C Fundamentals

What is C?

C is a general-purpose programming language created by Dennis Ritchie at the Bell Laboratories in 1972.

The main reason for its popularity is because it is a fundamental language in the field of computer science.

C is strongly associated with UNIX, as it was developed to write the UNIX operating system.

An IDE (**I**ntegrated **D**evelopment **E**nvironment) is used to edit and compile the code.

C Syntax

Syntax

Example

```c
#include <stdio.h>

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

Line 1: #include <stdio.h> is a **header file library** that lets us work with input and output functions, such as **printf()** (used in line 4). Header files add *functionality* to C programs.

Line 2: A *blank* line. C **ignores** white space. But we use it to make the code more ***readable***.

Line 3: Another thing that always appear in a C program is main(). This is called a ***function***. Any code inside its curly brackets {} will be *executed*.

Line 4: printf() is a function used to **output/print** text to the screen.

Note that: Every C statement ends with a semicolon **;**

Note: The body of int main() could also been written as:
int main(){printf("Hello World!");return 0;}

Line 5: return 0 **ends** the *main() function*.

Line 6: Do not forget to add the closing curly bracket } to actually **end** the main function.

Statements

A computer program is a list of "**instructions**" to be "**executed**" by a computer.

In a programming language, these programming instructions are called **statements**.

Most C programs contain *many* statements. The statements are executed, **one by one**, in the **same order** as they are *written*:

C Output

Output (Print Text)

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

Example

```
#include <stdio.h>

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

When you are working with text, it must be wrapped inside double quotations marks **""**.

If you forget the double quotes, an error occurs:

Example

```
printf("This sentence will work!");
```

```
printf(This sentence will produce an error.);
```

You can use as many printf() functions as you want. However, note that it does not insert a new line at the end of the output:

Example

```
#include <stdio.h>

int main() {
  printf("Hello World!");
  printf("I am learning C.");
  printf("And it is awesome!");
  return 0;
}
```

New Lines

To insert a new line, you can use the **\n** character:

Example

```
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  printf("I am learning C.");
  return 0;
}
```

You can also output *multiple* lines with a single printf() function:

Example

```
#include <stdio.h>

int main() {
  printf("Hello World!\nI am learning C.\nAnd it is awesome!");
  return 0;
}
```

Two \n characters after each other will create a blank line:

Example

```
#include <stdio.h>

int main() {
  printf("Hello World!\n\n");
  printf("I am learning C.");
  return 0;
}
```

The newline character (\n) is called an **escape sequence**, and it *forces* the *cursor* to change its *position* to the **beginning** of the next line on the screen. This results in a new line.

Examples of other valid escape sequences are:

Escape Sequence	Description
\t	Creates a horizontal tab
\\	Inserts a backslash character (\)
\"	Inserts a double quote character

C Comments

Comments can be used to *explain code*, and to make it more *readable*. It can also be used to *prevent execution when testing* alternative code.

Comments can be **singled-lined** or **multi-lined**.

Single-line Comments

Single-line comments start with two forward slashes (//).

Any text between // and the end of the line is **ignored** by the compiler

Example

```
// This is a comment
printf("Hello World!");
```

Example

```
printf("Hello World!"); // This is a comment
```

Multi-line Comments

Multi-line comments start with /* and ends with */.

Any text between /* and */ will be **ignored** by the compiler:

Example

```
/* The code below will print the words Hello World!
to the screen, and it is amazing */
printf("Hello World!");
```

C Variables

Variables" are *containers* for **storing data values**, like *numbers* and *characters*.

In C, there are different types of variables (defined with different keywords), for example:

- int - stores **integers** (whole numbers), ***without*** *decimals*, such as 123 or -123
- float - stores **floating point numbers**, ***with*** *decimals*, such as 19.99 or -19.99
- char - stores **single characters**, such as 'a' or 'B'. Characters are surrounded by ***single quotes***.

