CSE 439

DESIGN OF COMPILERS

JUNIOR CESS

SPRING 24

COURSE PROJECT

Name	ID
Seif Yasser Ahmed Kamal	21P0102
Mohammed Salah Fathy	21P0117
Youssef Tamer Hossam	21P0138
Muhammad Ehab	21P0267
Ahmad Abdelmaksoud	2101077

Table of Contents

Introduction

The compiler is software that converts a program written in a high-level language (Source Language) to a low-level language.

A translator or language processor is a program that translates an input program written in a programming language into an equivalent program in another language. The compiler is a type of translator, which takes a program written in a high-level programming language as input and translates it into an equivalent program in low-level languages such as machine language or assembly language.

The program written in a high-level language is known as a source program, and the program converted into a low-level language is known as an object (or target) program. Without compilation, no program written in a high-level language can be executed. For every programming language, we have a different compiler; however, the basic tasks performed by every compiler are the same. The process of translating the source code into machine code involves several stages, including lexical analysis, syntax analysis, semantic analysis, code generation, and optimization.

The aim of this project is to develop the lexical analysis and syntax analysis phases of the compiler for the C programming language.

We should search for the Language Specifications: Identify the basic constructs of the C programming language such as:

- Keywords
- Variable Identifiers
- Function Identifiers
- Data Types
- Functions
- Statements:
  - Assignment Statement
  - Declaration Statement
  - Return Statement
  - Iterative Statement
  - Conditional Statements
  - Function Call Statement
- Expressions
  - Arithmetic
  - Boolean

We will use the C++ programming language to implement a lexer and a parser for the C programming language.

The compiler is software that converts a program written in a high-level language (Source Language) to a low-level language.

Keywords

auto	static	const	_Alignas	sizeof
break	inline	while	_Alignof	_Generic
case	long	for	_Atomic	_Imaginary
char	short	if	_Bool	_Noreturn
int	struct	do	typedef	_Complex
float	union	return	else	_Static_assert
double	enum	extern	void	_Thread_local
signed	unsigned	register	switch	volatile
continue	goto	restrict	default	

- auto: Declares automatic (local) variables.
- break: Terminates loops or switch statements.
- case: Defines values in a switch statement.
- char: Represents character data.
- int: Declares integer variables.
- float: Represents single-precision floating-point numbers.
- double: Represents double-precision floating-point numbers.
- signed: Indicates signed integer types.
- continue: Skips the current iteration in a loop.
- static: Declares static variables with file scope.
- inline: Suggests inlining a function (since C99).
- long: Declares long integer variables.
- short: Declares short integer variables.
- struct: Defines user-defined structures.
- union: Defines a union of different data types.
- enum: Declares enumeration types.
- unsigned: Represents non-negative integer values.
- goto: Transfers control to a labeled statement.
- const: Declares constants.
- while: Introduces a while loop.
- for: Introduces a for loop.
- if: Creates conditional statements.
- do: Introduces a do-while loop.
- return: Exits a function and returns a value.
- extern: Declares external variables or functions.
- register: Suggests storing a variable in a CPU register.
- restrict: Hints compiler optimization (since C99).
- _Alignas: Specifies alignment requirements.
- _Alignof: Determines alignment of a data type.

- _Atomic: Used with atomic types for concurrency.
- _Bool: Represents boolean values (since C99).
- typedef: Creates type aliases.void
- else: Used in conditional statements to define the code block executed when the preceding "if" condition is false.
- void: Specifies that a function does not return a value or that a pointer does not point to any data type.
- switch: Enables multi-way decision-making based on the value of an expression.
- default: Specifies the code block executed in a switch statement when none of the case values match the expression.
- sizeof: Returns the size, in bytes, of a data type or an object.
- _Generic: Part of C11 standard, provides a generic selection mechanism based on the type of an expression.
- _Imaginary: Represents imaginary parts in complex numbers in C.
- _Noreturn: Informs the compiler that a function does not return to the calling function.
- _Complex: Declares a complex data type, representing a number with both real and imaginary parts.
- _Static_assert: Performs compile-time assertions to check conditions that must be true.
- _Thread_local: Specifies that a variable has thread-local storage duration.
- volatile: Indicates to the compiler that a variable's value may change at any time, without any action being taken by the code the compiler finds nearby.

Variable Identifiers

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 Identifiers Naming Conventions

While C does not enforce specific naming conventions for identifiers, adopting consistent naming practices enhances code readability and maintainability. Some commonly used conventions include:

- Using meaningful and descriptive names that reflect the purpose of the identifier.
- Employing camel case (e.g., myVariable) or underscore-separated (e.g., my_variable) naming styles.
- Avoiding single-character or overly cryptic names that may lead to confusion.

Function Identifiers

A function is a C language construct that associates a compound statement (the function body) with an identifier (the function name). Every C program begins execution from the main function, which either terminates or invokes other, user-defined or library functions.

A function is introduced by a function declaration or a function definition.

Function Declaration Syntax

returnType functionName(parameters);

returnType: Identifies what the data type of the value this function returns would be. It can be any valid C data type (such as primitive data types like ‘int’, ‘char’, etc.) as well as custom data types or void if the function doesn't return any value.

functionName: This is the function’s identifier. It should usually be a descriptive name that explains the purpose of the function or what it does.

parameters: Each parameter consists of a data type followed by a parameter name, separated by commas. They are inputs provided to the function and can be used within the function scope.

