Quick Study® COMPUTER

# C++ Guide

THE ULTIMATE GUIDE TO C++ A PROGRAMMING LANGUAGE.

## Introduction

### This Chart Assumes That All Readers:

1. Have a general understanding of programming languages.
2. Understand the PC / Wintel environment.

### The Programming Process

These are the five simple steps to follow when developing a program:

1. Decide what the program's primary function will be.
2. Decide how the program will execute.
3. Develop the code properly.
4. Test the program.
5. Compile the program.

### What is C++?

C++ is a programming language which evolved from C and was developed by AT&T in the 1980s. It is a structured programming language designed to build large programs out of smaller programs.

The program is rather small with only 74 keywords, but it has one of the largest assortments of operators.

C++ does not have any input or output statements. Instead, it has a library of standard I/O functions allowing any computer or device that conforms to C++ standard to execute the program.

*Note: This chart is based on a specific C++ compiler. There is no difference in coding between this compiler and all other marketed compilers.*

## Basic C++

### Definitions

**Compiler:** converts code into low level machine instruction.
**Logic Error:** an error difficult to find, the computer cannot find them.
**Literal:** a constant value stated in the program.
**Operator:** characters that perform specific functions.
**Syntax Error:** an error in the code stucture (i.e. spelling).
**Variable:** a defined value that holds changeable data.

### Basic Lines of Code:

There are at least six lines of code that will appear in every C++ Program:

```
1. //Filename: FILENAME.CPP
2. //Comment about what it does.
3. #include <iostream>
4.  using namespace std;
5. void main()
6.    {
7.    cout << "Hello World.";
8.    }
```

**Line 1:** designates the name of the file.
**Line 2:** comment that gives a short explanation of what the file is for.
**Line 3:** a statement making reference to an existing library. This statement asks the compiler to include the existing library in the compiling process. This library can be one from the standard library or from a recent development.
**Line 4:** this is needed to make the I/O facilities available.
**Line 5:** the `main()` function defines where the rest of the program will stem from. The `void` before `main()` should be taken as a given, until a working knowledge of the program exists.
*Note:* `main()` *is a function. Once it is defined another cannot be defined until the* `main()` *function ends.*
**Line 6 and 8:** the opening and closing braces for the entire program. All coding for the `main()` function will exist inside these braces.
**Line 7:** the only line that will do anything when the file is executed. It represents the code for `main()`.

### Comments

Comments are placed to the right of a line of code. They are designated by a `//`. The comments are used to explain what is happening within the program.

## Numbers and Characters

### Data Types

In C++ a progammer must specify the type of information (data) that is to be placed within the variable. C++ has 16 designated data types :

| Declaration Name | Data Type Description |
|---|---|
| char | Character |
| unsigned char | Unsigned character |
| signed char | Signed character (same as char) |
| int | Integer |
| unsigned int | Unsigned integer |
| signed int | Signed integer (same as int) |
| short int | Short integer |
| unsigned short int | Unsigned short integer |
| signed short int | Signed short integer (same as short int) |
| long | Long integer (same as int) |
| long int | Long integer (same as long) |
| signed long int | Signed long integer (same as long int) |
| unsigned long int | Unsigned long integer |
| float | Floating-point (Real) |
| double | Double floating-point (Real) |
| long double | Long double floating-point (Real) |

Data Type example: `char initial;`

Each data type has a specific range of characters which it can occupy. The table below explains each data type's range:

| Data Type | Range |
|---|---|
| char | -128 to 127 |
| unsigned char | 0 to 255 |
| signed char | -128 to 127 |
| int | -2147483648 to 2147483647 |
| unsigned int | 0 to 65535 |
| signed int | -2147483648 to 2147483647 |
| short int | -32768 to 32767 |
| unsigned short int | 0 to 65535 |
| signed short int | -32768 to 32767 |
| long int | -2147483648 to 2147483647 |
| signed long int | -2147483648 to 2147483647 |
| unsigned long int | 0 to 4294967296 |
| float | -3.4E38 to 3.4E+38 |
| double | -1.7E308 to 1.7E+308 |
| long double | -1.7E+308 to 1.7E+308 |
| float | -1.7E+308 to 1.7E+308 |
| double | -1.7E+308 to 1.7E+308 |
| long double | -1.7E+308 to 1.7E+308 |

### Adding Expressions to Variables

Now that the variable has been designated as a specific data type, an expression is assigned to the variable. An expression is assigned to a variable with the operator '='. The format of assignment looks like this:

```
variable = expression;
```

The expression can be assigned when defining the variable or later in the program.