Declaring Variables

To create a variable, *specify* the **type** and *assign* it a **value**:

Syntax

```
type variableName = value;
```

To create a variable that should store a number, look at the following example:

Example

Create a variable called **myNum** of type `int` and assign the value **15** to it:

```
int myNum = 15;
```

You can also declare a variable without assigning the value, and assign the value later:

Example

```
// Declare a variable
int myNum;

// Assign a value to the variable
myNum = 15;
```

Format Specifiers

Format specifiers are used together with the printf() function to tell the compiler what type of data the variable is storing. It is basically a **placeholder** for the variable value.

A format specifier starts with a **percentage sign %,** followed by a character.

For example, to output the value of an int variable, use the format specifier **%d** surrounded by double quotes (""), inside the printf() function:

Example

```c
int myNum = 15;
printf("%d", myNum);  // Outputs 15
```

%d - int
%c - char
%f - float

To combine both text and a variable, separate them with a comma inside the printf() function:

Example

```c
int myNum = 15;
printf("My favorite number is: %d", myNum);
```

To print different types in a single printf() function, you can use the following:

Example

```c
int myNum = 15;
char myLetter = 'D';
printf("My number is %d and my letter is %c", myNum, myLetter);
```

Print Values Without Variables

You can also just print a value without storing it in a variable, as long as you use the correct format specifier:

Example

```c
printf("My favorite number is: %d", 15);
printf("My favorite letter is: %c", 'D');
```

Change Variable Values

If you assign a new value to an existing variable, it will **overwrite** the previous value:

Example

```c
int myNum = 15;  // myNum is 15
myNum = 10;  // Now myNum is 10
```

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

Example

```c
int myNum = 15;

int myOtherNum = 23;

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

// myNum is now 23, instead of 15
printf("%d", myNum);
```

Or copy values to empty variables:

Example

```c
// Create a variable and assign the value 15 to it
int myNum = 15;

// Declare a variable without assigning it a value
int myOtherNum;

// Assign the value of myNum to myOtherNum
myOtherNum = myNum;

// myOtherNum now has 15 as a value
printf("%d", myOtherNum);
```

Add Variables Together

To add a variable to another variable, you can use the + operator:

Example

```c
int x = 5;
int y = 6;
int sum = x + y;
printf("%d", sum);
```

Declaring Multiple Variables

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

Example

```c
int x = 5, y = 6, z = 50;
printf("%d", x + y + z);
```

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

Example

```
int x, y, z;
x = y = z = 50;
printf("%d", x + y + z);
```

Variable Names (Identifiers)

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 (age, sum, totalVolume).

Example

```
// Good variable name
int minutesPerHour = 60;

// OK, but not so easy to understand what m actually is
int m = 60;
```

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 (myVar and myvar are different variables)
- Names cannot contain whitespaces or special characters like !, #, %, etc.
- Reserved words (such as int) cannot be used as names

C Data Types

Basic Data Types

Data Type	Size	Description	Example
`int`	2 or 4 bytes	Stores whole numbers, without decimals	`1`
`float`	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits	`1.99`
`double`	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits	`1.99`
`char`	1 byte	Stores a single character/letter/number, or ASCII values	`'A'`

Basic Format Specifiers

Format Specifier	Data Type
`%d` or `%i`	`int`
`%f` or `%F`	`float`
`%lf`	`double`
`%c`	`char`
`%s`	Used for **strings (text)**, which you will learn more about in a later chapter

Character Data Type

The **char** data type is used to store a **single** character.

The character must be surrounded by single quotes, like 'A' or 'c', and we use the %c format specifier to print it:

Example

```
char myGrade = 'A';
printf("%c", myGrade);
```

Alternatively, if you are familiar with ASCII, you can use ASCII values to display certain characters. Note that these values are not surrounded by quotes (' '), as they are numbers:

Example

```
char a = 65, b = 66, c = 67;
printf("%c", a);
printf("%c", b);
printf("%c", c);
```

Numeric Data Type

Use **int** when you need to store a whole number **without** decimals, like 35 or 1000, and **float** or **double** when you need a **floating point number**, like 9.99 or 3.14515.

int

```
int myNum = 1000;
printf("%d", myNum);
```

float

```
float myNum = 5.75;
printf("%f", myNum);
```

double

```
double myNum = 19.99;
printf("%lf", myNum);
```

float vs. double

The precision of a floating point value indicates how many digits the value can have after the decimal point. The precision of float is **six or seven** decimal digits, while double variables have a precision of about **15** digits. Therefore, it is often safer to use double for most calculations - but note that it takes up twice as much memory as float (**8 bytes vs. 4 bytes**).

