CS335A
Compiler Design
Indian Institute of Technology, Kanpur

Group Members:

1. Himanshu PS (190379)
2. Kindinti Uday Kiran (190434)
3. Viraj Ravjibhai Limbasiya (180870)
4. Anita Bugaliya (180093)

SIT Languages:

Source Language - C++
Implementation Language - C++
Target Language - MIPS

GitHub Link:

GitHub Group-27 -

Contents

1. ELEMENTS **4**

1.1 Tokens and Character Sets . . . 5
1.2 Comments . . . 5
1.3 Identifiers . . . 5
1.4 Keywords . . . 6
1.5 Punctuators . . . 6
1.6 Numeric, boolean, and pointer literals . . . 6
1.7 String and character literals . . . 6
1.8 User-defined literals . . . 6

2. DATA TYPES **7**

2.1 Native Data types: . . . 7
2.1.1 Integer . . . 7
2.1.2 Real numbers . . . 7
2.1.3 Boolean . . . 7
2.2 Unions . . . 7
2.2.1 Defining Unions . . . 7
2.2.2 Declaring Union Variables . . . 8
2.2.3 Initializing Union Members . . . 8
Accessing Union Members . . . 8
Size of Unions . . . 9
2.3 Structures . . . 9
2.3.1 Defining Structures . . . 9
2.3.2 Initializing Structure Members . . . 10
2.3.3 Accessing Structure Members . . . 11
2.3.4 Size of Structures . . . 11
2.4 Arrays . . . 11
2.4.1 Declaring Arrays . . . 11
2.4.2 Initializing Arrays . . . 11
2.4.3 Accessing Array elements : . . . 12
2.4.4 Multi-dimensional array : . . . 12
2.4.5 Arrays as strings : . . . 12
2.4.6 Arrays of structs or unions: . . . 13
2.5 Pointers . . . 13
2.5.1 Declaring Pointers . . . 13
2.5.2 Initializing Pointers . . . 14
2.5.3 Pointers to Unions . . . 14
2.5.4 Pointers to Structures . . . 14

3. EXPRESSIONS and OPERATORS **15**

Arithmetic Operators: . . . 15
Comparison operators: . . . 15
Logical operators: . . . 15
Bitwise operators: . . . 15
Assignment and compound assignment operators: . . . 15

4. STATEMENTS **16**
LABEL . . . 16
IF . . . 16
for . . . 16
While . . . 17
DO . . . 17
BREAK . . . 17
continue . . . 17
return . . . 17
typedef . . . 17
input . . . 18
output . . . 18

5. FUNCTIONS **19**
Declaration: . . . 19
Defining: . . . 19
Calling functions: . . . 19
Parameters: . . . 19
Function pointers: . . . 19
Recursive functions: . . . 20
The main function: . . . 20

6. PROGRAM STRUCTURE and SCOPE **21**
Program Structure . . . 21
Scope . . . 21

Resources **22**

1. ELEMENTS

This section introduces the fundamental elements of a C++ program. You use these elements, called " elements" or "tokens" to construct statements, definitions, declarations, and so on, which are used to construct complete programs.

The following elements are discussed in this section:

- Tokens and character sets
- Comments
- Identifiers
- Keywords
- Punctuators
- Numeric, boolean, and pointer literals
- String and character literals
- User-defined literals

1.1 Tokens and Character Sets

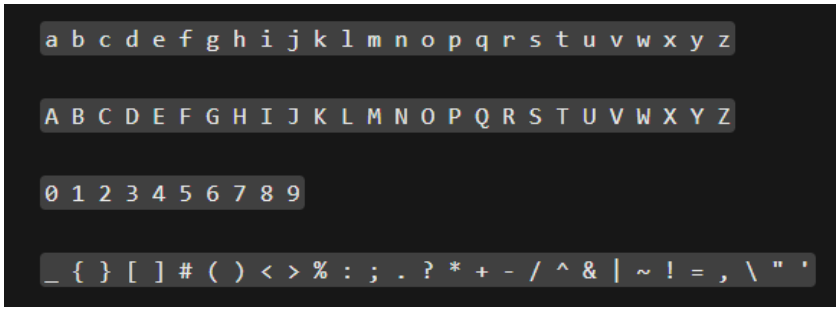
The text of a C++ program consists of tokens and white space. A token is the smallest element of a program that is meaningful to the compiler.
The implemented parser will recognize the following tokens:

