C Programming Language
INTRODUCTION

Introduction

https://en.wikipedia.org/wiki/C_(programming_language)

C (/ˈsiː/) is a general-purpose, imperative computer programming language. C was originally developed by Dennis Ritchie between 1969 and 1973 at Bell Labs and used to re-implement the Unix operating system

The C Language is developed for creating system applications that interacts directly to the hardware devices such as drivers, kernels etc. C programming is considered as the base for other programming languages, that is why it is known as ***mother language***.

Structure of the program

A ***computer program*** is a sequence of instructions that tell the computer what to do.

Statements and expressions

The most common type of instruction in a program is the ***statement***. A statement in C is the smallest independent unit in the language. In human language, it is analogous to a sentence. We write sentences in order to convey an idea. In C, we write statements in order to convey to the compiler that we want to perform a task.

Statements in C are terminated by a ***semicolon***.

There are many different kinds of statements in C. The following are some of the most common types of simple statements:

```
int y;
```

This is a ***declaration statement***. It tells the compiler that *y* is a ***variable***. You may remember variables from algebra in school. Variables serve the same purpose here: to provide a name for a value that can vary. All variables in a program must be declared before they are used. We will talk more about variables shortly.

```
y = 24;
```

This is an ***assignment statement***. It assigns a value (24) to a variable (*y*).

```
printf("%d",y);
```

This is an ***output statement***. It outputs the value of *y* (which we set to 24 in the previous statement) to the screen. Format "%d" means that we print integer value in ***decimal*** notation.

The compiler is also capable of resolving expressions. An expression is a mathematical entity that evaluates to a value. For example, in math, the expression 4 + 6 evaluates to the value 10. Expressions can involve values (such as 4), variables (such as *y*), operators (such as +) and functions (which return an output value based on some input value). They can be singular (such as 4, or *y*), or compound (such as 4 + 6, 4 + *x*, *x* + *y*, or (2 + *x*) * (*y* – 3)).

For example, the statement *y* = 4 + 6; is a valid assignment statement. The expression 4 + 6 evaluates to the value of 10. This value of 10 is then assigned to *y*.

What is the difference between a statement and an expression?

- A ***statement*** is a "complete sentence" that tells the compiler to perform a particular task. Statement: *y* = 4 + 6;
- An ***expression*** is a mathematical entity that evaluates to a value. Expressions are often used inside of statements. Expression: 4 + 6.

Functions

In C statements are typically grouped into units called ***functions***. A ***function*** is a collection of statements that executes sequentially. Every C program must contain a special function called **main**. When the C program is run, execution starts with the first statement inside of function **main**. Functions are typically written to do a very specific job. For example, a function named "**max**" might contain statements that figures out which of two numbers is larger. A function named "**calculateGrade**" might calculate a student's grade.

Libraries and the C Standard Library

A ***library*** is a collection of precompiled code (e.g. functions) that has been "packaged up" for reuse in many different programs. Libraries provide a common way to extend what your programs can do. For example, if you were writing a game, you'd probably want to include a sound library and a graphics library.

The C core language is actually very small and minimalistic. However, C also comes with a library called the C standard library that provides additional functionality for your use. One of the most commonly used parts of the C standard library is the ***stdio.h*** (STanDart Input Output) library, which contains functionality for writing to the screen and getting input from a console user.

What is the difference between a function and a library?

- A ***function*** is a collection of statements that executes sequentially, typically designed to perform a specific job.
- A ***library*** is a collection of functions packaged for use in multiple programs.

Taking a look at a sample program

Now that you have a brief understanding of what statements, functions, and libraries are, let's look at a simple "Hello World!" program:

```
#include <stdio.h>

int main(void)
{
  printf("Hello World!\n");
  return 0;
}
```

Line 1 is a special type of statement called a ***preprocessor directive***. Preprocessor directives tell the compiler to perform a special task. In this case, we are telling the compiler that we would like to add the contents of the ***stdio.h*** header to our program. The ***stdio.h*** header allows us to access functionality in the ***stdio.h*** library, which will allow us to write to the screen.

