CSE 1101: Structured Programming Language

Weeks 2-3: Fundamental Concepts in C

- Fundamental data types in C
- Keywords and Identifiers
- Variables and constants
- C Tokens
- Type modifiers and qualifiers
- Expression and Statement
- Type Conversion and Casting
- Input and output functions in C
- Format Specifiers

Lecture 4: Fundamental data types in C

2.1.1 Data Types

In C programming, a data type is a classification that specifies which type of value a variable can hold. It defines the set of operations that can be performed on the data, the meaning of the data, and the way values of that type can be stored. The choice of data type is crucial because it determines the amount of memory required to store the data and the operations that can be performed on the data.

C provides several built-in data types, and they can be broadly categorized into the following types:

- **Basic Data Types:**
  - **int**: Integer data type used to represent whole numbers.
  - **float**: Floating-point data type used to represent real numbers with single precision.
  - **double**: Floating-point data type used to represent real numbers with double precision.
  - **char**: Character data type used to represent individual characters.
- **Derived Data Types:**
  - **Arrays**: Collection of elements of the same data type stored in contiguous memory locations.
  - **Structures**: User-defined data type that groups different data types together.
  - **Pointers**: Variables that store memory addresses, pointing to the location of another variable.
- **Enumeration Data Type:**
  - enum: User-defined data type that consists of a set of named integer constants.
- **Void Data Type:**

- void: Represents the absence of a data type. It is often used in functions that do not return a value or to indicate generic pointers.

2.1.1.1 Introduction to fundamental data types (int, float, double, char)

In C, data types are used to define the type of data a variable can hold. The fundamental data types include:

- **char (Character)**
  - **Description**: Used for representing a single character.
  - **Range**: Typically 0 to 255 or -128 to 127 (if signed).
  - **Precision**: 8 bits, representing a single character.
- **int (Integer)**
  - **Description**: Used for representing whole numbers.
  - **Range**: Depends on the system architecture. Commonly -32768 to 32767 or -2147483648 to 2147483647 for 16-bit and 32-bit systems, respectively.
  - **Precision**: Typically 32 bits, providing about 9-10 decimal digits of precision.
- **float (Floating Point)**
  - **Description**: Used for representing real numbers (floating-point numbers).
  - **Range**: Approximately ±3.4E38 (7 decimal digits of precision).
  - **Precision**: Typically 32 bits, providing about 7 decimal digits of precision.
- **double (Double Precision Floating Point Number)**
  - **Description**: Used for representing real numbers with higher precision than float.
  - **Range**: Approximately ±1.7E308 (15 decimal digits of precision).
  - **Precision**: Typically 64 bits, providing about 15 decimal digits of precision.
- **void (Valueless)**
  - void: Represents the absence of a data type. It is often used in functions that do not return a value or to indicate generic pointers.

Data Type	Keyword	Size	Format Specifier(s)	Literals	Constant	Range
Char	**Char**	**1 byte**	**%c**	**'A', '\n'**	**CHAR_MAX: -128 to 127 CHAR_MIN: 0 to 255**	**-128 to 127 (signed) 0 to 255 (unsigned)**
Integer	**Int**	**System-dependent**	**%d**	**42, -32768**	**INT_MAX: Depends on system INT_MIN: Depends on system**	**Depends on system**
Floating point Number	**Float**	**4 bytes**	**%f**	**3.14f**	**FLT_MAX: ~3.4E38 FLT_MIN: ~1.2E-38**	**~3.4E38 to ~1.2E-38**
Double precision Floating Point Number	**Double**	**8 bytes**	**%lf**	**3.14**	**DBL_MAX: ~1.7E308 DBL_MIN: ~2.2E-308**	**~1.7E308 to ~2.2E-308**
Void	**Void**	**valueless**	**N/A**	**N/A**	**N/A**	**N/A**

2.1.1.2 Derived Data Types

- **Arrays**
  - **Description**: Collections of elements of the same data type.
  - **Declaration**: int numbers[5]; declares an array of 5 integers.
  - **Accessing Elements**: Elements are accessed using indices, e.g., numbers[0].
- **User Defined : Structures/Union**
  - **Description**: User-defined composite data type that groups variables of different data types.
  - **Declaration**:
    ```
    struct Person {
       char name[50];
       int age;
       float salary;
    };
    ```
- **Pointers**
  - **Description**: Variables that store memory addresses.
  - **Declaration**: int *ptr; declares a pointer to an integer.
  - **Usage**: Pointers are used for dynamic memory allocation and manipulation.