Constant variables are variables that cannot change during the program. To designate a variable as a constant follow this example:

```
const int day = 5;
```

The constant variable expression must be assigned when defining the variable.

### Character Literals and String Literals

C++ designates character literals as a single character between single quotation marks: '1'.

If the character literal does not have single quotation marks around it, C++ will think the program is defining a variable. C++ programs can define a variable with another variable. For example:

```
char initial;
initial = 'A'
initial = firstLetter
```

A *string literal* is one or more characters between double quotation marks, i.e. "mike".

C++ places a null zero (\0) at the end of a string literal during compiling to designate the end of the string. If the null zero is not added, the string end and the program will not know where the string ends. This is not necessarily a problem, but it may add memory space.

### Escape Sequences

An escape sequence informs the program that the following character(s) is (are) a special control character. The backslash (\) operator represents an escape sequence. The following characters may be for an ASCII character or for a special escape sequence character. The following list displays all special escape sequence characters:

| Escape Sequence | Meaning |
|---|---|
| \a | Alarm (a beep from the speaker) |
| \b | Backspace |
| \f | Form feed (new page on printer) |
| \n | New line (carriage return and line feed) |
| \r | Carriage return |
| \t | Tab |
| \v | Vertical tab |
| \\ | Backslash (\) |
| \? | Question mark |
| \' | Single quotation mark |
| \" | Double quotation mark |
| \000 | Octal number |
| \xhh | Hexadecimal number |
| \0 | Terminator (or null zero) |

### cout and cin

`cout` and `cin` are defined in the IOSTREAM.H. When the IOSTREAM.H library is included, I/O's with cout and cin are possible.

`cout` is for data output and `cin` is for data input. They are pronounced "see out" and "see in".

The insertion operator (<<) is used with cout and the extraction operator (>>) is used with cin.

cout and cin examples:

```
cout << "Hello world!";
cin >> Name; // Waits for users input
```

**Output Options:**

All float data types cout a certain number of digits. If the value was 7.8 the cout would be 7.800000. The answer is correct, but the extra zeros get in the way.

An I/O manipulator changes the way cout works. The output manipulator precision() limits the number of digits output for less precision. This is an example for a cout of 7.80:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
cout << 7.8;
```

**To set a fixed decimal point in any program** the only item in the above example that will ever change is the (2) in line 3, depending on the number of decimal places needed.

An easy way to **right justify** the cout output is with the width() manipulator. Simply add the highest number of digits beginning output in-between the parentheses:

```
cout.width(10)
cout << 543 << end 1;
cout.width(10)
cout << 12345678 << end 1;
cout.width(10)
cout << 746782 << end 1;
```

**Output:**


One difference between width() and precision() is that precision() will stay throughout the program unless otherwise changed and width() must always be reentered.

### Input Options:

If the cin requests more than one answer on one line, the user has three different options for placing the information, but each answer must be separated by a space. The user can type an answer and hit the space bar, tab key or return key before entering the next answer.

# Operators & More...

## Operators

C++ has a large number of operators to assist in the simplification of code. Operators are read in order of precedence. If two operators are of the same precedence, the file reads from left to right. The following is a full list of all operators in order of precedence:

**Precedence Level**

| Symbol | Description |
|---|---|
| **1** | |
| :: | C++ scope access / resolution |
| **2** | |
| ( ) | Function call |
| [ ] | Array subscript |
| -> | C++ indirect component selector |
| . | C++ direct component selector |
| **3 Unary** | |
| ! | Logical negation |
| ~ | Bitwise (1's) complement |
| + | Unary plus |
| - | Unary minus |
| & | Address of |
| * | Indirection |
| sizeof | Returns size of operand in bytes |
| new | Dynamically allocates C++ storage |
| delete | Dynamically deallocates C++ storage |
| type | Typecast |
| **4 Member Access** | |
| .* | C++ class member dereference |
| ->* | C++ class member dereference |
| () | Expression parentheses |
| **5 Multiplicative** | |
| * | Multiply |
| / | Divide |
| % | Remainder (modulus) |
| **6 Additive** | |
| + | Binary plus |
| - | Binary minus |
| **7 Shift** | |
| << | Left shift |
| >> | Right shift |
| **8 Relational** | |
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |
| **9 Equality** | |
| == | Equal to |
| != | Not equal to |
| **10** | |
| & | Bitwise AND |
| **11** | |
| ^ | Bitwise XOR |
| **12** | |
| \| | Bitwise OR |
| **13** | |
| && | Logical AND |
| **14** | |
| \|\| | Logical OR |
| **15 Ternary** | |
| ?: | Conditional |
| **16 Assignment** | |
| = | Simple assignment |
| *= | Compound assignment product |
| /= | Compound assignment quotient |
| %= | Compound assignment remainder |
| += | Compound assignment sum |
| -= | Compound assignment difference |
| &= | Compound assignment Bitwise AND |
| ^= | Compound assignment Bitwise XOR |
| \|= | Compound assignment Bitwise OR |
| <<= | Compound assignment left shift |
| >>= | Compound assignment right shift |
| **17 Comma** | |
| , | Sequence point |

### Code Snippet:

```
int age = 5;
age += (-5+11)/6+2;
cout << "In three years I will be
" << age << endl;
```

### Output:

```
Stdin/Stdout/Stderr
In three years I will be 8.
```

Following are some of the most useful operators:

### Unary and Binary Operators

Unary operators define a single value while binary operators operate on two values. For example:

```
int number = -1
int number2 = 1 - 2
```

In the first line of this code, the unary "-" makes "1" a negative number. In the second line of code, the value of variable `number2` uses a binary operator "-" to find the sum of two numbers.

### Assignment Operator

The "=" sign is the assignment operator. As shown in other sections, it assigns a value or expression to a variable. Multiple assignments can be used to shorten code. For example:

```
age1 = 21;
age2 = 21;
age3 = 21;
```

or

```
age1 = age2 = age3 = 21;
```

*Note: The above example is an interesting way to shorten code, but code is still more understandable when avoiding multiple assignments.*

Compound assignment operators are designed to simplify the adjustment of variables during program execution. A compound assignment operator takes the variable on the left and subjects it to the value on the right, giving a new value for the variable.

For example:

```
b = b + 100;
```

or

```
b += 100;
```

Both statements above produce the same answer.

### Adding and Subtracting One

The increment and decrement operators are similar to compound assignment operators. The variable is subject to an increase or decrease of 1. For example:

```
int q = 17;
++q;
```

or

```
int q = 17;
q++;
```

or

```
int q = 17;
q += 1;
```

or

```
int q = 17;
q = q + 1;
```

All four examples will produce the same answer, 18.

### Typecast Operators

Typecast operators change a variable from one data type to another. A typecast operator is an existing data type with the keyword `static_cast`. For example:

```
static_cast<int> (q)
```

When placed in C++ code, the operator looks like this:

```
int q = 17;
int answer;
float c = 8.7;
answer = q + static_cast<int> (c);
```

In the above code, the value for variable `answer` must be an integer. To ensure this, the author typecasted float `c`, which will remove all decimal places. Truncate is the term used for removing or eliminating part of an answer.

### The sizeof() Operator

sizeof() is not a function. It is an operator designed to determine how many memory bytes are needed to hold the value of a variable. For example:

```
memoryHeld = sizeof(float);
```

This line will hold the amount of memory required for a float.

### Logical Operators

Logical operators (i.e. `&&`, `||`, and `!`) add capabilities to `if` statements (next column). These operators combine the actions of two or more relations. For example:

```
if (age < 5 || age > 10)
```

## if / else command

All programming languages have `if` statements. In C++, `if` statements are designed to test operators. The answer from an `if` statement will determine which part of the program to execute next. Most `if` statements are based on relational test such as:

```
if (int x < 22)
  {a block of one or more C++
  statements here}
```

*Note: Indenting the block of statements in the above example is not necessary, but it is easier to find* `if` *statements this way.*

The `else` statement will tell the program what to do when an `if` statement is false. Add the `else` statement after the `if` statement's closing blocks like this:

```
if (int x < 22)
  {a block of one or more C++
  statements here}
else
  {a block of one or more C++
  statements here}
```

`if/else` statements only have one or two options (true or false). Nesting statements make the program choose between three or more potential options. Nesting statements means placing a statement inside of a statement.

### Code Snippet:

```
void main()
{
int age;
cout << "How old are you?";
cin >> age;
if (age <= 10)
  {
  if (age == 7)
   cout << "You are 7 years old.";
  else
   cout << "You are younger than
   10, but not 7 years old.";
  }
else
  {
  cout << "You are older than 10
  years old.";
  }
}
```

### Output:

```
Stdin/Stdout/Stderr
How old are you?
7
You are 7 years old.
```

## Arrays

An array is a defined number of memory slots for a variable's value. For example:

```
char yourName[4] = "Don";
```

When the array is placed in memory it looks like this:

```
[0]  D
[1]  o
[2]  n
[3]  \0
```

*Note: Make sure space is left for the null zero (page 1).*
*Note: After defining these memory slots, the value cannot exceed the array, otherwise the extra digits or characters will be removed.*

When using an array, it is easy to edit a letter of the expression. If Don's name was Dan, the editing code would look like this:

```
yourName[1] = a;
```

This code will edit the letter in the `[1]` spot of memory held by `yourName`.

When defining an array, if the expression is known when defining the variable it is not necessary to place a number between the array subscripts, but the expression must be assigned at that time. If not, the program assumes that the array is nothing.

# Statements & Loops

## switch Statements

`if/else` statements are best used with C++ code which must choose between two options. In an earlier section (page 2), nested `if/else` statements were discussed. The problems with `if/else` statements are:

1. The more nesting is used, the closer the code gets to the right margin.
2. Changing an extensive nested `if/else` statement is not easy.

This is where switch statements come in handy. A switch statement works the same as an `if/else` statement, by testing values using relationship operators. The main differences are:

1. `Switch` statements only search for one matching answer. Once the answer is found, the program stops looking.
2. If the program has no matching answer, the default statement is used. *The default statement is added by the user.*
3. When editing the statements, it is easier to make changes to `switch` statements when compared to nested `if/else` statements.
4. A `switch` is controlled by just one integer or character value instead of a logical test.

Here is an example of a switch statement:

```
void main()
{
int age;
cout << "How old are you?";
cin >> age;
switch (age)
{
  case 1 :
   {
   cout << "You are 1 year old.";
   break;
   }
  case 2 :
   {
   cout << "You are 2 years old.";
   break;
   }
  case 3 :
   {
   cout << "You are 3 years old.";
   break;
   }
  default :
   {
   cout << "You are older than
   3 years old.";
   }
 }
}
```

**Output:**

```
Stdin/Stdout/Stderr
How old are you?
2
You are 2 years old.
```

*Note: If the programmer sets up a* `switch` *statement with upper case characters, the user must type upper case characters into the* `cin` *prompt.* `Switch` *does not perform upper and lower case conversions.*

### The break Statement

In the above example the break command appeared at the end of every case. This causes the program to go to the end of the switch statement. If the break was not there, the output would be:

Assuming the user typed a 1:

```
You are 1 year old.
You are 2 years old.
You are 3 years old.
You are older than 3 years old.
```

Assuming the user typed a 3:

```
You are 3 years old.
You are older than 3 years old.
```

## Loops

A loop is the repeated execution of the same set of programming instructions. To stop the repeat, a variable must be added, otherwise the program will loop infinitely. The `count` variable in a control relationship is the standard variable used to stop a loop.

### The while Loop

A `while Loop` uses a relationship test to stop it from looping. Once the relationship is false, the loop is ended.

The following example prints Quick Study 5, times:

```
int count = 0;
while (count < 5)
  {
   cout << "Quick Study" << endl;
   count++;
  }
```

The fifth line of code in the above section is very important. Each time the loop repeats, '1' will be added to the variable `count`. When `count` reaches 5, the loop will stop. The '++' is the increment operator (discussed on page 2).

### The do – while Loop

Use the `do - while Loop` when the body of the loop must repeat at least once during execution. An iteration is a cycle throughout a body of the loop. The difference between a `while Loop` and a `do - while Loop` is where they test their control relationship. The `while Loop` tests at the beginning and the `do - while Loop` tests at the end of the code body.

The following is an example of a `do - while Loop`:

```
// FILENAME: DOWHILE.CPP
// An example of a do - while loop.
#include <iostream>
using namespace std;
void main()
{
float height;
do
  { cout << "How tall are you?";
   cin >> height;
   if ((height < 3.5) || (height >
9.5))
    {cout << "You can't be that
    size!" <<endl;
    cout << "Tell me the
    truth...";
  } while ((height < 3.5) ||
(height >
   9.5));
if (height < 4.5)
  { cout << "You must be at least
   4.5 feet tall to go on this
   ride." << endl;
   cout << "Sorry!" << endl;
  }
else
  { cout << "Enjoy the ride.";
  }
return;
}
```

**Output:**

```
Stdin/Stdout/Stderr
How tall are you?
16.5
You can't be that size!
Tell me the truth...

How tall are you?
.5
You can't be that size!
Tell me the truth...

How tall are you?
6.5
Enjoy the ride.
```


`endl` stands for end line. This command, placed at the end of a `cout`, forces a hard return when the program is executed.

### for Loop

`for Loops` are more complicated than the `while Loops`. To control a single `for Loop` the programmer needs three expressions. The following is the format of a `for Loop`:

```
for (startExpression; conditional;
countExpression)
  {
  // Block of one or more C++
    statements.
  }
```

*Note: Semicolons end all statements. It is required for* `for` *statements to use them to split the three expressions. All three expression are actually statements inside the* `for` *statement.*

**Code Snippet:**

```
for (down = 5; down >= 1; down--)
  {
  cout << down << endl;
  }
```

**Output:**

```
Stdin/Stdout/Stderr
5
4
3
2
1
```

The `for Loop` statement is not much different than that of the `while Loop` statements. The *startExpression* is executed before the loop begins, and the *countExpression* is done at the end of the body before testing the loop condition again.

### Nested Loops

Nested loops are similar to any nested statement. The nested statement is controlled by the outer statement. This adds power to the loop. An example of a nested loop is a car's tripmeter. Each set of numbers on the tripmeter represents a nested loop. The loop below will add just like a tripmeter:

```
for (sand = 0; sand <= 9; sand++)
  {for (dred = 0; dred <= 9; dred++)
    {for (ten = 0; ten <= 9; ten++)
      {for (mile = 0; mile <= 9;
       mile++)
       {for (tenth = 0; tenth <=
        9; tenth++)
         {cout << sand << dred <<
          ten << tenth << endl; }
       }
      }
    }
  }
```

### break Command

The break command will stop a loop before it would normally end. A break command only works on loops and switch statements. An example using a break command was shown in the first column of this page.

*Note: The exit() function will stop the program wherever the exit() is. To use the exit() function, the STDLIB.H file must be included.*

### return Command

The return command will stop a function before it would normally end. An example using a return command was shown in the second column of this page.

### continue Command

The continue command is the opposite of a break command. The continue command jumps back to the loop's start, skipping the rest of the statements in the loop body. In the following example the rae line will never execute, but the loop will repeat five times:

```
for (up = 0; up >= 5; up++)
  {
   cout << "doe" << endl;
   continue;
   cout << "rae" << endl;
  }
```

QuickStudy

# Functions & Advanced

## Functions

Functions are a good way to break down a program. C++ was designed to create large programs from little programs. Each function should be a self-contained mini-program. Mini-programs aren't necessary, but will help the organization of programs.

*Note: A structured program is set up with a single function for every task.*

The `main()` function, in a perfect program, should only be a starting or controlling function for all other functions.

All new functions must be defined during the function in which it will be used. When a function is used in another it is referred to as "calling". When C++ calls a function, the new function gains control until the code has been read, then the original function regains control until the next function is called.

*Note: Calling functions could create the same problem as an infinite loop. If one function calls itself or if two functions call each other, that is called recursion. Recursion may cause the program to never end.*

Some pre-made functions need special `#include` lines at the top of the program. For example, `strcopy()` needs this statement:

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

### Local and Global Variables

Any variable can be placed inside any function in any program. When a variable is defined inside a function, it is considered a local variable. A local variable only exists while its defining function's block exists. A function block is the code between a function's brackets. This is an example of a local variable:

```
void main()
{
int money = 5;
}
```

Integer money is a local variable to function main(). After the second bracket, function main ends, and integer money stops existing.

Global variables are defined after a function ends and before the next function begins. Usually, a global variable is defined before the main() function. These functions will exist from their defining point to the end of the program. They can be used by any function during the program. This is an example of a global variable:

```
int money = 5;
void main()
{
}
```

Global variables are very visible to all functions, whether the functions need the variable or not. A local variable places functions on a need-to-know basis.

### Code Example:

```
#include <iostream.h>
int drinkAge = 21;
void main()
{
int yourAge = 15;
cout << "You are" << yourAge <<
"and not old enough to drink," <<
drinkAge << "." <<endl;
  {
  int momAge = 38;
  cout << "Your mom is" << momAge
  << ". She is old enough to
  drink" << drinkAge << "." <<
  endl;
  }
{
```

### Output:


## Sharing Variables

Data or values can be shared (passed) between two functions. The value being passed is called an argument. The receiving variable is called the parameter. To define the passed value in a new function, the parameters must be placed, in parentheses, on the new function's definition line. The definition line is the first line of the function. This is an example of value passing:

```
FindMe (p4, p5);
// ...and later on in the program
void FindMe (p4, p5)
{
```

The first line of the above example calls for the FindMe function and passes two values. Line two separates two functions. Lines three and four are the first two lines of the FindMe function.

All functions in C++ must have a prototype. A prototype is what declares a function. The prototype is usually placed at the top of the code, before main().

The `#include` line at the beginning of all programs is a header file. These header files are prototypes for library functions, such as, `strcpy()`.

There are three different ways to pass values and expressions from one function to the next:

1. by value
2. by address
3. by reference

### Passing by Value

In this form of passing, the value is passed on to the next function, but the variable is not. If any changes occur to the value once it has been passed, the changes will not affect the original variable. The receiving function looks at this value as if it were a local value.

### Passing by Address

Passing by address means that the entire variable is moved from one function to the next. An address is a variable's location in memory. Address passing is most useful when passing an array. The address held by the array will also move.

### Passing by Reference

This form of value passing is designed to pass non-arrays. Reference passing works the same as address passing except reference passing doesn't work with arrays.

### Code Example:

```
#include <iostream>
using namespace std;
void GetValue(int height);
void GetAdd(char name[10]);
void GetRef(int &age);
void main()
{
int height;
char name[10];
int age;
cout << "How tall are you?";
cin >> height;
cout << "What is your first
name?";
cin >> name;
cout << "How old are you?";
cin >> age;
GetValue(height);
cout << "In six years you may
still be" << height << "." <<
endl;
GetAdd(name);
cout << "In six years you will
change your name to" << name <<
"." << endl;
GetRef(&age);
cout << "In six years you will be"
<< age << "years old." << endl;
return;
}

void GetValue(int height)
{
height += 3;
cout << "If you grow a half an
```

## Advanced Functions

### Return Values

The return statement was discussed earlier in this guide. The return statement not only stops a function prematurely, but can also take a value. It is possible to pass more than one value from one function to another, but it is not possible to return more than one value from a function.

The data type must be placed in the prototype of all returning functions. If there is no value, the program assumes it is an integer. If there is no returning value, the function prototype must start with void.

Here is an example of a return value:

```
int RemoveAge(int age);
void main()
{
  int ageDif = 0;
  int age;
  cout << "How old are you?";
  cin >> age;
  ageDif = RemoveAge(age);
  cout << "You have << ageDif
       << "until you are 100
       << "years old." << endl;
  return;
}

int RemoveAge(int age)
{
  return (100 - age);
}
```

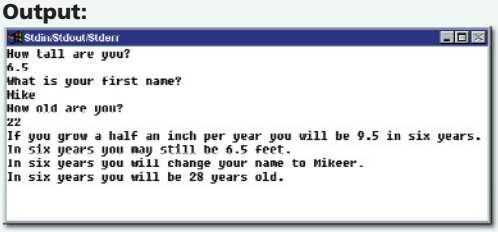
Line 1 in the prototype for the function `RemoveAge`. Line 8 calls the function. Line 14 defines the new function. Line 17 calculates the result and uses return to send the value back to `main`.

```
inch a year, you will be" << height
<< "in six years." <<endl;
return;
}

void GetAdd(char name[10])
{
name[4] = 'e';
name[5] = 'r';
name[6] = '\0'
return;
}

void GetRef(int &age)
{
age += 6;
return;
}
```

### Output:




ISBN 142320008-X

9 781423 200086

50395

PRICE

U.S. $3.95
CAN $5.95

NOV 2006

6 54614 00008 8