Function Identifiers Scope and Visibility

Function identifiers have a scope, determining where they could be accessed in the program.
By default, a function has “file scope” visibility.
An object has file scope if its definition appears outside of any block. Such an object is visible from the point where it is declared to the end of the source file.

The keyword “extern” can be used for external linkage, to make functions accessible from other files. This keyword informs the compiler that the function is defined in another file, allowing the function to be used in the current file.

Prototype Declaration

Those are declarations that define the function's signature
Function signature is composed of:

1) Function's name
2) Function's return type
3) Function's parameters types

The function's implementation is not provided in the prototype declaration, the prototype declaration just informs the compiler about the existence of that specific function it represents.

Static Functions

The static keyword can be used to declare functions with internal linkage. Functions declared as static are only accessible within the same source file and are not visible to other files linked in the program.

Standard Library Functions:

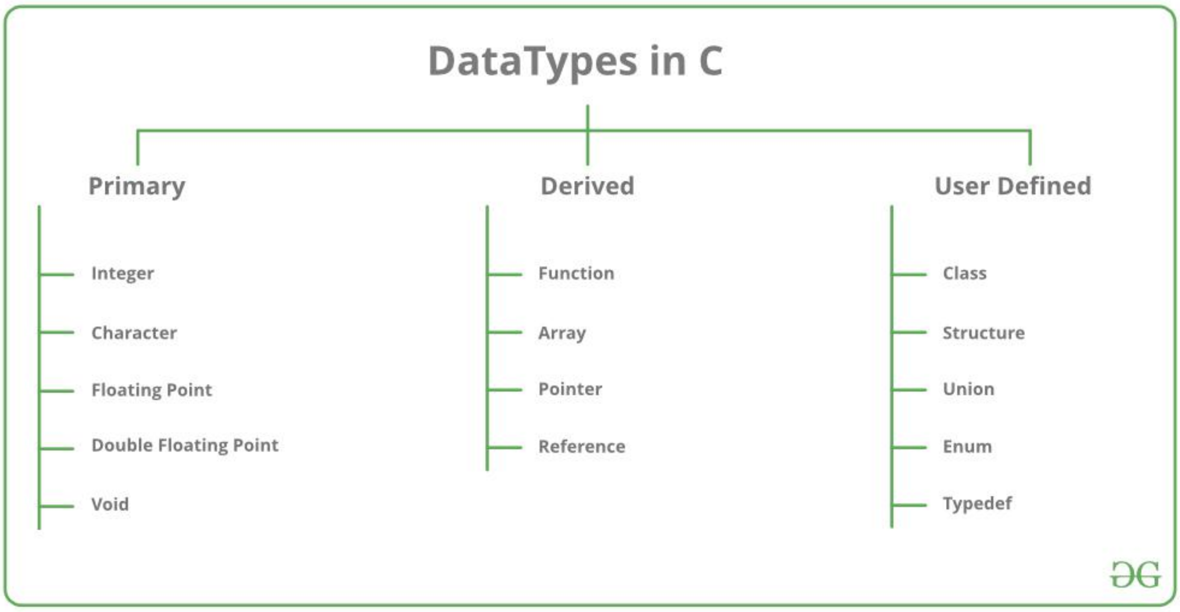
There is a significant number of standard library functions available in C17 to support a wide range of programming tasks, some of the most widely used standard library functions are:

- **printf()**: Used for formatted output to the standard output stream (usually the console).
- **scanf()**: Used for formatted input from the standard input stream (usually the keyboard).
- **strlen()**: Returns the length of a null-terminated string.
- **strcpy()**: Copies a null-terminated string.
- **strcmp()**: Compares two strings.
- **strcat()**: Concatenates two strings.
- **malloc()**: Allocates memory dynamically.
- **free()**: Deallocates memory allocated dynamically by malloc, calloc, or realloc.
- **fgets()**: Reads a line from a file stream.
- **fprintf()**: Writes formatted output to a file stream.
- **fscanf()**: Reads formatted input from a file stream.
- **toupper()**: Converts a lowercase character to upperc

Data Types

Each variable in C has an associated data type. It specifies the type of data that the variable can store like integer, character, floating, double, etc. Each data type requires different amounts of memory and has some specific operations which can be performed over it. The data type is a collection of data with values having fixed values, meaning as well as its characteristics.

Types	Description
Primitive data type	Primitive data types are the most basic data types that are used for representing simple values such as integers, float, characters, etc.
User-defined data type	The user-defined data types are defined by the user himself.
Derived types	The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types.



Ranges

on the 32-bit compiler

Data Type	Size (Bytes)	Range	Format Specifier
Short int	2	-32,768 to 32,767	%hd
Unsigned short int	2	0 to 65,535	%hu
Unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
Long int	4	-2,147,483,648 to 2,147,483,647	%ld
Unsigned long int	4	0 to 4,294,967,295	%lu
Long long int	8	$-(2^{63})$ to $(2^{63}) -1$	%lld
Unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
Signed char	1	-128 to 127	%c
Unsigned char	1	0 to 255	%c
float	4	1.2E-38 to 3.4E+38	%f
double	8	1.7E-308 to 1.7E+308	%lf
Long double	16	3.4E-4932 to 1.1E+4932	%Lf

