C Programming Language

Compendium

ANRICH TAIT

Written 2023

Contents

1 TODO Introduction

1.1 Example Template:

```
#include <stdio.h>

int main(){
  printf("This is a template!");
}
```

Output:

```
This is a template!
```

---

2 Output (printf):

To output values or print text in c, you can use the **printf()** function.

```
#include <stdio.h>

int main(){
  printf("hello, world");
}
```

Output:

```
hello, world
```

There is no limit to the amount of one function you can put in a program. In the case of the printf function keep in mind that using multiple **printf()** functions will not ouput a new line. See below:

```
#include <stdio.h>

int main(){
  printf("hello, world");
  printf("I am learning C!");
}
```

Output:

```
hello, worldI am learning C!
```

---

3 Basic escape sequences:

Escape sequences are used to do various things, like adding new lines and adding characters that the compiler would normally conflict with. See below:

ESCAPE SEQUENCE	DESCRIPTION
⁀	Creates a horizontal tab
\	Inserts a backslash character (\)
\"	Inserts a double quote character

3.1 Insert a new line:

To insert a new line we need to use the character.

```
#include <stdio.h>

int main(){
  printf("hello, world \n");
  printf("I am learning C! \n");

  // You can also use \n characters to create blank lines:
  printf("Do you see? \n\n");
  printf("YES/NO");
}
```

Output:

```
hello, world
I am learning C!
Do you see?

YES/NO
```

---

4 Comments:

Comments are used to explain code and make it more readable. It is also a good way to make sure the compiler doesn't execute a block of code when you are testing changes. Comments can be single-lined or multi-lined.

4.1 Single-lined comments:

```
#include <stdio.h>

int main(){
  printf("This is an example of a single line comment.");
//  printf("I hope this doesn't get executed.");
}
```

Output:

```
This is an example of a single line comment.
```

4.2 Multi-lined comments:

```
#include <stdio.h>

int main(){
  printf("This is an example of a multi-lined comment.");
  /*
    printf("This won't be executed.") <-- No syntax errors in comments
    printf("Found a bug? Maybe just comment the code out.");
    */
}
```

Output:

```
This is an example of a multi-lined comment.
```

---

5 Data types:

Data types in c refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The different C data types are classified as follows:

TYPE	DESCRIPTION
Basic types	Arithmetic types that are further classified into: a: Integer types b: Floating-point numbers
Enumerated types	Arithmetic types that are used to define variables that can only assign certain discrete integer values throughout the program
Void type	Indicates that no value is available
Derived types	They include: a: Pointer types b: Array types c: Structure types d: Union types e: Function types

The array types and structur etypes are reffered collectively as the aggregate types. The type of a function specifies the type of the functions return value. See the basic types in the following sections.

5.1 Basic Data types (quick reference):

In the following sub-sections detailed descriptions and uses for each data type are given. The table below is a quick reference for all the basic types and their format specifiers:

DATA TYPE	DESCRIPTION	FORMAT SPECIFIER
int	Stores whole numbers without decimals	%d or %i
float	Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits.	%f
double	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits.	%lf
char	Stores a single character/letter/number or ASCII value	%c
string	Stores two ore more characters	%s

5.2 Set decimal precision:

If you print a floating point number it will output many digits after the decimal point. If you want to remove the extra decimals (set decimal precision), you can use a dot (.) followed by a number that specifies how many digits that should be shown after the decimal point:

```
#include <stdio.h>

int main(){
  float myFloat = 3.5;
  printf("%f\n", myFloat); //Will show 6 digits after decimal point.
  printf("%.1f\n", myFloat); // Only show 1 digit.
  printf("%.2f\n", myFloat); //Only show 2 digits.
  printf("%.4f", myFloat); //Only show 4 digits.
}
```

Output:

```
3.500000
3.5
3.50
3.5000
```

5.3 Type conversion:

Sometimes you have to convert the value of one data type to another. This is known as type conversion: For example if you divide two integers, 5 and 2. You would expect the result to be 2.5 but since they are integers (and not floating-point) numbers the output will be 2.

```
#include <stdio.h>

int main(){
  int x = 5;
  int y = 2;
  int sum = 5 / 2;

  printf("%d", sum); // outputs 2
  return 0;
}
```

Output:

```
2
```

To get the right result, you need to know how type conversion works. There are two types of conversion in C:

- Implicit Conversion (automatically)
- Explicity Conversion (manually)

5.3.1 Implicit Conversion:

Implicit conversion is done automatically by the compiler when you assign a value of one type to another. For example if you assign int to a float type:

```
#include <stdio.h>

int main(){
  // Automatic conversion: int to float
  float myFloat = 9;

  printf("%f", myFloat);
  return 0;
}
```

Output:

```
9.0
```

As you can see the compiler automatically converts the int value 9 to a float value 9.0. This may be risky due to you losing control over specific values in certain situations. Especially when it is the other way around - the following example automatically converts the float value 9.99 to an int value of 9:

```
#include <stdio.h>

int main(){
  // automatic conversion: float to int
  int myInt = 9.99;
  printf("%d", myInt);
  return 0;
}
```

Output:

```
9
```

See how the output is just 9. For some programs that extra .99 may be necessary (most likley it is).

As another example, if you divide two integers: 5 by 2, you know the sum result should be 2.5. Like previously mentioned if you store the sum as an integer, the result will only display the number 2, therefore it would be better to store the sum as a float or a double (right?).

```
#include <stdio.h>

int main(){
  float sum = 5 / 2;
  printf("%f", sum);
}
```

Output:

```
2.0
```

As you can see the result is 2.0 not 2.5. This is because 5 and 2 are still integers in the division. In this case you will need to manually convert the integer values to floating-point values. For the we use Explicit Conversion.

5.3.2 Explicit Conversion:

Explicit conversion is done manually by placing the type in parentheses () in front of the value.

Considering our problem from the example above, we can now get the right result:

```
#include <stdio.h>

int main(){
  //manual conversion: int to float
  float sum = (float) 5 / 2;
  printf("%f", sum); //2.5
}
```

Output:

```
2.5
```

You can also place the type in front of the variable:

```
#include <stdio.h>

int main(){
  int num1 = 5;
  int num2 = 2;
  float sum = (float) num1 / num2;

  printf("%f", sum); //2.5
}
```

Output:

```
2.5
```

5.4 Integer Types:

The following table provides the details of standard integer tyoes with their storage size and value ranges:

TYPE	STORAGE SIZE	VALUE RANGE
char	1 byte	-128 to 127 or 0 to 225
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,768 or -2,147,483,648 to 2,147,483,647
unsigned it	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes or (4bytes for 32 bit OS)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

5.5 Floating-point numbers:

The following table provides the details of standard floating-point numbers with storage sizes and value ranges and their precision:

TYPE	STORAGE SIZE	VALUE RANGE	PRECISION
float	4 bytes	1.2E-38 to 3.4E+38	6 decimal places
double	8 bytes	2.3E-308 to 1.7E+308	15 decimal places
long double	10 bytes	3.4E-4932 to 1.1E+4932	19 decimal places

The header file <float.h> defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs.

5.6 Void Types:

The void type specifies that no value is availabe. It is used in three situtations:

TYPE	DESCRIPTION / USE
Functions returns as void	There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example: **void exit(int status);**
Function arguments as void	There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example: **int rand(void);**
Pointers to void	A pointer of type void * represents the address of an object, but not it's type. For example: a memory allocation function void *malloc(size_t size); returns a pointer to void which can be casted to any data type.

5.7 Derived types:

The derived data types are basically derived out of the fundamental datatypes. A derived data type won't typically create a new data type – but would add various new functionalities to the existing ones instead.

We can derive the derived data types out of the primitive data type by adding some extra relationships to the elements that are available with the primitive data types. We use the derived data types to represent multiple values as well as single values in a program.