2.1.2 Data Type Modifiers

In C, data type modifiers are keywords that are used to alter the properties of the basic data types. These modifiers provide additional information about the size or storage of the variables they modify. The main data type modifiers in C include:

- **short Modifier:**
  - It is used with int and unsigned int.
  - It reduces the size of the integer variable to save memory.
  - Example: short int x; or short x;
- **long Modifier:**
  - It is used with int, double, and unsigned int.
  - It increases the size of the variable.
  - Example: long int y; or long y;
- **signed and unsigned Modifiers:**
  - They are used with int, char, and short.
  - signed allows both positive and negative values.
  - unsigned allows only non-negative values.
  - Example: signed int z; or unsigned short w;
- **const Modifier:**
  - It is used to define constants.

  - A variable declared as const cannot be modified once assigned.
  - Example: const int MAX_SIZE = 100;
- **volatile Modifier:**
  - It informs the compiler that the variable may be changed at any time by external forces not known to the compiler.
  - It prevents the compiler from making assumptions about the variable.
  - Example: volatile int sensorValue;
- **register Modifier:**
  - It suggests to the compiler that the variable be stored in a register for faster access.
  - The compiler may or may not honor this suggestion.
  - Example: register int counter;

These modifiers allow developers to fine-tune the behavior of variables based on specific requirements, such as memory constraints or the need for faster access. Choosing the appropriate data type and modifiers is essential for optimizing code performance and ensuring correct behavior.

2.1.2.1 Integer Types

The range of integer types in C depends on the number of bits allocated for each type. Here are the typical ranges for various integer types

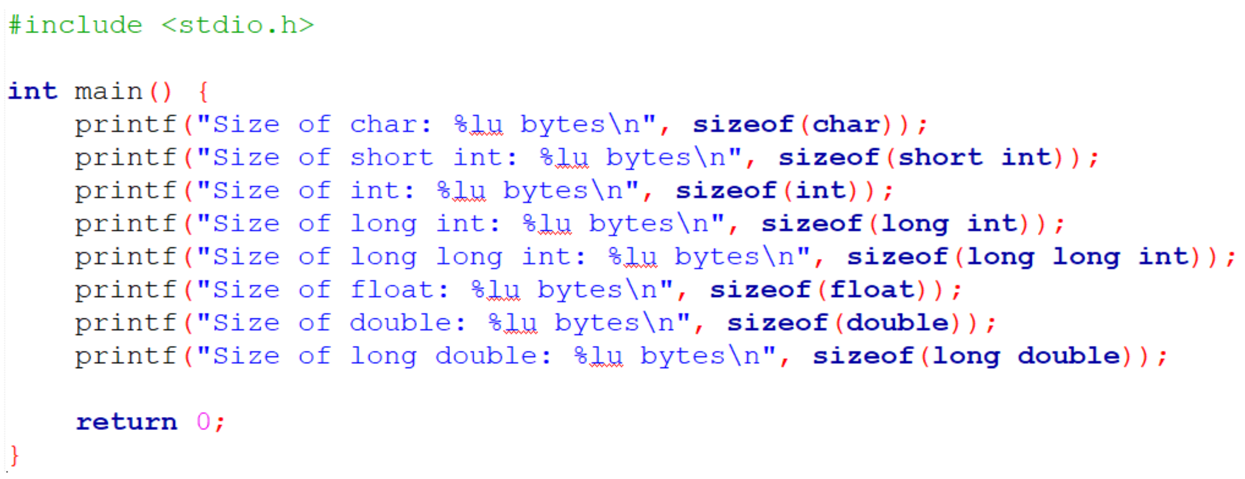
For signed types:

char: -128 to 127 (8 bits)
short int: -32768 to 32767 (16 bits)
int: -2147483648 to 2147483647 (32 bits)
long int: -2147483648 to 2147483647 (32 bits on most systems)
long long int: -9223372036854775808 to 9223372036854775807 (64 bits)