- Keywords
- Identifiers
- Numeric, Boolean and Pointer Literals
- String and Character Literals
- User-Defined Literals
- Operators
- Punctuators

Tokens are usually separated by white space, which can be one or more:

- Blanks
- Horizontal tab
- New lines
- Comments

The basic character set allowed in this language is the following:
The space character (' '), the horizontal tab ('\t') and new-line ('\n') control characters, along with



1.2 Comments

A comment is ignored by the compiler, but is very helpful for the programmer, both as authors and also viewers. The compiler treated them as white-spaces. Comments are written using either '\\', everything in the line is ignored, or '* *\, in which everything in between the start and the end symbols are ignored.
One major thing to keep in mind is, we cannot nest comments using the second format.

1.3 Identifiers

- Identifiers are a sequence of characters used for naming different kinds of stuff in the program.
- They include, names of objects, variabes, classes, structs, unions, enumerated type and typedef among others.
- Functions are also identified by these identifiers.

- The permitted character set for the naming of the identifiers are all alphabets in the English language, both lower and upper cases and digits.
- An additional character, underscore (_) is also permitted.
- The only rule of the identifier, is they can not start with a digit or an underscore.

1.4 Keywords

Keywords are predefined reserved identifiers that have special meanings. They can't be used as identifiers in our program.

1.5 Punctuators

Punctuators have syntactic and semantic meaning to the compiler but do not, of themselves, specify an operation that yields a value. Some punctuators, either alone or in combination.
Any of the following characters are considered punctuators:

1.6 Numeric, boolean, and pointer literals

A literal is a program element that directly represents a value. This article covers literals of type integer, floating-point, boolean, and pointer. For information about string and character literals

1.7 String and character literals

string and character types, and provides ways to express literal values of each of these types. In your source code, you express the content of your character and string literals using a character set

1.8 User-defined literals

There are six major categories of literals: integer, character, floating-point, string, boolean, and pointer.

2. DATA TYPES

2.1 Native Data types:

2.1.1 Integer

- Sequence of digits with optional prefix that denote integer base.
- Size of int ranges from 8 bits to 64 bits.

number of bits	type name
8	char
8	unsigned char
16	short int
16	unsigned short int
32	int
32	unsigned int
32	long int
32	unsigned long int
64	long long int
64	unsigned long long int

2.1.2 Real numbers

- used for representing fractional data types, numbers with decimals.
- These are approximations, since the computers have limited memory as to the number of digits after the decimal place.

type name	minimum	maximum
float	FLT_MIN	FLT_MAX
double	DBL_MIN	DBL_MAX
long double	LDBL_MIN	LDBL_MAX

2.1.3 Boolean

- Takes values true/false.
- True is for non-zero value or the condition is true.
- And false represents zero value or the condition is false.

2.2 Unions

A union is a custom data type used for storing several variables in the same memory space. Although you can access any of those variables at any time, you should only read from one of them at a time—assigning a value to one of them overwrites the values in the others.

2.2.1 Defining Unions

We define a union using the union keyword followed by the name of the union and the declarations of the union's members, enclosed in braces.
Each member is declared in the same manner a variable is declared.

Then end the union definition with a semicolon after the closing brace.
Eg: union numbers
int i;
float f;
;

2.2.2 Declaring Union Variables

You can declare variables of a union type when both you initially define the union and after the definition, provided you gave the union type a name. union numbers
int i;
float f;
first$_numb$$er, second_numb$$er$;
Put the variable names after the closing brace of the union definition, but before the final semicolon. You can declare more than one such variable by separating the names with commas.
// union numbers int i;
float f;
;
union numbers first$_numb$$er, second_numb$$er$;
Put the union keyword and the name you gave the union type, followed by one or more variable names separated by commas.

2.2.3 Initializing Union Members

You can initialize the first member of a union variable when you declare it:
union numbers
{
int i;
float f;
};
union numbers first$_numb$$er = \{5\}$;
In that example, the i member of first$_numb$$ergetsthevalue5.Thefmemberisleftalone.$
Another way to initialize a union member is to specify the name of the member to initialize. This way, you can initialize whichever member you want to, not just the first one. There are two methods that you can use—either follow the member name with a colon, and then its value, like this:
union number wp = { w: 2.33 }; or precede the member name with a period and assign a value with the assignment operator, like this:
union numberp wp = { .w = 2.33 }; You can also initialize a union member when you declare the union variable during the definition:
union numbers { int i; float f; } first$_numb$$er = \{5\}; Next : SizeofUnions, Previous :$
$DeclaringUnionVariables, Up : Unions[Contents][Index]$

Accessing Union Members

You 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$_number$; $first_number.i = 5$; $first_number.f = 3.9$;

Size of Unions

This size of a union is equal to the size of its largest member. Consider the first union example from this section:
union numbers { int i; float f; }; The size of the union data type is the same as sizeof (float), the union data type size just needs to be large enough to hold the largest member.