Below are the types of derived data types and their uses:

TYPE	DECRIPTION / USE
arrays	refers to a sequence (ordered sequence) of a finite number of data items from the same data type sharing one common name
function	refers to a self-contained block of single or multiple statements. It has it's own specified name.
pointers	refers to a some special form of variables that one can use for holding other variables' addresses.
structures	A collection of various different types of data type items that get stored in a contagious type of memory allocation is known as structure in C.

6 Variables:

Variables are containers for storing data values, like numbers and characters. Like mentioned previously there are different **data types** in C. These data types are used to declare variable types.

6.1 Syntax:

To create a variable specify the type and assign it a value.

type	variableName	=	value;
Example:			
int	myNum	=	32;

The above example will create an integer variable called myNum with the value of 15.

You can also declare a variable without assigning the value and assign it later.

```
#include <stdio.h>

int main(){
  int myNum; //declare the variable

  myNum = 32; //assign a value to it
}
```

6.2 Output variables:

To output variables in C you must use "format specifiers": These are used together with the **printf()** function to tell the compiler what data type the variable is. A format specifier starts with the % sign followed by a character. The most common format specifiers are listed below:

Format specifiers	Type of output
%d or %i	A decimal inter or signed integer
%c	Signed character
%f	Signed float
%e	A floating-point number
%s	A string or sequence of characters
%lf	double
%Lf	Long double
%o	Octal integer
%u	Short unsigned integer
%ld	Long decimal integer
%x	Hexadecimal integer
%p	Print memory address in the hexadecimal form

The below example uses the formula °C = (5/9)(°F - 32) to print a table of Fahrenheit temperatures and their centigrade or Celsius equivalents:

```
#include <stdio.h>

int main(){
  int fahr, celsius;
  int lower, upper, step;

  lower = 0; //lower limit of temperature table
  upper = 300; //upper limit
  step = 20; //step size

  fahr = lower;
  while (fahr <= upper) {
  celsius = 5 * (fahr-32) /9;
  printf("%d\t%d\n", fahr, celsius);
  fahr = fahr + step;
  }
}
```

Output:

```
0 -17
20 -6
40 4
60 15
80 26
100 37
120 48
140 60
160 71
180 82
200 93
220 104
240 115
260 126
280 137
300 148
```

The program consists of the definition of a single function names main. The program is longer than the one that printed "hello, world" but is no complicated. It makes use of several core concepts:

- comments
- declarations
- variables
- arithmetic expressions

- loops
- formatted output

In C all variables must be declared before they are used, this is usually done at the beginning of a function before any executable statements. A declaration announces the properties of variables; it consists of a type name and a list of variables such as:

```
int fahr, celsius;
int lower, upper, step;
```

Computation in the temperature conversion program begins with the assignment statements:

```
lower = 0;
upper = 300;
step = 20;
fahr = lower;
```

This sets the variables to their initial values. Individual statements are terminated by semicolons. Each line of the table is computed the same way, so we use a loop that repeats once per output line; this is the purpose of the while loop:

```
while (fahr <= upper) {
  ...
    }
```

The while loop operates as follows: The condition in parentheses is tested. If it is true (fahr is than or equal to upper0, the body of the loop (the three statements enclosed in braces) is executed. Then the condition is re-tested and if true the body is executed again. When the test becomes false (fahr exceeds upper) the loop ends, and excecution continues at the statement that follows the loop. There are no further statments in this program, so it terminates.

The body of a while can be one or more statements enclosed in braces, as in the temperature converter, or a single statemnt without braces, as in:

```
while (i < j)
  i = 2 * j
```

In either case, we will always indent the statments controlled by the while by one tab stop so you can see at a glance which statements are inside the loop. The indentation emphasizes the logical structure of the program. ALthough C compilers do not care about how a program looks, proper indentation and spacing are essential to building clear and read-able code.

Most of the work get's done in the body of the loop. The celsius temperature is computed and assigned to the variable celsius by the statement:

```
celsius = 5 * (fahr-32) / 9;
```

The reason for multiplying by 5 and then dividing by 9 instead of just multiplying by 5/9 is that in C integer division truncates: and fractional part is discareded. Since 5 and 9 are integers, 5/9 would be truncated to zero and so all the Celsius temperatures would be reported as zero.

The above example is also a good represenation of the basic functionality of 'printf'. 'printf' is a general purpose output formatting function. It's first argument is a string of characters to be printed, with each '%' indicating where one of the other (second, third, ...) arguments is to be substituted, and in what form it is to be printed. For instance, %d specifies an integer argument so the statement:

```
printf("%d\t%d", fahr, celsius);
```

causes the values of the two integers fahr and celsius to be printed, with a tab ($\widehat{()}$) between them. Each % construction in the first argument of 'printf' is paired with the corresponding second argument, third argument, etc. They must match up properly by number and type, or you get wrong answers.

There are a couple of probelms with the temperature conversion program. The simple one is that the output isn't very pretty because the numbers are not right-justified. This can be fixed by augmenting each '%d' in the printf statement with a width, the numbers printed will be right-justified in their fields. For instance:

```
printf("%3d, %6d\n", fahr, celsius);
```

This will print the first number of each line in a field three digits wide, and the second in a field six digits wide.

The more serious issue is that integer arithmetic was used, this means that the celsius temperatures are not very accurate, for example: 0°F is actually about -17.8°C not -17. To get more accurate answers 'floating-point arithmetic' should be used. With those improvements in mind here is the second version of the program:

```
#include <stdio.h>

int main(){
  float fahr, celsius;
  int lower, upper, step;

  lower = 0; // lower limit of temp table
```

```
  upper = 300;// upper limit
  step = 20; // step size

  fahr = lower;
  while (fahr <= upper) {
    celsius = (5.0/9.0) * (fahr-32.0);
    printf("%3.0f %6.1f\n", fahr, celsius);
    fahr = fahr + step;
    }
}
```

Output:

```
  0  -17.8
 20   -6.7
 40    4.4
 60   15.6
 80   26.7
100   37.8
120   48.9
140   60.0
160   71.1
180   82.2
200   93.3
220  104.4
240  115.6
260  126.7
280  137.8
300  148.9
```

The changes are:

- fahr and celsius are now declared to be float
- the formula for conversion is written in a more natural way
- 5/9 would truncate to zero (a decimal point in a constant indicates that it is floating point) so 5.0/9.0 was used and will not truncate because it is the ratio of two floating-point values.

If an arithmetic operator has integer operands, an integer operation is performed. If an arithmetic operator has one floating-point operand and one integer operand, however, the integer will be converted to floating point before the operation is done. If fahr-32 was written it would be converted to a floating-point number, but writing floating-point contstants with explicit decimal points even when they have integral values emphasizes their floating-point nature for human readers.

Notice that the assignment:

```
fahr = lower;
```

and the test

```
while (fahr <= upper)
```

also work in the natural way-the int is converted to float before the operation is done. The printf conversion specification %3.0f says that a floating-point number (here fahr) is to be printed at least three characters wide, with no decimal point and no fraction digits. %6.1f describes another number (celsius) that is to be printed at least six characters wide, with 1 digit after the decimal point. Witdth and precision may be omitted from a specification: %6f says that the number is to be atleast six characters wide, %.2f specifies two characters after the decimal point, but the width is no constrained; and %f merely says to print the number as floating point.

```
%d   //print as decimal integer
%6d  // print as decimal integer, at least 6 chars wide
%f   //print as floating point
%f6f // print as floating point, at least 6 chars wide
%.2f //print as floating point, 2 chars after decimal point
%6.2f//print as floating point, at least 6 wide and 2 after decimal point
```

6.2.1 %d (Decimal Integer):

```
#include <stdio.h>

int main(){
  int a=50;
  printf("The integer value of a is %d \n",a);
  return 0;
}
```

6.2.2 %c (Character):

```
#include <stdio.h>

int main(){
  char myChar = 'a';

  printf("The first letter of the alphabet is: %c", myChar);
  return 0;
}
```