Scientific Numbers

A floating point number can also be a **scientific number** with an "e" to indicate the power of 10:

Example

```
float f1 = 35e3;
double d1 = 12E4;

printf("%f\n", f1);
printf("%lf", d1);
```

Decimal Precision

If you want to remove the extra zeros (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:

Example

```
float myFloatNum = 3.5;

printf("%f\n", myFloatNum);   // Default will show 6 digits after the decimal point
printf("%.1f\n", myFloatNum); // Only show 1 digit
printf("%.2f\n", myFloatNum); // Only show 2 digits
printf("%.4f", myFloatNum);   // Only show 4 digits
```

Memory Size

The memory size refers to how much space a type occupies in the computer's memory.

Data Type	Size
`int`	2 or 4 bytes
`float`	4 bytes
`double`	8 bytes
`char`	1 byte

To actually get the size (in bytes) of a data type or variable, use the sizeof operator:

Example

```
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));
```

Note; ***We use the %lu format specifer to print the result, instead of %d. It is because the compiler expects the sizeof operator to return a long unsigned int (%lu), instead of int (%d). On some computers it might work with %d, but it is safer to use %lu.***

Type Conversion

Sometimes, you have to convert the value of one data type to another type. This is known as **type conversion**.

There are two types of conversion in C:

- **Implicit Conversion** (automatically)
- **Explicit Conversion** (manually)

Implicit Conversion

Implicit conversion is done automatically by the compiler when you assign a value of one type to another.

Example

```
// Automatic conversion: int to float
float myFloat = 9;

printf("%f", myFloat); // 9.000000
```

Example

```
// Automatic conversion: float to int
int myInt = 9.99;

printf("%d", myInt); // 9
```

Example

```
float sum = 5 / 2;

printf("%f", sum); // 2.000000
```

WHY? Well, it is because 5 and 2 are still integers in the division before the conversion.

Explicit Conversion

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

Example

```
// Manual conversion: int to float
float sum = (float) 5 / 2;

printf("%f", sum); // 2.500000
```

Example

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

printf("%.1f", sum); // 2.5
```

Constants

If you don't want others (or yourself) to change existing variable values, you can use the const keyword.

This will declare the variable as "constant", which means **unchangeable** and **read-only**:

Example

```
const int myNum = 15;  // myNum will always be 15
myNum = 10;  // error: assignment of read-only variable 'myNum'
```

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

Example

Like this:

```
const int minutesPerHour = 60;
```

This however, **will not work**:

```
const int minutesPerHour;
minutesPerHour = 60; // error
```

Operators

Operators are used to perform operations on variables and values.

C divides the operators into the following groups:

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

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example
+	Addition	Adds together 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 remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

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
<<=	x <<= 3	x = x << 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.

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**.

Operator	Name	Example	Description
==	Equal to	x == y	Returns 1 if the values are equal
!=	Not equal	x != y	Returns 1 if the values are not equal
>	Greater than	x > y	Returns 1 if the first value is greater than the second value
<	Less than	x < y	Returns 1 if the first value is less than the second value
>=	Greater than or equal to	x >= y	Returns 1 if the first value is greater than, or equal to, the second value
<=	Less than or equal to	x <= y	Returns 1 if the first value is less than, or equal to, the second value

Logical Operators

You can also test for true or false values with logical operators.

Logical operators are used to determine the logic between variables or values, by combining multiple conditions:

Operator	Name	Example	Description
&&	AND	x < 5 && x < 10	Returns 1 if both statements are true
||	OR	x < 5 || x < 4	Returns 1 if one of the statements is true
!	NOT	!(x < 5 && x < 10)	Reverse the result, returns 0 if the result is 1

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 values that are either true or false.

Boolean Variables

In C, the bool type is not a built-in data type, like int or char.

It was introduced in C99, and you must import the following header file to use it:

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

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

```
bool isProgrammingFun = true;
bool isFishTasty = false;
```

Before trying to print the boolean variables, you should know that boolean values are returned as **integers**. Therefore, you must use the %d format specifier to print a boolean value:

Example

```
// Create boolean variables
bool isProgrammingFun = true;
bool isFishTasty = false;

// Return boolean values
printf("%d", isProgrammingFun);   // Returns 1 (true)
printf("%d", isFishTasty);        // Returns 0 (false)
```

Comparing Values and Vairables

Comparing values are useful in programming, because it helps us to find answers and make decisions:

Example

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

Condition and If Statements

You can use these conditions to perform different actions for different decisions.

C has the following conditional statements:

- Use **if** to specify a block of code to be executed, if a specified condition is **true**

Syntax

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

- Use **else** to specify a block of code to be executed, if the same condition is **false**

Syntax

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

- Use **else if** to specify a new condition to test, if the first condition is **false**

Syntax

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is true
} else {
  // block of code to be executed if the condition1 is false and condition2 is false
}
```

- Use **switch** to specify many alternative blocks of code to be executed

Syntax

```
switch (expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

- 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

Example

```
int day = 4;

switch (day) {
  case 1:
    printf("Monday");
    break;
  case 2:
    printf("Tuesday");
    break;
  case 3:
    printf("Wednesday");
    break;
  case 4:
    printf("Thursday");
    break;
  case 5:
    printf("Friday");
    break;
  case 6:
    printf("Saturday");
    break;
  case 7:
    printf("Sunday");
    break;
}

// Outputs "Thursday" (day 4)
```

Short Hand If...Else (Ternary Operator)

There is also a short-hand if else, which is known as the **ternary operator** because it consists of three operands. It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements:

Syntax

```
variable = (condition) ? expressionTrue : expressionFalse;
```

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 make code more readable.

While Loop

The while loop loops through a block of code as long as a specified condition is true:

Syntax

```
while (condition) {
  // code block to be executed
}
```

Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

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

For Loops

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

Syntax

```
for (expression 1; expression 2; expression 3) {
  // code block to be executed
}
```

Expression 1 is executed (one time) before the execution of the code block.

Expression 2 defines the condition for executing the code block.

Expression 3 is executed (every time) after the code block has been executed.

Nested Loops

It is also possible to place a loop inside another loop. This is called a nested loop.

Example

```
int i, j;

// Outer loop
for (i = 1; i <= 2; ++i) {
  printf("Outer: %d\n", i);  // Executes 2 times

  // Inner loop
  for (j = 1; j <= 3; ++j) {
    printf(" Inner: %d\n", j);  // Executes 6 times (2 * 3)
  }
}
```

Break and Continue

Break

The break statement can also be used to jump out of a loop.

Example

```
int i;

for (i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  printf("%d\n", i);
}
```

Continue

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

Example

```
int i;

for (i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  printf("%d\n", i);
}
```

Arrays

Arrays are used to store **multiple values** in a **single variable**, instead of declaring separate variables for each value.

To create an array, define the data type (like *int*) and specify the name of the array followed by square brackets [].

To insert values to it, use a comma-separated list inside curly braces, and make sure all values are of the same data type:

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

Access the Elements of an Array

To access an array element, refer to its **index number**.

Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Example

```
int myNumbers[] = {25, 50, 75, 100};
printf("%d", myNumbers[0]);

// Outputs 25
```

Change an Array Element

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

Example

```
int myNumbers[] = {25, 50, 75, 100};
myNumbers[0] = 33;

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

// Now outputs 33 instead of 25
```

Set Array Size

Another common way to create arrays, is to specify the size of the array, and add elements later:

Example