2.3 Structures

A structure is a programmer-defined data type made up of variables of other data types (possibly including other structure types).

2.3.1 Defining Structures

You define a structure using the struct keyword followed by the declarations of the structure's members, enclosed in braces. You declare each member of a structure just as you 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 end the structure definition with a semicolon after the closing brace.
You should also include a name for the structure in between the struct keyword and the opening brace. This is optional, but if you leave it out, you can't refer to that structure data type later on (without a typedef, see The typedef Statement).
Eg:
struct point
{ int x, y;
};
That defines a structure type named struct point, which contains two members, x and y, both of which are of type int.
Structures (and unions) may contain instances of other structures and unions, but of course not themselves. It is possible for a structure or union type to contain a field which is a pointer to the same type (see Incomplete Types).
Next: Accessing Structure Members, Previous: Defining Structures, Up: Structures
[Contents][Index]
2.4.2 Declaring Structure Variables You can declare variables of a structure type when both you initially define the structure and after the definition, provided you gave the structure type a name. define the structure type by putting the variable names after the closing brace of the structure definition, but before the final semicolon. You can declare more than one such variable by separating the names with commas.
struct point
{
int x, y;
} first$_point$, $second_point$;
You can declare variables of a structure type after defining the structure by using the struct keyword and the name you gave the structure type, followed by one or more variable names separated by commas.

struct point { int x, y; }; struct point
first$_point, second_point; That example declares two variables of type struct point, first_point and second_point.$

2.3.2 Initializing Structure Members

You can initialize the members of a structure type to have certain values when you declare structure variables.
If you do not initialize a structure variable, the effect depends on whether it has static storage (see Storage Class Specifiers) or not.
If it is, members with integral types are initialized with 0 and pointer members are initialized to NULL; otherwise, the value of the structure's members is indeterminate.
One way to initialize a structure is to specify the values in a set of braces and separated by commas.
Those values are assigned to the structure members in the same order that the members are declared in the structure in definition.

struct point
{
int x, y;
};
struct point first$_point = \{5, 10\};$
In that example, the x member of first$_point gets the value 5, and the y member gets the value 10.$
Another way to initialize the members is to specify the name of the member to initialize. This way, you can initialize the members in any order you like, and even leave some of them uninitialized.
There are two methods that you can use. The first method is available in C99 and as a C89 extension in GCC:
struct point first$_point = \{.y = 10, .x = 5\}; You can also omit the period and use a colon instead of `=' , though this is a GNU C extension :$
struct point first$_point =$
$y : 10, x : 5; You can also initialize the structure variable's members when you declare the variable during the structure de fi nit$
struct point {
int x, y;
} first$_point = 5, 10;$
You can also initialize fewer than all of a structure variable's members:

struct pointy
{
int x, y;
char *p;
};
struct pointy first$_pointy = \{5\};$
Here, x is initialized with 5, y is initialized with 0, and p is initialized with NULL. The rule here is that y and p are initialized just as they would be if they were static variables.
Here is another example that initializes a structure's members which are structure variables themselves:

struct point
{
int x, y;

};
struct rectangle
{
struct point top$_left, bottom_right$;
};
struct rectangle my$_rectangle =$
$\{\{0,5\},\{10,0\}\}; That example defines the rectangle structure to consist of two point structure variables.$
Then it declares one variable of type struct rectangle and initializes its members.
Since its members are structure variables, we used an extra set of braces surrounding the members that belong to the point structure variables.
However, those extra braces are not necessary; they just make the code easier to read.

2.3.3 Accessing Structure Members

Accessing the members of a struct variable is exactly the same way of accessing the members of a union.
We can also access the members of a structure variable which is itself a member of a structure variable.
struct hostel;
{
struct wing A1, A2;
};
struct hostel da$_hostel$;

da$_hostel.A1.x = 0$;

2.3.4 Size of Structures

The size of a structure type is equal to the sum of the size of all of its members and might include padding to cause the structure type to align to a certain byte boundary.