Output:

```
The first letter of the alphabet is: a
```

6.2.3 %f (Floating Point):

```
#include <stdio.h>

int main(){
  float a = 3;
  printf("The floating point of a is %f \n", a);
  return 0;
}
```

Output:

```
The floating point of a is 3.000000
```

6.2.4 %e (Floating Pointer Number):

```
#include <stdio.h>

int main(){
  float a = 12.5;
  printf("The floating-point of a is %e\n", a);
  return 0;
}
```

Output:

```
The floating-point of a is 1.250000e+01
```

6.2.5 %s (String):

```
#include <stdio.h>

int main(){
  char s[15] = "String";
  printf("The string value of s is %s\n", s);
  return 0;
}
```

Output:

```
The string value of s is String
```

6.2.6 %lf (Double):

```
#include <stdio.h>

int main(){
  double d = 12.5;
  printf("The double value of d is %lf\n", d);
  return 0;
}
```

Output:

```
The double value of d is 12.500000
```

6.2.7 %o (octal integer):

```
#include <stdio.h>

int main(){
  int oct = 11;
  printf("The octal integer value of oct is %o\n", oct);
  return 0;
}
```

Output:

```
The octal integer value of oct is 13
```

6.2.8 %x (Hexadecimal Integer):

```
#include <stdio.h>

int main(){
  int h = 14;
  printf("The hexadecimal value of h is %x\n", h);
  return 0;
}
```

Output:

```
The hexadecimal value of h is e
```

6.2.9 %p (Prints Memory Address):

To find the memory address that holds values of a variable, we use the %p format specifier and it prints in hexadecimal form.

```
#include <stdio.h>

int main(){
  int sum = 0;
  printf("The memory address of sum is %p\n", &sum);
  return 0;
}
```

Output:

```
The memory address of sum is 0x7ffdf9e40ce4
```

6.3 Changing variable values:

Note: if you assign values to an existing variable, it will overwrite the previous value.

```
#include <stdio.h>

int main(){
  int myNum = 15; //myNum is 15
  myNum = 10;     //myNum is now 10

  printf("%d", myNum);
  return 0;
}
```

Output:

```
10
```

You can also assign the value of one variable to another:

```
#include <stdio.h>

int main(){
  int myNum = 15;
  int myNumTwo = 23;

  //Assign the value of myNumTwo (23) to myNum
  myNum = myNumTwo;

  printf("myNum= %d\n", myNum);
```

```
  printf("myNumTwo= %d", myNumTwo);
  return 0;
}
```

Output:

```
myNum= 23
myNumTwo= 23
```

6.4 Add variables together:

To add variables together use the + operator:

```
#include <stdio.h>

int main(){
  int x = 5;
  int y = 6;
  int sum = x + y;
  printf("%d", sum);
  return 0;
}
```

Output:

```
11
```

6.5 Declare multiple variables:

To declare more than one variable of the same type use a comma-seperated list:

```
#include <stdio.h>

int main(){
  int x = 5, y = 6, z = 50;
  printf("%d", x + y + z);
  return 0;
}
```

Output:

```
61
```

You can also assign the same value to multiple variables of the same type:

```
#include <stdio.h>

int main(){
  int x, y, z;
  x = y = z = 50;
  printf("%d", x + y + z);
  return 0;
}
```

Output:

```
150
```

6.6 Variable names:

All C variables must be identified with unique names. These unique names are called **identifiers**. Identifiers can be short names (like x and y) or more descriptive names (ages, sum, totalVolume). **Note:** It is recommended to use descriptive names in order to create understandable and maintainable code:

The general rules for naming variables are:

- Names can contain letters, digits and underscores.
- Names must begin with a letter or an underscore(_).
- Names are case sensitive.
- Names cannot contain whitespaces or special characters like !, #, %, etc.
- Reserved words (such as int) cannot be used as names.

6.7 Real life example:

```
  #include <stdio.h>

int main(){
// Student data
int studentID = 15;
int studentAge = 23;
float studentFee = 75.25;
char studentGrade = 'B';

// Print variables
printf("Student id: %d\n", studentID);
printf("Student age: %d\n", studentAge);
printf("Student fee: %f\n", studentFee);
printf("Student grade: %c", studentGrade);
  }
```

---

7 Constants:

To prevent a variable from being changed you can use the **const** keyword. This will declare the variable as "constant", which means unchangeable and read-only:

```
#include <stdio.h>

int main(){
  const int myNum = 15; //myNum will always be 15
  myNum = 10; //Attempting to change the variable will output an error.
}
```

You should always declare the variable as constant when you have values that are unlikely to change:

```
#include <stdio.h>

int main(){
  const int minutesPerHour = 60;
  const float PI = 3.14;

  printf("Minutes per hour: %i\n", minutesPerHour);
  printf("PI: %f", PI);
}
```

Output:

```
Minutes per hour: 60
PI: 3.140000
```

7.1 Things to note:

When you declare a constant variable, it must be assigned with a value: **Like this:**

```
#include <stdio.h>

int main(){
  const int minutesPerHour = 60
}
```

This will not work:

```
#include <stdio.h>

int main(){
  const int minutesPerHour;
  minutesPerHour = 60; //error
}
```

Another thing to note about constants is that it is considered good practice to declare them in CAPS. It isn't required but makes your code more readable and ensures that constant variables are easily distinguishable.

```
#include <stdio.h>

int main(){
  const int BIRTHYEAR = 2023;
}
```

---

8 Operators:

Operators are used to perform operations on variables and values. In the example below, the + operator is used to add two values together.

```
#include <stdio.h>

int main(){
  int myNum = 1+2;
  printf("%i", myNum);
  return 0;
}
```

Output:

```
3
```

Although the + operator is often used to add together two values like in the example above it can also be used to add together a variable and a value or a variable and another variable:

```
#include <stdio.h>

int main(){
  int sum1 = 100 + 50; //150
  int sum2 = sum1 + 250; //400
  int sum3 = sum2 + sum2; //800?

  printf("%i", sum3);
}
```

Output:

```
800
```

8.1 Operator groups:

C divides operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators

8.1.1 Arithmetic operators:

Arithmetic operators are used to perfrom common mathematical operations.

OPERATOR	NAME	DESCRIPTION	EXAMPLE
"+"	addition	Adds two values	x + y
"-"	subtraction	Subtracts one value from another	x - y
"*"	multiplication	Multiplies two values	x * y
"/"	division	Divides one value by another	x / y
%	modulus	Returns the division remained	x % y
++	increment	Increases the value of a variable by 1	++x
–	decrement	Decreases the value of a variable by 1	–x

8.1.2 Assignment operators:

Assignment operators are used to assign values to variables.

OPERATOR	EXAMPLE	SAME AS
"="	x = 5	x =5
"+="	x += 3	x = x + 3
"-="	x -= 3	x = x - 3
"*="	x *= 3	x = x * 3
"/="	x /= 3	x = x / 3
"%="	x %= 3	x = x % 3
"&="	x &= 3	x = x & 3
"^="	x ^=3	x = x ^ 3
"»="	x »= 3	x = x » 3
"«="	x «= 3	x = x « 3

8.1.3 Comparison operators:

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

OPERATOR	NAME	EXAMPLE
"=="	Equal to	x == y
"!="	Not equal	x != y
">"	Greater than	x > y
"<"	Less than	x < y
">="	Greater than or equal to	x >= y
"<="	Less than or equal to	x <= y

The return value of a comparison is either 1 or 0, which means true(1) or false(0). These values are known as **Boolean values**. The boolean concept is better explained in the **booleans and if. . . else** sections.