Line 2 has nothing on it, and is ignored by the compiler.

Line 3 declares the **main**() function, which as you learned above, is mandatory. Every program must have a **main**() function.

Lines 4 and 7 tell the compiler which lines are part of the main function. Everything between the opening curly brace on line 4 and the closing curly brace on line 7 is considered part of the **main**() function.

Line 5 is our first statement (you can tell it's a statement because it ends with a semicolon), and it is an output statement. Function ***printf*** prints the string "Hello World!" on the screen. The character '\n' means a new line character.

Line 6 is a new type of statement, called a ***return statement***. When an executable program finishes running, the **main**() function sends a value back to the operating system that indicates whether it was run successfully or not.

This particular ***return*** statement returns the value of 0 to the operating system, which means "everything went okay!". Non-zero numbers are typically used to indicate that something went wrong, and the program had to abort.

All of the programs we write will follow this template, or a variation on it.

What symbol do statements in C end with?

- The semicolon (;)

Try the next example:

```
#include <stdio.h>

int main(void)
{
  printf("Hello\n Wor\nld!\n");
  return 0;
}
```

Syntax and syntax errors

In English, sentences are constructed according to specific grammatical rules that you probably learned in English class in school. For example, normal sentences end in a period. The rules that govern how sentences are constructed in a language is called syntax. If you forget the period and run two sentences together, this is a violation of the English language syntax.

C has a syntax too: rules about how your programs must be constructed in order to be considered valid. When you compile your program, the compiler is responsible for making sure your program follows the basic syntax of the C language. If you violate a rule, the compiler will complain when you try to compile your program, and issue you a syntax error.

For example, you learned above that statements must end in a semicolon.

Let's see what happens if we omit the semicolon in the following program:

```
#include <stdio.h>

int main(void)
{
  printf("Hello World!\n")
  return 0;
}
```

Visual studio produces the following error:

```
d:\microsoft visual studio 8\work\ex\ex\ex.cpp(6) : error C2143: syntax error : missing ';' before 'return'
```

This is telling you that you have an syntax error on line 6: You've forgotten a semicolon before the return. In this case, the error is actually at the end of line 5. Often, the compiler will pinpoint the exact line where the syntax error occurs for you. However, sometimes it doesn't notice until the next line.

Syntax errors are common when writing a program. Fortunately, they're often easily fixable. The program can only be fully compiled (and executed) once all syntax errors are resolved.

What is a syntax error?

A ***syntax error*** is a compiler error that occurs at compile-time when your program violates the grammar rules of the C language.

E-OLYMP 1024. Hello World! Print the message "Hello World!".

► Use the program given above.

E-OLYMP 990. Hello World! Print the digits 1, 2, 3, 4, 5, each in a separate line.

► Use '\n' to separate the digits.

E-OLYMP 5133. abc In the first line print one letter ***a***. In the second line print two letters ***b***. In the third line print three letters ***c***.

► Use '\n' to separate the lines.

Comments

A ***comment*** is a line (or multiple lines) of text that are inserted into the source code to explain what the code is doing. In C there are two kinds of comments.

The // symbol begins a C single-line comment, which tells the compiler to ignore everything to the end of the line. For example:

```
#include <stdio.h>

int main(void)
{
  printf("Hello World!\n"); // this is a comment
  return 0; // return statement
}
```

The `/*` and `*/` pair of symbols denotes a C-style multi-line comment. Everything in between the symbols is ignored.

```
#include <stdio.h>

int main(void)
{
  printf("Hello World!\n");
  /* This is a multi-line comment.
     This line will be ignored.
     So will this one. */
  return 0;
}
```

A first look at variables, initialization and assignment

Variables

A statement such as $x = 5$; seems obvious enough. As you would guess, we are assigning the value of 5 to *x*. But what exactly is *x*? *x* is a ***variable***.

A ***variable*** in C is a name for a piece of memory that can be used to store information. You can think of a variable as a mailbox, or a cubbyhole, where we can put and retrieve information. All computers have memory, called RAM (random access memory), that is available for programs to use. When a variable is defined, a piece of that memory is set aside for the variable.

A ***computer's memory*** is a contiguous sequence of slots, or ***memory cells***. Similar to how a street address can be used to find a given house on a street, the memory address allows us to find and access the contents of memory at a particular location. All types of data – whole numbers, floating point numbers, strings of characters, boolean values – can be stored in memory cells.

A ***variable*** is a named ***memory cell***. We call it a variable because the value in the cell can change. Memory cells can be of different sizes: 8 bits, 16 bits, 32 bits, 64 bits. Different sized cells are used to store different ***types*** of data.

Basic types

The C programming language provides the programmer with a set of *data types* for storing information and building up data types that are not part of the language itself. The former data types are called ***built-in types***, and the latter are called ***user-defined types***.

The three basic ***built-in types*** are: Characters, Integer numbers and Floating-point numbers.

The integer data types come in two flavors – ***signed*** and ***unsigned*** – which permit the programmer to specify values greater than and less than zero. In other words, positive and negative numbers. All of these basic built-in types have a specific *size* (amount of memory required) and range of values that they can represent. The bigger the numbers, the more memory is required.

The smallest unit of memory is a binary digit (***bit***), which can hold a value of 0 or 1. You can think of a bit as being like a traditional light switch – either the light is off (0), or it is on (1). Perhaps surprisingly, in modern computers, each bit does not get its own address. The smallest addressable unit of memory is a group of 8 bits known as a ***byte***.

Type	*Description*
`int`	integer, 4 bytes
`long long`	integer, 8 bytes
`float`	real, 4 bytes
`double`	real, 8 bytes
`char`	character, 1 byte

Variable declaration

All the variables that a program is going to use must be ***declared*** prior to use. Declaration of a variable serves two purposes:

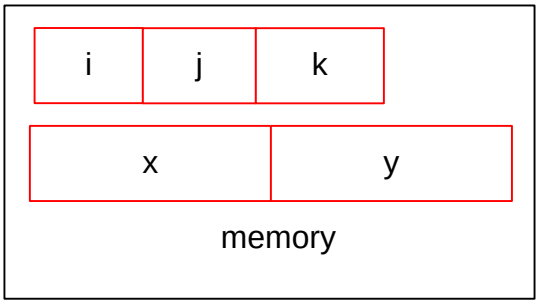
- It associates a type and an identifier (or name) with the variable. The type allows the compiler to interpret statements correctly. For example in the CPU the instruction to add two integer values together is different from the instruction to add two floating-point values together. Hence the compiler must know the type of the variables so it can generate the correct add instruction.
- It allows the compiler to decide how much *storage space* to allocate for storage of the value associated with the identifier and to assign an address for each variable which can be used in code generation.

A variable declaration has the form:

```
type identifier-list;
```

type specifies the type of the variables being declared. The identifier-list is a list of the identifiers of the variables being declared, separated by commas.

```
int i, j, k;
double x, y;
```



Input and output with scanf / printf

The ***standard input stream*** (**stdin**) is the default source of data for applications. In most systems, it is usually directed by default to the ***keyboard***.

The ***standard output stream*** (**stdout**) is the default destination of output for applications. In most systems, it is usually directed by default to the ***text console*** (generally, on the ***screen***).

printf() is an I/O function that prints the ***formatted*** data to the **stdout**. This function returns the total number of characters, but on failure a negative value is returned.

```
int printf ( const char * format, ... );
```

If format includes format specifiers (subsequences beginning with %), the additional arguments following format are formatted and inserted in the resulting string replacing their respective specifiers.

