ccpy: Specification Document

CS335 2021-22 II - Compiler Design

Group 10 - Naman (190527), Soham (180771), Ayush (180178), Lakshay (180378)
GitHub Repository: https://github.com/namangup/ccpy

Jan 24, 2021

Contents

1 Lexical Elements

This section describes the lexical elements that make up C source code after preprocessing. These elements are called tokens. There are five types of tokens: keywords, identifiers, constants, operators, and separators. White space, sometimes required to separate tokens, is also described in this section.

1.1 Identifiers

These are sequences of characters used for naming variables and functions. There are certain rules which we have followed while naming identifiers:

- Identifiers can include letters, decimal digits and underscore.
- First character of an identifier cannot be a digit.
- Identifiers are case sensitive.

1.2 Keywords

Keywords are special identifiers reserved for use as part of the programming language itself. We cannot use them for any other purpose.
Our compiler supports the following keywords:

break case char continue default do switch
float for if else int return short signed null
struct union unsigned void while bool true false

1.3 Constants

Constants are the literal alphanumeric characters which make up numbers/ characters or strings in the language. Usually, each constant is of a certain data type, but we have the liberty to typecast certain constants into another. This typecasting is supported for translating ASCII characters into their ASCII codes in the integer datatype. Here are the prominently used constants our compiler supports:

- Integer Constants
- Real Number Constants
- Character Constants
- String Constants

Some characters cannot be represented using only one character. Such characters are represented by escape sequences, which are enumerated below.

- \\- Backslash
- \? - Question mark
- \' - Single quotation mark
- \" - Double quotation mark
- \b - Backspace
- \n - Newline
- \t - Tab
- \0 - Null Character

1.4 Operators and Expressions

Expressions usually consist of one or more operands and operators to result in a meaningful mathematical statement which can be comprehended by the compiler. Operands are either identifiers or constants which are connected by operators.
Operators supported by our compiler are:

- Arithmetic Operators (+, -, *, /, %, - and + (unary))
- Shorthand Operators (+=, -=, *=, . . .)
- Increment and Decrement (a++, ++b, c - -, – - d)
- Comparison Operators (>, >=, <, <=,==, ! =)
- Logical Operators (&&, ||, !)
- Bitwise Operators (>>, <<, &, |, ˆ, ˜)
- Pointer Operators (*, &, ., ->)
- Comma operator (,)
- Ternary Operators (a?b:c)

In certain scenarios, an expression may comprise multiple operators in the same level of bracketing. To determine the order in which the expression is evaluated, we define some operator precedence rules. Stated below is the precedence of operators.
Incase where certain operators have same precedence, they are clubbed in the same precedence levels :

- Function calls, array subscripting
- Unary operators, including logical negation, bitwise complement, increment, decrement, unary positive, unary negative, indirection operator, address operator, and sizeof expressions. Incase of consecutive unary operators, the later ones are nested within the earlier ones: !-x means !(-x).
- Multiplication, division, and modular division expressions.
- Addition and subtraction expressions.
- Bitwise shifting expressions.
- Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to expressions.
- Equal-to and not-equal-to expressions.
- Bitwise AND expressions.
- Bitwise exclusive OR expressions.
- Bitwise inclusive OR expressions.
- Logical AND expressions.
- Logical OR expressions.
- Conditional expressions (using ?:). When used as subexpressions, these are evaluated right to left.
- All assignment expressions, including compound assignment. When multiple assignment statements appear as subexpressions in a single larger expression, they are evaluated right to left.
- Comma operator expressions.

1.5 Separators and Whitespaces

Separators and Whitespaces are the characters which separate the tokens. Separators themselves are tokens while whitespaces aren't. Following are the list of supported separators and whitespaces :

- Separators are
  - ⇒ (
  - ⇒)
  - ⇒ [
  - ⇒]
  - ⇒ {
  - ⇒ }
  - ⇒ ;
  - ⇒ ,
  - ⇒ .
  - ⇒ :
- White Spaces are
  - ⇒ ' '
  - ⇒ '\t'
  - ⇒ '\n'

1.6 Comments

Comments serve as a sort of in-code documentation. When inserted into a program, they are effectively ignored by the compiler; they are solely intended to be used as notes by the humans that read source code.
Comments can be of two primary types:

- Single Line Comment: //
- Multi Line Comment: /* */

For example:

```
//Single Line comment


/*Multi
Line
Comment*/
```

2 Primitive Data Types

2.1 Integer Type

The integer data types range in size from at least 1 byte to at least 4 bytes. One should use integer types for storing whole number values (and the char data type for storing characters).