2.4 Arrays

2.4.1 Declaring Arrays

- An array is declared by specifying the data type for its elements, name of the array, and the number of elements it can store.
- Basic syntax:
  type name[n]; // here 'type' is the data type and 'n' is the number of the elements to
  // be stored, and 'name' is the name of the array.
- type name [ ]= {1,4,6,7}; // this declares an array of size 4 and also initializes its
  // elements as 1,4,6,7 respectively.

2.4.2 Initializing Arrays

- An array can be initialized by initializing all its elements or just some of the elements.
- Following are the allowed ways of initializing an array.
- 1. int arr[7] = $\{1,2,3,4,5,6,7\}$;

2. int arr[7] = $\{1, 4, 6, 7\}$;

- In the second method mentioned above, the first four elements of the array are initialized to $1, 4, 6, 7$ respectively.

2.4.3 Accessing Array elements :

- The element at a particular position of an array can be initialized/accessed as follows.
- arr[4] $= 5$;
- The index numbers are non-negative integers, till $n - 1$ where n is the size of the array.
- 0^{th} index refers to the leftmost element, and $n - 1$ is for the rightmost one in the array.

2.4.4 Multi-dimensional array :

- A multi-dimensional array can be declared and used by just adding the required number of brackets in the declaration as follows.
- type two_d_arr [4][6];
- this is a $2D$ array consisting of 4 rows and 6 columns totalling to 24 elements.
- Can be accessed similar to simple arrays.
- We can think these as arrays of arrays and so on.

2.4.5 Arrays as strings :

- An array of characters can be used to hold a string.
- The array may be built of either signed or unsigned characters.
- When you declare the array, you can specify the number of elements it will have.
- This number will be the maximum number of characters that should be in the string, including the null character used to end the string.
- If you choose this option, then you do not have to initialize the array when you declare it.
- Alternately, you can simply initialize the array to a value, and its size will then be exactly large enough to hold whatever string you used to initialize it.
- There are two different ways to initialize the array.
- You can specify of comma-delimited list of characters enclosed in braces, or you can specify a string literal enclosed in double quotation marks.
- Here are some examples:
  - char pant[26];
  - char virat[26] = 'l', 'e', 'g', 'e', 'n', 'd', '';
  - char dev[ ] = 'd', 'e', 'v', '';
  - char rahul[26] = "player";
  - char rohit[ ] = "captain_player";
- In each of these cases, the null character is included at the end of the string, even when not explicitly stated.
- If in the case where the number of elements is mentioned, but the string is larger than the specified number of characters, the extra part of it written to a memory not allocated to it.
- After initialization, you cannot assign a new string literal to an array using the assignment operator.There are functions (including copy) on string arrays.
- You can also change one character at a time, by accessing individual string elements as you would any other array:

2.4.6 Arrays of structs or unions:

- Declaration and initialization of arrays of structs and unions is done as follows.
- Consider the following as the definition for the struct 'grade'.

```
struct grade {
    int cno;
    int gd;
};
```

- struct grade arr_grade[n] ;
- The above statement declares an array arr_grade of struct grade type and the array can hold upto 'n' structs grade.
- struct grade arr_grade[5];
  struct grade arr_grade[5] = { {201, 7}, {202, 9}, {203, 8};
  struct grade arr_grade[5] = { {201}, {202, 9}, {203, 8} {242}, {220, 8} };
- As seen above, e can partially initialize some of the structures in the array, and fully initialize others
- Accessing the elements of the structs in the array is done by using the very method used to access elements of a normal struct, but the difference is that , here the name of the struct is replaced by the name of the array and the index in square brackets.
- arr_grade[0].cno = 201;
- arr_grade[2].gd = 8;
- The same is the case with array of unions.

2.5 Pointers

Pointers hold memory addresses of stored constants or variables. We will deal with the following main subsections:

- Declaring Pointers:
- Initializing Pointers:
- Pointers to Unions:
- Pointers to Structures:

2.5.1 Declaring Pointers

- You declare a pointer by specifying a name for it, a data type of variable the pointer will hold memory addresses and include the indirection operator before the identifier.
- Basic syntax: data-type * name;
- White space is not significant around the indirection operator.
- Ex: int *ip;
- When declaring multiple pointers in the same statement, you must explicitly declare each as a pointer, using the indirection operator.
- int *foo, *bar; $\rightarrow$ two pointers
- int *baz, quux; $\rightarrow$ A pointer and an integer variable.

2.5.2 Initializing Pointers

- You can initialize a pointer by first declaring a variable whose address can be stored in the pointer.
- E.g:-

```
int n;
int *ptr = &n; // pointer is initialized with the address of 'n'.
```

- Address operator (&) is used to get the memory address of a variable.
- After a pointer is declared, you do not use the indirection operator with the pointer's name when assigning it a new address to point to.
- The value stored in a pointer is an integral number, since declared using int, which is in a location within the computer's memory space.
- Pointer values can be explicitly assigned using literal integers, casting them to the appropriate pointer type.

2.5.3 Pointers to Unions

-
```
union grade {
    int c;
    float g;
};
union grad abc = {240};
union grad *grad_ptr = &abc;
```

- In this example we create a new union type, union numbers, and declares (and initializes the first member of) a variable of that type named grad.
- Finally, it declares a pointer to the type union numbers, and gives it the address of grad_ptr.
- You can access the members of a union variable through a pointer, but you have to use the indirect member access operator '.'
- The following example will change the value of the first member of abc:

2.5.4 Pointers to Structures

  -
```
struct ball {
    float radius, weight;
};
struct ball cric = {4.3, 5.8};
struct ball *cric_ptr = &cric;
```

  - In this example we create a new structure type, struct ball, and declares (and initializes) a variable of that type, named cric.
  - Then we declare a pointer to the type struct ball, and gives it the address of cric.
  - You can access the members of a structure variable through a pointer, but you have to use the indirect member access operator '.'
  - The following example will change the values of the members of cric:

```
cric_ptr-¿radius = 5.1;
cric_ptr-¿weight = 6.2;
```

  - Now the radius and weight members in cric are 5.1 and 6.2, respectively.

3. EXPRESSIONS and OPERATORS

Arithmetic Operators:

- Operators for arithmetic operations
- Include addition(+), subtraction(-), multiplication(*), division(/), remainder, (%)
- Syntax: a *op* b , where *op* is an arithmetic operator and a,b are operands.

Comparison operators:

- To compare operands.
- Gives true if the condition is true.
- Includes operators (==(equal), !=(not equal), ¿=(greater than or equal), ¡=(less than or equal), ¿(greater than), ¡(less than))
- Syntax: a *op* b , where *op* is a comparison operator and a,b are operands.

Logical operators:

- They test the truth values of operands.
- Includes operators (&& (and), —— (or), ! (not))
- Syntax : c1 *op* c2 , where *op* is a logical operator and c1, c2 are comparison statements.
- a&&b - returns true if both conditions are true
- a——b - returns true if one of the condition is true
- !c - returns true if c is false

Bitwise operators:

- Includes &(bitwise and), —(bitwise or), ∧(bitwise xor), ˜(bitwise not)
- Syntax: a *op* b , where *op* is a bitwise operator and a,b are operands. & - if corresponding bits are both 1 then the resulting bit is 1 else 0
- — - if one of the corresponding bits is 1 then the resulting bit is 1 else 0
- ∧ - if both corresponding bits are not same then the resulting bit is 1
- ˜- 1 becomes 0 and 0 becomes 1

Assignment and compound assignment operators:

- = is the assignment operator.
- Used to assign a value to a variable. Eg: int a=2;
- Compound assignment operators perform operations involving both left and right operands and return value to the left variable. Eg: int a += 2 ; (i.e a = a+2)
- Compound assignment operators are +=, -=, *=, /=, %=, &=, —=,∧=,˜=.
-

4. STATEMENTS

You write statements to cause actions and to control flow within your programs. You can also write statements that do not do anything at all, or do things that are uselessly trivial.

LABEL

- A labeled statement labels a statement for control flow purposes.
- Can be used in multiple situations:
  * identifier: statement → target for goto
  * **case** constexpr: statement → case label in switch statement
  * **default** : statement → default label in switch statement

IF

- Used to conditionally execute a part of the code, based on the truth value of the given expression.
- Else statement is the optional statement, to be executed when the statement turns out to be false.
- Independent else statements can't exist.
- If statements without else make send, but not the other way around.
- If the body of either of If or Else statement is a single line, the curly braces may be omitted.
- Eg1:-

```
if(x == 5) {
    cout ≪ "x is 5";
}
```

- Eg2:-

```
if(x%2 == 0) {
    cout ≪ "x is even";
}
else {
    cout ≪ "x is odd";
}
```

for

- This loop statement allow variable initialization, validation expression truth value and variable update operation easily.
- Basic syntax is of the form:

```
for(initialize; test; step) { body }
```

- Similar to If and Else, single line body can omit the curly braces
- Eg:-

```
for(int i=0;i¡10) {
    cout ≪ x ≪ " ";
}
```

While

- In this loop statement expression truth value is checked in the beginning of the loop, inside parenthesis of while();.
- Basic syntax is of the form:
  while(condition) { body }
- Eg:-

```
int x = 0;
while(x < 5){ // the loop runs as long as the test condition is true.
    x = x+1;
}
cout << x << " "; // will output 10
```

DO

- The do statement is a loop statement with an exit test at the end of the loop.
-
- Basic syntax is of the form:
  do { body } while(condition);

BREAK

- It is used to terminate a while, do, for, or switch statement
- Eg:-

```
int x = 0
for(int i=0;i¡10) {
    if (x == 6) break;
    x+ = 1;
}
cout << x; // output is 6
```

continue

- it is used to skip an iteration and begin with next iteration.
- Everything written after the continue statement, is skipped for the current iteration, and they are executed without any change for other iterations.

return

- It is used to end function execution and return control to function it is called.
- if the function's return type is void, no return statement is included in the body of the function.
- Basic syntax: return return_value.

typedef

- It is used to create new names for data types.
- Especially helpful for aliasing long user defined data types to short ones.

- Basic syntax : typedef name1 name2.
- Old variable type name1 can also be used as the new variable type, name2.

input

scanf reads the input from user. Eg:- scanf(“input string is read into the address str.

output

printf() prints the output. Eg:- printf(“

5. FUNCTIONS

Declaration:

- Function declaration is used to specify the name of a function, a list of parameters, and the function's return type, followed by semicolon.
- The function name can be any valid identifier.
- Parameter-list can be of zero or more.

Defining:

- Function definition includes function name, types and names of parameter and the return type(a void type may not return any thing).
- Eg:

```
int multiply(int x, int y){          // function's return type followed by name
    int z = x · y;                   // body of the function.
    return z;                        // return value
}
```

- multiply → the function's name
- int x, int y → parameter list
- int → return tyep

Calling functions:

- Function is called by its name along with parameters followed by a semicolon in the end.
- Parameters can be any literal or a value itself.
- Eg: → **foo(**15**);**

Parameters:

- Takes literal, value stored or complex expressions as input.

Function pointers:

- Calling function by a pointer.
- Eg:-

```
int add( int x, int y){
    return x+y;
}
int main() {
    int (*funcptr)(int,int);
    funcptr = add;
    int sum=funcptr(5,5);
   return 0;
}
```

Recursive functions:

- A recursive function is a function that calls itself during its execution.
- The process may repeat several times, outputting the result and the end of each iteration.
- One important point to note, is to include the base case, so as to avoid the infinite loop during recursion.
- Eg:-

```
int factorial(int x) {
    if(x == 0||x == 1) return 1;
    return factorial(x − 1)*x;
}
```

The main function:

- Every program shall contain a global function named main.
- It is the designated start of the program in hosted environment.
- Return type should be int. 0 is for success, 1 is for error.
- It shall have one of the following forms:
  * int main() { body }
  * int main(int argc, char* argv[]) { body }
- Eg:-

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

6. PROGRAM STRUCTURE and SCOPE

Program Structure

Trivial C++ programs can be written in a single file, but non-trivial programs are usually consisting of custom header files(with extension *.h*, source files and existing libraries.
By convention, header files (with a ".h" extension) contain variable and function declarations, and source files (with a ".c" extension) contain the corresponding definitions. Source files may also store declarations, if these declarations are not for objects which need to be seen by other files. However, header files almost certainly should not contain any definitions.
If we want a function to be accessible in files other than where it is defined, we need to define the function in a header file and include it in every source file where it might be used.

Scope

The part(s) of the program where the variable is recognisable is called the scope of the variable.
A declared object can be visible only within a particular function, or within a particular file, or may be visible to an entire set of files by way of including header files and using extern declarations.
Unless explicitly stated otherwise, declarations made at the top-level of a file (i.e., not within a function) are visible to the entire file, including from within functions, but are not visible outside of the file.
Declarations made within functions are visible only within those functions.

Resources

https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html https://docs.microsoft.com/en-us/cpp/cpp/user-defined-literals-cpp?view=msvc-170