8.1.4 Logical operators:

You can also test for true or false with logical operators. Logical operators are used to determine the logic between variables or values:

OPERATOR	NAME	DESCRIPTION	EXAMPLE
"&&"	Logical and	Returns true if both statements are true	x < 5 && x < 10
"!"	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < (10)

8.1.5 Sizeof operator:

The memory size (in bytes) of a data type or a variable can be found with the **sizeof** operator:

```
#include <stdio.h>

int main(){
  int myInt;
  float myFloat;
  double myDouble;
  char myChar;

  printf("%lu\n", sizeof(myInt));
  printf("%lu\n", sizeof(myFloat));
  printf("%lu\n", sizeof(myDouble));
  printf("%lu\n", sizeof(myChar));

}
```

Output:

```
4
4
8
1
```

Note: the **%lu** format specifier is used to print the result, instead of &d. This is because the compiler expects the sizeof operator to return a **long unsigned int** (%lu) instead of int (%d).

9 Booleans:

Very often in programming you will need a data type that can only have one of two values, like:

- YES / NO
- ON / OFF
- True / FALSE

For this, C has a bool data type, which is known as booleans. Booleans represent data values that are either true or false.

9.1 Boolean variables:

In C, the bool type is not a built-in data type, like int or char. It was introduced in C99 and must be imported with the following header file to be used:

```
#include <stdbool.h>
```

A boolean variable is declared with the **bool** keyword and can only take the values true or false:

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

int main(){
  bool isProgrammingFun = true;
  bool isFishTasty = false;

  printf("%d\n", isProgrammingFun); //Returns 1 (true)
  printf("%d", isFishTasty);      //Returns 0 (false)
}
```

Output:

```
1
0
```

Note: boolean values are returned as integers:

- 1 (or any number that isn't 0) represents true
- 0 represents false

Therefore, you must use the %d format specifier to print a boolean variable.

9.2 Comparing Values and Variables:

Comparing values can be very useful in programming, because it helps us find answers and make decisions. For example, you can use a comparison operator, such as greater than (>) to compare two values:

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

int main(){
  printf("%d", 10 > 9); //Returns 1 (true) because 10 is greater than 9
}
```

Output:

```
1
```

In the same way you can compare two variables:

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

int main(){
  int x = 10;
  int y = 9;
  printf("%d", x > y);
}
```

Output:

```
1
```

In the example below, the equal to (==) operator is used to compare different values:

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

int main(){
  printf("%d", 10 == 10); //Returns 1 because 10 is equal to  10
  printf("%d", 10 == 15); //Returns 0 because 10 is not equal to 15
}
```

Output:

```
10
```

You are not limited to only compare numbers, You can compare variables or even special structures like arrays:

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

int main(){
  bool isProgrammingFun = true;
  bool isProgrammingTasty = false;

  //find out if both statements are true:
  printf("%d", isProgrammingFun == isProgrammingTasty);
}
```

Output:

```
0
```

9.3 Real life example:

This example checks if a person is old enough to vote.

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

int main(){
  int myAge = 22;
  int votingAge = 18;

  if (myAge >= votingAge) {
  printf("Old enough to vote!");
  } else {
  printf("Not old enough to vote.");
    }
}
```

Output:

```
Old enough to vote!
```

---

10 if Statements:

In C there are several types of **'if'** statements that are used to control the flow of execution in a program. The main difference between the different types of 'if' statements is in their syntax and how they are used in different situations. Here are the different types of 'if' statements and their syntax:

10.1 Conditions and if statements:

In previous sections the use of logical conditions were mentioned:

- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b
- Equal to: a == b
- Not equal to: a != b

In C programming, conditional statements are used to perfrom different actions based on different conditions. C has the following conditional statements:

- Use **if** to specify a block of code to be executed if a specified condition is **true**
- Use **else** to specify a block of code to be executed, if the same condition is **false**
- Use **else if** to specify a new condition to test, if the first condition is **false**
- Use **switch** to specify many alternative blocks of code to be executed

10.2 if Statement:

The "**if**" statement is used for making decisions based on certain conditions. It allows a program to execute different statements based on whether a specified condition is true or false.

10.2.1 Syntax and examples:

```
if (condition) {
  //code to be executed if the condition is true
}
```

The **"condition"** is an expression that is evaluated to either true or false. If the **"condition"** is true then the code inside the curly braces is executed, otherwise the code is skipped. For example:

```
#include <stdio.h>

int main(){
  int x = 5;
  if (x > 3) {
    printf("x is greater than 3");
  }
}
```

Output:

```
x is greater than 3
```

In this example, the **"if"** statement check if "**x**" is greater than 3. Since **x** is indeed greater than 3, the message "x is greater than 3" is printed in the output.

10.3 if . . . else Statement:

The **'if else'** statement is used for making decisions based on certain conditions. It allows a program to execute different statements based on whether a specified condition is true or false. The '**if else**' statement provides an alternative execution path if the condition is false.

10.3.1 Syntax and examples:

```
if (condition) {
  //condition to be if the condition is true
} else {
  //code to be executed if the condition is false
}
```

The '**condition**' is an expression that is evaluated to either true or false. If the '**condition**' is true, then the code inside the first block of curly braces is executed, otherwise the code inside the inside the second block of curly braces is executed. For example:

```
#include <stdio.h>

int main(){
  int x = 5;
  if (x > 3 ) {
    printf("x is greater than 3");
  } else {
    printf("x is less than or equal to 3");
  }
}
```

Output:

```
x is greater than 3
```

In this example the '**if**' statements checks if '**x**' is greater than 3. Since '**x**' is indeed greater than 3, the message "x is greater than 3" is printed in the output. If '**x**' were less than or equal to 3, the message "x is less than or equal to 3" would be outputted instead.

The '**if else**' statement can also be nested, which means that an '**if else**' statement can be placed inside another '**if else**' statement. This is useful for testinng multiple conditions. The syntax for a nested '**if else**' statement is as follows:

```
if (condition) {
//code to be executed if condition is true
} else if (condition2) {
//code to be executed if condition1 is false and condition2 is true
} else {
//code to be executed if both condition1 and condition2 are false
}
```

For example:

```
#include <stdio.h>
#include <stdbool.h>
  int main() {
    int x = 5;
    int y = 2;
    if (x > 3) {
      if (y > 1) {
      printf("x is greater than 3 and y is greater than 1");
      } else {
      printf("x is greater than 3 but y is less than or equal to 1");
      }
    } else {
      printf("x is less than or equal to 3");
    }
  }
```

Output:

```
x is greater than 3 and y is greater than 1
```

In this example the out '**if**' statement checks if '**x**' is greater than 3. Since '**x**' is greater than 3, the inner '**if else**' statement is executed, which checks if '**y**' is greater than 1. Since '**y**' is greater than 1, the message "x is greater than 3 and y is greater than 1" is printed in the console. If "**y**' were less than or equal to 1, the message "x is greater than 3 but y is less than or equal to 1" would be printed instead. If '**x**' were less than or equal to 3, the message "x is less than or equal to 3" would be printed instead.

10.4 else ... if Statement:

In C the '**else if**' statement is used to test multiple conditions in sequence. It allows a program to executre different statments based on different conditions in a hierarchical manner. The '**else if**' statement is placed after an initial '**if**' statement and before the '**else**' statement.

10.4.1 Syntax and examples:

```
if (condition1) {
//code to be executed if condition1 is true
} else if (condition2) {
//code to be executed if condition1 is false and condition2 is true
} else {
//code to be executed if both condition1 and condition2 are false
}
```

In this syntax, '**condition1**' is the initial condition that is tested in the '**if**' statement. If '**condition1**' is false, the '**else if**' statement is executed and the program tests the second condition '**condition2**'. If '**condition2**' is true, then the code inside the second block of curly braces is executed, otherwise, the code inside the third block of curly braces is executed.

Here is an example of the '**else if**' statement:

```
#include <stdio.h>

int main(){
  int score = 85;

  if (score >= 90) {
  printf("You got an A");
  } else if (score >= 80) {
    printf("You got a B");
  } else if (score >= 70) {
    printf("You got a C");
  } else if (score >= 60) {
    printf("You got a D");
  } else {
    printf("You failed");
  }        }
```

Output:

```
You got a B
```

In this example, the program checks the value of the 'score' variable and prints a message depending on the score. The 'if' statements test if the score is greater than or equal to 90. If it is, the message "You got an A" is printed. If not, the

next 'else if' statement is executed, which tests if the score is greater than or equal to 80. If it is, the message "You got a B" is printed. This process repeats for each 'else if' statement until a condition is met. If none of the conditions are met, the message "you failed" is printed.