```
// Declare an array of four integers:
int myNumbers[4];

// Add elements
myNumbers[0] = 25;
myNumbers[1] = 50;
myNumbers[2] = 75;
myNumbers[3] = 100;
```

Get Array Size/Length

Size

To get the size of an array, you can use the sizeof operator:

Example

```
int myNumbers[] = {10, 25, 50, 75, 100};
printf("%lu", sizeof(myNumbers)); // Prints 20
```

Length

To find out how many elements an array has, you can use the following formula (which divides the size of the array by the size of the first element in the array):

Example

```
int myNumbers[] = {10, 25, 50, 75, 100};
int length = sizeof(myNumbers) / sizeof(myNumbers[0]);

printf("%d", length);  // Prints 5
```

For Loop in Arrays

Example

```
int myNumbers[] = {25, 50, 75, 100};
int length = sizeof(myNumbers) / sizeof(myNumbers[0]);
int i;

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

MultiDimensional Arrays

A multidimensional array is basically an array of arrays.

Arrays can have any number of dimensions.

Two Dimensional Arrays

A 2D array is also known as a matrix

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };
```

The first dimension represents the number of rows [2], while the second dimension represents the number of columns [3].

Accessing the Elements of a 2-D Array

To access an element of a two-dimensional array, you must specify the index number of both the row and column.

Example

```c
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };

printf("%d", matrix[0][2]);  // Outputs 2
```

Strings

Strings are used for storing text/characters.C **does not** have a String type to easily create string variables. Instead, you must use the char type and create an array of characters to make a string in C:

```c
char greetings[] = "Hello World!";
```

To output the string, you can use the printf() function together with the format specifier %s to tell C that we are now working with strings:

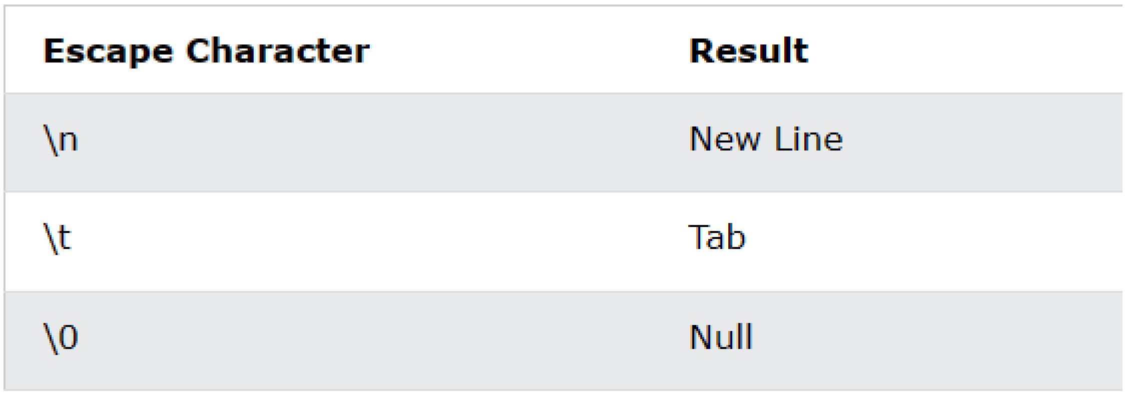
Example

```c
char greetings[] = "Hello World!";
printf("%s", greetings);
```

Special Characters

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

Escape character	Result	Description
\'	'	Single quote
\"	"	Double quote
\\\\	\\	Backslash



Escape Character	Result
\n	New Line
\t	Tab
\0	Null

String Functions

C also has many useful string functions, which can be used to perform certain operations on strings.

To use them, you must include the <string.h> header file in your program.

- **String Length:**

To get the length of a string, you can use the strlen() function:

Example

```
char alphabet[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
printf("%d", strlen(alphabet));
```

- ## Concatenate Strings:

To concatenate (combine) two strings, you can use the strcat() function:

Example

```
char str1[20] = "Hello ";
char str2[] = "World!";