- **bool**-The 8-bit bool data type can hold values of true (evaluates to 1 in terms of int) and false (evaluates to 0 in terms of int).
- **signed char**-The 8-bit signed char data type can hold integer values in the range of -128 to 127.
- **unsigned char**-The 8-bit unsigned char data type can hold integer values in the range of 0 to 255.
- **short int**-The 16-bit short int data type can hold integer values in the range of -32,768 to 32,767.
- **unsigned short int**-The 16-bit unsigned short int data type can hold integer values in the range of 0 to 65,535.
- **int**-The 32-bit int data type can hold integer values in the range of -2,147,483,648 to 2,147,483,647.
- **unsigned int**-The 32-bit unsigned int data type can hold integer values in the range of 0 to 4,294,967,295.

Declaration and Definition:

```
int a;
bool d;
unsigned int b=5;
char c='a';
```

2.2 Real types

- **float** - Its minimum value is stored in the FLT_MIN, and should be no greater than 1E-37. Its maximum value is stored in FLT_MAX, and should be no less than 1E37.All floating point data types are signed; trying to use an unsigned float, for example, will cause a compile-time error.

Declaration and Definition:

```
float p;
float q=41.33;
```

A floating number is internally expressed as the product of two parts: the mantissa and an exponent. It can also be expressed in scientific notation.
For example:

```
float f=5.3e-2;
```

Here, f takes the value $5.3 * 10^{-2}$

3 Conditionals

3.1 If-else statement

The if statement is used to conditionally execute the part of the program, based on if the expression evaluates to True. The general form of the if statement can be given by :

```
if(test)
    if-statement
else if(test2)
    else-if-statement
else
    else-statement
```

If the test evaluates true, then if-statement is executed. If test evaluates to false and test2 evaluates to true, then else-if statement is executed. If test evaluates to false and test2 evaluates to false, then else-statement is executed and then-statement is not. The else clause is optional.

3.2 Switch Case

The switch statement is used to compare one particular expression with many expressions given sequentially. According to which expression matches, we can then carry on a series of statements . A general form of statements is:

```
switch(test)
{
    case compare-statement-1:
        execution-statement-1;
        break;
    case compare-statement-2:
        execution-statement-2;
        break;
    .....
    default:
        default-statement;
}
```

The switch statement compares the test statement sequentially with all the compares-statements. On reaching a successful compare-statement, that is the one which evaluates to true, the execution statement with respect to that compare statement is executed and all the statements for the following case statements are executed. Once all the compare-statements before the default statement are checked, and none evaluate to true, then the default statement is executed. The default case is optional. All of the expressions compared must be of an integer type, and the compare-N expressions must be of a constant integer type.

4 Loops

Loops are used to execute a set of statements repeatedly, depending on a test condition. The break and continue keywords cause the control to exit out of the loop, skip to the next iteration respectively.

4.1 For Loop

The for statement is a loop statement whose structure allows easy variable initialization, expression testing, and variable modification. It is very convenient for making counter-controlled loops. Here is the general form of the for statement:

```
for(initialise;test;step)
{
    statements
}
```

The for statement first evaluates the expression initialise. Then it evaluates the expression test. If the test is false, then the loop ends and program control resumes after the statement. Otherwise, if the test is true, then the statement is executed. Finally, the step is evaluated, and the next iteration of the loop begins with evaluating the test again.
All three of the expressions in a for statement are optional, and any combination of the three is valid.
Multiple variables can be used in initialise and step expressions by separating using comma, the test expression can monitor multiple variables using logical operators.

4.2 While Loop

The while statement is a loop statement with an exit test at the beginning of the loop. Here is the general form of the while statement:

```
while(test)
{
    statements
}
```

The while statement first evaluates the test. If the test evaluates to true, the statement is executed, and then the test is evaluated again. statement continues to execute repeatedly as long as the test is true after each execution of statement.

4.3 Do-While Loop

The do statement is a loop statement with an exit test at the end of the loop. Here is the general form of the do statement:

```
do{
    statements
}while(test);
```

The do statement first executes the statement. After that, it evaluates the test. If the test is true, then the statement is executed again. statement continues to execute repeatedly as long as the test is true after each execution of the statement.

5 Statements

5.1 Break

One can use the break statement to terminate a while, do, for, or switch statement. For example:

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

The above example prints numbers from 0 to 1.When i is incremented to 2, i == 2 is true, so the break statement is executed, terminating the for loop prematurely.
If we put a break statement inside of a loop or switch statement which itself is inside of a loop or switch statement, the break only terminates the innermost loop or switch statement.

```
int x,y;
for(x=1;x<=10;x++){
    for(y=1;y<=10;y++){
        if(y==5) break;
    }
}
```

Here, break will only terminate the inner for loop.

5.2 Continue

One can use the continue statement in loops to terminate an iteration of the loop and begin the next iteration.
For example:

```
int x;
int sum_of_odd_numbers=0;
for(x=0;x<100;x++){
    if(x%2==0) continue;
    else sum_of_odd_numbers+=x;
}
```

The above example will force the next iteration of the loop, in case of even numbers
If we put a continue statement inside a loop which itself is inside a loop, then it affects only the innermost loop.

5.3 Return

We can use the return statement to end the execution of a function and return program control to the function that called it. Here is the general form of the return statement:

return return-value;

```
int func(){
    int x=5;
    return x;
}
```

6 Arrays

6.1 Declaring Arrays

You declare an array after specifying the data type for its elements, its name, and the number of elements it can store. Following is an example to declare an array of 10 elements :

```
int arr[10];
```

While declaring the array only a constant can be used to define the space taken up by the array. We cannot use expressions or statements in there.

6.2 Initialising Arrays

Initialising the elements in an array can be done by listing the initialising array elements. For example:

```
int arr[3]={1,2,3};
```

There is no need to explicitly initialise all the array elements, you can only specify some of them. In any case, you always have to specify the number of elements in the array while declaring or initialising.

6.3 Accessing Array Elements

We can access the elements of an array by specifying the array name, followed by the element index enclosed in square brackets. The array elements start with 0. For example:

```
arr[3]=5;
```

This assigns the value 5 to the third element in the array.

6.4 Multi-Dimensional Arrays

We can make multidimensional arrays. This is done by adding an extra set of brackets and array lengths for every additional dimension that we want our multidimensional array to have. We would want all the array lengths for each dimension to be specified while declaring and while initialising.
For example:

```
int my_arr[2][3]={{1,2,3},{4,5,6}};
```

For accessing the element at a particular position we have to choose the desired index at both positions.

```
my_arr[2][2]=4;
```

7 Input/Output

Input is accepted from stdin using the function scanf, while output is printed to stdout using printf. Following is the declaration for scanf() function:

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

format : This is the C string that contains one or more of the following items:
Whitespace character, Non-whitespace character and Format specifiers. A format specifier will be like

%[width]specifier

where specifiers are : **%d** for int and bool, **%c** for char, **%f** for float. Width specifies the maximum number of characters to be read in the current reading operation.

additional arguments : Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter, if any. There should be the same number of these arguments as the number of %-tags that expect a value.
Following is the declaration for printf() function :

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

format : This is the string that contains the text to be written to stdout. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format tags prototype is

%[width][.precision]specifier

where specifiers are : **%d** for int and bool, **%c** for char, **%f** for float. Width specifies the minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. Precision is only specified for floats, it denotes the number of digits after the decimal to be printed.

additional arguments : Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter (if any). There should be the same number of these arguments as the number of %-tags that expect a value.

Return Value If successful, the total number of characters read/written is returned. On failure, a negative number is returned.

8 Functions

One can write functions to separate parts of your program into distinct subprocedures.Every program requires at least one function, called “main”. That is where the program’s execution begins.

8.1 Function Definition

One writes a function definition to specify what a function actually does. A function definition consists of information regarding the function’s name, return type, and types and names of parameters, along with the body of the function. The function body is a series of statements enclosed in braces.
A general form of function definition is:

return-type function-name (parameter-list)
{ function-body}

Example:

```
int dummy(int x){
    return x;
}
```

8.2 Calling Functions

One can call a function by using its name and supplying any needed parameters.
Here is the general form of a function call:

function-name(parameters)

```
dummy(21);
```

Here, the function 'dummy' is being called with parameter 21. A function call can also be used as a subexpression.

```
int a=dummy(21);
```

Here, the 'dummy' function is called, and the variable a gets the value 21.

8.3 Main Function

'Main' function is compulsory. This is where the program begins executing. We do not need to write a declaration or prototype for main, but we do need to define it.
The return type for main is always int. We do not have to specify the return type for main. However, we cannot specify that it has a return type other than int.
In general, the return value from main indicates the program's exit status. A value of zero or EXIT _ SUCCESS indicates success and EXIT _ FAILURE indicates an error.

8.4 Recursive Functions

Recursive function is a function which calls itself.
For example:

```
int factorial(int x){
    if(x<1) return 1;
    else return (x*factorial(x-1));
}
```

9 Pointers

Pointers are required to store the memory addresses of constants or variables. We can create a pointer to a memory address of an instance of a data type, for any data type, including both primitive and custom types.

9.1 Declaring Pointers

We declare a pointer by specifying a name for it and a data type. The data type tells us what is the data type to which the variable pointed to by the pointer belongs to.
Format:

data-type* name;

White space is not significant around the indirection operator:
Example:

```
int* p;
```

9.2 Initialising Pointers

We can initialise the pointer when you first declare it by specifying a variable address to store in it.
For example:

```
int* p=&n;
```

Now in order to change the memory address to which a pointer points to once it has been declared, we will have to use the * symbol or the dereferencing operator.
For example:

```
int x;
int *p=&n;(p stores address of n)
*p=&x;
```

9.3 Dereferencing Pointers

We can dereference a pointer, that is, get the value of the variable stored at the memory address pointer to by the pointer by using the * operator along with the name of the pointer.
For example:

```
int *p=&n;
int x=*p;
```

9.4 Multilevel Pointers

Pointer to a pointer is a multi-level pointer. So in this case, the pointer stores the memory address where a pointer is stored. On dereferencing this pointer, we get another pointer which in turn could point to another memory address.
For example -

```
int n=10;
int *p1=&n;
int **p2=&p1;
```

Here p1 stores the memory address of n, while p2 stores the memory address of p1. Multilevel pointers can be used to implement multi-dimensional arrays.

10 Library Functions

10.1 Strings

A string is a character array terminated by the null character '\0'. The following functions are supported:

- strlen - Returns length of the string, that is, the number of characters excluding the null character. The syntax is - int strlen (char *s).
- strcpy- Copy contents of one string to another. The syntax is - char* strcpy (char *dest, char *source). It returns a pointer to the destination string dest.
- strcat - Contacts or joins 2 strings in sequence storing the result in the first string. The syntax is - char* strcat (char *first, char *second). It returns a pointer to the first string.

- strcmp - Compares 2 strings based on their lexicographical ordering. The syntax is - int strcat (char *first, char *second). It returns 0 if they are equal, else the difference between the ASCII values of the first unmatched character in first and second strings in both the cases.
- strrev - Reverse the given string. The syntax is - char* strrev(char *str). This function returns the string after reversing the given string.
- strcat - Contacts or joins 2 strings in sequence storing the result in the first string. The syntax is - char* strcat (char *first, char *second). It returns a pointer to the first string.

10.2 Math

The compiler supports a variety of built-in math which have similar meaning in mathematical literature. The following functions are supported:

- float exp(float x) - Returns the exponential of x.
- float log(float x) - Returns the natural logarithm of x. Note, here x>0.
- float sqrt(float x) - Returns the square root of x.
- float pow(float base, float power) - Returns baseˆpower
- float abs(float x) - Returns absolute value of x, i.e. $|x|$
- int floor(float x) - Returns the rounded down value of x, i.e., integer value which is less than or equal to x
- int ceil(float x) - Returns the rounded up value of x, i.e., integer value which is greater than or equal to x

11 Dynamic Memory Allocation

The compiler allows dynamic memory allocation to data structures like arrays so that the size can be changed during the runtime. The functions provided to support these operations are :

- **malloc()** : It serves as the memory allocation method in C which can dynamically allocate one single large block of memory space as per the user request. The return type of the function is a void type pointer however this can be converted into a pointer of any other type as required. The memory is initialised with garbage values and needs to be filled before proper usage. Syntax :

```
void* ptr=malloc(byte_size);
```

- **free()** : This method is used by the compiler to free or to deallocate the memory consumed by any of the malloc statements after their use is over. This function call is essential as it reduces the wastage of memory since malloc doesn't deallocate memory on it's own. Thus these two functions are used in a pair for dynamic memory allocation. Syntax :

```
free(ptr);
```

12 Global Variables

Declaring variables as global is the simplest way of using them. Local variables are restricted as their scope is limited to the functions or parenthesis they are defined under, but global variables are freely available everywhere. They can be declared several times but defined only once. They must be declared (not necessarily defined) before their usage. They belong to the data section of the memory layout.
Usage:

```
int cnt;
int func(){
    cnt=0;
    cnt++;
    return cnt;
}
```

13 Structure

A structure is used to define a custom data type made up of variables of other data types (possibly including other structure types).

13.1 Defining Structures

We can define a struct using the struct keyword, and enclose the members of the struct within curly brackets. Within the curly brackets, each member is declared just as they would be normally - that is the data type followed by one or more variable names separated by commas and ending with a semicolon. We also need to end the struct with a semicolon in the end.
Example:

```
struct line
{
    int slope;
    int intercept;
};
```

13.2 Declaring structure variables

To declare a structure variable, we use the struct keyword followed by the name of the struct which is then followed by the name of the struct.
Example:

```
struct line;
```

13.3 Initialising structure and accessing members

There are 2 major ways to work with the members of struct :

- Using curly brackets:
  Example:

```
struct line
{
    int slope;
    int intercept;
};
struct line aline={3,7};
```

- Using dot notation:
  Example:

```
struct line aline;
aline.intercept=3;
aline.slope=7;
```

14 Union

A union is a custom data type used for storing several variables in the same memory space.

14.1 Defining Union

We define a union using the union keyword followed by the declarations of the union's members, enclosed in braces. We declare each member of a union just as we would normally declare a variable—using the data type followed by one or more variable names separated by commas, and ending with a semicolon. Then we should end the union definition with a semicolon after the closing brace.
We should also include a name for the union between the union keyword and the opening brace.
For example:

```
union numbers{
    int i;
    float f;
};
```

That defines a union named numbers, which contains two members, i and f, which are of type int and float, respectively.

14.2 Declaring union variables

There are 2 ways of declaring union variables.

- At definition:

```
union numbers{
    int i;
    float f;
} first_number,second_number;
```

- After definition:

```
union numbers{
    int i;
    float f;
}
union numbers first_number,second_number;
```

14.3 Accessing union members

We can access the members of a union variable using the member access operator. You put the name of the union variable on the left side of the operator, and the name of the member on the right side.

```
union numbers{
    int i;
    float f;
}
union numbers first_number;
first_number.i=5;
first_number.f=3.9;
```

15 Floating Point Operations

Floating-point types would support most of the same arithmetic and relational operators as integer types; x > y, x / y, x + y, as in all of these relations would make sense when x and y are floats. If you mix two different floating-point types together, the less-precise one will be extended to match the precision of the more-precise one; this also works if you mix integer and floating-point types as in 2 / 3.0.

As an example, when a floating point division occurs, then the fractional part is not discarded, which is the case in integer division. Numerically, 2.0/3 gives 0.66666666666666663 which is not quite exact. Whereas 2/3 gives 0. As this is integer division, which gets round off.

16 Short circuit expression evaluation

Short circuiting is the concept where the compiler skips the execution or evaluation of some sub-expressions in a logical expression.The compiler stops evaluating further sub-expressions as soon as the value of the expression is determined.

```
if(a==b||c==d||e==f){

}
```

Here,if a=b, then c==d and e==f are never evaluated because the expression's result has already been determined.
Similarly, in case of logical AND, if a is not equal to b, the compiler will skip evaluating the other expressions.

17 Fork

The fork function creates a new process by duplicating the calling process.
It's function signature is:

pid_t fork();

Here, pid_t is the same as the int data type, and the return value for the parent process is the pid of the child process while the return value for the child process is 0.
In case of any error, the return value is negative.

```
void forkexample()
{
    // child process because return value zero
    if (fork() == 0)
        printf("Hello from Child!\n");

    // parent process because return value non-zero.
    else
        printf("Hello from Parent!\n");
}
```

18 File Handling

18.1 Opening a file

The open function opens the file specified by pathname. If the specified file does not exist, it may optionally (if O_CREAT is specified in flags) be created by open(). It's function signature is:

int open(const char *pathname, int flags);

The argument flags must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR. In addition, the file creation flag O_CREAT, or the append flag O_APPEND can be bitwise-or'd in flags.

This function returns a non negative file descriptor that is an index to an entry in the process's table of open file descriptors. In case of an error, the return value is negative.

18.2 Closing a file

The close function closes an existing file descriptor. It returns zero on success, -1 on error. It's function signature is:

int close(int fd);

18.3 Writing to a file

The write function writes specified number of bytes from the buffer to the file. It's function signature is:

int write(int fd, const void *buf, unsigned int count);

On success, the number of bytes written is returned. On error, -1 is returned.

18.4 Reading from a file

The read function attempts to read specified number of bytes from an already opened file. On success, the number of bytes read is returned, which can as well be less than the specified number in case of reaching EOF. On error, -1 is returned. It's function signature is:

int read(int fd, void *buf, unsigned int count);

19 References

- Linux Programmer's Manual
- GNU C Reference Manual
- Geeks for Geeks
- Tutorials Point