Design of Compilers
Spring 2024
Language Specifications Document

Presented to : Dr. Wafaa Samy, Eng. Ahmed Salama

Table 1: Team 15

ID	Team Member
21P0180	Patrick Ramez Eskander Bolous
21P0173	Monica Hany Makram Derias
21P0082	Youssef Hany Emil Ayad
21P0206	Mark Saleh Sobhi
2100588	Zeynat Haytham Farouk Ghallab

Contents

1 Introduction

This document serves as a comprehensive guide to the language specifications for the C programming language. Its primary purpose is to provide a clear understanding of the fundamental constructs and rules inherent to the C language, essential for the development of the lexical and syntax analysis phases of the compiler project.
Understanding the language specifications is paramount as it lays the foundation for building an accurate and efficient compiler. The lexical analysis phase, facilitated by the lexer, relies on precise rules to tokenize the input source code accurately. Similarly, the syntax analysis phase, driven by the parser, necessitates adherence to the grammar rules to construct a meaningful parse tree.
By delineating the basic constructs, lexical rules, and grammar guidelines, this document aims to equip the project team with the necessary knowledge to implement robust lexer and parser components. A thorough comprehension of the language specifications ensures that the compiler accurately interprets and processes C code, paving the way for the successful compilation of programs written in the C programming language.

2 Keywords

Below is a comprehensive list of keywords of the C programming language. However, it's important to note that while the list includes additions introduced in the C23 standard, these are not implemented, they are just included for completeness, acknowledging their existence in the language's evolution.

- **alignas (C23)**
  - **Purpose:** Specifies alignment requirements for a variable or structure.
  - **Syntax:**

```
alignas(4) int x;
```

  - **Usage Restrictions:** Introduced in C23; used to control memory alignment.
  - **Standardization:** C23
- **alignof (C23)**
  - **Purpose:** Returns the alignment requirement of a type.
  - **Syntax:**

```
size_t alignment = alignof(double);
```

  - **Usage Restrictions:** Introduced in C23; used to determine the alignment of types.
  - **Standardization:** C23

- **auto**
  - **Purpose:** Declares automatic variables.
  - **Syntax:**

```
auto int x;
```

  - **Usage Restrictions:** Limited use due to automatic storage duration; generally unnecessary as storage duration is implied.
  - **Standardization:** Standard C
- **bool (C23)**
  - **Purpose:** Boolean data type.
  - **Syntax:**

```
bool flag = true;
```

  - **Usage Restrictions:** Introduced in C23; used for Boolean logic and conditions.
  - **Standardization:** C23
- **break**
  - **Purpose:** Exits from the current loop or switch statement.
  - **Syntax:**

```
break;
```

  - **Usage Restrictions:** Can only be used within loops or switch statements.
  - **Standardization:** Standard C
- **case**
  - **Purpose:** Labels a statement within a switch statement.
  - **Syntax:**

```
switch (variable) { case 1:
    /* Code */
break;
case 2:
    /* Code */
break;
default:
    /* Default code */ }
```

  - **Usage Restrictions:** Must be used within a switch statement; each case must end with a break statement or another control flow statement.
  - **Standardization:** Standard C

- **char**
  - **Purpose:** Declares a character data type.
  - **Syntax:**

```
char c = 'A';
```

  - **Usage Restrictions:** Represents a single character or small integer; occupies 1 byte of memory.
  - **Standardization:** Standard C
- **const**
  - **Purpose:** Declares a read-only variable.
  - **Syntax:**

```
const int MAX_VALUE = 100;
```

  - **Usage Restrictions:** Once initialized, the value cannot be modified.
  - **Standardization:** Standard C
- **constexpr (C23)**
  - **Purpose:** Specifies that a variable or function is constant and can be evaluated at compile time.
  - **Syntax:**

```
constexpr int SQUARE(int x) { return x * x; }
```

  - **Usage Restrictions:** Introduced in C23; used for compile-time evaluation of expressions.
  - **Standardization:** C23
- **continue**
  - **Purpose:** Skips the current iteration of a loop and continues with the next iteration.
  - **Syntax:**

```
continue;
```

  - **Usage Restrictions:** Can only be used within loops.
  - **Standardization:** Standard C

- **default**
  - **Purpose:** Specifies the default case in a switch statement.
  - **Syntax:**

```
switch (variable) {
case 1: /* Code */
break;
case 2: /* Code */
break;
default: /* Default code */ }
```

  - **Usage Restrictions:** Must be used within a switch statement; typically appears after all case labels.
  - **Standardization:** Standard C
- **do**
  - **Purpose:** Initiates a do-while loop.
  - **Syntax:**

```
do {
   /* Code */
} while (condition);
```

  - **Usage Restrictions:** None
  - **Standardization:** Standard C
- **double**
  - **Purpose:** Declares a double-precision floating-point data type.
  - **Syntax:**

```
double value = 3.14;
```

  - **Usage Restrictions:** Represents double-precision floating-point numbers.
  - **Standardization:** Standard C

- **else**
  - **Purpose:** Specifies an alternative statement in an if-else statement.
  - **Syntax:**

```
if (condition) {
  /* Code block */
 } else {
  /* Alternative code block */
 }
```

  - **Usage Restrictions:** None
  - **Standardization:** Standard C
- **enum**
  - **Purpose:** Declares an enumeration type.
  - **Syntax:**

```
enum Weekday { Monday, Tuesday, Wednesday, Thursday, Friday };
```

  - **Usage Restrictions:** None
  - **Standardization:** Standard C
- **extern**
  - **Purpose:** Declares a variable or function that is defined in another file or external to the current scope.
  - **Syntax:**

```
extern int x;
```

  - **Usage Restrictions:** Typically used in header files to declare functions or variables defined elsewhere.
  - **Standardization:** Standard C
- **false (C23)**
  - **Purpose:** Boolean false value.
  - **Syntax:**

```
bool flag = false;
```

  - **Usage Restrictions:** Introduced in C23; used to represent a false Boolean value.
  - **Standardization:** C23

- **float**
  - **Purpose:** Declares a floating-point data type.
  - **Syntax:**

```
float value = 3.14f;
```

  - **Usage Restrictions:** Represents single-precision floating-point numbers.
  - **Standardization:** Standard C
- **for**
  - **Purpose:** Initiates a for loop.
  - **Syntax:**

```
for (initialization; condition; increment) {
     /* Code */
 }
```

  - **Usage Restrictions:** None
  - **Standardization:** Standard C
- **goto**
  - **Purpose:** Transfers control to a labeled statement.
  - **Syntax:**

```
goto label;
label:
     /* Code */
```

  - **Usage Restrictions:** Use sparingly; can lead to spaghetti code.
  - **Standardization:** Standard C
- **if**
  - **Purpose:** Initiates an if statement.
  - **Syntax:**

```
if (condition) {
  /* Code block */
 }
```

  - **Usage Restrictions:** None
  - **Standardization:** Standard C

- **inline (C99)**
  - **Purpose:** Suggests to the compiler that a function should be inlined.
  - **Syntax:**

```
inline void functionName() { /* function body */ }
```

  - **Usage Restrictions:** Introduced in C99; used to suggest to the compiler that a function should be expanded in-place at the point of call.
  - **Standardization:** C99
- **int**
  - **Purpose:** Declares an integer data type.
  - **Syntax:**

```
int value = 10;
```

  - **Usage Restrictions:** Represents signed integers.
  - **Standardization:** Standard C
- **long**
  - **Purpose:** Declares a long integer data type.
  - **Syntax:**

```
long value = 1000000L;
```

  - **Usage Restrictions:** Represents long signed integers.
  - **Standardization:** Standard C
- **nullptr (C23)**
  - **Purpose:** Pointer literal representing a null pointer.
  - **Syntax:**

```
int* ptr = nullptr;
```

  - **Usage Restrictions:** Introduced in C23; used to represent a null pointer.
  - **Standardization:** C23

- **register**
  - **Purpose:** Suggests that a variable should be stored in a register for faster access.
  - **Syntax:**

```
register int count;
```

  - **Usage Restrictions:** Implementation-dependent; may be ignored by the compiler.
  - **Standardization:** Standard C
- **restrict (C99)**
  - **Purpose:** Specifies that a pointer is not aliased.
  - **Syntax:**

```
void func(int* restrict ptr);
```

  - **Usage Restrictions:** Introduced in C99; used to optimize code performance.
  - **Standardization:** C99
- **return**
  - **Purpose:** Exits from a function and returns a value.
  - **Syntax:**

```
return expression;
```

  - **Usage Restrictions:** Must be used within a function; terminates the function's execution and returns the specified value to the caller.
  - **Standardization:** Standard C
- **short**
  - **Purpose:** Declares a short integer data type.
  - **Syntax:**

```
short value = 10;
```

  - **Usage Restrictions:** Represents short signed integers.
  - **Standardization:** Standard C

- **signed**
  - **Purpose:** Specifies a signed data type.
  - **Syntax:**

```
signed int x;
```

  - **Usage Restrictions:** Specifies that a variable can hold negative values.
  - **Standardization:** Standard C
- **sizeof**
  - **Purpose:** Returns the size of a data type or variable in bytes.
  - **Syntax:**

```
size_t size = sizeof(int);
```

  - **Usage Restrictions:** Used to calculate the size of objects and data types.
  - **Standardization:** Standard C
- **static**
  - **Purpose:** Specifies that a variable retains its value between function calls.
  - **Syntax:**

```
static int count = 0;
```

  - **Usage Restrictions:** Static variables are initialized once and retain their value throughout program execution.
  - **Standardization:** Standard C
- **true (C23)**
  - **Purpose:** Boolean true value.
  - **Syntax:**

```
bool flag = true;
```

  - **Usage Restrictions:** Introduced in C23; used to represent a true Boolean value.
  - **Standardization:** C23

- **typeof (C23)**
  - **Purpose:** Returns the type of an expression.
  - **Syntax:**

```
typeof(expression);
```

  - **Usage Restrictions:** Introduced in C23; used to determine the type of an expression.
  - **Standardization:** C23
- **void**
  - **Purpose:** Specifies an empty data type or indicates that a function returns no value.
  - **Syntax:**

```
void function();
```

  - **Usage Restrictions:** Used as a placeholder for functions with no return value or to indicate an empty parameter list.
  - **Standardization:** Standard C
- **while**
  - **Purpose:** Initiates a while loop.
  - **Syntax:**

```
while (condition) {
    /* Code */
}
```

  - **Usage Restrictions:** None
  - **Standardization:** Standard C
- **volatile**
  - **Purpose:** Specifies that a variable can be modified externally.
  - **Syntax:**

```
volatile int x;
```

  - **Usage Restrictions:** Prevents the compiler from optimizing away accesses to the variable.
  - **Standardization:** Standard C

- **static_assert (C23)**
  - **Purpose:** Performs compile-time assertion checking.
  - **Syntax:**

```c
static_assert(sizeof(int) >= 4, "int must be at least 4 bytes");
```

  - **Usage Restrictions:** Introduced in C23; used to check conditions at compile-time.
  - **Standardization:** C23
- **switch**
  - **Purpose:** Initiates a switch statement. Syntax:
  - **Syntax:**

```c
   switch (expression) {
    case constant1:
        /* Code */
        break;
    case constant2:
        /* Code */
        break;
    default:
        /* Code */
}
```

  - **Usage Restrictions:** None
  - **Standardization:** Standard C
- **struct**
  - **Purpose:** Declares a structure type.
  - **Syntax:**

```c
struct MyStruct {
    int member1;
    float member2;
};
```

  - **Usage Restrictions:** Used to group related data members into a single unit.
  - **Standardization:** Standard C

- **thread_local (C23)**
  - **Purpose:** Performs compile-time assertion checking.
  - **Syntax:**

```
static_assert(sizeof(int) >= 4, "int must be at least 4 bytes");
```

  - **Usage Restrictions:** Introduced in C23; used to check conditions at compile-time.
  - **Standardization:** C23
- **typedef**
  - **Purpose:** Creates a type alias.
  - **Syntax:**

```
typedef int Integer;
```

  - **Usage Restrictions:** Used to create shorter or more descriptive names for data types.
  - **Standardization:** Standard C
- **typeof_unqual (C23)**
  - **Purpose:** Returns the type of an expression without qualifiers.
  - **Syntax:**

```
typeof_unqual(expression);
```

  - **Usage Restrictions:** Introduced in C23; used to obtain the type of an expression without qualifiers.
  - **Standardization:** C23
- **union**
  - **Purpose:** Declares a union type.
  - **Syntax:**

```
union MyUnion {
     int intValue;
     float floatValue;
 };
```

  - **Usage Restrictions:** Used to group multiple data members that share the same memory location.
  - **Standardization:** Standard C

- **unsigned**
  - **Purpose:** Declares an unsigned integer data type.
  - **Syntax:**

```
unsigned int value = 10;
```

  - **Usage Restrictions:** Represents non-negative integers.
  - **Standardization:** Standard C
- **_Alignas (C11)**
  - **Purpose:** Specifies alignment requirements for a variable or structure.
  - **Syntax:**

```
_Alignas(4) int x;
```

  - **Usage Restrictions:** Introduced in C11; used to control memory alignment.
  - **Standardization:** C11
- **_Alignof (C11)**
  - **Purpose:** Returns the alignment requirement of a type.
  - **Syntax:**

```
size_t alignment = _Alignof(int);
```

  - **Usage Restrictions:** Introduced in C11; used to determine the alignment requirement of a type.
  - **Standardization:** C11
- **_Atomic (C11)**
  - **Purpose:** Specifies atomic data types.
  - **Syntax:**

```
_Atomic(int) atomicInt;
```

  - **Usage Restrictions:** Introduced in C11; used to specify atomic data types.
  - **Standardization:** C11

- **_BitInt (C23)**
  - **Purpose:** Boolean type with explicitly defined size.
  - **Syntax:**

```
_BitInt(1) bit;
```

  - **Usage Restrictions:** Introduced in C23; used to specify Boolean types with explicitly defined size.
  - **Standardization:** C23
- **_Bool (C99)**
  - **Purpose:** Boolean type.
  - **Syntax:**

```
_Bool flag = 1;
```

  - **Usage Restrictions:** Introduced in C99; used to represent Boolean values.
  - **Standardization:** C99
- **Complex (C99)**
  - **Purpose:** Represents complex numbers.
  - **Syntax:**

```
complex float z = 1.0 + 2.0i;
```

  - **Usage Restrictions:** Introduced in C99; used to represent complex numbers.
  - **Standardization:** C99
- **_Decimal128 (C23)**
  - **Purpose:** Decimal floating-point type with 128-bit precision.
  - **Syntax:**

```
_Decimal128 d;
```

  - **Usage Restrictions:** Introduced in C23; used to specify decimal floating-point types with 128-bit precision.
  - **Standardization:** C23

- **_Decimal132 (C23)**
  - **Purpose:** Decimal floating-point type with 132-bit precision.
  - **Syntax:**

```
_Decimal132 d;
```

  - **Usage Restrictions:** Introduced in C23; used to specify decimal floating-point types with 132-bit precision.
  - **Standardization:** C23
- **_Decimal64 (C23)**
  - **Purpose:** Decimal floating-point type with 64-bit precision.
  - **Syntax:**

```
_Decimal64 d;
```

  - **Usage Restrictions:** Introduced in C23; used to specify decimal floating-point types with 64-bit precision.
  - **Standardization:** C23
- **_Generic (C11)**
  - **Purpose:** Allows generic programming in C.
  - **Syntax:**

```
_Generic(expression,
         int: "Integer",
         float: "Float",
         default: "Other"
         );
```

  - **Usage Restrictions:** Introduced in C11; used to write generic code based on the type of an expression.
  - **Standardization:** C11
- **_Imaginary (C99)**
  - **Purpose:** Represents imaginary numbers.
  - **Syntax:**

```
_Imaginary im = 1.0i;
```

  - **Usage Restrictions:** Introduced in C99; used to represent imaginary numbers.
  - **Standardization:** C99

- **_Noreturn (C11)**
  - **Purpose:** Indicates that a function does not return.
  - **Syntax:**

```
_Noreturn void function();
```

  - **Usage Restrictions:** Introduced in C11; used to indicate that a function does not return.
  - **Standardization:** C11
- **_Static_assert(C11)**
  - **Purpose:** Performs compile-time assertion checking.
  - **Syntax:**

```
_Static_assert(sizeof(int) >= 4, "int must be at least 4 bytes");
```

  - **Usage Restrictions:** Introduced in C11; used to check conditions at compile-time.
  - **Standardization:** C11
- **_Thread_local (C11)**
  - **Purpose:** Specifies that a variable has thread-local storage duration.
  - **Syntax:**

```
_Thread_local int x;
```

  - **Usage Restrictions:** Introduced in C11; specifies that each thread has its own instance of the variable.
  - **Standardization:** C11

3 Variable Identifiers

In C programming, identifiers are crucial as they represent variables, functions, and data types. It's essential to follow strict rules when naming identifiers to ensure clean and compliant codebases. In C, identifiers encompass a broad spectrum of characters, ranging from digits, underscores, lowercase, and uppercase Latin letters to Unicode characters, as facilitated by the \u and \U escape notation, a capability introduced since C99. Notably, with the advent of C23, Unicode characters utilized in identifiers must conform to the class XID Continue, enhancing the language's inclusivity and expressive power.

Before C23, identifiers in C could start with a non-digit character, like letters (Latin or Unicode), underscores, or other non-digit Unicode characters. But with C23, this changed. Now, the first character of an identifier must be a Unicode character from the XID Start class. This makes the language more consistent and reliable.

Identifiers in C are case-sensitive, meaning that lowercase and uppercase letters are considered different. This emphasizes the need for careful naming to avoid confusion and ensure clarity within the code. Additionally, every identifier must follow Normalization Form C standards, which helps ensure consistent representation across various systems and platforms, promoting compatibility and standardization.

Variable identifiers in C are essential for labeling storage locations of different data types, facilitating data access and manipulation. Adhering to strict naming rules, including case sensitivity and character usage, ensures consistency. Following standards like Normalization Form C promotes uniformity across systems. Understanding and adhering to naming conventions are vital for effective management of program logic and data.

Here's a concise list of all the entities that identifiers can represent in the C programming language:

1. Objects
2. Functions
3. Tags (struct, union, or enumerations)
4. Structure or union members
5. Enumeration constants
6. Typedef names
7. Label names
8. Macro names
9. Macro parameter names

It's important to note that every identifier, except for macro names or macro parameter names, has scope, belongs to a namespace, and may have linkage. Besides, the same identifier can denote different entities at different points in the program, or even at the same point if the entities are in different namespaces.

4 Function Identifiers

In C, function identifiers are names given to functions, they play a crucial role in organizing code and facilitating modular programming. The syntax for function identifiers is as follows:

Rules for Naming Identifiers:

- Identifiers must begin with a letter (uppercase or lowercase) or an underscore (_).
- After the first character, identifiers can include letters, digits, and underscores.
- C is case-sensitive, so uppercase and lowercase letters are distinct.
- Certain words, known as keywords (e.g., `int`, `char`, `for`, `while`), are reserved and cannot be used as identifiers.
- Identifiers should not exceed 31 characters (though some compilers may support longer identifiers).

Examples:

- Valid function identifier: `calculateSum`
- Valid variable identifier: `total_amount`
- Valid constant identifier: `MAX_SIZE`
- Invalid identifier: `3count` (starts with a digit)
- Invalid identifier: `for` (using a reserved keyword)

Function Declaration and Definition:

- Functions must be declared before they are used, either explicitly or implicitly.
- A function declaration specifies the function's name, return type, and parameter types (if any).
- A function definition provides the actual implementation of the function.

Remember that proper naming conventions and meaningful identifiers contribute to the readability and maintainability of the code. It's a good practice to choose descriptive names that reflect the purpose or functionality of the variable or function.

5 Data Types

C provides various data types that define the type and size of data that a variable can hold.

5.1 Primitive Data Types

In C, primitive data types represent the fundamental building blocks for variables. These include:

- `int`: Integer data type.
- `float`: Single-precision floating-point data type.
- `double`: Double-precision floating-point data type.
- `char`: Character data type.
- `void`: Represents the absence of a type or an empty set of values.

5.2 Derived Data Types

5.2.1 Arrays

Arrays are collections of elements of the same data type. The syntax for declaring an array is as follows:

```
data_type array_name[size];
```

5.2.2 Pointers

Pointers are variables that store memory addresses. They are used to manipulate data indirectly. The syntax for declaring a pointer is:

```
data_type *pointer_name;
```

5.2.3 Structures

Structures allow the creation of user-defined data types that group related variables under one name. The syntax is:

```
struct structure_name {
    data_type member1;
    data_type member2;
    // ... additional members
};
```

5.2.4 Unions

Unions are similar to structures, but different data members share the same memory location. The syntax is:

```
union union_name {
    data_type member1;
    data_type member2;
    // ... additional members
};
```

5.3 Modifiers

C provides modifiers to adjust the size of data types. These include:

- `short`: Reduces the size of integer data types.
- `long`: Increases the size of integer data types.
- `signed` and `unsigned`: Specify whether a data type can represent both positive and negative numbers or only non-negative numbers.

5.4 Typedef

The `typedef` keyword allows the creation of custom names for existing data types. It is particularly useful for enhancing code readability. The syntax is:

```
typedef existing_data_type new_type_name;
```

For example:

```
typedef int myInt;
myInt num = 5;
```

These data types, along with modifiers and typedef, form the foundation for variable declarations and manipulations in C.

6 Functions

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.

```
// function definition.
// defines a function with the name "sum" and with the body "{ return x+y; }"
int sum(int x, int y)
{
    return x + y;
}
```

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

Functions may accept zero or more parameters, which are initialized from the arguments of a function call operator, and may return a value to its caller by means of the return statement.

```
int n = sum(1, 2); // parameters x and y are initialized with the arguments 1 and 2
```

The body of a function is provided in a function definition. Each non-inline(since C99) function that is used in an expression (unless unevaluated) must be defined only once in a program.

There are no nested functions (except where allowed through non-standard compiler extensions): each function definition must appear at file scope, and functions have no access to the local variables from the caller:

```
int main(void) // the main function definition
{
    int sum(int, int); // function declaration (may appear at any scope)
    int x = 1;  // local variable in main
    sum(1, 2); // function call

//    int sum(int a, int b) // error: no nested functions
//    {
//        return  a + b;
//    }
}
int sum(int a, int b) // function definition
{
//    return x + a + b; //  error: main's x is not accessible within sum
    return a + b;
}
```

6.1 Function declarations

A function declaration introduces an identifier that designates a function and, optionally, specifies the types of the function parameters (the prototype). Function declarations (unlike definitions) may appear at block scope as well as file scope.

Syntax

In the declaration grammar of a function declaration, the type-specifier sequence, possibly modified by the declarator, designates the return type (which may be any type other than array or function type), and the declarator has one of three forms:

```
noptr-declarator ( parameter-list ) attr-spec-seq
noptr-declarator ( identifier-list ) attr-spec-seq
noptr-declarator ( ) attr-spec-seq
```

noptr-declarator - any declarator except unparenthesized pointer declarator. The identifier that is contained in this declarator is the identifier that becomes the function designator. *parameter-list* - either the single keyword void or a comma-separated list of parameters, which may end with an ellipsis parameter *identifier-list* - comma-separated list of identifiers, only possible if this declarator is used as part of old-style function definition *attr-spec-seq* - (C23)an optional list of attributes, applied to the function type

- New-style (C89) function declaration: This declaration both introduces the function designator itself and also serves as a function prototype for any future function call expressions, forcing conversions from argument expressions to the declared parameter types and compile-time checks for the number of arguments.

  ```
  int max(int a, int b); // declaration
  int n = max(12.01, 3.14); // OK, conversion from double to int
  ```

- Old-style (KR) function definition. This declaration does not introduce a prototype and any future function call expressions will perform default argument promotions and will invoke undefined behavior if the number of arguments doesn't match the number of parameters.

  ```
  int max(a, b)
      int a, b; // definition expects ints; the second call is undefined
  {
      return a > b ? a : b;
  }

  int n = max(true, (char)'a'); // calls max with two int args (after promotions)

  int n = max(12.01f, 3.14); // calls max with two double args (after promotions)
  ```

- Non-prototype function declaration: This declaration does not introduce a prototype(until C23). A new style function declaration equivalent to the parameter-list void

Explanation

The return type of the function, determined by the type specifier in specifiers-and-qualifiers and possibly modified by the declarator as usual in declarations, must be a non-array object type or the type void. If the function declaration is not a definition, the return type can be incomplete. The return type cannot be cvr-qualified: any qualified return type is adjusted to its unqualified version for the purpose of constructing the function type.

```
void f(char *s);                    // return type is void
int sum(int a, int b);              // return type of sum is int.
int (*foo(const void *p))[3];       // return type is pointer to array of 3 int

double const bar(void);             // declares function of type double(void)
double (*barp)(void) = bar;         // OK: barp is a pointer to double(void)
double const (*barpc)(void) = barp; // OK: barpc is also a pointer to double(void)
```

Function declarators can be combined with other declarators as long as they can share their type specifiers and qualifiers

```
int f(void), *fip(), (*pfi)(), *ap[3]; // declares two functions and two objects
inline int g(int), n; // Error: inline qualifier is for functions only
typedef int array_t[3];
array_t a, h(); // Error: array type cannot be a return type for a function
```

If a function declaration appears outside of any function, the identifier it introduces has file scope and external linkage, unless static is used or an earlier static declaration is visible. If the declaration occurs inside another function, the identifier has block scope (and also either internal or external linkage).

```
int main(void)
{
    int f(int); // external linkage, block scope
    f(1); // definition needs to be available somewhere in the program
}
```

The parameters in a declaration that is not part of a function definition(until C23) do not need to be named:

```
int f(int, int); // declaration
// int f(int, int) { return 7; } // Error: parameters must be named in definitions
// This definition is allowed since C23
```

Each parameter in a parameter-list is a declaration that introduced a single variable, with the following additional properties:

- the identifier in the declarator is optional (except if this function declaration is part of a function definition)

```
int f(int, double); // OK
int g(int a, double b); // also OK
// int f(int, double) { return 1; } // Error: definition must name parameters
// This definition is allowed since C23
```

- the only storage class specifier allowed for parameters is register, and it is ignored in function declarations that are not definitions

```
int f(static int x); // Error
int f(int [static 10]); // OK (array index static is not a storage class specifier)
```

- any parameter of array type is adjusted to the corresponding pointer type, which may be qualified if there are qualifiers between the square brackets of the array declarator

```
int f(int[]); // declares int f(int*)
int g(const int[10]); // declares int g(const int*)
int h(int[const volatile]); // declares int h(int * const volatile)
int x(int[*]); // declares int x(int*)
```

- any parameter of function type is adjusted to the corresponding pointer type

```
int f(char g(double)); // declares int f(char (*g)(double))
int h(int(void)); // declares int h(int (*)(void))
```

- the parameter list may terminate with , ... or be ...(since C23), see variadic functions for details.

```
int f(int, ...);
```

- parameters cannot have type void (but can have type pointer to void). The special parameter list that consists entirely of the keyword void is used to declare functions that take no parameters.

```
int f(void); // OK
int g(void x); // Error
```

- any identifier that appears in a parameter list that could be treated as a typedef name or as a parameter name is treated as a typedef name: int f(size_t, uintptr_t) is parsed as a new-style declarator for a function taking two unnamed parameters of type size_t and uintptr_t, not an old-style declarator that begins the definition of a function taking two parameters named "size_t" and "uintptr_t"
- parameters may have incomplete type and may use the VLA notation [*](since C99) (except that in a function definition, the parameter types after array-to-pointer and function-to-pointer adjustment must be complete)

Notes

Unlike in C++, the declarators f() and f(void) have different meaning: the declarator f(void) is a new-style (prototype) declarator that declares a function that takes no parameters. The declarator f() is a declarator that declares a function that takes unspecified number of parameters (unless used in a function definition)

```
int f(void); // declaration: takes no parameters
int g(); // declaration: takes unknown parameters

int main(void) {
    f(1); // compile-time error
    g(2); // undefined behavior
}

int f(void) { return 1; } // actual definition
int g(a,b,c,d) int a,b,c,d; { return 2; } // actual definition
```

Unlike in a function definition, the parameter list may be inherited from a typedef

```
typedef int p(int q, int r); // p is a function type int(int, int)
p f; // declares int f(int, int)
```

6.2 Function definitions

A function definition associates the function body (a sequence of declarations and statements) with the function name and parameter list. Unlike function declaration, function definitions are allowed at file scope only (there are no nested functions).

C supports two different forms of function definitions:

```
attr-spec-seq(optional) specifiers-and-qualifiers parameter-list-declarator function-body

specifiers-and-qualifiers identifier-list-declarator declaration-list function-body
```

attr-spec-seq - (C23)an optional list of attributes, applied to the function *specifiers-and-qualifiers* - a combination of type specifiers that, possibly modified by the declarator, form the return type storage class specifiers, which determine the linkage of the identifier (static, extern, or none) function specifiers inline, _Noreturn, or none *parameter-list-declarator* - a declarator for a function type which uses a parameter list to designate function parameters *identifier-list-declarator* - a declarator for a function type which uses a identifier list to designate function parameters *declaration-list* - sequence of declarations that declare every identifier in identifier-list-declarator. These declarations cannot use initializers and the only storage-class specifier allowed is register. *function-body* - a compound statement, that is a brace-enclosed sequence of declarations and statements, that is executed whenever this function is called

- New-style (C89) function definition. This definition both introduces the function itself and serves as a function prototype for any future function call expressions, forcing conversions from argument expressions to the declared parameter types.

```
int max(int a, int b)
{
    return a>b?a:b;
}

double g(void)
{
    return 0.1;
}
```

- Old-style (KR) function definition. This definition does not behave as a prototype and any future function call expressions will perform default argument promotions.

```
int max(a, b)
int a, b;
{
    return a>b?a:b;
}
double g()
{
    return 0.1;
}
```

Explanation

As with function declarations, the return type of the function, determined by the type specifier in specifiers-and-qualifiers and possibly modified by the declarator as usual in declarations, must be a complete non-array object type or the type void. If the return type would be cvr-qualified, it is adjusted to its unqualified version for the purpose of constructing the function type.

```
void f(char *s) { puts(s); } // return type is void
int sum(int a, int b) { return a+b: } // return type is int
int (*foo(const void *p))[3] { // return type is pointer to array of 3 int
    return malloc(sizeof(int[3]));
}
```

As with function declarations, the types of the parameters are adjusted from functions to pointers and from arrays to pointers for the purpose of constructing the function type and the top-level cvr-qualifiers of all parameter types are ignored for the purpose of determining compatible function type.

```
int f(int, int); // declaration
// int f(int, int) { return 7; } // Error until C23, OK since C23
int f(int a, int b) { return 7; } // definition
int g(void) { return 8; } // OK: void doesn't declare a parameter
```

Within the function body, every named parameter is an lvalue expression, they have automatic storage duration and block scope. The layout of the parameters in memory (or if they are stored in memory at all) is unspecified: it is a part of the calling convention.

```
int main(int ac, char **av)
{
    ac = 2; // parameters are lvalues
    av = (char *[]){"abc", "def", NULL};
    f(ac, av);
}
```

Notes

The argument list must be explicitly present in the declarator, it cannot be inherited from a typedef

```
typedef int p(int q, int r); // p is a function type int(int, int)
p f { return q + r; } // Error
```

7 Statements:

7.1 Assignment Statement

an assignment statement is used to assign a value to a variable. The syntax is straightforward:

Syntax:

`variable = value;`
`variable` is the name of the variable, and `value` is the data you want to store in that variable. The assignment operator (`=`) is used to perform this operation.

For example:

```
/*int num;          // Declare an integer variable named num
num = 42;         // Assign the value 42 to the variable num */
```

In this case, the variable `num` is declared as an integer, and the assignment statement sets its value to 42.
You can also perform calculations during assignment:

```
/*int a = 5;
int b = 7;
int sum = a + b;  // The value of sum is now 12 */
```

It's crucial to understand that the right-hand side (RHS) of the assignment statement is evaluated first, and then the result is stored in the left-hand side (LHS) variable.

Assignment statements are fundamental for manipulating data in C and are used extensively in creating algorithms, performing calculations, and managing program state.

7.2 Declaration Statement

A declaration is a C language construct that introduces one or more identifiers into the program and specifies their meaning and properties. Declarations may appear in any scope. Each declaration ends with a semicolon (just like a statement).

Syntax:

1. `specifiers-and-qualifiers declarators-and-initializers(optional) ;`
2. `attr-spec-seq specifiers-and-qualifiers declarators-and-initializers ;`
3. `attr-spec-seq ;`

`specifiers-and-qualifiers` - whitespace-separated list of, in any order,

- type specifiers:
  - `void`
  - the name of an arithmetic type
  - the name of an atomic type
  - a name earlier introduced by a typedef declaration
  - struct, union, or enum specifier
  - a typeof specifier (since C23)
- zero or one storage-class specifiers: `typedef`, `constexpr`, `auto`, `register`, `static`, `extern`, `_Thread_local`
- zero or more type qualifiers: `const`, `volatile`, `restrict`, `_Atomic`
- (only when declaring functions), zero or more function specifiers: `inline`, `_Noreturn`
- zero or more alignment specifiers: `_Alignas`

`declarators-and-initializers` - comma-separated list of declarators (each declarator provides additional type information and/or the identifier to declare). Declarators may be accompanied by initializers. The enum, struct, and union declarations may omit declarators, in which case they only introduce the enumeration constants and/or tags.

`attr-spec-seq` - optional list of attributes, applied to the declared entities, or forms an attribute declaration if it appears alone.

Explanation:

Simple declaration. Introduces one or more identifiers which denote objects, functions, struct/union/enum tags, typedefs, or enumeration constants. Attribute declaration. Does not declare any identifier and has implementation-defined meaning if the meaning is not specified by the standard.
For example,

```
/*
int a, *b=NULL; // "int" is the type specifier,
                // "a" is a declarator
                // "*b" is a declarator and NULL is its initializer
const int *f(void); // "int" is the type specifier
                    // "const" is the type qualifier
                    // "*f(void)" is the declarator
enum COLOR {RED, GREEN, BLUE} c;
// "enum COLOR {RED, GREEN, BLUE}" is the type specifier
                     // "c" is the declarator
*/
```

The type of each identifier introduced in a declaration is determined by a combination of the type specified by the type specifier and the type modifications applied by its declarator. The type of a variable might also be inferred if the auto specifier is used. Attributes may appear in `specifiers-and-qualifiers`, in which case they apply to the type determined by the preceding specifiers.

7.3 Return Statement

Terminates the current function and returns the specified value to the caller function.

Syntax:

1. `attr-spec-seq(optional) return expression ;`
2. `attr-spec-seq(optional) return ;`

`expression` - expression used for initializing the return value of the function
`attr-spec-seq` - optional list of attributes, applied to the return statement

Explanation:

Evaluates the expression, terminates the current function, and returns the result of the expression to the caller (the value returned becomes the value of the function call expression). Only valid if the function return type is not `void`. Terminates the current function. Only valid if the function return type is `void`. If the type of the expression is different from the return type of the function, its value is converted as if by assignment to an object whose type is the return type of the function, except that overlap between object representations is permitted:

```
/*
struct s { double i; } f(void); // function returning struct s
union { struct { int f1; struct s f2; } u1;
        struct { struct s f3; int f4; } u2; } g;
struct s f(void)
{
    return g.u1.f2;
}
int main(void)
{
// g.u2.f3 = g.u1.f2; // undefined behavior (overlap in assignment)
   g.u2.f3 = f();     // well-defined
}*/
```

If the return type is a real floating type, the result may be represented in greater range and precision than implied by the new type. Reaching the end of a function returning `void` is equivalent to `return;`. Reaching the end of any other value-returning function is undefined behavior if the result of the function is used in an expression (it is allowed to discard such return value). For `main`, see main function.

7.4 Iterative Statement

- **for loop**
  Executes a loop.
  Used as a shorter equivalent of while loop.

Syntax:

1. attr-spec-seq(optional) for (init-clause ; cond-expression ; iteration-expression) loop-statement

Explanation:

Behaves as follows: init-clause may be an expression or a declaration.

An init-clause, which is an expression, is evaluated once, before the first evaluation of cond-expression and its result is discarded.

An init-clause, which is a declaration, is in scope in the entire loop body, including the remainder of init-clause, the entire cond-expression, the entire iteration-expression and the entire loop-statement. Only auto and register storage class specifiers are allowed for the variables declared in this declaration.

cond-expression is evaluated before the loop body. If the result of the expression is zero, the loop statement is exited immediately.

iteration-expression is evaluated after the loop body and its result is discarded. After evaluating iteration-expression, control is transferred to cond-expression.

init-clause, cond-expression, and iteration-expression are all optional. If cond-expression is omitted, it is replaced with a non-zero integer constant, which makes the loop endless:

```
/*for(;;) {
   printf("endless loop!");
}*/
loop-statement is not optional, but it may be a null statement:
/*for(int n = 0; n < 10; ++n, printf("%d\n", n))
    ; // null statement
*/
```

If the execution of the loop needs to be terminated at some point, a break statement can be used anywhere within the loop-statement.

The continue statement used anywhere within the loop-statement transfers control to iteration-expression.

- **while loop**
  Executes a statement repeatedly, until the value of expression becomes equal to zero. The test takes place before each iteration.

Syntax:

1. attr-spec-seq(optional) while (expression) statement

expression - any expression of scalar type. This expression is evaluated before each iteration, and if it compares equal to zero, the loop is exited.
Statement - any statement, typically a compound statement, which serves as the body of the loop
attr-spec-seq - optional list of attributes, applied to the loop statement

Explanation:

A while statement causes the statement (also called the loop body) to be executed repeatedly until the expression (also called controlling expression) compares equal to zero. The repetition occurs regardless of whether the loop body is entered normally or by a goto into the middle of statement.

The evaluation of expression takes place before each execution of statement (unless entered by a goto). If the controlling expression needs to be evaluated after the loop body, the do-while loop may be used.

If the execution of the loop needs to be terminated at some point, break statement can be used as a terminating statement.

If the execution of the loop needs to be continued at the end of the loop body, continue statement can be used as a shortcut.

A program with an endless loop has undefined behavior if the loop has no observable behavior (I/O, volatile accesses, atomic or synchronization operation) in any part of its statement or expression. This allows the compilers to optimize out all unobservable loops without proving that they terminate. The only exceptions are the loops where expression is a constant expression; while(true) is always an endless loop.

- **do-while loop**
  Executes a statement repeatedly until the value of the condition expression becomes false. The test takes place after each iteration.

Syntax:

attr-spec-seq(optional) do statement while (expression) ;
expression - any expression of scalar type. This expression is evaluated after each iteration, and if it compares equal to zero, the loop is exited.
statement - any statement, typically a compound statement, which is the body of the loop
attr-spec-seq - optional list of attributes, applied to the loop statement

Explanation:

A do-while statement causes the statement (also called the loop body) to be executed repeatedly until the expression (also called controlling expression) compares equal to 0. The repetition occurs regardless of whether the loop body is entered normally or by a goto into the middle of statement.

The evaluation of expression takes place after each execution of statement (whether entered normally or by a goto). If the controlling expression needs to be evaluated before the loop body, the while loop or the for loop may be used.

If the execution of the loop needs to be terminated at some point, break statement can be used as terminating statement.

If the execution of the loop needs to be continued at the end of the loop body, continue statement can be used as a shortcut.

A program with an endless loop has undefined behavior if the loop has no observable behavior (I/O, volatile accesses, atomic or synchronization operation) in any part of its statement or expression. This allows the compilers to optimize out all unobservable loops without proving that they terminate. The only exceptions are the loops where expression is a constant expression; do ... while(true); is always an endless loop.

7.5 Conditional Statements

- **if statement**
  Conditionally executes code.
  Used where code needs to be executed only if some condition is true.

Syntax:

1. attr-spec-seq(optional) if (expression) statement-true
2. attr-spec-seq(optional) if (expression) statement-true else statement-false

attr-spec-seq - optional list of attributes, applied to the if statement
expression- an expression of any scalar type
statement-true - any statement (often a compound statement), which is executed if expression compares not equal to 0
statement-false - any statement (often a compound statement), which is executed if expression compares equal to 0

Explanation:

expression must be an expression of any scalar type. If expression compares not equal to the integer zero, statement-true is executed.
In the form (2), if expression compares equal to the integer zero, statement_false is executed.
As with all other selection and iteration statements, the entire if-statement has its own block scope:

```
/*enum {a, b};
int different(void)
{
    if (sizeof(enum {b, a}) != sizeof(int))
        return a; // a == 1
    return b; // b == 0 in C89, b == 1 in C99
}
*/
```

- **switch statement** Executes code according to the value of an integral argument. Used where one or several out of many branches of code need to be executed according to an integral value.

Syntax:

1. attr-spec-seq(optional) switch (expression) statement
   attr-spec-seq - optional list of attributes, applied to the switch statement
   expression - any expression of integer type (char, signed or unsigned integer, or enumeration)
   statement - any statement (typically a compound statement). case: and default: labels are permitted in statement, and break; statement has special meaning.

2. **case constant-expression :** statement
3. **attr-spec-seq(optional) case constant-expression :** statement(optional)
4. **default :** statement
5. **attr-spec-seq(optional) default :** statement(optional)
   constant-expression - any integer constant expression
   attr-spec-seq - optional list of attributes, applied to the label

Explanation:

The body of a switch statement may have an arbitrary number of case: labels, as long as the values of all constant-expressions are unique (after conversion to the promoted type of expression). At most one default: label may be present (although nested switch statements may use their own default: labels or have case: labels whose constants are identical to the ones used in the enclosing switch).

If expression evaluates to the value that is equal to the value of one of constant-expressions after conversion to the promoted type of expression, then control is transferred to the statement that is labeled with that constant-expression.

If expression evaluates to a value that doesn't match any of the case: labels, and the default: label is present, control is transferred to the statement labeled with the default: label. If expression evaluates to a value that doesn't match any of the case:

labels, and the default: label is not present, none of the switch body is executed.

The break statement, when encountered anywhere in statement, exits the switch statement:

```
/*switch(1) {
    case 1 : puts("1"); // prints "1",
    case 2 : puts("2"); // then prints "2" ("fall-through")
}
*/
/*switch(1) {
    case 1 : puts("1"); // prints "1"
             break;     // and exits the switch
    case 2 : puts("2");
             break;
}
*/
```

7.6 Function Call Statement

Syntax:

1. **expression (argument-list(optional))**
   expression - any expression of pointer-to-function type.
   argument-list - comma-separated list of expressions (which cannot be comma operators) of any complete object type. May be omitted when calling functions that take no arguments.

7.7 Jump Statement

- **continue statement**
  Causes the remaining portion of the enclosing for, while or do-while loop body to be skipped.
  Used when it is otherwise awkward to ignore the remaining portion of the loop using conditional statements.

Syntax:

1. attr-spec-seq(optional) continue ;
   attr-spec-seq - optional list of attributes, applied to the continue statement

Explanation:

The continue statement causes a jump, as if by goto, to the end of the loop body (it may only appear within the loop body of for, while, and do-while loops).

For while loop, it acts as

```
/*while (/* ... */) {
   // ...
   continue; // acts as goto contin;
   // ...
   contin:;
}*/
For do-while loop, it acts as:
/*do {
    // ...
    continue; // acts as goto contin;
    // ...
    contin:;
} while (/* ... */); */
For for loop, it acts as:
/*for (/* ... */) {
    // ...
    continue; // acts as goto contin;
    // ...
    contin:;
}*/
```

- **break statement** Causes the enclosing for, while or do-while loop or switch statement to terminate.
  Used when it is otherwise awkward to terminate the loop using the condition expression and conditional statements.

Syntax:

1. attr-spec-seq (optional) break ;
   attr-spec-seq - optional list of attributes, applied to the break statement
   Appears only within the statement of a loop body (while, do-while, for) or within the statement of a switch.

Explanation:

After this statement the control is transferred to the statement or declaration immediately following the enclosing loop or switch, as if by goto.

- **goto statement** Transfers control unconditionally to the desired location.
Used when it is otherwise impossible to transfer control to the desired location using conventional constructs.

Syntax:

1. attr-spec-seq(optional) goto label ;
   label - target label for the goto statement
   attr-spec-seq - optional list of attributes, applied to the goto statement

Explanation:

The goto statement causes an unconditional jump (transfer of control) to the statement prefixed by the named label (which must appear in the same function as the goto statement), except when this jump would enter the scope of a variable-length array or another variably-modified type.

A label is an identifier followed by a colon (:) and a statement.Labels are the only identifiers that have function scope: they can be used (in a goto statement) anywhere in the same function in which they appear. There may be multiple labels before any statement.

Entering the scope of a non-variably modified variable is permitted:

```
/* goto lab1; // OK: going into the scope of a regular variable
    int n = 5;
lab1:; // Note, n is uninitialized, as if declared by int n;

//   goto lab2;   // Error: going into the scope of two VM types
     double a[n]; // a VLA
     int (*p)[n]; // a VM pointer
lab2: */
If goto leaves the scope of a VLA, it is deallocated
(and may be reallocated if its initialization is executed again):
/*{
   int n = 1;
label:;
   int a[n]; // re-allocated 10 times, each with a different size
   if (n++ < 10) goto label; // leaving the scope of a VM
}*/
```

8 Expressions

8.1 Arithmetic

1. **Unary Plus Operator (+):**
   - **Description:** The unary plus operator (+) is used to indicate a positive value.
   - **Usage:** It retains the value of the operand without any change.
   - **Example:** ‘+a‘ preserves the value of ‘a‘ after promotions.
2. **Unary Minus Operator (-):**
   - **Description:** The unary minus operator (-) is used to negate the value of its operand.
   - **Usage:** It negates the value of the operand.
   - **Example:** ‘-a‘ represents the negative value of ‘a‘.
3. **Addition Operator (+):**
   - **Description:** The addition operator (+) is used to perform arithmetic addition between two operands.
   - **Usage:** It adds the value of the operand on the left-hand side to the value of the operand on the right-hand side.
   - **Example:** ‘a + b‘ computes the addition of ‘a‘ and ‘b‘.
4. **Subtraction Operator (-):**
   - **Description:** The subtraction operator (-) is used to subtract the value of the right-hand operand from the value of the left-hand operand.
   - **Usage:** It subtracts the value of the operand on the right-hand side from the value of the operand on the left-hand side.
   - **Example:** ‘a - b‘ calculates the subtraction of ‘b‘ from ‘a‘.
5. **Multiplication Operator (*):**
   - **Description:** The multiplication operator (*) is used to perform multiplication between two operands.
   - **Usage:** It multiplies the value of the operand on the left-hand side by the value of the operand on the right-hand side.
   - **Example:** ‘a * b‘ computes the product of ‘a‘ and ‘b‘.
6. **Division Operator (/):**
   - **Description:** The division operator (/) is used to divide the value of the left-hand operand by the value of the right-hand operand.
   - **Usage:** It divides the value of the operand on the left-hand side by the value of the operand on the right-hand side.
   - **Example:** ‘a / b‘ computes the division of ‘a‘ by ‘b‘.

7. **Modulus Operator (%):**
    - **Description:** The modulus operator (%) returns the remainder of dividing the left-hand operand by the right-hand operand.
    - **Usage:** It computes the remainder when the value of the left-hand operand is divided by the value of the right-hand operand.
    - **Example:** `a % b` calculates the remainder of `a` divided by `b`.
8. **Bitwise NOT Operator (~):**
    - **Description:** The bitwise NOT operator (~) inverts each bit of its operand.
    - **Usage:** It flips all the bits in the binary representation of the operand.
    - **Example:** `~a` represents the bitwise NOT of `a`.
9. **Bitwise Left Shift Operator (<<):**
    - **Description:** The bitwise left shift operator (<<) shifts the bits of its left operand to the left by a number of positions specified by the right operand.
    - **Usage:** It shifts the bits to the left, padding with zeros on the right.
    - **Example:** `a <<b` performs a left shift of `a` by `b` positions.
10. **Bitwise OR Operator (|):**
    - **Description:** The bitwise OR operator (|) performs a bitwise OR operation between corresponding bits of its operands.
    - **Usage:** It evaluates to 1 if at least one corresponding bit is 1; otherwise, it evaluates to 0.
    - **Example:** `a | b` computes the bitwise OR of `a` and `b`.
11. **Bitwise XOR Operator (^):**
    - **Description:** The bitwise XOR operator (^) performs a bitwise XOR (exclusive OR) operation between corresponding bits of its operands.
    - **Usage:** It evaluates to 1 if the corresponding bits are different; otherwise, it evaluates to 0.
    - **Example:** `a ^ b` computes the bitwise XOR of `a` and `b`.
12. **Bitwise Right Shift Operator (>>):**
    - **Description:** The bitwise right shift operator (>>) shifts the bits of its left operand to the right by a number of positions specified by the right operand.
    - **Usage:** It shifts the bits to the right, preserving the sign bit for signed types and padding with zeros for unsigned types.
    - **Example:** `a >> b` performs a right shift of `a` by `b` positions.
13. **Bitwise AND Operator (&):**
    - **Description:** The bitwise AND operator (&) performs a bitwise AND operation between corresponding bits of its operands.
    - **Usage:** It evaluates to 1 if both corresponding bits are 1; otherwise, it evaluates to 0.
    - **Example:** `a & b` computes the bitwise AND of `a` and `b`.

8.2 Boolean

1. **Logical AND Operator (`&&`):**
   - **Description:** The logical AND operator (`&&`) returns true if both operands are true; otherwise, it returns false.
   - **Usage:** It evaluates to true if both operands are true; otherwise, it evaluates to false.
   - **Example:** a `&&` b returns true if both a and b are true.
2. **Logical OR Operator (`||`):**
   - **Description:** The logical OR operator (`||`) returns true if at least one of the operands is true; otherwise, it returns false.
   - **Usage:** It evaluates to true if at least one operand is true; otherwise, it evaluates to false.
   - **Example:** `a || b` returns true if either `a` or `b` is true.
3. **Logical NOT Operator (!):**
   - **Description:** The logical NOT operator (!) negates the value of its operand; if the operand is true, it returns false, and vice versa.
   - **Usage:** It flips the boolean value of its operand.
   - **Example:** !a returns true if a is false, and false if a is true.
4. **Relational Operators:**
   - **Description:** Relational operators are used to compare the values of operands.
   - **List of Relational Operators:**
     - **Equal to (==): Returns true if the operands are equal; otherwise, it returns false.**
     - **Not equal to (!=): Returns true if the operands are not equal; otherwise, it returns false.**
     - **Greater than (>): Returns true if the left operand is greater than the right operand; otherwise, it returns false.**
     - **Less than (<): Returns true if the left operand is less than the right operand; otherwise, it returns false.**
     - **Greater than or equal to (>=): Returns true if the left operand is greater than or equal to the right operand; otherwise, it returns false.**
     - **Less than or equal to (<=): Returns true if the left operand is less than or equal to the right operand; otherwise, it returns false.**
   - **Usage:** These operators are used to form conditions in if statements, loops, and other control flow structures.
   - **Example:** a == b returns true if a is equal to b.

9 Conclusion

In conclusion, understanding expressions and statements in the C programming language is fundamental for developing efficient and robust software. By grasping the intricacies of iterative, conditional, function call, and jump statements, as well as arithmetic and boolean expressions, programmers gain the ability to control program flow, perform complex computations, and make decisions based on various conditions. With diligent adherence to syntax rules and careful consideration of program logic, developers can write clear, maintainable, and portable code that meets the demands of diverse computing environments. Mastery of these concepts empowers programmers to create software solutions that fulfill a wide range of requirements, making them essential components of any programmer's toolkit.

10 References

- https://en.cppreference.com/w/c/language