// Concatenate str2 to str1 (result is stored in str1)
strcat(str1, str2);

// Print str1
printf("%s", str1);
```

- ## Copy Strings:

To copy the value of one string to another, you can use the strcpy() function:

Example

```
char str1[20] = "Hello World!";
char str2[20];

// Copy str1 to str2
strcpy(str2, str1);

// Print str2
printf("%s", str2);
```

- ## Compare Strings

To compare two strings, you can use the strcmp() function.

It returns 0 if the two strings are equal, otherwise a value that is not 0:

Example

```
char str1[] = "Hello";
char str2[] = "Hello";
char str3[] = "Hi";

// Compare str1 and str2, and print the result
printf("%d\n", strcmp(str1, str2));  // Returns 0 (the strings are equal)

// Compare str1 and str3, and print the result
printf("%d\n", strcmp(str1, str3));  // Returns -4 (the strings are not equal)
```

User Input

To get user input, you can use the scanf() function:

Example

Output a number entered by the user:

```c
// Create an integer variable that will store the number we get from the user
int myNum;

// Ask the user to type a number
printf("Type a number: \n");

// Get and save the number the user types
scanf("%d", &myNum);

// Output the number the user typed
printf("Your number is: %d", myNum);
```

Multiple Inputs

Example

```c
// Create an int and a char variable
int myNum;
char myChar;

// Ask the user to type a number AND a character
printf("Type a number AND a character and press enter: \n");

// Get and save the number AND character the user types
scanf("%d %c", &myNum, &myChar);

// Print the number
printf("Your number is: %d\n", myNum);

// Print the character
printf("Your character is: %c\n", myChar);
```

Example

```c
char fullName[30];

printf("Type your full name: \n");
fgets(fullName, sizeof(fullName), stdin);

printf("Hello %s", fullName);

// Type your full name: John Doe
// Hello John Doe
```

Memory Address

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.

To access it, use the reference operator (&), and the result represents where the variable is stored:

Example

```c
int myAge = 43;
printf("%p", &myAge); // Outputs 0x7ffe5367e044
```

Pointers

A pointer is a variable that stores the memory address of another variable as its value.

A pointer variable points to a data type (like int) of the same type, and is created with the * operator.

Example

```c
int myAge = 43;     // An int variable
int* ptr = &myAge;  // A pointer variable, with the name ptr, that stores the address of myAge

// Output the value of myAge (43)
printf("%d\n", myAge);

// Output the memory address of myAge (0x7ffe5367e044)
printf("%p\n", &myAge);

// Output the memory address of myAge with the pointer (0x7ffe5367e044)
printf("%p\n", ptr);
```

Dereference

You can also get the value of the variable the pointer points to, by using the * operator (the dereference operator):

Example

```c
int myAge = 43;     // Variable declaration
int* ptr = &myAge;  // Pointer declaration

// Reference: Output the memory address of myAge with the pointer (0x7ffe5367e044)
printf("%p\n", ptr);

// Dereference: Output the value of myAge with the pointer (43)
printf("%d\n", *ptr);
```

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.

Predefined Functions

main() is a function, which is used to execute code, and printf() is a function; used to output/print text to the screen. These are predefined functions.

Creating a Function

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

Syntax

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

Calling a Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed when they are called.

To call a function, write the function's name followed by two parentheses () and a semicolon ;

Example

Inside `main`, call `myFunction()`:

```
// Create a function
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
```

Function Parameters

Parameters and Arguments

Information can be passed to functions as a parameter. Parameters act as variables inside the function.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma:

Syntax

```
returnType functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

When a parameter is passed to the function, it is called an argument.

Return Values

The void keyword, used in the previous examples, indicates that the function should not return a value. If you want the function to return a value, you can use a data type (such as int or float, etc.) instead of void, and use the return keyword inside the function:

Example

```
int myFunction(int x) {
  return 5 + x;
}

int main() {
  printf("Result is: %d", myFunction(3));
  return 0;
}

// Outputs 8 (5 + 3)
```