Note that in the example aboce each 'else if' statement is nested inside the previous one. This creates a sequence of conditions that are tested one after the other, allowing the program to choose the apporopriate message to print based on the value of the 'score' variable.

10.5 Short hand if ... else:

There is also a shorthand if else, which is known as the ternary operator[1] because it consists of three operands[2]. The shorthand 'if-else' statement is a compact way of writing an 'if-else' statement that consists of a single line of code for each condition. It is also known as a conditional operator.

10.5.1 Syntax:

```
(condition) ? expression1 : expression2;
```

In this syntax 'condition' is the condtition that is tested, 'expression' is the expression that is evaluated if this condition is true, and 'expression2' is the expression that is evaluated if the condition is false.

Here's an example:

```
#include <stdio.h>

int main(){
  int score = 85;
  char grade = (score >= 60) ? 'P' : 'F';
  printf("Your grade is %c", grade);
  return 0;
}
```

Output:

```
Your grade is P
```

In this example, the program checks the value of the 'score' variable and assigns the grade 'P' if the score is greater than or equal to 60, and 'F' otherwise. The shorthand 'if-else' statement is used to evaluate the condition and assign the apporopriate grade to the 'grade' variable.

Note that the shorthand 'if-else' statement is often used in situations where a simple condition needs to be tested and the result of the condition needs to be

[1] Operand: a term used to refer to a value or a variable that is operated on by an operator.

[2] Operator is a symbol that performs some operation on one or more operands to produce a result.

assigned to a variable. However, it can also be used in more complex expressions, such as nested expressions or expressions involving multiple operators.

It's important to note that the shorthand 'if-else' statement should be used judiciously, as it can make code more difficult to read and understand if used excessively. It's generally best to use the full 'if-else' statement for more complex conditions and the shorthand 'if-else' statement for simpler conditions.

11 Switch statement:

Instead of writing many 'if ... else' statements, you can use the switch statement.

The 'switch' statement is a control statement that allows the execution of different sections of code depending on the value of a variable or an expression. The 'switch' statement is commonly used when there are many possible cases to consider, and it provides a more concise and readable way to express complex conditional logic than a series of nested 'if' statements.

11.1 Syntax:

```
  switch (expression) {
   case constant1:
     //statements to be executed if expression == constant1
     break;
   case constant2:
    //statements to be executed if expression == constant2
     break;


   case constantN:
     //statements to be executed if expression == constantN
     break;
   default:
     //statements to be executed if none of the above cases are ture
     break;
}
```

In this syntax, 'expression' is the variable or expression that is being tested, and 'constant1' to 'constantN' are the values or expressions that are being compared to the value of 'expression'. If the value of 'expression' matches one of the constants, the statements inside that case block are executed. If none of the cases match, the statements inside the 'default' block are executed.

- The 'switch' expression is evaluated once
- The value of the expression is compared with the values of each 'case'
- If there is a match, the associated block of code is executed

- The 'break' statement breaks out of the switch block and stops the execution
- The 'default' statement is optional, and specifies some code to run if there is no case match.

11.2 Example:

```
#include <stdio.h>

int main(){
  char grade = 'B';

  switch (grade) {
  case 'A':
      printf("Excellent!\n");
      break;
  case 'B':
    printf("Good job!\n");
    break;
  case 'C':
    printf("Needs improvement.\n");
    break;
  case 'D':
    printf("Invalid grade.\n");
    break;
  }
  return 0;
}
```

Output:

```
Good job!
```

In this example, the program check the value of the 'grade' variable using a 'switch' statement and prints a message depending on the value of 'grade'. If the value of 'grade' is 'A', the program prints "Excellent!". If the value of 'grade' is 'B', the program prints "Good job!". If the value of 'grade' is 'C', the program prints "Needs improvement.". If the value of 'grade' is anything else, the program prints "Invalid grade.".

Note that each case block must end with a 'break' statement. This is because the 'switch' statement will continue to execute the statements in the subsequent case blocks until it encounters a 'break' statement. The 'default' block is optional and it is executed if none of the cases match. It is recommended to always include a 'default' block to handel unexpected cases.

11.2.1 'Break' keyword:

When C reaches a break keyword, it breaks out of the switch block. This will stop the execution of more code and case testing inside the block. When a

match is found and the job is done, it's time for a break. There is no need for more testing.

Note: a break can save a lot of execution time because it "ignore" the execution of all the rest of the code in the switch block.

11.2.2 'Default' keyword:

The 'default' keyword specifies some code to run if there is no case match:

```
    #include <stdio.h>

int main(){
      int day = 4;

      switch (day) {
    case 6:
      printf("Today is Saturday");
      break;
    case 7:
      printf("Today is Sunday");
      break;
    default:
      printf("Looking forward to the Weekend");
  }

  // Outputs "Looking forward to the Weekend"
    }
```

Output:

```
Looking forward to the Weekend
```

Note: The default keyword must be used as the last statement in the switch, and it does not need a break.

12 Loops:

Loops can execute a block of code as long as a specified condition is reached. Loops are handy because they save time, reduce errors, and they makde code more readable.

There are 3 types of loops in C: 'for', 'while', and 'do-while'.

12.1 for Loop:

The 'for' loop is a control flow statement that repeatedly executes a block of code as long as a particular condition is true. The syntax is as follows:

```
for (initialization; condition; increment/decrement) {
  //statment to be executed
}
```

The initialization step initializes a loop counter variable and the condition is evaluated at the beginning of each iteration. If the condition is true, the statements inside the loop are executed. After executing the statements, the increment/decrement statement is executed and the condition is evaluated again. The loop continues until the condition becomes false.

- Statement 1 is executed (one time) before the execution of the code block.
- Statement 2 defines the condition for executing the code block.
- Statement 3 is executed (every time) after the code block has been executed.

Here's an example:

```
#include <stdio.h>

int main(){
  for (int i = 0; i < 10; i++) {
      printf("%d\n", i);
      }
    }
```

Output:

```
0
1
2
3
4
5
6
7
8
9
```

Example using the fahrenheit celsius program mentioned before: Below is a variation of the termperature converter used in the table:

```
#include <stdio.h>

int main(){
  int fahr;

  for (fahr = 0; fahr <= 300; fahr = fahr + 20)
    printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Output:

```
  0  -17.8
 20   -6.7
 40    4.4
 60   15.6
 80   26.7
100   37.8
120   48.9
140   60.0
160   71.1
180   82.2
200   93.3
220  104.4
240  115.6
260  126.7
280  137.8
300  148.9
```

One of the major changes in this example compared to the one shown previously is the elimination of most of the variables, only 'fahr' remains and it was made an 'int' type. The 'lower' and 'upper' limits as well as the 'step size' appear only as constants in the 'for' statement. The expression that computes the celsius temperature now appears as the third argument of printf instead of as a seperate assignment statement.

This last change is an instance of a general rule-in any context where it is permissible to use the value of a variable of some type, you can use a more compliced expression of that type. Since the third argument of printf must be a floating-point value to math the %6.1f, any floating-point expression can occur there.

The 'for' statement is a loop, a generalization of the 'while'. If you compare it to the earlier 'while', it's operation should be clear. The first part, the initialization

```
fahr = 0
```

is done once, before the loop proper is entered. The scond part is the test or condition that controls the loop:

```
fahr <= 300
```

This condition is evaluated; if it is true, the body of the loop (here a single printf) is executed. Then the increment step:

```
fahr = fahr + 20
```

is executed, and the condition re-evaluated. The loop terminates if the condition has become false. As with the 'while', the body of the loop can be a single statement, or a group

12.2 while Loop:

```
while (condition) {
    // statements to be executed
}
```

The condition is evaluated at the beginning of each iteration. If the condition is true, the statements inside the loop are executed. After executing the statements, the condition is evaluated again. The loop continues until the condition becomes false. Here's an example:

```
#include <stdio.h>

  int main(){
int i = 0;
while (i < 10) {
printf("%d\n", i);
i++;
    }
      }
```

Output:

```
0
1
2
3
4
5
6
7
8
9
```

12.3 do-while Loop:

The do-while loop is a control flow statement that repeatedly executes a block of code as long as a particular condition is true. The syntax of the do-while loop is as follows:

```
do {
    // statements to be executed
} while (condition);
```

The statements inside the loop are executed at least once, and then the condition

is evaluated. If the condition is true, the statements inside the loop are executed again. The loop continues until the condition becomes false. Here's an example:

```
#include <stdio.h>

  int main(){
    int i = 0;
    do {
      printf("%d\n", i);
      i++;
    } while (i < 10);
  }
```

Output:

```
0
1
2
3
4
5
6
7
8
9
```

Loops are useful when you need to perform the same operation multiple times, or when you need to iterate over a collection of data. It's important to make sure that the loop condition will eventually become false, otherwise the loop will continue indefinitely, resulting in an infinite loop.

12.4 Nested Loops:

Nested loops in C are loops that are placed inside another loop. This allows you to perform more complex operations that require multiple iterations.

The basic syntax for a nested loop in C is as follows:

```
for (initialization; condition; increment/decrement) {
    for (initialization; condition; increment/decrement) {
// statements to be executed
    }
}
```

In this example, the outer loop controls the iteration of the inner loop. The statements inside the inner loop will be executed for each iteration of the outer loop.

Here's an example of a nested loop that prints out a multiplication table:

```
  #include <stdio.h>

  int main(){
    for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10; j++) {
printf("%d ", i*j);
    }
    printf("\n");
}
  }
```

Output:

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

It's important to note that nested loops can significantly increase the execution time of a program, especially if the inner loop is executed many times. Therefore, it's important to carefully consider whether a nested loop is the best solution for a particular problem, or if there is a more efficient way to achieve the same result.

13 Break and Continue:

13.1 Break:

The 'break' statement is a control statement that is used to exit a loop or switch statement. When the 'break' statement is encountered the program jumps out of the loop or switch statement, regardless of whether the loop condition or switch case condition is still true.

In a previous example the 'break' statement was used to jump out of a 'switch' statement.

Below is an example of break used to jump out of a for loop where it is used to jump out when i is equal to 4:

```
#include <stdio.h>

int main(){
  int i;

  for (i = 0; i < 10; i++) {
```

```
    if (i == 4) {
      break;
      }
    printf("%d\n", i);
    }
}
```

Output:

```
0
1
2
3
```

Here is an example of using 'break' in a 'switch' statement:

```
#include <stdio.h>

int main(){
  int day = 3;
  switch (day) {
  case 1:
      printf("Monday");
      break;
  case 2:
      printf("Tuesday");
      break;
  case 3:
      printf("Wednesday");
      break;
  default:
    printf("Invalid day");
  }
}
```

Output:

```
Wednesday
```

In this example, the switch statement evaluates the value of the variable day, and executes the appropriate case statement. When day is equal to 3, the code inside the case 3: block is executed. After the code is executed, the break statement is encountered, which causes the program to exit the switch statement and continue with the rest of the program.

In summary, the break statement is a useful tool for controlling the flow of a program, and can be used to exit loops or switch statements when certain conditions are met.

13.2 Continue:

The 'continue' statement is a control statement that is used to skip the current iteration of a loop and move on to the next iteration. When the 'continue' statement is encountered inside a loop, the program skips over any remaining statements in the loop for that iteration and goes directly to the next iteration.

Below is an example of using 'continue' in a 'for' loop:

```
  #include <stdio.h>

  int main(){
    for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
continue;
    }
    printf("%d\n", i);
}
  }
```

Output:

```
1
3
5
7
9
```

In this example, the loop will execute 10 times, with the variable i starting at 0 and increasing by 1 each time through the loop. However, when i is an even number (i.e., when i % 2 == 0 is true), the continue statement is executed, which causes the program to skip over the remaining statements in the loop for that iteration and move directly to the next iteration. This means that the printf statement is only executed for odd numbers.

Here's another example of using continue in a while loop:

```
  #include <stdio.h>

  int main(){
    int i = 0;
    while (i < 10) {
    i++;
    if (i % 2 == 0) {
continue;
    }
    printf("%d\n", i);
}
  }
```

Output:

```
1
3
5
7
9
```

In this example, the loop will continue to execute as long as i is less than 10. The variable i is incremented by 1 at the beginning of each iteration. However, when i is an even number, the continue statement is executed, which causes the program to skip over the remaining statements in the loop for that iteration and move directly to the next iteration. This means that the printf statement is only executed for odd numbers.

In summary, the continue statement is a useful tool for controlling the flow of a loop, and can be used to skip over certain iterations when certain conditions are met.

14 Arrays:

An array is a collection of elements of the same data type, stored in contiguous memory locations in C. Each element in the array is accessed using an index, which is an integer value that represents the position of the element in the array. The first element in the array has an index of 0, and the last element has an index of n-1, where n is the size of the array.

Here's an example of creating and accessing an array in C:

```
#include <stdio.h>

int main(){
 int numbers[5];   // creates an array of 5 integers

 numbers[0] = 10;  // assigns 10 to the first element of the array
 numbers[1] = 20;  // assigns 20 to the second element of the array
 numbers[2] = 30;  // assigns 30 to the third element of the array
 numbers[3] = 40;  // assigns 40 to the fourth element of the array
 numbers[4] = 50;  // assigns 50 to the fifth element of the array

 printf("%d\n", numbers[2]);   // prints the value of the third element of the array (30)

}
```

Output:

```
30
```

In this example, we create an array of 5 integers called numbers, and assign values to each element of the array. We then use the index notation numbers[2] to access the third element of the array, which has a value of 30.

Arrays can also be initialized at the time of creation, like this:

```
int numbers[5] = {10, 20, 30, 40, 50};
```

In this example, we create an array of 5 integers called numbers and initialize each element with a value.

Arrays can also be used to represent multi-dimensional data, such as matrices. In a two-dimensional array, each element is identified by two indices: a row index and a column index. The elements are stored in row-major order, meaning that the elements of each row are stored together in contiguous memory locations.

Here's an example of creating and accessing a two-dimensional array in C:

```
  #include <stdio.h>

  int main(){
    int matrix[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

printf("%d\n", matrix[1][2]);  // prints the value of the element in
       //the second row and third column (6)

  }
```

Output:

```
6
```

In this example, we create a 3x3 matrix called matrix, and initialize each element with a value. We then use the index notation matrix[1][2] to access the element in the second row and third column, which has a value of 6.

Arrays in C are a powerful tool for storing and manipulating collections of data of the same type, and are widely used in many different programming applications.

14.1 Change an Array Element:

To change the value of a specific element, refer to the index number:

```
myNumbers[0] = 33;
```

```
#include <stdio.h>

int main(){
  int myNumbers[] = {25, 50, 75, 100};
```

```
myNumbers[0] = 33;

printf("%d", myNumbers[0]);

    // Now outputs 33 instead of 25
      }
```

Output:

```
33
```

14.2 Loop through an array:

You can loop through the array elements with a 'for' loop. The following example outputs all elements in the myNumbers array:

```
  #include <stdio.h>

  int main(){
    int myNumbers[] = {25, 50, 75, 100};
    int i;

    for (i = 0; i < 4; i++) {
      printf("%d\n", myNumbers[i]);
}
  }
```

Output:

```
25
50
75
100
```

---

15 Strings:

A string is a sequence of characters that are stored in an array. A string is represented as an array of characters terminated by a null character, which is a character with the ASCII value of 0.

For example, the string "Hello, world!" would be represented in C as an array of characters:

```
#include <stdio.h>

int main(){
  char str[] = "Hello, world!";
  printf("%s", str);
}
```

Output:

```
Hello, world!
```

In the above example 'str' is an array of characters with a size of 14, which includes the 13 characters of the string plus the null character that terminates the string.

Strings in C can be manipulated using various standard library functions that are defind in the 'string.h' header file. Some of the commonly used string functions include:

- 'strlen()' - Returns the length of a string.
- 'strcpy()' - Copies a string to another string.
- 'strcat()' - Concatenates two strings.
- 'strcmp()' - Compares two strings.

Here is an example of using the 'strlen()' function:

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

int main() {
    char str[] = "Hello, world!";
    int length = strlen(str);
    printf("Length of string: %d\n", length);
    return 0;
}
```

Output:

```
Length of string: 13
```

15.1 Modify strings:

To change the value of a specific character in a string, refer to the index number, and use single quotes:

```
  #include <stdio.h>

  int main(){
    char greetings[] = "Hello World!";
    greetings[0] = 'J';
    printf("%s", greetings);
// Outputs Jello World! instead of Hello World!
  }
```

Output:

```
Jello World!
```

15.2 Loop Through a String:

In the same way you loop through an array, it can be done to loop through a string:

```
  #include <stdio.h>

  int main(){
char carName[] = "Volvo";
int i;

for (i = 0; i < 5; ++i) {
  printf("%c\n", carName[i]);
}
  }
```

Output:

```
V
o
l
v
o
```

15.3 Alternate Way of Creating Strings:

In the examples above, we used a "string literal" to create a string variable. This is the easiest way to create a string in C.

You should also note that you can create a string with a set of characters. This example will produce the same result as the example in the beginning of this page:

```
char greetings[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!', '\0'};
printf("%s", greetings);
```

Note: The '\0\ character at the end is known as a "null terminating character", This tells C that it is the end of the string.

15.4 Special Characters in String:

Because strings must be written within quotes, C will misunderstand this string, and generate an error:

```
char txt[] = "We are the so-called "Vikings" from the north.";
```

The solution to avoid this problem, is to use the backslash escape character.

The backslash (\) escape character turns special characters into string characters:

ESCAPE CHARACTER	RESULT	DESCRIPTION
\'	'	Single quote
\"	"	Double quote
\	\	Backslash
		New line
‿		Tab
\0		Null

So the correct way to type the above example would be:

```
char txt[] = "We are the so-called \"Vikings\" from the north.";
```

15.5 String functions:

In C, strings are represented as arrays of characters, terminated by a null character (\0). There are various standard library functions that are defined in the string.h header file that allow you to manipulate strings. Here are some commonly used string functions in C:

15.5.1 'strlen()'

Returns the length of a string. This function takes a string as input and returns the number of characters in the string, excluding the null character.

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

int main(){
  char str[] = "Hello, world!";
  int len = strlen(str);
  printf("The length of the string is %d\n", len);
  return 0;
}
```

Output:

```
The length of the string is 13
```

15.5.2 'strcpy()'

Copies one string to another. This function takes two strings as input, and copies the contents of the second string into the first string.

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

```
int main(){
  char src[] = "Hello, world!";
  char dest[20];
  strcpy(dest, src);
  printf("The copied string is %s\n", dest);
  return 0;
}
```

Output:

```
The copied string is Hello, world!
```

15.5.3 'strcat()'

Concatenates two strings. This function takes two strings as input, and appends the contents of the second string to the end of the first string.

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

int main(){
  char str1[20] = "Hello, ";
  char str2[] = "world!";
  strcat(str1, str2);
  printf("The concatenated string is %s\n", str1);
  return 0;
}
```

Output:

```
The concatenated string is Hello, world!
```

15.5.4 'strcmp()'

Compares two strings. This function takes two strings as input and returns an integer value that indicates the lexicographic relationship between the two strings.

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

int main(){
  char str1[] = "Hello, world!";
  char str2[] = "Hello, World!";
  int cmp = strcmp(str1, str2);
  if(cmp == 0) {
    printf("The strings are equal\n");
```

```
    } else if(cmp < 0) {
    printf("The first string is less than the second string\n");
    } else {
    printf("The first string is greater than the second string\n");
    }
}
```

Output:

```
The first string is greater than the second string
```

These are just a few of the many string functions available in C, It's important to read the documentation carefully to understand how to use each function correctly.

16 User Input:

In C, you can read user input from the keyboard using the scanf() function. This function reads input from the standard input stream (stdin) and stores the values in variables.

Here is an example of how to use scanf() to read a single integer from the user:

```
#include <stdio.h>

int main(){
    int num;
  printf("Enter a number: ");
  scanf("%d", &num);
  printf("You entered: %d\n", num);
  return 0;
}
```

In this example, the printf() function is used to prompt the user to enter a number, and the scanf() function is used to read the number from the keyboard and store it in the num variable. The format specifier %d tells scanf() to read an integer value, and the & operator is used to pass the address of the variable num to scanf().

Here is an example of how to use scanf() to read multiple values from the user:

```
#include <stdio.h>

int main(){
  int num1, num2;
  printf("Enter two numbers: ");
  scanf("%d %d", &num1, &num2);
```

```
  printf("You entered: %d and %d\n", num1, num2);
  return 0;
}
```

In this example, the scanf() function is used to read two integer values separated by a space. The values are stored in the variables num1 and num2, respectively.

Note that scanf() can be tricky to use correctly, especially when reading strings or other types of data that require more complex input formatting.

'scanf()' has some serious limitations regarding strings. One being that it considers space (whitespace, tabs, etc) as terminating characters, which means that it will only display a single word (even if you input many words). That is why when working with strings it is at times better to use 'fgets()' to read a line of text. Note that you must include the following arguments:

- name of the string variable
- sizeof ($\text{string}_{\text{name}}$)
- stdin

17 Memory Address:

In C, memory addresses are used to refer to specific locations in the computer's memory. Every variable declared in a C program has a memory address associated with it. You can think of a memory address as a unique identifier that tells the program where a particular value is stored in memory. When a variable is created in C, a memory address is assigned to the variable. The memory address is the location of where the variable is stored on the computer. When we assign a value to the variable, it is stored in this memory address.

Memory addresses are represented as hexadecimal numbers in C. For example, the memory address of a variable called x can be obtained using the & operator, like this:

```
#include <stdio.h>

int main(){
  int x = 42;
  printf("The address of x is: %p\n", &x);
}
```

Output:

```
The address of x is: 0x7ffd2d641154
```

In this example, the %p format specifier is used to print the memory address of x in hexadecimal format. The & operator is used to obtain the address of x and pass it to printf().

You can also use pointers in C to work with memory addresses directly. A pointer is a variable that stores the memory address of another variable. Here

is an example of how to declare a pointer and initialize it with the address of a variable:

```
#include <stdio.h>

int main(){
  int x = 42;
  int *ptr = &x;
  printf("The value of x is: %d\n", *ptr);
}
```

Output:

```
The value of x is: 42
```

In this example, the * operator is used to declare ptr as a pointer to an integer. The address of x is obtained using the & operator and stored in ptr.

You can use pointers to access the value stored at a particular memory address. The * operator is used to dereference a pointer and obtain the value stored at the memory address it points to.

In this example, the * operator is used to dereference ptr and obtain the value stored at the memory address it points to. The output of the printf() statement will be The value of x is: 42.

Working with memory addresses and pointers can be tricky and requires a good understanding of how memory works in the computer. It's important to be careful when working with memory addresses and avoid common errors like dereferencing a null pointer or accessing memory that is out of bounds.

17.1 Why is it useful to know the memory address?

Pointers are important in C, because they allow us to manipulate the data in the computer's memory - this can reduce the code and improve the performance.

Pointers are one of the things that makes C stand out from other programming languages.

18 Pointers:

In the previous section pointers were briefly mentioned with regards to memory addresses.

A pointer is a variable that stores the memory address of another variable. Pointers provide a way to work with memory addresses and access the values stored at those addresses directly.

To declare a pointer variable, you use the * operator in the variable declaration. For example, to declare a pointer to an integer variable x, you would use the following syntax:

```
int *prt;
```

This declares a pointer variable called ptr that can store the memory address of an integer value. The * operator is used to indicate that ptr is a pointer variable.

To initialize a pointer variable with the memory address of a variable, you use the & operator. For example, to initialize ptr with the memory address of x, you would use the following syntax:

```
#include <stdio.h>

int main(){
  int x = 42;
  int *ptr = &x;
  printf("The value of x is: %d\n", *ptr);
}
```

Output:

```
The value of x is: 42
```

This initializes ptr with the memory address of x. The & operator is used to obtain the memory address of x, which is then assigned to ptr.

To access the value stored at the memory address pointed to by a pointer variable, you use the * operator again.

This dereferences ptr using the * operator and prints the value stored at the memory address pointed to by ptr, which is the value stored in x.

Pointers can be used for a wide variety of purposes in C, including dynamic memory allocation, working with arrays, and passing variables to functions by reference. However, working with pointers requires a good understanding of memory management and can be error-prone if not done carefully.

18.1 Dereference:

Note that the * sign can be confusing here, as it does two different things in our code:

When used in declaration (int* ptr), it creates a pointer variable. When not used in declaration, it act as a dereference operator. Good To Know: There are two ways to declare pointer variables in C:

```
int* myNum;
int *myNum;
```

Pointers are one of the things that make C stand out from other programming languages, like Python and Java.

They are important in C, because they allow us to manipulate the data in the computer's memory. This can reduce the code and improve the performance. If you are familiar with data structures like lists, trees and graphs, you should

know that pointers are especially useful for implementing those. And sometimes you even have to use pointers, for example when working with files.

But be careful; pointers must be handled with care, since it is possible to damage data stored in other memory addresses.

19 Pointers and Arrays:

You can also use pointers to access arrays. Consider the following array of integers:

```
int myNumbers[4] = {25, 50, 75, 100};
```

Loop through the array elements with a for loop:

```
#include <stdio.h>

int main(){
  int myNumbers[4] = {25, 50, 75, 100};
  int i;

  for (i = 0; i < 4; i++) {
    printf("%d\n", myNumbers[i]);
  }
}
```

Output:

```
25
50
75
100
```

Now instead of printing the value of each array element, print the memory address of each array element:

```
#include <stdio.h>

int main(){
  int myNumbers[4] = {25, 50 , 75, 100};
  int i;

for (i = 0; i < 4; i++) {
  printf("%p\n", &myNumbers[i]);
}
}
```

Output:

```
0x7ffc285a9830
0x7ffc285a9834
0x7ffc285a9838
0x7ffc285a983c
```

Note that the last number of each elements' memory is address is different, with an addition of 4. This is due to the size of an 'int' type being typically 4 bytes.

```
#include <stdio.h>

int main(){
  int myInt;

printf("%lu", sizeof(myInt));

}
```

Output:

```
4
```

From the memory address example above you can see that the compiler reserves 4 bytes of memory for each array, element, which means that the entire array takes up 16 bytes (4*4) of memory storage:

```
#include <stdio.h>

int main(){
  int myNumbers[4] = {25, 50, 75, 100};

  printf("%lu", sizeof(myNumbers));
}
```

Output:

```
16
```

19.1 How are pointers related to arrays:

The name of an array is actually a pointer to the first element of the array. See in the example below that the memory address of the first element is the same as the name of the array:

```
#include <stdio.h>

int main(){
  int myNumbers[4] = {25, 50, 75, 100};
```

```
  printf("%p\n", myNumbers);
  printf("%p\n", &myNumbers[0]);
}
```

Output:

```
0x7ffcd8fd0b30
0x7ffcd8fd0b30
```

This shows that it is possible to work with arrays through pointers. Since myNumbers is a pointer to the first element in myNumbers you can use the * operator to access it.

```
#include <stdio.h>

int main(){
  int myNumbers[4] = {25, 50, 75, 100};

  //Get the value of the fist element in myNumbers
  printf("%d\n", *myNumbers);

  //Get the value of the second element in myNumbers
  printf("%d\n", *(myNumbers + 1));

  //Get the value of the third element in myNumbers
  printf("%d\n", *(myNumbers + 2));

  //Get the value of the fourth element in myNumbers
  printf("%d\n", *(myNumbers + 3));
}
```

Output:

```
25
50
75
100
```

Or loop through the array:

```
#include <stdio.h>

int main(){
  int myNumbers[4] = {25, 50, 75, 100};
  int *ptr = myNumbers;
  int i;

  for (i = 0; i < 4; i++) {
```

```
    printf("%d\n", *(ptr + i));
    }
}
```

Output:

```
25
50
75
100
```

It is also possible to change the value of array elements with pointers:

```
#include <stdio.h>

int main(){
  int myNumbers[4] = {25, 50, 75, 100};
  //Change the value of the first element to 13
  *myNumbers = 13;

  //Change the value of the second element to 17
  *(myNumbers + 1) = 17;

  //Print value of the first element
  printf("%d\n", *myNumbers);

  //Get the value of the second element
  printf("%d\n", *(myNumbers + 1));
}
```

Output:

```
13
17
```

This way of working with arrays might seem a bit excessive. Especially with simple arrays like in the examples above. However, for large arrays, it can be much more efficient to access and manipulate arrays with pointers.

It is also considered faster and easier to access two-dimensional arrays with pointers.

And since strings are actually arrays, you can also use pointers to access strings.

20 Functions:

A function is a block of code which only runs when it is called. You can pass data known as parameters into a function. Functions are used to perform certain actions, and they are important for reusing code: Define the code once and use it many times. Functions are essentially used to break down a large program into smaller, manageable parts. Functions in C have the following characteristics:

1. A function is defined using the "function" keywrod followed by the return type, function name, and parameters (if any) in parentheses.
2. The code insude the function is enclosed in curly braces {}.
3. A function can have zero or more parameters, which are used to pass data to the function.
4. A function can have a return type, which indicates the type of data the function will return when it completes its task.
5. When a function is called, control is transferred to the function, and the code inside the function is executed. After the function completes it's task, control is returned to the calling function.
6. Functions can be called from other functions, or from the main program.

   Below is an example of a simple function in C:

```
#include <stdio.h>

int square(int num) {
  int result;
  result = num * num;
  return result;
  }

int main(){
  int num = 5;
  int sqr;
  sqr = square(num);
  printf("The square of %d is %d\n", num, sqr);
  return 0;
}
```

Output:

```
The square of 5 is 25
```

In this example, the "square" function takes an integer parameter "num", calculates the square of "num", and returns the result. The "main" function calls the "square" function, passing it the value 5. The returned result is then assigned to the "sqr" variable, which is then printed to the console using the printf function.

Functions are an essential part of C programming, allowing you to break down complex programs into smaller, more manageable parts.

20.1 Create a function:

To create (often referred to as declare) a function, specify the name of the function followed by parentheses() and curly brackets {}:

```
void myFunction(){
  //code to be executed
}
```

20.2 Call a function:

Declared functions are not executed immediatley. They are "saved for later use" and will be executed when they are called. To call a function, write the functions name followed by two parenthese () and a semicolon ; The following example shows how myFunction() is to print text when it's called:

```
  #include <stdio.h>


//Create function
void myFunction() {
      printf("I just got executed!");
    }
int main() {
  myFunction(); //call  function (off with his head!)
  return 0;
}
```

Output:

```
I just got executed!
```

A function can be called multiple times:

```
#include <stdio.h>

void myFunction() {
  printf("I just got executed!");
}

int main(){
  myFunction();
  myFunction();
  myFunction();
  return 0;
}
```

Output:

```
I just got executed!I just got executed!I just got executed!
```

---

21 Footnotes: