1 is 2 and num2 is 1
Before invoking the swap function, num1 is 2 and num2 is 1
After invoking the swap function, num1 is 1 and num2 is 2
Before invoking the swap function, p1 is 0028FB84 and p2 is 0028FB78
After invoking the swap function, p1 is 0028FB78 and p2 is 0028FB84

```

### Array Parameter or Pointer Parameter

An array parameter in a function can always be replaced using a pointer parameter.

For example:



Recall that a C-string is an array of characters that ends with a null terminator. The size of a C-string can be detected from the C-string itself.

If a value does not change, you should declare it `const` to prevent it from being accidentally modified. Listing 11.5 gives an example.

### Listing 11.5 ConstParameter.cpp

```
#include <iostream>

using namespace std;

void printArray(const int*, const int);

int main()
{
    int list[6] = {11, 12, 13, 14, 15, 16};
    printArray(list, 6);
    return 0;
} // End main

void printArray(const int* list, const int size)
{
    for (int i = 0; i < size; i++)
        cout << list[i] << " ";
} // End printArray function
```

### 11.9: Dynamic Persistent Memory Allocation

Write a function an array argument and returns a new array that is the reversal of the array argument, so the original array have to be unchanged.

#### WrongReverse.Cpp

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


int* reverse(const int* list, int size)
{
    int result[6];
    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        result[j] = list[i];
    }

    return result;
}

void printArray(const int* list, int size)
{
    for (int i = 0; i < size; i++)
        cout << list[i] << " ";
}

int main()
{
    int list[] = {1, 2, 3, 4, 5, 6};
    int* p = reverse(list, 6);
    printArray(p, 6);

    return 0;
}
```



- The array **result** is stored in the activation record in the call stack.
- The memory in the call stack does not persist; when the function returns, the activation record used by the function in the call stack are thrown away from the call stack.
- Attempting to access the array via the pointer will result in erroneous and unpredictable values.

#### Fixing the Problem: the new Operator

To fix this problem, you have to allocate persistent storage for the **result** array so that it can be accessed after the function returns. We discuss the fix next.

C++ supports **dynamic memory allocation**, which enables you to **allocate persistent storage dynamically**. The memory is created using the **new** operator.

For example, `int* p = new int(4) ;`

Here, `new int` tells the computer to allocate memory space for an `int` variable initialized to 4 at runtime, and the address of the variable is assigned to the pointer `p`. So you can access the memory through the pointer.

New operator is specifically to allocate memory in the heap only. We can use it on primitive and non-primitive data type. Don't forget to use the delete operator when using new.

### Create Array Dynamically

```
cout << "Enter the size of the array: ";
```

```
int size;
```

```
cin >> size;
```

```
int* list = new int[size];
```

Here, `new int[size]` tells the computer to allocate memory space for an `int` array with the specified number of elements, and the address of the array is assigned to `list`.

The array created using the `new` operator is also known as a dynamic array.

Note that when you create a regular array, its size must be known at compile time. It cannot be a variable. It must be a constant.

The memory allocated using the `new` operator is persistent and exists until it is explicitly deleted or the program exits. Now you can fix the problem in the preceding example by creating a new array dynamically in the `reverse` function.

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

int* reverse(const int* list, int size)
{
    int* result = new int[size];

    for (int i = 0, j = size - 1; i < size; i++, j--)
    {
        result[j] = list[i];
    }

    return result;
}

void printArray(const int* list, int size)
{
    for (int i = 0; i < size; i++)
        cout << list[i] << " ";
}

int main()
{
    int list[] = {1, 2, 3, 4, 5, 6};
    int* p = reverse(list, 6);
    printArray(p, 6);

    return 0;
}
```

C++ allocates **local variables** in **the stack**, but the **memory allocated by the new operator** is in an area of memory called the freestore or **heap**.

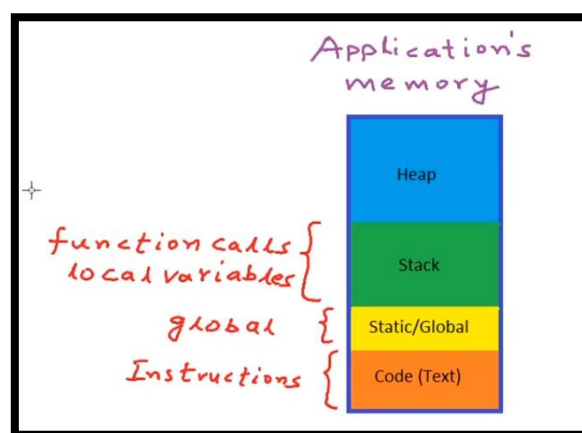
The **heap memory** remains **available until you explicitly free it or the program terminates**. If you **allocate heap memory for a variable while in a function, the memory is still available after the function returns**.

The result array is created in the function (line 6). After the function returns in line 25, the result array is intact. So, you can access it in line 26 to print all the elements in the result array.

### Memory Structure: Heap and Stack

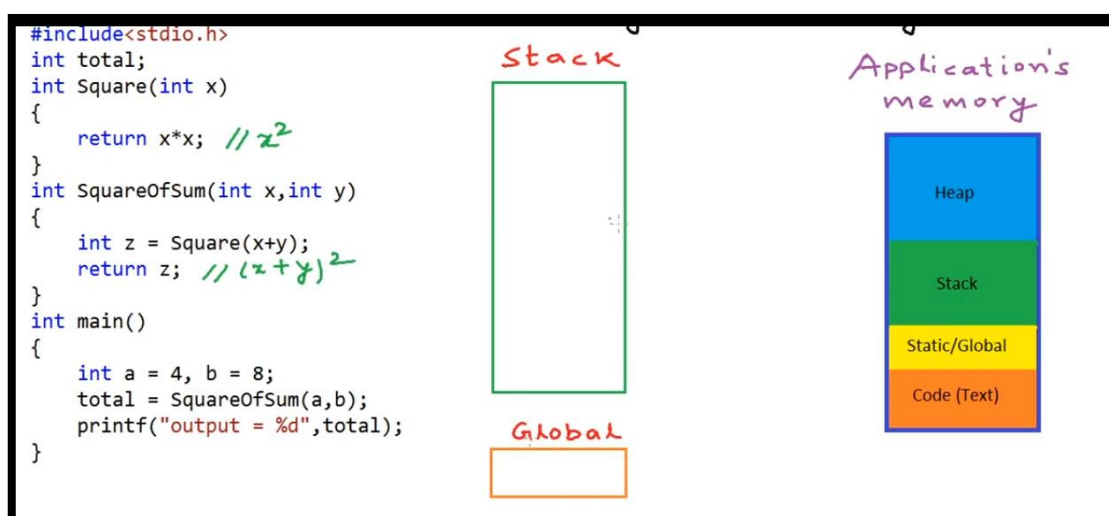
We will now see how the memory is organized when we write a program.

Normally, the memory is divided into 4 segments as shown in the figure below.



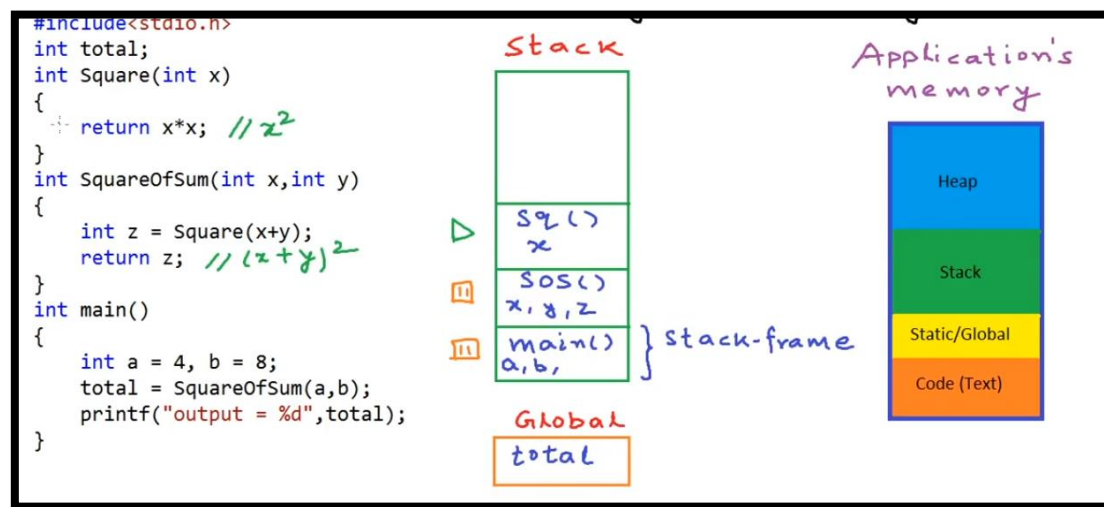
The stack, static/global and code segments **don't grow while the program is running**.

We will see the trace of the program and what is happening to memory while a program is being executed.





When the program is being executed, all the functions will be pushed into the stack. Main is called 1<sup>st</sup> so it will be pushed 1<sup>st</sup>, then we have squareofsum function, so it will be pushed and finally we have the function square.



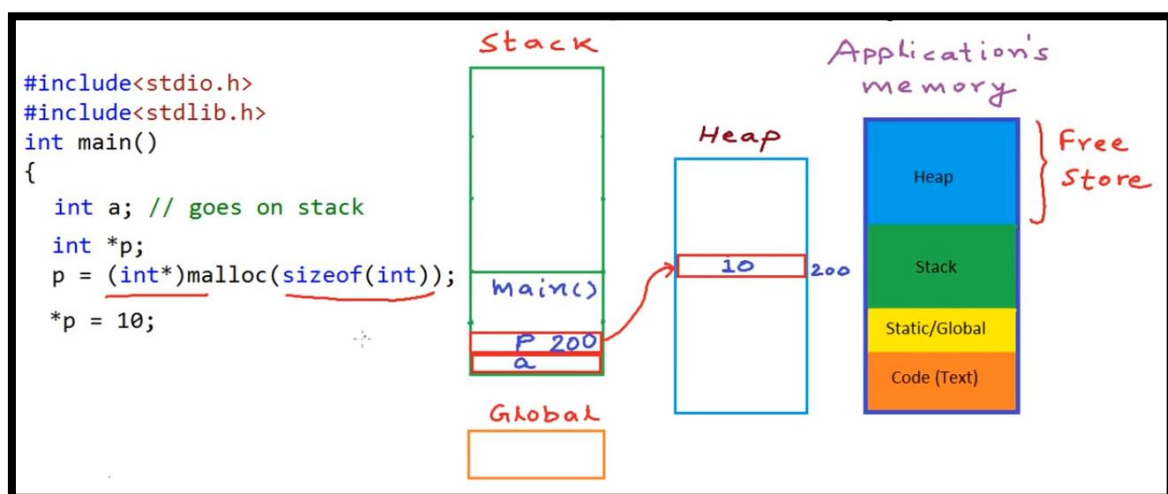
When we reach return statement in square function, the square function will be popped out from the stack. Same for sos, and main() when the program terminate.

The stack rules said that when a function is called, it will be pushed into the stack, and when the function has finished, it will be popped out from the stack. Also in a stack, the variables are cleared just when the function has finished or the main() is terminated.

A stack can be run out from memory (known as stack overflow), when we don't have any more space in the stack (like doing an infinite recursion).

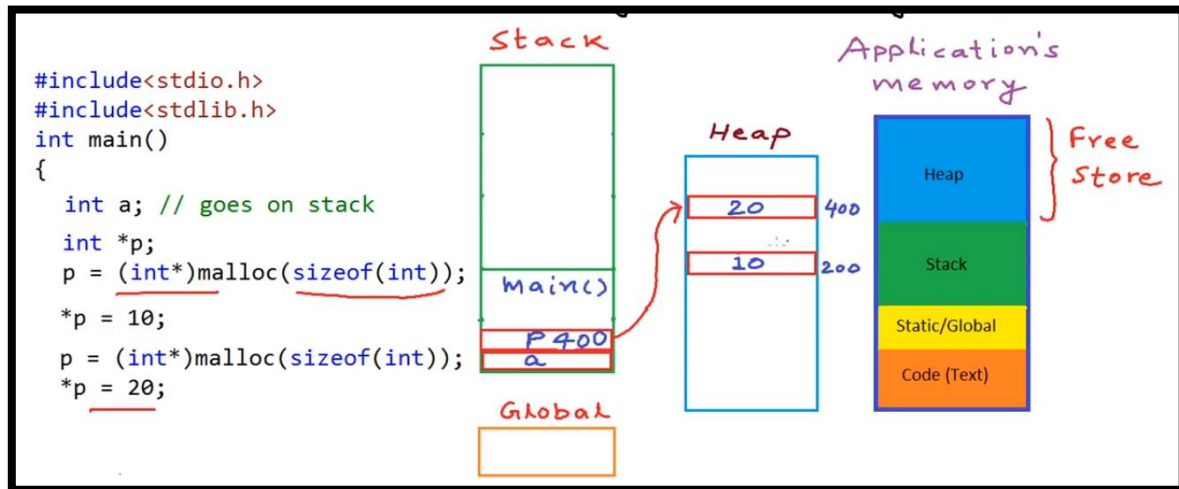
In contrast, a **heap is dynamic**, it can change through the execution of the program, and we can delete whenever we want a variable, and not necessary wait till the program finish.

In C, we have 4 functions to do allocation: malloc, realloc, calloc, free and in C++ we have 2 operators: new and delete.

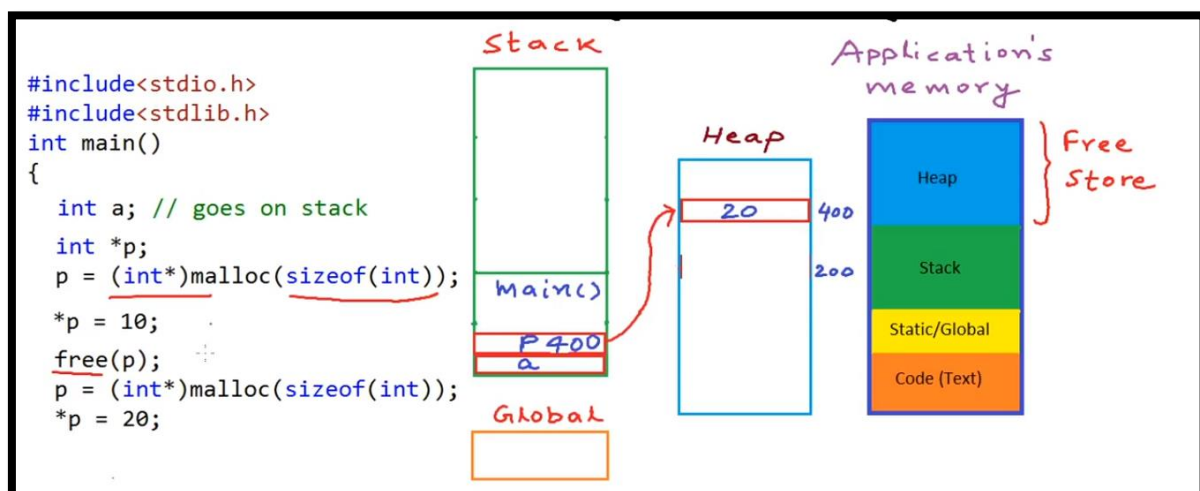


Malloc return a void pointer, so we need to do casting.

Suppose we are allocating again.



We can see that the old value at address 200 (10) did not freed automatically, we need to free it before allocating again because it is a waste of memory.



In free function, we pass the name of the pointer.

### Delete Operator

To explicitly free the memory created by the **new** operator, use the **delete** operator for the pointer. For example,

```
delete p;
```

If the memory is allocated for an array, the `[]` symbol must be placed between the **delete** keyword and the pointer to the array to release the memory properly. For example,

```
delete [] list;
```

### Dangling Pointers

After the memory pointed by a pointer is freed, the value of the pointer becomes undefined. Moreover, if some other pointer points to the same memory that was freed, this other pointer is also undefined. These undefined pointers are called **dangling pointers**.

Don't apply the dereference operator `*` on dangling pointer. Doing so would cause serious errors.

### Caution

Use the **delete** keyword only with the pointer that points to the memory created by the **new** operator. Otherwise, it may cause unexpected problems. For example, the following code is erroneous, because **p** does not point to a memory created using **new**.

```
int x = 10;  
int* p = &x;  
delete p; // This is wrong
```

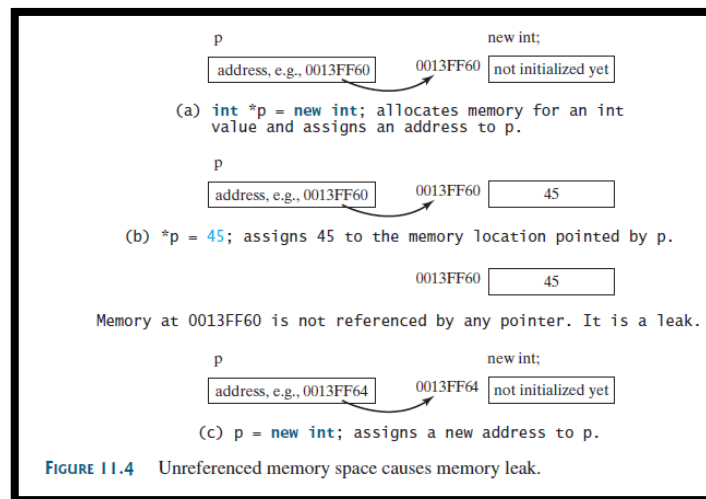
## Memory Leak

Line 1 declares a pointer assigned with a memory address for an **int** value, as shown in Figure 11.4a.

Line 2 assigns **45** to the variable pointed by **p**, as shown in Figure 11.4b.

Line 3 assigns a new memory address to **p**, as shown in Figure 11.4c.

The original memory space that holds value **45** is not accessible, because it is not pointed to by any pointer. This memory cannot be accessed and cannot be deleted. This is a **memory leak**.



Dynamic memory allocation is a powerful feature, but you must use it carefully to avoid memory leaks and other errors. As a good programming practice, every call to **new** should be matched by a call to **delete**.

### 11.10 Creating and Accessing Dynamic Objects

To create an object dynamically, invoke the constructor for the object using the syntax `new ClassName(arguments)`.

You can also create objects dynamically on the heap using the syntax shown below.

`ClassName* pObj = new ClassName();` or  
`ClassName* pObj = new ClassName;`

Creates an object using the no-arg constructor and assigns the object address to the pointer

`ClassName* pObj = new ClassName(arguments);`


Creates an object using the constructor with arguments and assigns the object address to the pointer.

### Example:

```
string* p = new string("abcdefg");
```

```
cout << "The first three characters in the string are "
```

```
<< (*p).substr(0, 3) << endl;
```




To **access object members via a pointer**, you must **dereference the pointer** and use the **dot(.) operator** to object's members.

```
cout << "The length of the string is " << (*p).length() << endl;
```

```
cout << "The first three characters in the string are "
```

```
<< p->substr(0, 3) << endl;
```



C++ also provides a shorthand member selection operator for accessing object members from a pointer:  
  
*arrow operator* (->), which is a dash (-) immediately followed by the greater than (>) symbol

```
cout << "The length of the string is " << p->length() << endl;
```

The objects are destroyed when the program is terminated. To explicitly destroy an object, invoke delete p;

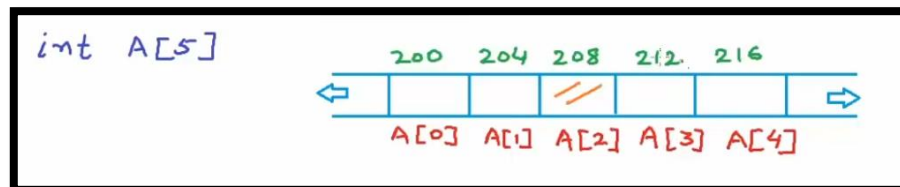
### 11.11: The this Pointer

Sometimes you need to reference a class's hidden data field in a function. For example, a data field name is often used as the parameter name in a set function for the data field. In this case, you need to reference the hidden data field name in the function in order to set a new value to it. A hidden data field can be accessed by using the 'this' keyword, which is a special built-in pointer that references the calling object.

### 11.12: Pointers in 2D Array

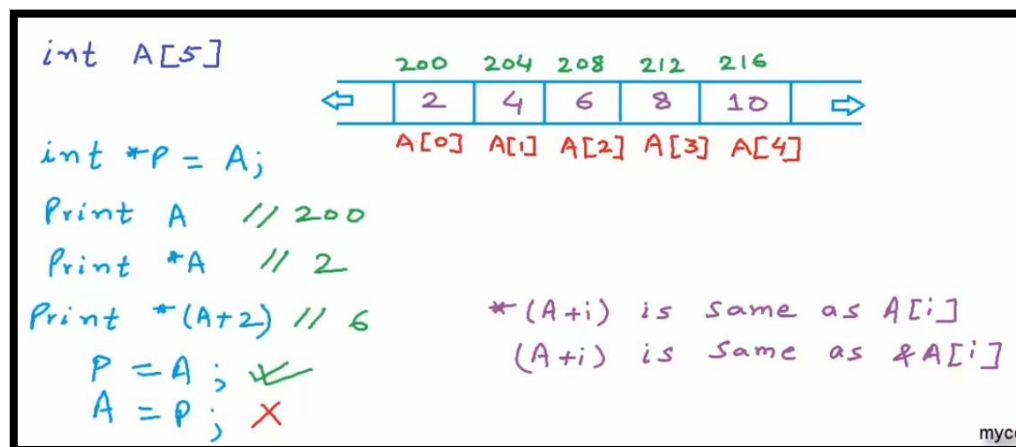
1<sup>st</sup> we will understand how arrays are organized in memory. We will begin with 1D array.

Remember that each byte has its starting address in memory. If we declare an array of int, which has a size of 4 bytes on most operating system, we will have the following configuration.



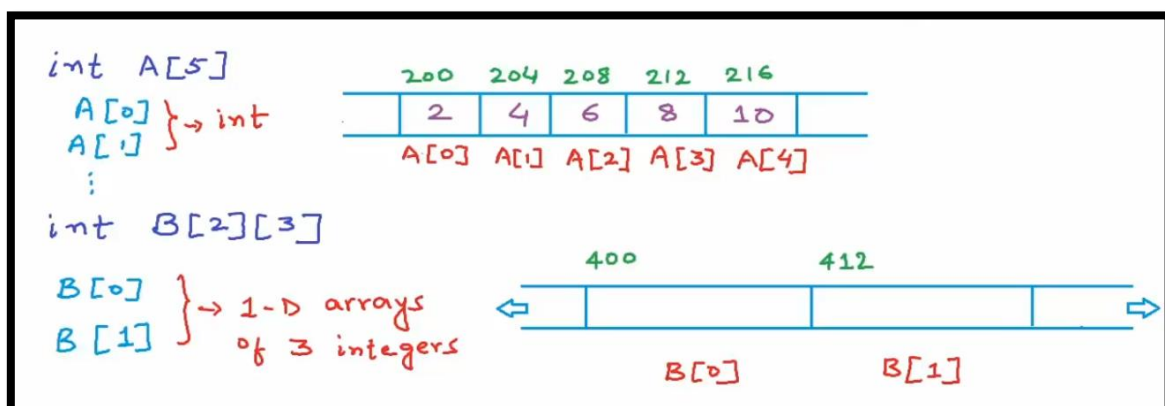
Name of the array return a pointer to the 1<sup>st</sup> element in the array.

We will present a summary about pointer operation in 1D array.



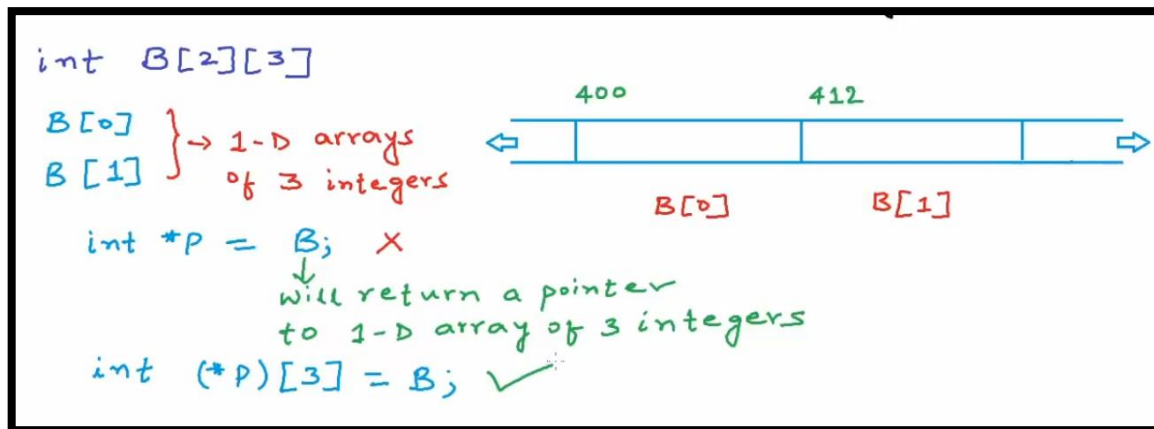
`A = P` is wrong because the name of the array is a constant pointer.

Now if we work with 2D array, like `int B[2][3]`, so we are creating 2 1D array, each of 3 elements.

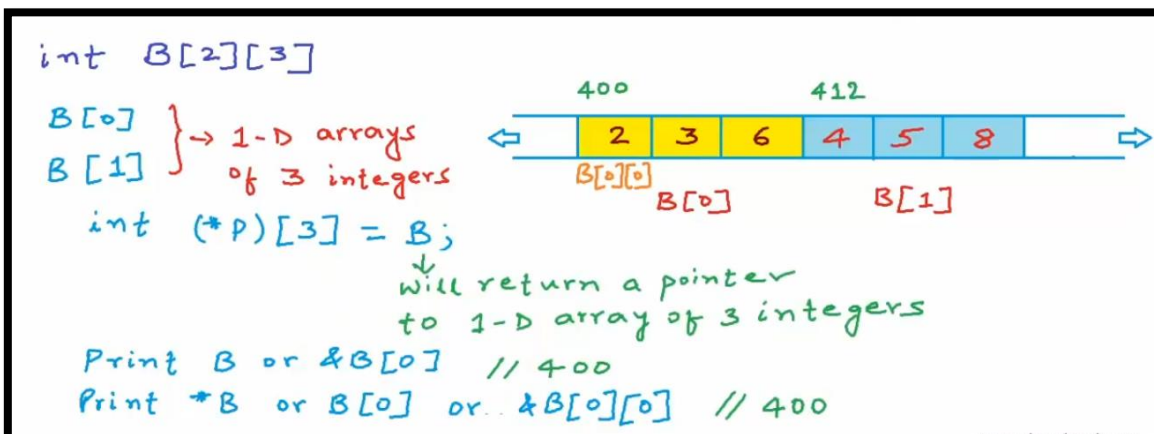


Each element in B, B[0] and B[1] is a 1D array of 12 bytes (4bytes (size int) x 3(nb of columns)).

Now the name of the array return a pointer to the 1<sup>st</sup> element in the array, in 2D array case it is tricky, we have to pay attention how to use the name of the 2D array.



Pointer type is important when we are dereferencing the pointer.



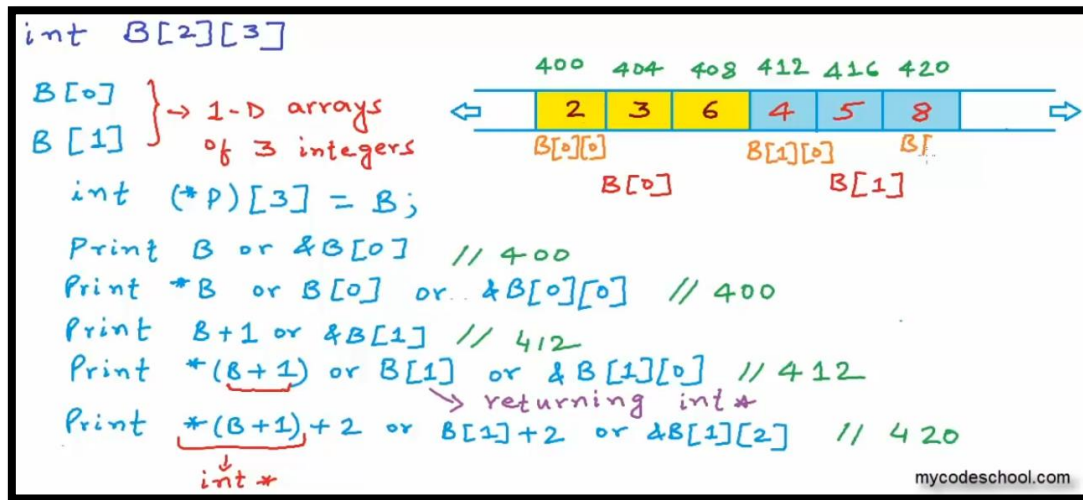
If we try with B+1: B + size of 3 integers in 1D arrays = B + 12 = 412.

➤ Very Important:

Here (1) is a step of 3 integers (so 12 bytes) because B is a pointer of type int\* [3], so it is pointing to 3 elements in this case, it is not like in the ordinary case of 1D array, where the name of the array is pointing to only 1 element (4 bytes in case of integers 1D array).

Now the different part in 2D array in using  $*(B+1)$ ? It will not return a value as in 1D array case.

$*(B+1) = *(B + \text{size of 3 integers}) = *(& B[1]) = B[1]$  which is  $\text{int}^*$  to the 1<sup>st</sup> element in  $B[1]$ .



➤ Compute the expression  $*(B+1)$ :

$*(B+1) = *(&B[0]+1) = *(B[0]+1)$ , where  $B[0]$  is a pointer here to the 1<sup>st</sup> element in array  $B[0]$ , which is  $B[0][0]$  (note that we have use 2 index to capture the element in array  $B$ ).

To continue:  $*(B[0]+1) = *(400 + 1) = *(404) = 3 = B[0][1]$  (since  $B[0]$  is a pointer to  $B[0][0]$  which has a size of 4 bytes, so the 1 here is a jump of 4 bytes)

**Be aware that**

- $*(B+1)$  is different than  $*(B+1)$ , where the 1 in  $*(B+1)$  is a step of 3 int.

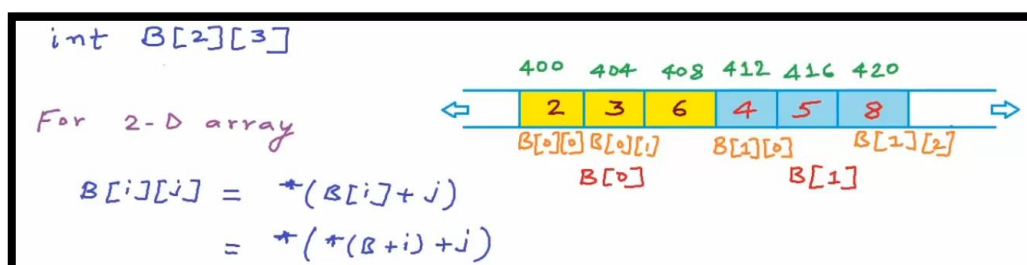
So we can say:

- $*(B) = B[0] = \text{pointer to } B[0][0]$ , which is the 1<sup>st</sup> element in array  $B[0]$
- $*(B+1) = B[1] = \text{pointer to } B[1][0]$ , which is the 1<sup>st</sup> element in array  $B[1]$
- The number of these pointers are equal to the number of lines.

And don't forget  $B$  alone, is  $\text{int}^* [3]$ , so it is a pointer to 3 integer elements, not like the ordinary case (pointer to 1 elements), so any jump of  $B$  (like  $(B+1)$ ), the 1 will be equivalent to  $3 \times 4\text{bytes}$  (size of  $\text{int}$ ).

Last note:

- $B$  is a pointer of type  $\text{int}^* [3]$ , so the step with  $B$  is a step of 3 int.
- Dereferencing  $B$  will give us:  $*B = B[0] = \text{pointer of type } \text{int}^*$ , so the step here is a step of int





### 11.13: Pointers in Higher Dimensional Arrays

1<sup>st</sup> we will put a general summary about pointers in 2D arrays.

```

int B[2][3]
int (*P)[3] = B; ✓
Print B //400
Print *B //400
Print B[0] //400
Print &B[0][0] //400
        
```

$B[i][j] = *(B[i] + j) = *((*B + i) + j)$   
↓ ↓  
int\* int (\*)[3]

Now in case of 3 dimensional arrays, it is a collection of 2D arrays (because remember we see multidimensional arrays as arrays of arrays).

```

int C[3][2][2]
        
```

```

int C[3][2][2]
int (*P)[2][2] = C; ✓
Print C //800
Print *C or C[0] or &C[0][0]
        
```

$C[i][j][k] = *(C[i][j] + k) = *((*C[i] + j) + k)$   
 $= *((*(*C + i) + j) + k)$

- C is a pointer to 2x2 array (so 2D array).
- C[0][1] is a pointer to 1 element.
- C[0] is a pointer to a 1D array.

```

int C[3][2][2]
int (*P)[2][2] = C; ✓
Print C // 800
Print *C or C[0] or &C[0][0] // 800
Print *(C[0][1] + 1) or C[0][1][1] // 9
Print *(C[1] + 1) or C[1][1] or &C[1][1][0] // 824
          ↓
        int (*)[2]

```

mycodeschool.com

## Chapter 15: Inheritance and Polymorphism

### ➤ Points about Inheritance

- **Private data fields in a base class are not accessible outside the class.** Therefore, they cannot be used directly in a derived class. They can, however, be accessed/mutated through public accessor/mutator if defined in the base class.
- Not all *is-a* relationships should be modeled using inheritance. For example, a square is a rectangle, but you should not define a Square class to extend a Rectangle class, because there is nothing to extend (or supplement) from a rectangle to a square. Rather you should define a Square class to extend the GeometricObject class. For class A to extend class B, A (**derived class**) should **contain more detailed information than** B (**base class**).
- Inheritance is used to model the *is-a* relationship. **Do not blindly extend a class just for the sake of reusing functions.** For example, it makes no sense for a Tree class to extend a Person class, even though they share common properties such as height and weight. A derived class and its base class must have the *is-a* relationship.
- **C++ allows you to derive a derived class from several classes.** This capability is known as **multiple inheritance**, which is discussed in Supplement IV.A.

## 15.8 The protected Keyword

*A protected member of a class can be accessed from a derived class.*

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

class B
{
public:
    int i;

protected:
    int j;

private:
    int k;
};

class A: public B
{
public:
    void display() const
    {
        cout << i << endl; // Fine, can access it
        cout << j << endl; // Fine, can access it
        cout << k << endl; // Wrong, cannot access it
    }
};

int main()
{
    A a;
    cout << a.i << endl; // Fine, can access it

    cout << a.j << endl; // Wrong, cannot access it
    cout << a.k << endl; // Wrong, cannot access it

    return 0;
}
```

Here (a) is an **object of the derived class** A, so (a) cannot access members j and k.

## 15.9 Abstract Classes and Pure Virtual Functions

*An abstract class cannot be used to create objects. An abstract class can contain abstract functions, which are implemented in concrete derived classes.*

In the inheritance hierarchy, classes become more specific and concrete with each new derived class. If you move from a derived class back up to its parent and ancestor classes, the classes become more general and less specific. Class design should ensure that a base class contains common features of its derived classes.

Sometimes a **base class** is so abstract that it **cannot have any specific instances**. Such a class is referred to as an **abstract class**.

In Section 15.2, GeometricObject was defined as the base class for Circle and Rectangle. GeometricObject models common features of geometric objects.

Both Circle and Rectangle contain the getArea() and getPerimeter() functions for computing the area and perimeter of a circle and a rectangle.

Since you can compute areas and perimeters for all geometric objects, it is better to define the getArea() and getPerimeter() functions in the GeometricObject class. However, these **functions cannot be implemented in the GeometricObject class (the base class)**, because **their implementation is dependent on the specific type of geometric object**. Such functions are referred to as **abstract functions**.

After you **define the abstract functions in GeometricObject**, GeometricObject becomes an **abstract class**.

### Listing 15.13 AbstractGeometricObject.h

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

class GeometricObject
{
protected:
    GeometricObject();
    GeometricObject(const string& color, bool filled);

public:
    string getColor() const;
    void setColor(const string& color);
    bool isFilled() const;
    void setFilled(bool filled);
    string toString() const;

    virtual double getArea() const = 0;
    virtual double getPerimeter() const = 0;

private:
    string color;
    bool filled;
}; // Must place semicolon here

#endif
```

GeometricObject is just like a regular class, except that *you cannot create objects from it because it is an abstract class*. If you attempt to create an object from GeometricObject, the compiler will report an error.

The (= 0) notation indicates that getArea is a pure virtual function. A *pure virtual function does not have a body or implementation in the base class*

### Listing 15.14 AbstractGeometricObject.cpp

```
#include "AbstractGeometricObject.h"

GeometricObject::GeometricObject()
{
    color = "white";
    filled = false;
}

GeometricObject::GeometricObject(const string& color, bool filled)
{
    setColor(color);
    setFilled(filled);
}

string GeometricObject::getColor() const
{
    return color;
}

void GeometricObject::setColor(const string& color)
{
    this->color = color;
}

bool GeometricObject::isFilled() const
{
    return filled;
}
```

```

void GeometricObject::setFilled(bool filled)
{
    this->filled = filled;
}

string GeometricObject::toString() const
{
    return "Geometric Object";
}

```

### **Listing 15.15** DerivedCircleFromAbstractGeometric-Object.h

```

#ifndef CIRCLE_H
#define CIRCLE_H
#include "AbstractGeometricObject.h"

class Circle: public GeometricObject
{
public:
    Circle();
    Circle(double);
    Circle(double radius, const string& color, bool filled);
    double getRadius() const;
    void setRadius(double);
    double getArea() const;
    double getPerimeter() const;
    double getDiameter() const;

private:
    double radius;
}; // Must place semicolon here

#endif

```

### **Listing 15.16** DerivedCircleFromAbstractGeometricObject. Cpp

```

#include "DerivedCircleFromAbstractGeometricObject.h"

// Construct a default circle object
Circle::Circle()
{
    radius = 1;
}

// Construct a circle object with specified radius
Circle::Circle(double radius)
{
    setRadius(radius);
}

// Construct a circle object with specified radius, color, filled
Circle::Circle(double radius, const string& color, bool filled)
{
    setRadius(radius);
    setColor(color);
    setFilled(filled);
}

// Return the radius of this circle
double Circle::getRadius() const
{
    return radius;
}

```

```

}

// Set a new radius
void Circle::setRadius(double radius)
{
    this->radius = (radius >= 0) ? radius : 0;
}

// Return the area of this circle
double Circle::getArea() const
{
    return radius * radius * 3.14159;
}

// Return the perimeter of this circle
double Circle::getPerimeter() const
{
    return 2 * radius * 3.14159;
}

// Return the diameter of this circle
double Circle::getDiameter() const
{
    return 2 * radius;
}

```

### **Listing 15.17** DerivedRectangleFromAbstractGeometric- Object.h

```

#ifndef RECTANGLE_H
#define RECTANGLE_H
#include "AbstractGeometricObject.h"

class Rectangle: public GeometricObject
{
public:
    Rectangle();
    Rectangle(double width, double height);
    Rectangle(double width, double height, const string& color, bool filled);
    double getWidth() const;
    void setWidth(double);
    double getHeight() const;
    void setHeight(double);
    double getArea() const;
    double getPerimeter() const;

private:
    double width;
    double height;
}; // Must place semicolon here

#endif

```



### **Listing 15.18** `DerivedRectangleFromAbstractGeometricObject.Cpp`

```
#include "DerivedRectangleFromAbstractGeometricObject.h"

// Construct a default rectangle object
Rectangle::Rectangle()
{
    width = 1;
    height = 1;
}

// Construct a rectangle object with specified width and height
Rectangle::Rectangle(double width, double height)
{
    setWidth(width);
    setHeight(height);
}

// Construct a rectangle object with width, height, color, filled
Rectangle::Rectangle(double width, double height,
    const string& color, bool filled)
{
    setWidth(width);
    setHeight(height);
    setColor(color);
    setFilled(filled);
}

// Return the width of this rectangle
double Rectangle::getWidth() const
{
    return width;
}

// Set a new radius
void Rectangle::setWidth(double width)
{
    this->width = (width >= 0) ? width : 0;
}

// Return the height of this rectangle
double Rectangle::getHeight() const
{
    return height;
}

// Set a new height
void Rectangle::setHeight(double height)
{
    this->height = (height >= 0) ? height : 0;
}

// Return the area of this rectangle
double Rectangle::getArea() const
{
    return width * height;
}

// Return the perimeter of this rectangle
double Rectangle::getPerimeter() const
{
    return 2 * (width + height);
}
```

You may be wondering whether the abstract functions `getArea` and `getPerimeter` should be removed from the `GeometricObject` class.

The following example in Listing 15.19 shows the benefits of defining them in the `GeometricObject` class.

This example presents a program that:

- creates two geometric objects (a circle and a rectangle)
- invokes the `equalArea` function to check whether the two objects have equal areas.
- invokes the `displayGeometricObject` function to display the objects.

### **Listing 15.19** [TestAbstractGeometricObject.cpp](#)

```
#include "AbstractGeometricObject.h"
#include "DerivedCircleFromAbstractGeometricObject.h"
#include "DerivedRectangleFromAbstractGeometricObject.h"
#include <iostream>
using namespace std;

// A function for comparing the areas of two geometric objects
bool equalArea(const GeometricObject& g1,
               const GeometricObject& g2)
{
    return g1.getArea() == g2.getArea();
}

// A function for displaying a geometric object
void displayGeometricObject(const GeometricObject& g)
{
    cout << "The area is " << g.getArea() << endl;
    cout << "The perimeter is " << g.getPerimeter() << endl;
}

int main()
{
    Circle circle(5);
    Rectangle rectangle(5, 3);

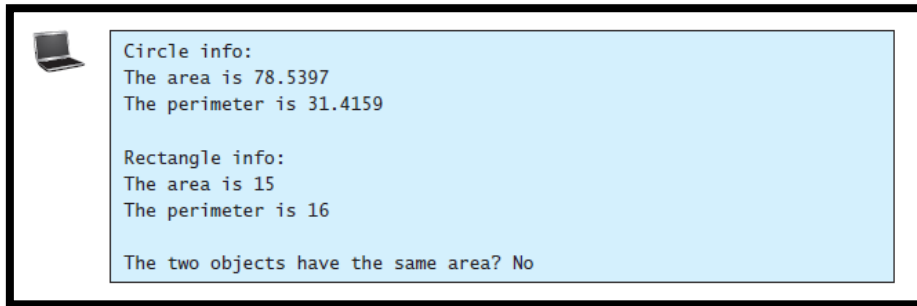
    cout << "Circle info: " << endl;
    displayGeometricObject(circle);

    cout << "\nRectangle info: " << endl;
    displayGeometricObject(rectangle);

    cout << "\nThe two objects have the same area? " <<
        (equalArea(circle, rectangle) ? "Yes" : "No") << endl;

    return 0;
}
```

### Program Output:



```
Circle info:  
The area is 78.5397  
The perimeter is 31.4159  
  
Rectangle info:  
The area is 15  
The perimeter is 16  
  
The two objects have the same area? No
```

The program creates a Circle object and a Rectangle object in lines 23–24.

The pure virtual functions `getArea()` and `getPerimeter()` defined in the `GeometricObject` class are overridden in the `Circle` class and the `Rectangle` class.

When invoking `displayGeometricObject(circle1)` (line 27), the functions `getArea` and `getPerimeter` defined in the `Circle` class are used, and when invoking `displayGeometricObject(rectangle)` (line 30), the functions `getArea` and `getPerimeter` defined in the `Rectangle` class are used.

C++ dynamically determines which of these functions to invoke at runtime, depending on the type of object.

Similarly, when invoking `equalArea(circle, rectangle)` (line 33), the `getArea` function defined in the `Circle` class is used for `g1.getArea()`, since `g1` is a circle.

Also, the `getArea` function defined in the `Rectangle` class is used for `g2.getArea()`, since `g2` is a rectangle.

Note that if the `getArea` and `getPerimeter` functions were not defined in `GeometricObject`, you cannot define the `equalArea` and `displayObject` functions in this program. So, you now see the benefits of defining the abstract functions in `GeometricObject`.