Type description	Type	Format
integer, 4 bytes	int	%d
integer, 8 bytes	long long	%lld
real, 4 bytes	float	%f
real, 8 bytes	double	%lf
character, 1 byte	char	%c
array of chars	char[], string	%s

scanf() is an I/O function which is used to read the **stdin** to store the data in the variables pointed by the argument list. The function returns the number of items read on success, it can be zero if a matching failure occurs.

```
int scanf(const char *format,..);
```

Read and print one number:

```
#include <stdio.h>

int x;

int main(void)
{
  scanf("%d",&x);
  printf("%d\n",x);
  return 0;
}
```

Find the sum of two integers:

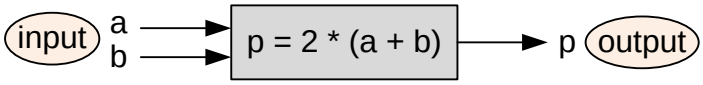
```
#include <stdio.h>

int a, b, res;

int main(void)
{
  scanf("%d %d",&a,&b);
  res = a + b;
  printf("%d + %d = %d\n", a, b, res);
  return 0;
}
```

E-OLYMP **7943. Perimeter of rectangle** Find the perimeter of rectangle with integer sides *a* and *b*.

► Write a program according to the next steps:

1. Read the values of *a* and *b*;
2. Find the perimeter according to formula: $p = 2 * (a + b)$;

3. Print the value of perimeter *p*.



E-OLYMP 7944. Area of rectangle Find the area of rectangle with integer sides *a* and *b*.

► Use formula *area* = *a* * *b* for area of rectangle.

Operation	C notation	Sample
integer division	/	26 / 10 = 2
remainder, mod	%	26 % 10 = 6

E-OLYMP 5175. The last digit Find the last digit of a positive integer *n*.

► Use formula *d* = *n* % 10 to find the last digit *d* of number *n*.

Data type	range	format
int	$[-2^{31}; 2^{31} - 1]$	%d
long long	$[-2^{63}; 2^{63} - 1]$	%lld

Note that

- $2^{31} - 1 = 2\ 147\ 483\ 647 \approx 2 * 10^9$;
- $2^{63} - 1 = 9\ 223\ 372\ 036\ 854\ 775\ 807 \approx 9 * 10^{18}$;

E-OLYMP 519. Sum of squares Find the sum of squares of two integers *a* and *b*. The numbers do not exceed 10^9 by absolute value.

► We must find the value *res* = *a* * *a* + *b* * *b*. We know that *a*, $b \le 10^9$, so *a* * *a* + *b* * *b* $\le 10^{18}$. We must use ***long long*** data type in this problem.

sizeof operator

The *sizeof* is a keyword, but it is a compile-time operator that determines the *size*, in bytes, of a variable or data type. The *sizeof* operator can be used to get the size of classes, structures, unions and any other user defined data type.

The syntax of using *sizeof* is as follows:

```
sizeof(data type)
```

or

```
sizeof(variable)
```

The next program prints the sizes of built-in data types:

```c
#include <stdio.h>

int main(void)
{
  printf("%d\n",sizeof(int));        // 4
  printf("%d\n",sizeof(long long)); // 8
  printf("%d\n",sizeof(float));      // 4
  printf("%d\n",sizeof(double));     // 8
  printf("%d\n",sizeof(char));      // 1
  return 0;
}
```

The next program prints the sizes of variables:

```c
#include <stdio.h>

int i, j, k;
double x, y;
char c, d;

int main(void)
{
  printf("%d %d %d\n",sizeof(i),sizeof(j),sizeof(k));
  printf("%d %d\n",sizeof(x),sizeof(y));
  printf("%d %d\n",sizeof(c),sizeof(d));
  return 0;
}
```

l-values and *r*-values

In C, variables are a type of ***l-value*** (pronounced ell-value). An *l*-value is a value that has an address (in memory). Since all variables have addresses, all variables are *l*-values. The name *l*-value came about because *l*-values are the only values that can be on the ***left side*** of an assignment statement. When we do an assignment, the left hand side of the assignment operator must be an *l*-value. Consequently, a statement like 5 = 6; will cause a compile error, because 5 is not an *l*-value. The value of 5 has no memory, and thus nothing can be assigned to it. 5 means 5, and its value can not be reassigned. When an l-value has a value assigned to it, the current value at that memory address is overwritten.

The opposite of *l*-values are ***r-values*** (pronounced arr-values). An *r*-value refers to any value that can be assigned to an *l*-value. *r*-values are always evaluated to produce a single value. Examples of *r*-values are single numbers (such as 5, which evaluates to 5), variables (such as *x*, which evaluates to whatever value was last assigned to it), or expressions (such as 2 + *x*, which evaluates to the value of *x* plus 2).

Here is an example of some assignment statements, showing how the *r*-values evaluate:

```
#include <stdio.h>

int main(void)
{
  int y;      // define y as an integer variable
  y = 4;      // 4 evaluates to 4, which is then assigned to y
  y = 2 + 5; // 2 + 5 evaluates to 7, which is then assigned to y

  int x;      // define x as an integer variable
  x = y;      // y evaluates to 7 (from before),
              // which is then assigned to x.
  x = x;    // x evaluates to 7, which is then assigned to x (useless!)
  x = x + 1;  // x + 1 evaluates to 8, which is then assigned to x.

  printf("%d %d\n", x, y);
  return 0;
}
```

Initialization vs assignment

C supports two related concepts that new programmers often get mixed up: ***assignment*** and ***initialization***.

After a variable is defined, a value may be assigned to it via the assignment operator (the '=' sign):

```
int x; // this is a variable definition
x = 5; // assign the value 5 to variable x
```

C will let you both define a variable AND give it an initial value in the same step. This is called initialization.

```
int x = 5; // initialize variable x with the value 5
```

A variable can only be initialized when it is defined.

Uninitialized variables

Unlike some programming languages, C does not initialize variables to a given value (such as zero) automatically (for performance reasons). Thus when a variable is assigned to a memory location by the compiler, the default value of that variable is whatever garbage happens to already be in that memory location! A variable that has not been assigned a value is called ***an uninitialized variable***.

Note: Some compilers, such as Visual Studio, will initialize the contents of memory when you're using a debug build configuration. This will not happen when using a release build configuration.

Uninitialized variables can lead to interesting (and by interesting, we mean unexpected) results. Consider the following short program:

```c
#include <stdio.h>

int x;

int main(void)
{
  int y;
  printf("%d %d\n",x,y);
  return 0;
}
```

Using uninitialized variables is one of the most common mistakes that novice programmers make, and unfortunately, it can also be one of the most challenging to debug (because the program may run fine anyway if the uninitialized value happened to get assigned to a spot of memory that had a reasonable value in it, like 0).

Keywords and naming identifiers

Keywords

C reserves a set of words for its own use. These words are called ***keywords***, and each of these keywords has a special meaning within the C language.

For example, the keywords are: **goto**, **static**, **sizeof**, **do**, **double**, **void** **virtual**, **unsigned**.

Along with a set of operators, these keywords define the entire language of C (preprocessor commands excluded). Because these keywords have special meaning, your IDEs will change the text color of these words (usually to blue) to make them more visible.

Identifiers

The name of a variable, function, class, or other object in C is called an ***identifier***. C gives you a lot of flexibility to name identifiers as you wish. However, there are a few rules that must be followed when naming identifiers:

- The identifier can not be a keyword. Keywords are reserved.
- The identifier can only be composed of letters (lower or upper case), numbers, and the underscore character. That means the name can not contain symbols (except the underscore) nor whitespace.
- The identifier must begin with a letter (lower or upper case) or an underscore. It can not start with a number.
- C distinguishes between lower and upper case letters. **nvalue** is different than **nValue** is different than **NVALUE**.

In the next program we have two different variables _x1 (lower-case letter ’*x*’) and _X1 (upper-case letter ’*x*’).

```c
#include <stdio.h>

int _x1, _X1;

int main(void)
{
  _x1 = 3;
  _X1 = _x1 + 2;
  printf("%d %d\n", _x1, _X1);
  return 0;
}
```

A first look at operators

An ***expression*** is a combination of literals, variables, functions, and operators that evaluates to a value.

Literals

A ***literal*** is a fixed value that has been inserted (hardcoded) directly into the source code, such as 5, 3.14159 or 'a'. Literals always evaluate to themselves, and have no representation in memory. Here's an example that uses literals:

```c
#include <stdio.h>

int main(void)
{
  int x = 2;  // x is a variable, 2 is a literal
  printf("%d\n", 3 + 4); // 3 + 4 is an expression,
                         // 3 and 4 are literals
  printf("Hello World!\n"); // "Hello, world\n" is a literal too
  return 0;
}
```

Literals, variables, and functions are all known as ***operands***. Operands supply the data that the expression works with. We just introduced literals, which evaluate to themselves. Variables evaluate to the values they hold. Functions evaluate to produce a value of the function's return type (unless the return type is void).

Operators

Operators tell the expression how to combine one or more operands to produce a new result. For example, in the expression "3 + 4", the + is the ***plus operator***. The + operator tells how to combine the operands 3 and 4 to produce a new value (7).

You are likely already quite familiar with standard arithmetic operators from common usage in math, including ***addition*** (+), ***subtraction*** (-), ***multiplication*** (*), and ***division*** (/). ***Assignment*** (=) is an operator as well. Some operators use more than one symbol, such as the ***equality operator*** (==), which allows us to compare two values to see if they are equal.

Note: One of the most common mistakes the new programmers make is to confuse the assignment operator (=) with the equality operator (==). Assignment (=) is used to assign a value to a variable. Equality (==) is used to test whether two operands are equal in value.

```
#include <stdio.h>

int main(void)
{
  int x;
  x = 10; // assignment operator
  if (x == 10) printf("x = 10\n"); // equality operator
  return 0;
}
```

Operators come in three types:

Unary operators act on one operand. An example of a unary operator is the - operator. In the expression -5, the - operator is only being applied to one operand (5) to produce a new value (-5).

Binary operators act on two operands (known as left and right). An example of a binary operator is the + operator. In the expression 3 + 4, the + operator is working with a left operand (3) and a right operand (4) to produce a new value (7).

Ternary operators act on three operands. There is only one of these in C, which we'll cover later.

Also note that some operators have more than one meaning. For example, the - operator has two contexts. It can be used in unary form to invert a number's sign (e.g. to convert 5 to -5, or vice versa), or it can be used in binary form to do arithmetic subtraction (e.g. 4 – 3).

Increment ++ and decrement -- operators

The increment operator ++ increases the value of a variable by 1. Similarly, the decrement operator -- decreases the value of a variable by 1.

- If you use the ++ operator as **prefix** like: *++var*. The value of *var* is incremented by 1 and then it returns the value.
- If you use the ++ operator as **postfix** like: *var++*. The original value of *var* is returned first and then *var* is incremented by 1.

```
#include <stdio.h>

int main(void)
{
  int a = 2, b = 6;
  printf("%d %d\n",++a, b++);  // 3 6
  printf("%d %d\n", a, b);     // 3 7
  printf("%d %d\n", --a, b--); // 2 7
```

```
  printf("%d %d\n", a, b);      // 2 6
  return 0;
}
```

E-olymp Problems to solve:
www.e-olymp.com

Main contest

Print the data:

- 1024. Hello World!
- 990. 12345
- 5133. abc

Simple math formula (build an expression for answer):

- 4716. Divide the apples – 1
- 4717. Divide the apples – 2
- 7401. Stepan Friends
- 7943. Perimeter of rectangle
- 7944. Area of rectangle
- 1286. Tuck-shop

Working with digits:

- 1. Simple problem?
- 5175. The last digit
- 906. Product of digits
- 939. The square of sum
- 943. Swap the digits
- 945. Without the middle
- 949. Two digits from four digits

long long type:

- 519. Sum of squares
- 8809. Marathon
- 8810. School concert
- 8811. Product of two integers
- 2860. Sum of integers on the interval

Math problems:

- 248. Young gardener
- 8254. Hotel rooms
- 9405. Professor and baloons
- 9406. Professor and batteries

Additional contest

Print the data:

- 8800. Hello, Python!

Simple math formula:

- 8801. Next number
- 8802. Previous number
- 8806. Number of students
- 8807. Opposite number
- 8813. Surface area and volume
- 8815. Surface area and volume 2
- 8824. Find the number
- 8837. Quotent and remainder

Working with digits:

953. Remainder
955. The square of sum
959. Sum of digits
8599. Digits of 3-digit number
8600. Sum and product 2
8601. Swap the digits in two-digit integer
8602. Third from right
8603. Sum and product 3
8607. Sum of squares of digits
8638. Append three
8855. Find the number 1

long long type:

7491. Integer

QUIZ

1. What is the output of following program?

```c
#include <stdio.h>

int main(void)
{
  int x = printf("Hello World!");
  printf("%d", x);
  return 0;
}
```

2. What is the output of following program?

```c
#include <stdio.h>

int main()
{
  printf("%d", printf("%d", 1234));
  return 0;
}
```

3. What is the output of following program?

```c
#include <stdio.h>

int main()
{
  int a = 10;
  int b = 15;

  printf("=%d", (a + 1), (b = a + 2));
  printf(" %d=", b);
  return 0;
}
```

4. What is the output of following program?

```c
#include <stdio.h>

int main()
{
  int a = 10;
  printf("%d\n", 2 + a++);
  printf("%d\n", 2 + ++a);
  return 0;
}
```

5. What is the output of following program?

```c
#include <stdio.h>

int main()
{
 int a, b, c;
 printf("%d\n", scanf("%d %d %d",&a,&b,&c));
 return 0;
}
```

6. What is the output of following program?

```c
#include <stdio.h>

int main()
{
  printf("50%% + 50%% = 100%%\n");
  return 0;
}
```

7. What is the output of following program?

```c
#include <stdio.h>

int main()
{
  int x = 1234;
  printf("%d", printf("%d", printf("%d", x)));
  return 0;
}
```

8. What is the output of following program?

```c
#include <stdio.h>

int main()
{
  int i = 5;
  printf("%d %d %d", i, ++i, i++);
  return 0;
}
```

9. What is the output of following program?

```c
#include <stdio.h>
```

```c
int main()
{
 int i = 3;
 printf("%d\n", i++ + ++i);
 return 0;
}
```

10. What is the output of following program?

```c
#include <stdio.h>

int a, b;

int main(void)
{
  a = 3; b = 5;
  a = a + b;
  b = a + b;
  printf("%d\n", a + b);
  return 0;
}
```

11. What is the output of following program?

```c
#include <stdio.h>

int a, b;

int main(void)
{
  a = 5; b = 3;
  a = a * b / 4;
  b = b * a % 5;
  printf("%d\n", a + b);
  return 0;
}
```

HINTS

1. The printf function returns the number of characters successfully printed on the screen. The string "Hello World!" has 12 characters, so the first printf prints Hello World! and returns 12.

answer: Hello World!12

2. The printf function returns the number of characters successfully printed on the screen.

answer: 12344

3. All the arguments of printf() are evaluated irrespective of whether they get printed or not. That's why (*b*=*a*+2) would also be evaluated and value of *b* would be 12 after first printf().

answer: =11 12=

4. First printf statement prints 2 + 10 as output. Then ++ will increment the *a* value, *a* becomes 11.

Second printf statement prints 2 + 12 as output. First increment *a*, then use *a* in expression 2 + *a*.

answer:

12

14

5. scanf returns the number of inputs it has successfully read.

answer: 3

6. we can print “%” using “%%”

7.

answer: 123441

8. Depends on compiler

answer: 7 7 5 (my compiler, evaluates from right to left)

9. Depends on compiler. If you modify a variable more than one time in a single statement the behavior is undefined according to the C standard.

As usually, evaluation runs from left to right.

First summand equals to 3, then *i* is increased by 1 and becomes 4.

Second summand: *i* is increased by 1 first (*i* = 5) and then used in expression

answer: 3+ 5 = 8

10. ***answer***: 21

```
a = 3; b = 5;
a = a + b; // a = 8, b = 5
b = a + b; // a = 8, b = 13
printf("%d\n", a + b); // a + b = 21
```

11. ***answer***: 7

```
a = 5; b = 3;
a = a * b / 4; // a = 3, b = 3
b = b * a % 5; // a = 3, b = 4
printf("%d\n", a + b); // a + b = 7
```