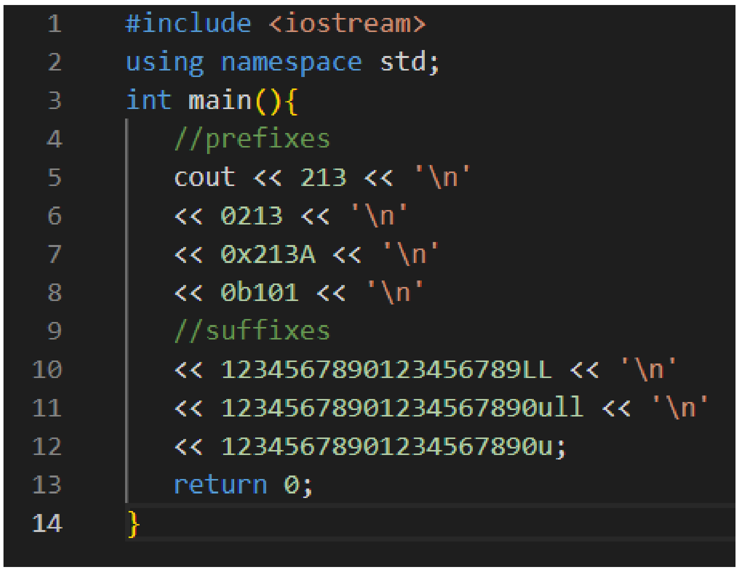
Integer literal

Integer literal is a type of literal for an integer whose value is directly represented in source code. For example, in the assignment statement **x = 1**, the string 1 is an integer literal indicating the value 1, while in the statement **x = 0x10** the string *0x10* is an integer literal indicating the value 16(in decimal), which is represented by 10 in hexadecimal (indicated by the 0x prefix)
Integer literals are expressed in two types:

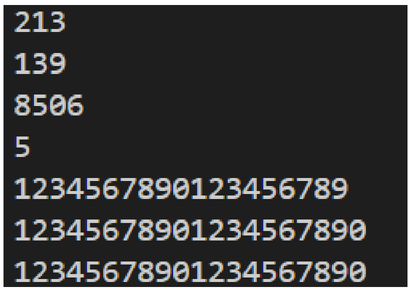
- **Prefixes**: which indicate the base. For example, 0x10 indicates the value *16* in hexadecimal having prefix 0x.
  - **Decimal-literal**:(base 10) a non-zero decimal digit followed by zero or more decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). For example, 56, 78.
  - **Octal-literal**:(base 8) a zero followed by zero or more octal digits(0, 1, 2, 3, 4, 5, 6, 7). For example, 045, 076, 06210.
  - **Hex-literal:**(base 16) 0x or 0X followed by one or more hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F). For example, 0x23A, 0Xb4C, 0xFEA.
  - **Binary-literal**:(base 2) 0b or 0B followed by one or more binary digits (0, 1). For example, 0b101, 0B111.

- **Suffixes**: which indicates the type. For example, 12345678901234LL indicates the value *12345678901234* as an long long integer having suffix LL.
  - **int**: No suffixes are required because integer constant are by default assigned as int data type.
  - **unsigned int**: character u or U at the end of integer constant.
  - **long int**: character l or L at the end of integer constant.
  - **unsigned long int**: character ul or UL at the end of integer constant.
  - **long long int**: character ll or LL at the end of integer constant.
  - **unsigned long long int**: character ull or ULL at the end of integer constant.

Example



```cpp
#include <iostream>
using namespace std;
int main(){
    //prefixes
    cout << 213 << '\n'
    << 0213 << '\n'
    << 0x213A << '\n'
    << 0b101 << '\n'
    //suffixes
    << 1234567890123456789LL << '\n'
    << 12345678901234567890ull << '\n'
    << 12345678901234567890u;
    return 0;
}
```



```
213
139
8506
5
1234567890123456789
12345678901234567890
12345678901234567890
```

Expressions

Arithmetic Expressions

Arithmetic expressions in C involve mathematical operations on variables, constants, and literals. These expressions evaluate to a single value.

Operators

Symbol	Name
- +	Additive operators
/ * %	Multiplicative operators
++ - -	Unary increment and decrement operators

Precedence and Associativity

Operators have different precedence levels. For example, multiplication has higher precedence than addition.Parentheses can be used to override the default precedence.

Boolean Expressions

Boolean expressions evaluate to either true or false. They are essential for control flow and decision-making.

Operators

Symbol	Name
==, !=, >, <, >=, <=	Relation Operators
&&, ||, !	Logical Operators
&, |, ^ ,~,<<,>>	Bitwise Operators
?:	Conditional/Ternary operator
. , -> ,	Member access operators
*	Dereferencing operator
&	Address of operator

Separators

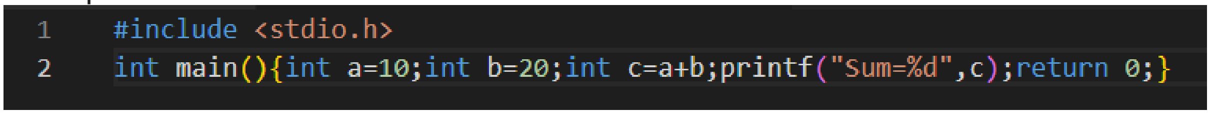
Separators are used to separate one programming element from other. Such as separating keyword from keyword, keyword from identifier, identifier from other identifier etc. They are similar to punctuation marks in English paragraphs.

In C programming every expression is separated using white space character/s, statements are separated from other using semicolon ;

We can use any number of white space characters to separate two expressions. However we must use at least a single white space character to separate one programming element from another. We can also use a number of semicolons to separate one statement from another.

Note: We can write the entire C program in two lines if proper separators are used. Take an example of the two programs.

Example



```c
#include <stdio.h>
int main(){int a=10;int b=20;int c=a+b;printf("Sum=%d",c);return 0;}
```

Order of execution

1. Parenthesis
2. Multiplicative operators (if more than one exists then evaluate from left → right), it may matters.
3. Additive operators (if more than one exists then evaluate from left →right), usually don’t matter.

Statements

The bodies of C functions (including the main function) are made up of **statements**. These can either be **simple statements** that do not contain other statements, or **compound statements** that have other statements inside them. **Control structures** are compound statements like if/then/else, while, for, and do..while that control how or whether their component statements are executed.

Declaration Statements

A declarative statement is a type of statement that declares or defines variables and their types. It specifies the names and data types of variables, but it doesn't perform any specific computation or action. Declarations provide information to the compiler about the existence of variables and their characteristics.

Declarative statements are essential for informing the compiler about the types of variables used in a program. They allocate memory for the variables and set aside storage space based on the specified data types. Once declared, variables can be used in subsequent statements for computation, assignment, and other operations.

The general syntax for a declarative statement in C looks like this:

```
data_type variable_name;
```

`Data_type represents the type of the variable (e.g,int, float,char)`

`Variable_name is the name given to the variable`

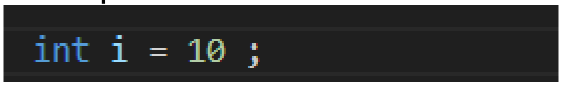
`Multiple variables of the same data type can be declared in a single statement, separated by commas.`

`Examples of declarative statements :`

- `int age ;`
- `float salary ;`
- `char garde , gender ;`

Assignment Statement

This statement assigns a value produced by expression to a variable
Syntax: variable = expression ;
Example:



```
int i = 10 ;
```

Parameters–Assignment Statement
Variable: Specifies the variable name to which the value is assigned. The variables are always to the left of the assignment operator (=)

Expression: Specifies any legal 4GL expression. The data types of the expression and the variable must be compatible. The expression is always to the right of the assignment operator (=).

Simple Assignment Statements

A simple assignment statement assigns the value of an expression to a simple variable, that is, a variable that represents a single data value
A simple variable can be:

- A scalar variable. For example: service_fee = 5.50;
- An individual attribute of a reference variable. For example: newcustomer.Name = ‘Jones’
- An individual data value in an array. For example: client[4].Address = ‘4307 Olive St’;

Reference Assignment Statements

A reference assignment statement redirects either a reference variable or an array variable. A reference variable is a pointer to an object. An array variable points to a set of reference variables, which points to a set of objects. When you assign a new value to a reference variable or array variable, you point the variable to a new object or set of objects, respectively.

Valid assignments for a reference variable include:

- Another reference variable. For example: newcustomer = customer ;
- A row reference in a dynamic array. For example: newcustomer = client[5];
- A null. For example: newcustomer = null ;

Conditional Statement

We have previously talked about the expressions in C that support the usual logical conditions from mathematics:

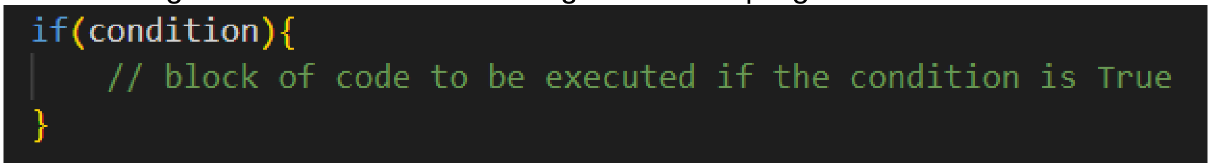
- Less than a < b
- Less than or equal to a <= b
- Greater than: a > b
- Greater than or equal to a >= b
- Equal to a == b
- Not Equal to a != b

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

If statement

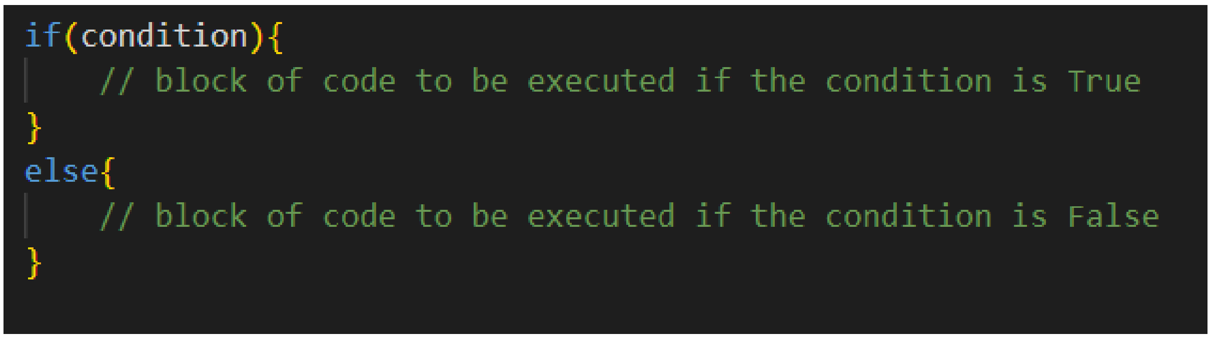
It allows the program to execute a block of code if the specified condition is true and we can imagine it as some sort of intelligence of the program



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

else statement

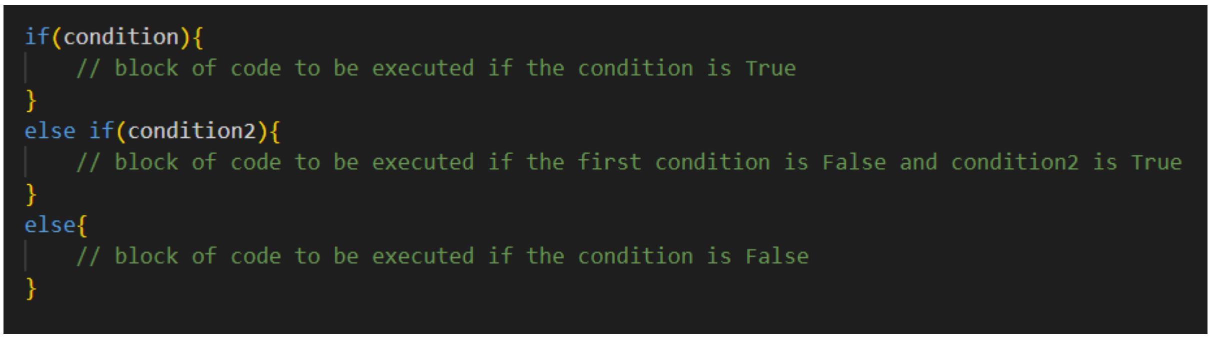
It allows the program to execute a block of code if the specified condition in the above if statement is false



```
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
}
```

else if statement

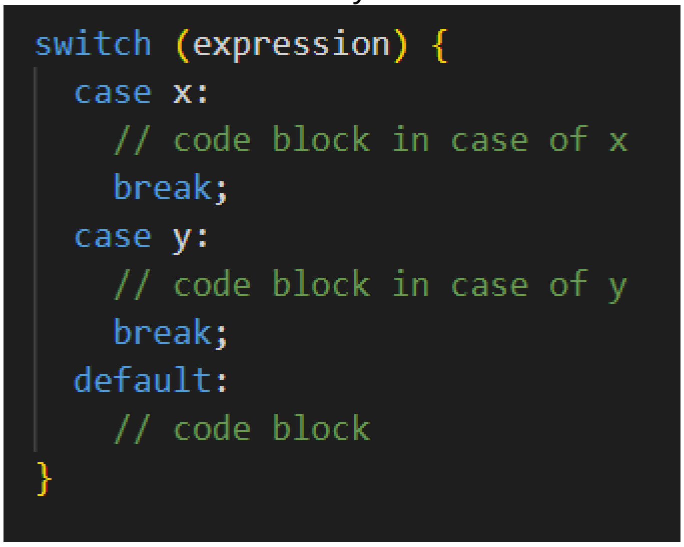
Use the else if statement to specify a new condition if the first condition is false



```
if(condition){
    // block of code to be executed if the condition is True
}
else if(condition2){
    // block of code to be executed if the first condition is False and condition2 is True
}
else{
    // block of code to be executed if the condition is False
}
```

Switch statement

Instead of writing too many if…else statements, you can use switch case
It selects one of too many blocks of code to be executed



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

Iterative Statement

- Iteration statements are used to set up loops.
- A loop is a statement whose job is to repeatedly is a statement whose job is to repeatedly execute some other statement (the loop body).
- Every loop has a controlling expression
- Each time the loop body is executed (an iteration of the loop), the controlling expression is evaluated.
- If the expression is true (has a value that’s not zero) the loop continues to execute.

C provides three iteration statements :

- The while statement is used for loops whose controlling expression is tested before the loop body is executed
- The do statement is used if the expression is tested after the loop body is executed

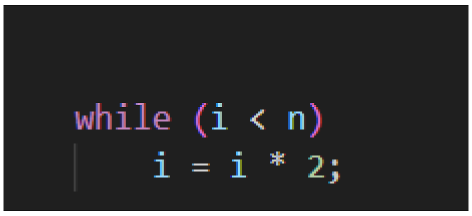
- The for statement is convenient for loops that increment or decrement a counting variable

while statement

- Using a while statement is the easiest way to set up a loop
- The while statement has the form: while (expression) {statement}
- Expression is the controlling expression
- Statement is the loop body
- When a while statement is executed, the controlling expression is evaluated first.
- If its value is nonzero (true), the loop body is executed and the expression is tested again
- The process continues until the controlling expression eventually has the value zero.
- C programmers sometimes deliberately create an infinite loop by using a nonzero constant as the controlling expression: while (1)
- A while statement of this form will execute forever unless its body contains a statement that transfers control out of the loop (break, goto, return) or calls a function that causes the program to terminate.

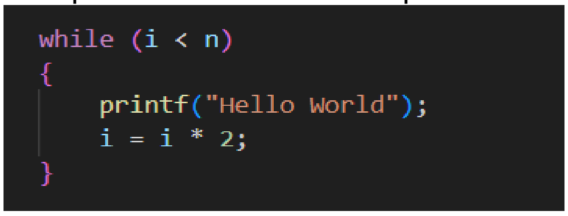
Examples

- Single Statement while loop



```
while (i < n)
    i = i * 2;
```

- Multiple Statement while loop

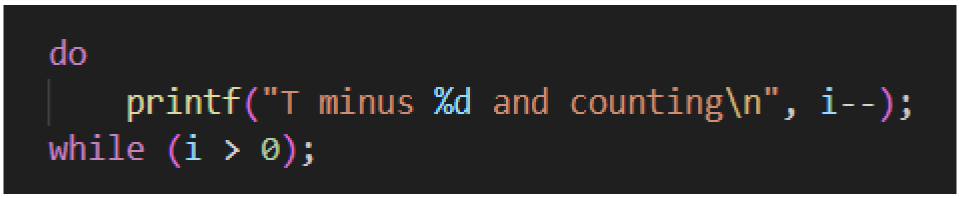


```
while (i < n)
{
    printf("Hello World");
    i = i * 2;
}
```

do Statement

- It’s a good idea to use braces in all do statements, whether or not they whether or not they're needed because a needed because a do statement without braces can easily be mistaken for a while statement:

Examples

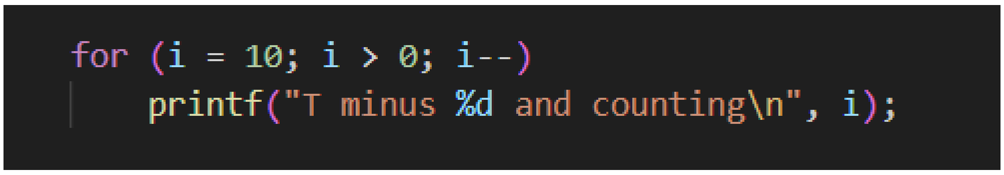


```c
do
    printf("T minus %d and counting\n", i--);
while (i > 0);
```

for statement

- The for statement is ideal for loops that have a “counting” variable but it is variable, but it's versatile enough to be used for other kinds of loops as well.
- The general form of the for statement: for (expr1; expr2; expr3) statement expr1, expr2, and expr3 are expressions.

Examples



```c
for (i = 10; i > 0; i--)
    printf("T minus %d and counting\n", i);
```

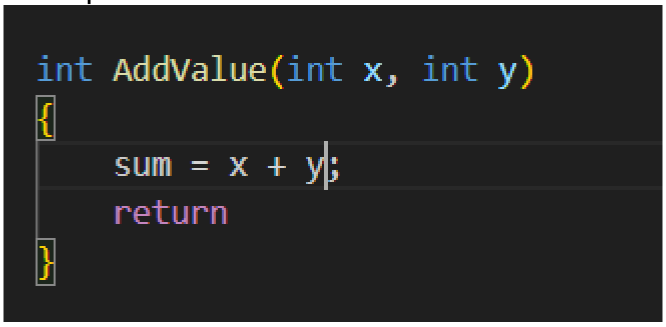
Return Statement

Its primary purpose is to terminate the execution of a function and optionally return a value to the calling code. When a function encounters a return statement, it immediately exits the function, regardless of its position within the function body. Execution resumes in the calling function at the point immediately following the function call.

Non-void Functions

For functions with non-void return types (int, double,...) the return statement must be followed by an expression of the corresponding type.

Example:

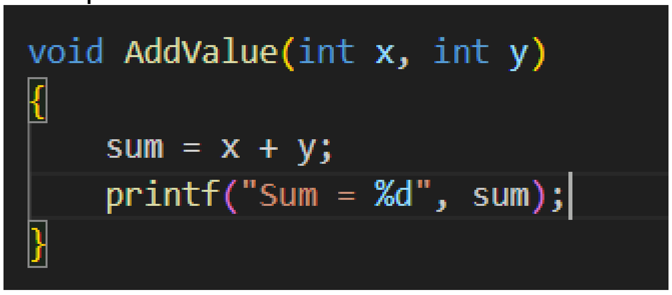


```c
int AddValue(int x, int y)
{
    sum = x + y;
    return
}
```

Void Functions

In void functions, the return statement can be used for control flow without returning value.

Example

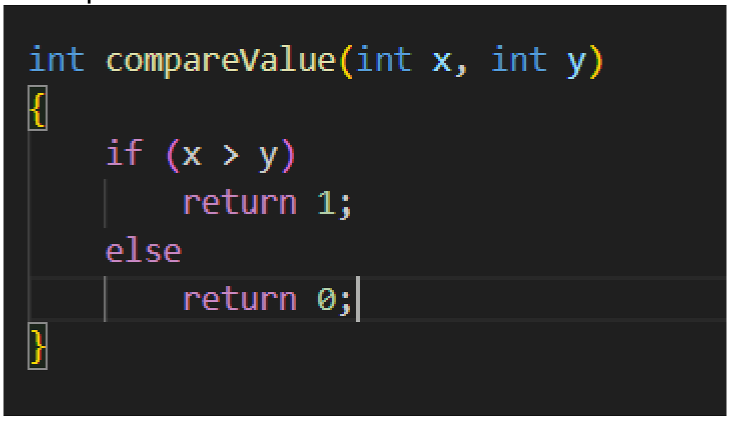


```c
void AddValue(int x, int y)
{
    sum = x + y;
    printf("Sum = %d", sum);
}
```

Multiple Return Statements

A function can have multiple return statements, but only one will be executed. The choice of which return statement to execute depends on program logic.

Example:

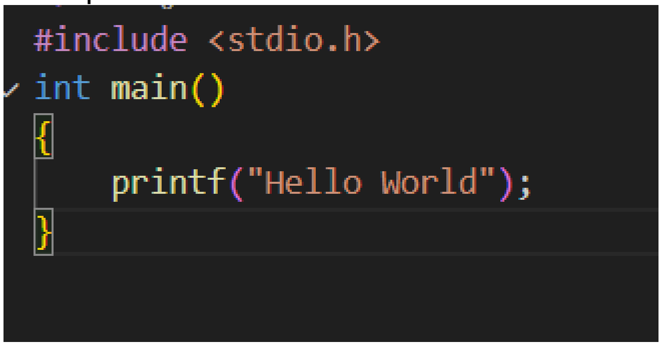


```c
int compareValue(int x, int y)
{
    if (x > y)
        return 1;
    else
        return 0;
}
```

Returning From Main

In the main function (entry point of a C program) a return statement is optional. If omitted, the compiler implicitly adds return 0; at the end of main.

Example :

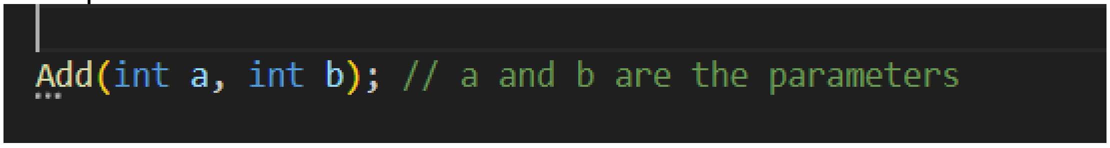


```c
#include <stdio.h>
int main()
{
    printf("Hello World");
}
```

Function Call Statement

A function call is an important part of the C programming language. It is called inside a program whenever it is required to call a function. It is only called by its name in the main() function of a program. We can pass the parameters to a function calling in the main() function.

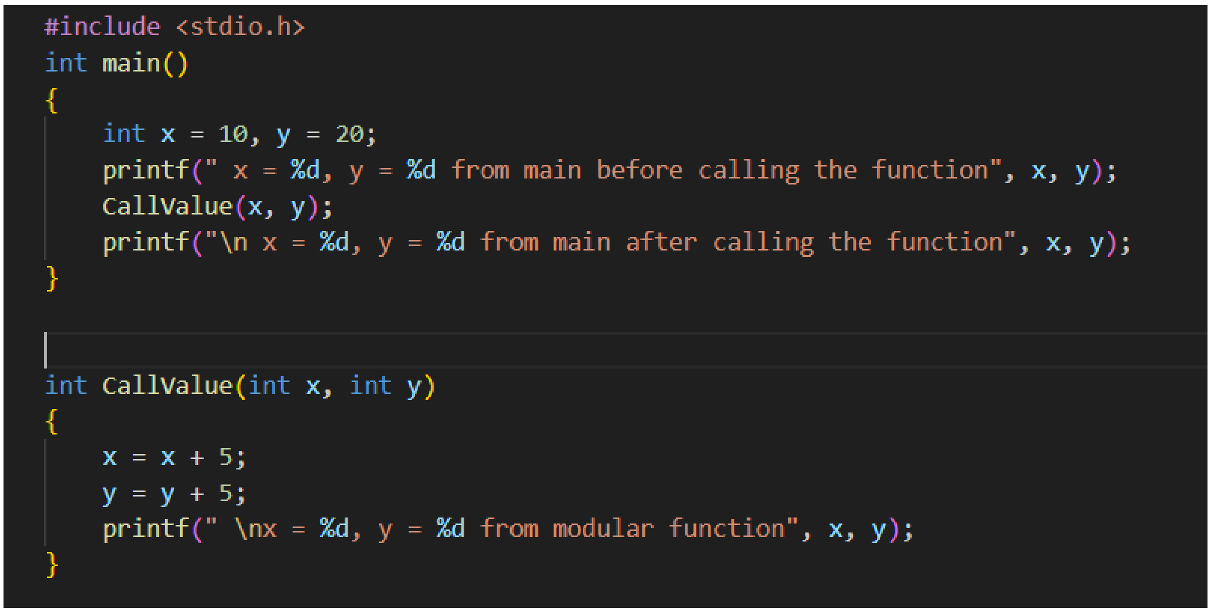
Example



```
Add(int a, int b); // a and b are the parameters
```

Call by Value

When single or multiple values of an actual argument are copied into the formal parameter of a function, the method is called the **Call by Value**. Hence, it does not alter the function's actual parameter using the formal parameter.

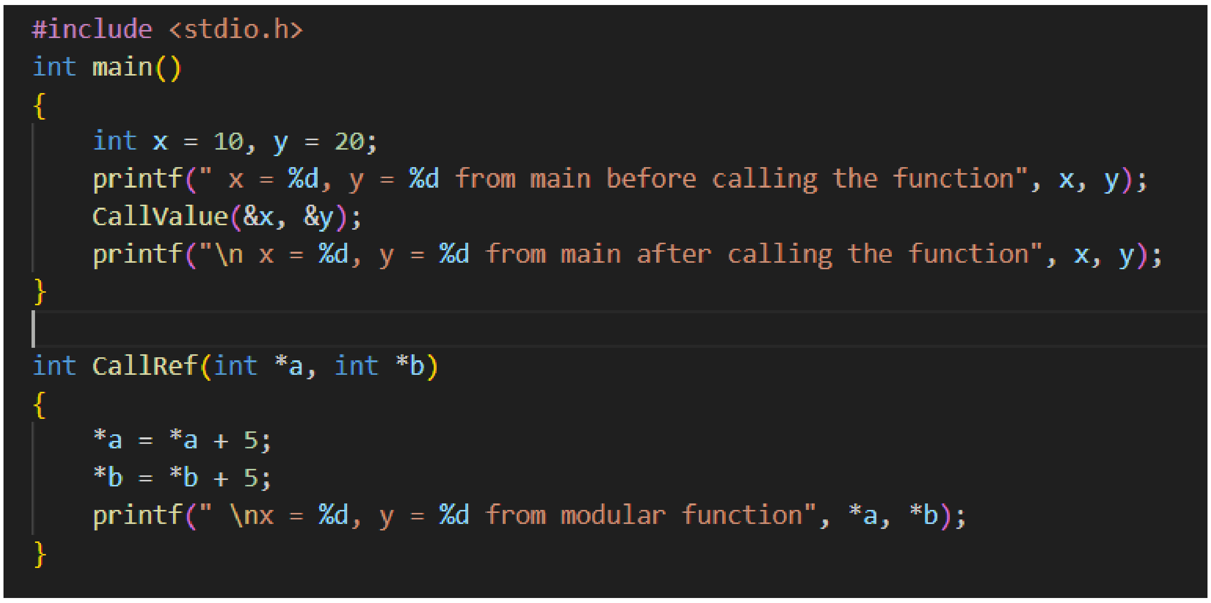


```
#include <stdio.h>
int main()
{
    int x = 10, y = 20;
    printf(" x = %d, y = %d from main before calling the function", x, y);
    CallValue(x, y);
    printf("\n x = %d, y = %d from main after calling the function", x, y);
}

int CallValue(int x, int y)
{
    x = x + 5;
    y = y + 5;
    printf(" \nx = %d, y = %d from modular function", x, y);
}
```

Call by Reference

In this method, the address of the actual argument is copied into the function call's formal parameter; the method is known as **Call by Reference**. If we make some changes in the formal parameters, it shows the effect in the value of the actual parameter.



```c
#include <stdio.h>
int main()
{
    int x = 10, y = 20;
    printf(" x = %d, y = %d from main before calling the function", x, y);
    CallValue(&x, &y);
    printf("\n x = %d, y = %d from main after calling the function", x, y);
}

int CallRef(int *a, int *b)
{
    *a = *a + 5;
    *b = *b + 5;
    printf(" \nx = %d, y = %d from modular function", *a, *b);
}
```

Escape Sequence

The escape sequence in C is the characters or the sequence of characters that can be used inside the string literal. The purpose of the escape sequence is to represent the characters that cannot be used normally using the keyboard. Some escape sequence characters are the part of ASCII charset but some are not.
Different escape sequences represent different characters but the output is dependent on the compiler you are using.

Escape Sequence	**Name**	**Description**
\a	Alarm or Beep	It is used to generate a bell sound in the C program.
\b	Backspace	It is used to move the cursor one place backward.
\f	Form Feed	It is used to move the cursor to the start of the next logical page.

\n	New Line	It moves the cursor to the start of the next line.
\r	Carriage Return	It moves the cursor to the start of the current line.
\t	Horizontal Tab	It inserts some whitespace to the left of the cursor and moves the cursor accordingly.
\v	Vertical Tab	It is used to insert vertical space.
\\	Backlash	Use to insert backslash character.
\’	Single Quote	It is used to display a single quotation mark.
\”	Double Quote	It is used to display double quotation marks.
\?	Question Mark	It is used to display a question mark.
\ooo	Octal Number	It is used to represent an octal number.
\xhh	Hexadecimal Number	It represents the hexadecimal number.

\0	NULL	It represents the NULL character.
\e	Escape sequence	It represents the ASCII escape character.
\s	Space Character	It represents the ASCII space character.
\d	Delete Character	It represents the ASCII DEL character.

References

- www.w3schools.com/c/index.php
- https://docs.actian.com/openroad/6.2/LangRef/Assignment_Statement.htm#ww308121
- http://www.di.uniba.it/~lanubile/fisica/ch06.pdf
- https://en.cppreference.com/w/c
- https://www.cs.yale.edu/homes/aspnes/pinewiki/C(2f)Statements.html#:~:text=C%2FStatements&text=The%20bodies%20of%20C%20functions,%2C%20for%2C%20and%20do
- https://data-flair.training/blogs/identifiers-and-variables-in-c/
- https://learn.microsoft.com/en-us/cpp/c-language/functions-c?view=msvc-170
- https://github.com/e3b0c442/keywords
- https://www.geeksforgeeks.org/escape-sequence-in-c/
- https://codeforwin.org/c-programming/operators-separators-c-programming#:~:text=In%20C%20programming%20every%20expression,one%20programming%20element%20from%20other.
- https://www.tutorialspoint.com/integer-literal-in-c-cplusplus-prefixes-and-suffixes#:~:text=Prefixes%20%E2%88%92%20Prefixes%20denotes%20the%20base,denotes%20a%20long%20long%20integer.
- https://www.geeksforgeeks.org/integer-literal-in-c-cpp-prefixes-suffixes/