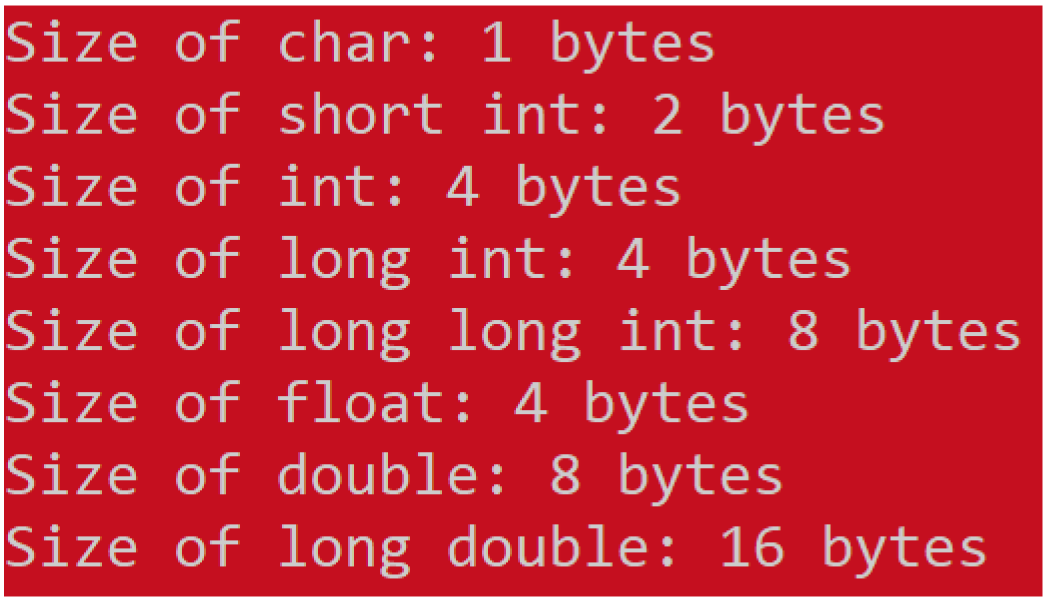
For unsigned types:

unsigned char: 0 to 255 (8 bits)
unsigned short int: 0 to 65535 (16 bits)
unsigned int: 0 to 4294967295 (32 bits)
unsigned long int: 0 to 4294967295 (32 bits on most systems)
unsigned long long int: 0 to 18446744073709551615 (64 bits)

It's important to note that the sizes of these types can vary between different systems and compilers. The ranges mentioned above are based on common conventions, but you can use the limits.h header to get these values on your specific system using the CHAR_BIT, SCHAR_MIN, SCHAR_MAX, SHRT_MIN, SHRT_MAX, INT_MIN, INT_MAX, LONG_MIN, LONG_MAX, LLONG_MIN, and LLONG_MAX macros.



```c
#include <stdio.h>

int main() {
    printf("Size of char: %lu bytes\n", sizeof(char));
    printf("Size of short int: %lu bytes\n", sizeof(short int));
    printf("Size of int: %lu bytes\n", sizeof(int));
    printf("Size of long int: %lu bytes\n", sizeof(long int));
    printf("Size of long long int: %lu bytes\n", sizeof(long long int));
    printf("Size of float: %lu bytes\n", sizeof(float));
    printf("Size of double: %lu bytes\n", sizeof(double));
    printf("Size of long double: %lu bytes\n", sizeof(long double));

    return 0;
}
```



```
Size of char: 1 bytes
Size of short int: 2 bytes
Size of int: 4 bytes
Size of long int: 4 bytes
Size of long long int: 8 bytes
Size of float: 4 bytes
Size of double: 8 bytes
Size of long double: 16 bytes
```

Engr. Mohammad Mamun Hossain
BSc.(Eng.) and MSc.(Thesis) in CSE, SUST
PhD (Pursuing) in CSE, RUET.
Assistant Professor, Dept. of CSE, BAUST

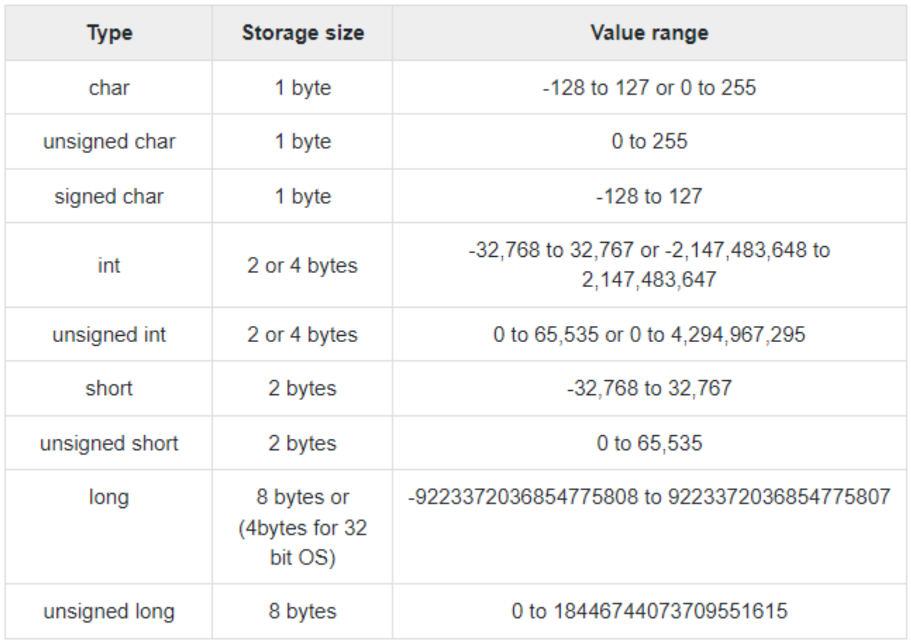
Contact: mhossain@baust.edu.bd
linkedIn:mamunsust12
Resources: mamunsust12@GitHub
Emergency: 01717690847

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <float.h>

int main(int argc, char** argv) {

    printf("CHAR_BIT    :   %d\n", CHAR_BIT);
    printf("CHAR_MAX    :   %d\n", CHAR_MAX);
    printf("CHAR_MIN    :   %d\n", CHAR_MIN);
    printf("INT_MAX     :   %d\n", INT_MAX);
    printf("INT_MIN     :   %d\n", INT_MIN);
    printf("LONG_MAX    :   %ld\n", (long) LONG_MAX);
    printf("LONG_MIN    :   %ld\n", (long) LONG_MIN);
    printf("SCHAR_MAX   :   %d\n", SCHAR_MAX);
    printf("SCHAR_MIN   :   %d\n", SCHAR_MIN);
    printf("SHRT_MAX    :   %d\n", SHRT_MAX);
    printf("SHRT_MIN    :   %d\n", SHRT_MIN);
    printf("UCHAR_MAX   :   %d\n", UCHAR_MAX);
    printf("UINT_MAX    :   %u\n", (unsigned int) UINT_MAX);
    printf("ULONG_MAX   :   %lu\n", (unsigned long) ULONG_MAX);
    printf("USHRT_MAX   :   %d\n", (unsigned short) USHRT_MAX);

    return 0;
}
```

When you compile and execute the above program, it produces the following result on Linux –

```
CHAR_BIT    :   8
CHAR_MAX    :   127
CHAR_MIN    :   -128
INT_MAX     :   2147483647
INT_MIN     :   -2147483648
LONG_MAX    :   9223372036854775807
LONG_MIN    :   -9223372036854775808
SCHAR_MAX   :   127
SCHAR_MIN   :   -128
SHRT_MAX    :   32767
SHRT_MIN    :   -32768
UCHAR_MAX   :   255
UINT_MAX    :   4294967295
ULONG_MAX   :   18446744073709551615
USHRT_MAX   :   65535
```

The following table provides the details of standard integer types with their storage sizes and value ranges –



Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes or (4bytes for 32 bit OS)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

To get the exact size of a type or a variable on a particular platform, you can use the sizeof operator. The expressions sizeof(type) yields the storage size of the object or type in bytes. Given below is an example to get the size of various type on a machine using different constant defined in limits.h header file –

2.1.2.2 Floating-Point Types

The following table provide the details of standard floating-point types with storage sizes and value ranges and their precision –

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. The following example prints the storage space taken by a float type and its range values –

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <float.h>

int main(int argc, char** argv) {

    printf("Storage size for float : %d \n", sizeof(float));
    printf("FLT_MAX     :   %g\n", (float) FLT_MAX);
    printf("FLT_MIN     :   %g\n", (float) FLT_MIN);
    printf("-FLT_MAX    :   %g\n", (float) -FLT_MAX);
    printf("-FLT_MIN    :   %g\n", (float) -FLT_MIN);
    printf("DBL_MAX     :   %g\n", (double) DBL_MAX);
    printf("DBL_MIN     :   %g\n", (double) DBL_MIN);
    printf("-DBL_MAX     :  %g\n", (double) -DBL_MAX);
    printf("Precision value: %d\n", FLT_DIG );

    return 0;
}
```

When you compile and execute the above program, it produces the following result on Linux –

```
Storage size for float : 4
FLT_MAX      :   3.40282e+38
FLT_MIN      :   1.17549e-38
-FLT_MAX     :   -3.40282e+38
-FLT_MIN     :   -1.17549e-38
DBL_MAX      :   1.79769e+308
DBL_MIN      :   2.22507e-308
-DBL_MAX     :  -1.79769e+308
Precision value: 6
```

2.1.2.3 The void Type

The void type specifies that no value is available. It is used in three kinds of situations −

Sr.No.	Types & Description
1	**Function returns as void** There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, **void exit (int status);**
2	**Function arguments as void** There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, **int rand(void);**
3	**Pointers to void** A pointer of type void * represents the address of an object, but not its type. For example, a memory allocation function **void *malloc(size_t size);** returns a pointer to void which can be casted to any data type.

Ref : https://www.tutorialspoint.com/cprogramming/c_data_types.htm#

Summary
Understanding fundamental data types is essential for writing C programs. The choice of data type impacts the memory usage and precision of variables. Derived data types like arrays, structures, and pointers provide versatility for handling complex data structures and dynamic memory.

Lecture 5: Variable and Identifiers

2.2 Variables

Variables are names used in a C program to store values. In C, you need to declare a variable before using it by specifying its type and name.
The syntax for declaring a variable in C is:

type variable_name;

For example:

int age;
float height;
char grade;

Where type is the data type of the variable (such as int, float, char, etc.), and variable_name is the name you want to give to the variable.
Once a variable is declared, you can assign a value to it using the assignment operator =. For example:

age = 30;
height = 5.8;
grade = 'A';

You can then use the variable in expressions and statements in your program.
For example:

int total = 10 + age;
printf("Your grade is %c", grade);

Variables: containers for holding data.
Variables are containers for holding data. They provide names to memory locations, allowing us to refer to specific data in our programs. Each variable has a data type that determines the type of data it can store.

2.2.1 Declaring, defining, and initializing variables.

2.2.1.1 Declaring Variables:

Declaring a variable in C involves specifying its name and data type. For example:

```
int age; // Declaring an integer variable named 'age'
```

Here, int is the data type, and age is the variable name.

2.2.1.2 Defining and Initializing Variables:

Defining a variable means allocating memory for it. Initialization is the process of assigning an initial value to a variable. Both declaration and initialization can be done together:

```
int age = 25; // Defining and initializing 'age' with the val
```

Engr. Mohammad Mamun Hossain Contact: mhossain@baust.edu.bd
BSc.(Eng.) and MSc.(Thesis) in CSE, SUST linkedIn:mamunsust12
PhD (Pursuing) in CSE, RUET. Resources: mamunsust12@GitHub
Assistant Professor, Dept. of CSE, BAUST Emergency: 01717690847

2.2.2 Scope and lifetime of Variables:

2.2.2.1 Scope: The scope of a variable refers to the region of the program where the variable is accessible. In C, variables can have local or global scope.

Local Variables: Declared within a specific block of code (e.g., inside a function). They are only accessible within that block.

```
void exampleFunction() {
   int localVar = 10; // Local variable
}
```

Global Variables: Declared outside of any function. They can be accessed throughout the entire program.

```
int globalVar = 20; // Global variable

void anotherFunction() {
   // Accessible here
}
```

2.2.2.2 Lifetime: The lifetime of a variable is the duration during which it exists in memory. It is influenced by its scope and storage class.

Automatic (Local) Variables: Created when the block is entered and destroyed when the block is exited. Their lifetime is limited to the function or block in which they are defined.

```
void exampleFunction() {
   int localVar = 10; // Automatic variable
   // ...
} // localVar ceases to exist here
```

Static Variables: Retain their values between function calls. They have a lifetime equal to the entire program.

```
void staticExample() {
   static int staticVar = 5; // Static variable
   // ...
} // staticVar retains its value between calls
```

Understanding the scope and lifetime of variables is crucial for writing efficient and bug-free C programs. Proper variable management ensures that data is accessed and modified in a controlled and predictable manner.

Note: In C, variables must be declared before they are used, and the type of a variable cannot be changed once it is declared. Additionally, variable names in C are case-sensitive, so age and Age are considered two different variables.

2.3 Identifiers

An identifier in C is a name used to identify variables, functions, arrays, structures, unions, and other elements in a C program.

The rules for naming identifiers in C are:

- An identifier can only contain letters, digits, and underscores.
- An identifier must start with a letter or an underscore.
- C is case-sensitive, so age and Age are considered two different identifiers.
- An identifier cannot be a keyword or a predefined identifier (such as printf).
- An identifier should not be more than 31 characters long

In C, the length of an identifier is implementation-defined, which means it can vary between different compilers and platforms. However, the C standard specifies a maximum limit of 31 characters for an identifier.

In practice, most compilers support much longer identifier names. However, it is still recommended to keep identifier names short and descriptive to make the code easier to read and maintain.

Here are some examples of valid identifiers in C:

```
int age;
float height_in_meters;
char _grade;
```

Note: It is good programming practice to give descriptive names to identifiers in C, such as age or height_in_meters, to make the code easier to read and understand.

2.3.1 Identifiers: names of program elements, including variables.

In programming, identifiers are names given to various program elements, including variables, functions, arrays, and other user-defined entities. Identifiers are crucial for making code readable and understandable.
Examples:

```
int age; // 'age' is an identifier for a variable
void calculateSalary() { /* ... */ } // 'calculateSalary' is an identifier for a function
```

2.3.2 Rules for naming identifiers and best practices

Identifiers in C must adhere to certain rules and best practices for clarity and maintainability.

Rules for Naming Identifiers:

- **Valid Characters:** Use letters (both uppercase and lowercase), digits, and underscores.
- **Start with a Letter or Underscore:** The first character must be a letter or underscore.
- **Case-Sensitive:** C is case-sensitive, so name and Name are different identifiers.
- **No Reserved Words:** Avoid using reserved words (keywords) as identifiers.

- **No Spaces or Special Characters:** Spaces and special characters (except underscore) are not allowed.
- **Meaningful Names:** Choose names that convey the purpose of the variable or function.

Best Practices:

- **CamelCase or Underscore Separation:** Choose a consistent naming convention (e.g., totalAmount or total_amount).
- **Descriptive Names:** Use names that clearly indicate the purpose of the identifier.
- **Avoid Abbreviations**: Aim for clarity; avoid cryptic abbreviations.
- **Maintain Consistency**: Be consistent with naming across the codebase for readability.
- **Follow Project Conventions**: Adhere to any established naming conventions within a project or organization.

Examples:

```
int totalAmount; // CamelCase
int total_amount; // Underscore Separation
```

2.3.3 Identifiers to variables and other program entities.

Identifiers are linked to variables and other program entities, acting as a reference to access or modify them.
Assigning Identifiers to Variables:

```
int age; // 'age' is an identifier assigned to a variable
```

Assigning Identifiers to Functions:

```
void calculateSalary() { /* ... */ } // 'calculateSalary' is an identifier assigned to a function
```

Identifiers play a crucial role in making code human-readable and maintaining code quality. By following naming conventions and best practices, developers can create code that is not only functional but also comprehensible and maintainable over time.

Engr. Mohammad Mamun Hossain
BSc.(Eng.) and MSc.(Thesis) in CSE, SUST
PhD (Pursuing) in CSE, RUET.
Assistant Professor, Dept. of CSE, BAUST

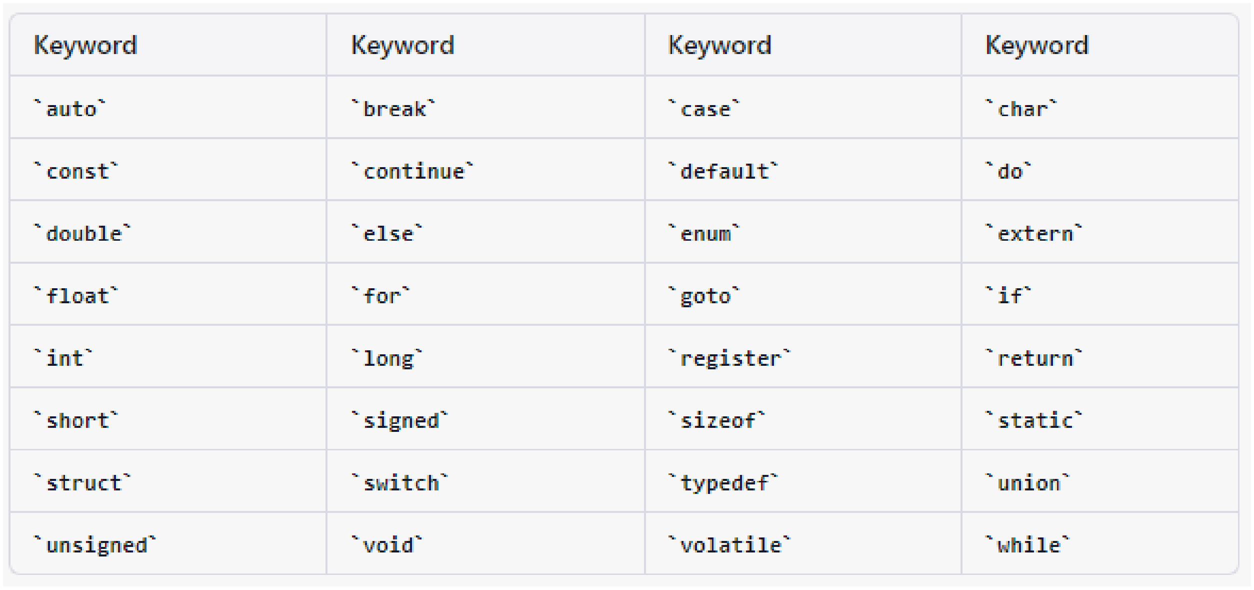
Contact: mhossain@baust.edu.bd
linkedIn:mamunsust12
Resources: mamunsust12@GitHub
Emergency: 01717690847

Lecture 6: Keywords and Constant/Literals

2.4 Keywords:

Keywords are reserved words in C that have a special meaning and cannot be used as identifiers (variable names, function names, etc.). Keywords are also sometimes referred to as "reserved words" or "predefined words".

There are 32 keywords in C. Here's a table of all 32 keywords in C:



Keyword	Keyword	Keyword	Keyword
`auto`	`break`	`case`	`char`
`const`	`continue`	`default`	`do`
`double`	`else`	`enum`	`extern`
`float`	`for`	`goto`	`if`
`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`union`
`unsigned`	`void`	`volatile`	`while`

Note: The list of keywords in C may vary depending on the compiler and the version of the C language standard that you are using.
Here is a list of all the keywords in C and a brief description of their purpose:

1. **auto**: Specifies that a variable declared within a function has automatic storage duration, meaning it will be created and destroyed each time the function is called.
2. **break**: Terminates the execution of a loop or switch statement.
3. **case**: Used in a switch statement to specify a particular branch of code to execute.
4. **char**: Specifies that a variable is of type character, representing a single character in the ASCII character set.
5. **const**: Specifies that a variable is a constant and cannot be modified.
6. **continue**: Skips the current iteration of a loop and continues with the next iteration.
7. **default**: Used in a switch statement to specify a default branch of code to execute when none of the other cases are met.
8. **do**: Begins a do-while loop, which will execute the loop body at least once before testing the loop condition.
9. **double**: Specifies that a variable is of type double-precision floating point, representing a real number with a larger range and precision than a single-precision float.
10. **else**: Used in an if-else statement to specify the code to execute if the condition is false.

Engr. Mohammad Mamun Hossain Contact: mhossain@baust.edu.bd
BSc.(Eng.) and MSc.(Thesis) in CSE, SUST linkedIn:mamunsust12
PhD (Pursuing) in CSE, RUET. Resources: mamunsust12@GitHub
Assistant Professor, Dept. of CSE, BAUST Emergency: 01717690847

11. **enum**: Declares an enumerated type, a type consisting of a set of named integer constants.
12. **extern**: Specifies that a variable or function is declared in another source file and can be used in the current file.
13. **float**: Specifies that a variable is of type single-precision floating point, representing a real number with a limited range and precision.
14. **for**: Begins a for loop, which will execute a specified number of iterations.
15. **goto**: Jumps to another location in the program specified by a label.
16. **if**: Begins an if statement, which will conditionally execute code based on a specified condition.
17. **int**: Specifies that a variable is of type integer, representing a whole number.
18. **long**: Specifies that a variable is of type long integer, representing a whole number with a larger range than a regular integer.
19. **register**: Specifies that a variable should be stored in a register instead of memory, potentially improving performance.
20. **return**: Terminates the execution of a function and returns a value.
21. **short**: Specifies that a variable is of type short integer, representing a whole number with a smaller range than a regular integer.
22. **signed**: Specifies that a variable is of a signed integer type, allowing for positive and negative values.
23. **sizeof**: Returns the size in bytes of a variable or type.
24. **static**: Specifies that a variable has static storage duration, meaning it will persist for the duration of the program.
25. **struct**: Declares a structure type, a composite data type consisting of a set of named variables.
26. **switch**: Begins a switch statement, which will conditionally execute code based on the value of a specified expression.
27. **typedef**: Defines a type alias for a previously declared type.
28. **union**: Declares a union type, a composite data type consisting of a set of named variables

The other 4 keywords in C are:

- "**signed**" - Specifies the type of data being stored as a signed integer.
- "**unsigned**" - Specifies the type of data being stored as an unsigned integer.
- "**register**" - Suggests to the compiler to store a variable in the CPU register for faster access.
- "**short**" - Specifies the type of data being stored as a short integer.

Engr. Mohammad Mamun Hossain Contact: mhossain@baust.edu.bd
BSc.(Eng.) and MSc.(Thesis) in CSE, SUST linkedIn:mamunsust12
PhD (Pursuing) in CSE, RUET. Resources: mamunsust12@GitHub
Assistant Professor, Dept. of CSE, BAUST Emergency: 01717690847

2.5 Constants / Literals

2.5.1 Fixed values Constant: Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals.

Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are enumeration constants as well.

Constants are treated just like regular variables except that their values cannot be modified after their definition.

1. **Numeric Constants:** Numeric constants can be either integer or floating-point values. For example: 10, 3.14, -27, etc.
2. **Character Constants:** A character constant is a single character surrounded by single quotes, such as 'A', 'a', '+', etc.
3. **String Literals:** A string literal is a sequence of characters surrounded by double quotes, such as "hello", "world", etc.
4. **Enumeration Constants:** Enumeration constants are user-defined constants with unique integer values. For example:
   enum colors {RED, GREEN, BLUE};
5. **Define Constants:** The #define directive is used to define constants in C. For example:
   #define PI 3.14

Constants are often used in C programs to store values that do not change, such as the value of PI in the example above. Using constants instead of hard-coded values makes the code easier to maintain and understand.

2.5.2 Symbolic constant

A symbolic constant in C is a constant that has a name and a value, and is defined using the #define preprocessor directive.

For example:

#define PI 3.14

In this example, PI is a symbolic constant with a value of 3.14. The value of a symbolic constant cannot be changed during the execution of a program.

Symbolic constants are often used in C programs to store values that do not change, such as the value of PI in the example above. Using symbolic constants instead of hard-coded values makes the code easier to maintain and understand, as the value can be changed in one place if necessary.

Note that symbolic constants do not have a data type, unlike variables. They are simply replaced with their values wherever they appear in the code during preprocessing.

Engr. Mohammad Mamun Hossain
BSc.(Eng.) and MSc.(Thesis) in CSE, SUST
PhD (Pursuing) in CSE, RUET.
Assistant Professor, Dept. of CSE, BAUST
Contact: mhossain@baust.edu.bd
linkedIn:mamunsust12
Resources: mamunsust12@GitHub
Emergency: 01717690847