NEHRU INSTITUTE
OF TECHNOLOGY
AUTONOMOUS

Lecture notes on C Programming

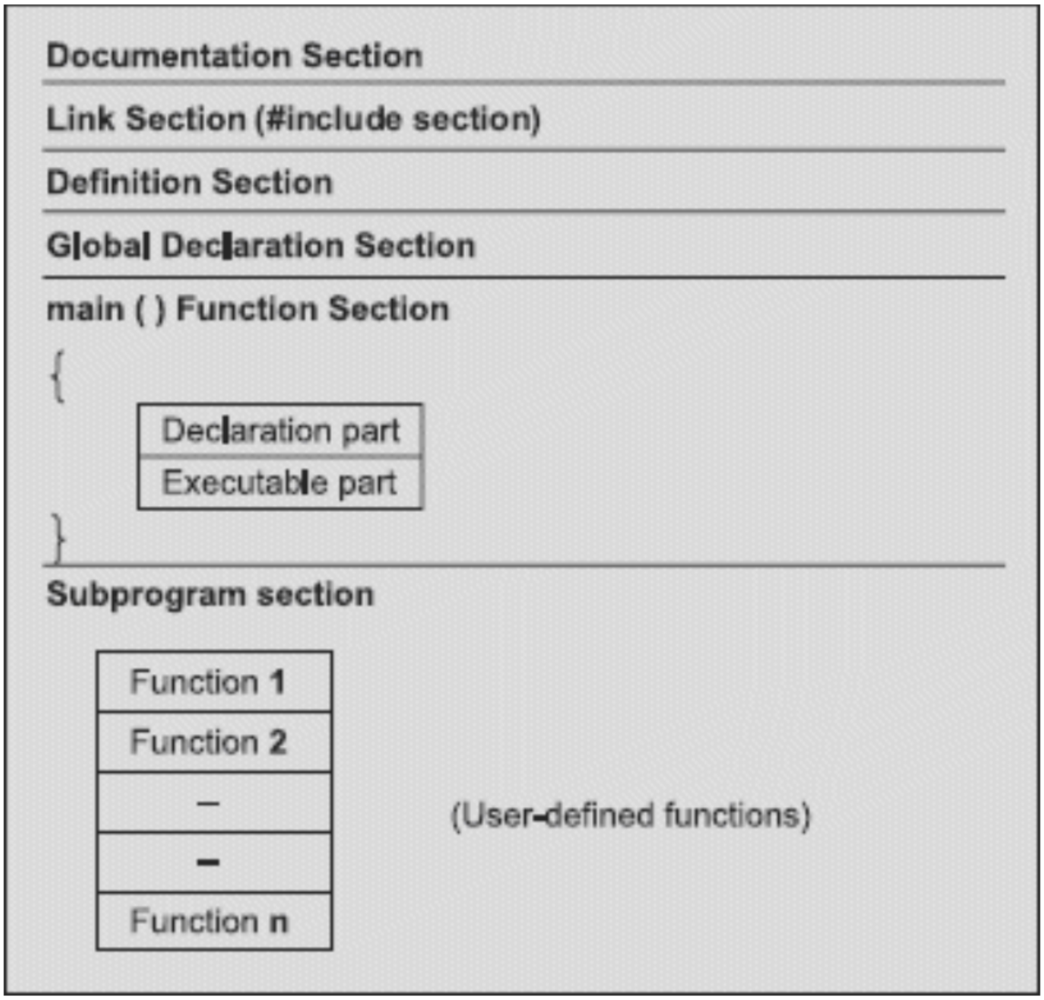
By Jason Pandian

Assistant Professor, Department of Information Technology

Structure of C Program _ C Tokens: Constants, Variables _ Data Types: Primitive Data Types, Type Definition, Operators and Expressions _ Managing Input and Output Operations

Structure of C Program

- A C program can be viewed as a group of building blocks called functions.
- A function is a subroutine that may include one or more statements designed to perform a specific task.
- To write a C program, we first create functions and then put them together.



Structure of C Program

In [ ]:

```
/*
 * Program: [Program Name]
 * Author: [Your Name]
 * Date: [Date of Creation]
 * Purpose: [Brief Description of the Program]
 */


#include <stdio.h>

int main(void) {
    // Program logic goes here
    printf("Hello, World!");
    return 0; // Indicates successful execution
}
```

C Tokens

- In C programming, a token is the unit that a compiler can understand.

- A program that you write is parsed as tokens and then executed to binary
- C tokens are classified into six categories:

1.Keywords

- All keywords have fixed meanings and these meanings cannot be changed. Keywords serve as basic building blocks for program statements.
- Examples: int, char, if, else, for, while, etc.

2.Identifiers

- Identifiers refer to the names of variables, functions and arrays.
- These are user-defined names and consist of a sequence of letters and digits, with a letter as a first character.
- Both uppercase and lowercase letters are permitted, although lowercase letters are commonly used.
- Must begin with a letter or underscore, followed by letters, digits, or underscores.
- Examples: main, count, _value, etc.

3. Constants

- Constants in C refer to fixed values that do not change during the execution of a program.

Types

- **Integer Constants:** 10, -20.
- **Floating-point Constants:** 3.14.
- **Character Constants:** 'A', '\n'.

In [ ]:

```
int const value = 10; // 10
value = value + 1; // produce a error
```

4. String Literals

- Sequence of characters enclosed in double quotes.
- Example: "Hello, World!"

5. Operators

- Symbols that perform operations on variables and values.

- Examples: +, -, *, /, %, ==, !=, &&, ||, etc.

6. Punctuation Symbols

- Symbols used to define the structure of C programs.
- Examples: ; (semicolon), , (comma), . (dot), () (parentheses), {} (braces), [] (square brackets), etc.

7. Comments

- Used for documentation and are ignored by the compiler.
- Single-line comment: // This is a comment
- Multi-line comment: /* This is a multi-line comment */
- Here's a simple example that includes different tokens:

Variables

- A variable is a data name that may be used to store a data value.
- Unlike constants that remain unchanged during the execution of a program,
- a variable may take different values at different times during execution.

Syntax

data_type variable_name;

In [ ]:

```
int value = 10;
// 10
value = value + 1;
//11
```

1.Use Descriptive Names

In [ ]:

```
int totalAmount = 1000;
```

2.Use CamelCase or Underscore Notation

In [ ]:

```
// CamelCase
int studentCount;

// Underscore Notation
int student_count;
```

3.Use Uppercase for Constants

In [ ]:

```
#define MAX_SIZE 100
const int BUFFER_SIZE = 256;
```

4.Global Variables and Local Variables

In [ ]:

```
// Global variable
int g_globalVar;

int main() {
    // Local variable
    int localVar;
    // ...
}
```

5.Use Plural for Arrays or Collections:

In [ ]:

```
// Good
int numbers[10];
char names[MAX_NAMES];
```

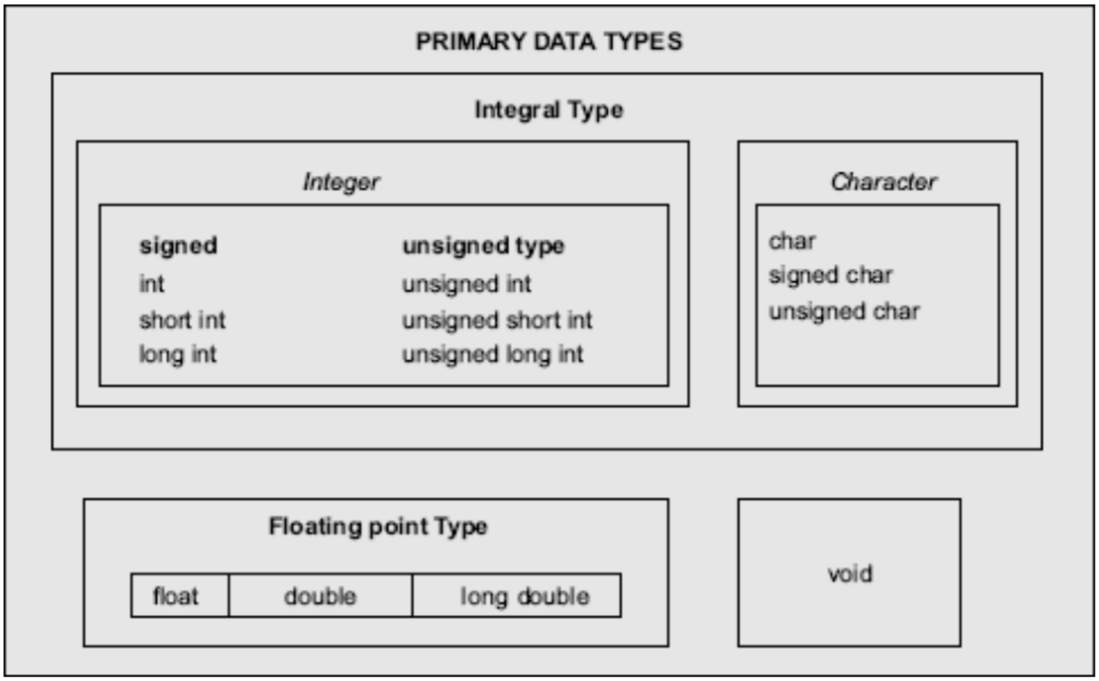
- **C language is rich in its data types.**
- **Storage representations and machine instructions to handle constants differ from machine to machine.**
- **The variety of data types available allows the programmer to select the type appropriate to the needs of the application as well as the machine.**

ANSI C supports three classes of data types:

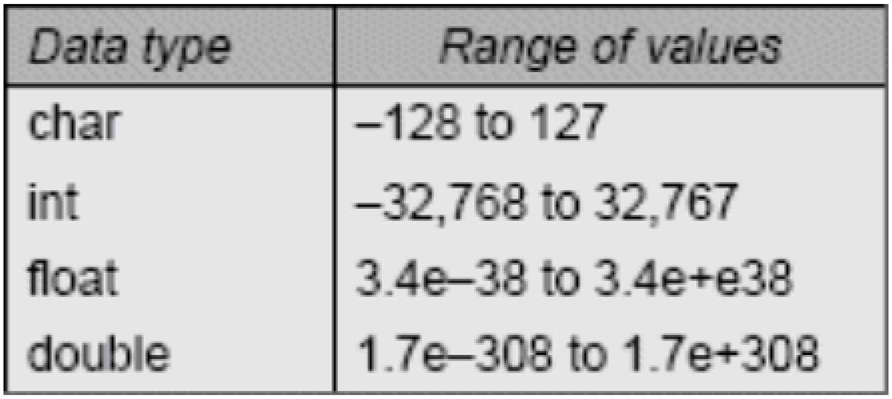
- Primary (or fundamental) data types
- Derived data types
- User-defined data types

- All C compilers support five fundamental data types, namely integer (`int`), character (`char`), floating point (`float`), double-precision floating point (`double`), and `void` . Many of them also offer extended data types such as `long int` and `long double` .

Primary Data Types

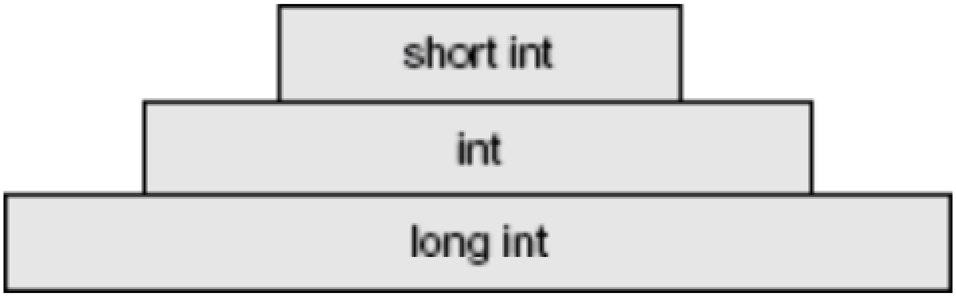


Size and Range of Basic Data Types on 16-bit Machines



Data type	*Range of values*
char	–128 to 127
int	–32,768 to 32,767
float	3.4e–38 to 3.4e+e38
double	1.7e–308 to 1.7e+308

Integer types



In [ ]:

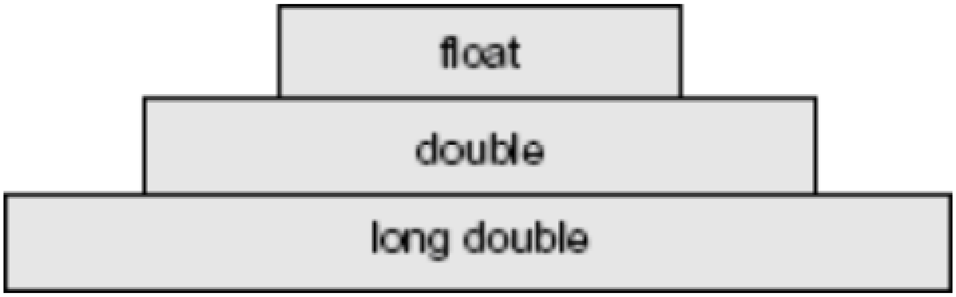
```
// 16-bit integer (2 bytes)
short int shortInteger = 32767;

// 32-bit integer (4 bytes)
int regularInteger = 2147483647;

// 32 or 64-bit integer (platform-dependent)
long longInteger = 2147483648;
```

Float types

Float is a decimal point data type which has double, and long double for extra precision.



In [ ]:

```
// 32-bit floating-point (4 bytes)
float pi = 3.14159;

// 64-bit floating-point (8 bytes)
double myDoubleVariable = 42.5678;

// 80-bit or 128-bit floating-point (10 or 16 bytes)
long double myLongDoubleVariable = 123.456789012345;
```

Char types

In [ ]:

```
// 8-bit character (1 byte)
char myCharVariable = 'A';
```

Bool type

- In C, the keyword for boolean values is typically `bool`, and it is provided by including the `<stdbool.h>` header.
- However, it's essential to note that the C standard doesn't specify a fixed size for `bool`.

include <stdbool.h>

// Implementation-dependent size (commonly 1 byte) bool myBoolVariable = true;

Operators and Expressions

Arithmetic Operators

- `+` (addition)
- `-` (subtraction)
- `*` (multiplication)
- `/` (division)
- `%` (modulus)

In [ ]:

```
#include <stdio.h>

int main()
{
    // Arithmetic Operators
    int a = 10, b = 5;
    printf("Addition: %d\n", a + b);
    printf("Subtraction: %d\n", a - b);
    printf("Multiplication: %d\n", a * b);
    printf("Division: %d\n", a / b);
    printf("Modulus: %d\n\n", a % b);

    return 0;
}
```

Relational Operators:

- `==` (equal to)
- `!=` (not equal to)
- `>` (greater than)
- `<` (less than)
- `>=` (greater than or equal to)
- `<=` (less than or equal to)

In [ ]:

```
#include <stdio.h>

int main()
{
 // Relational Operators
    int x = 8, y = 12;
    printf("Equal to: %d\n", x == y); // false
    printf("Not equal to: %d\n", x != y); // true
    printf("Greater than: %d\n", x > y); // false
    printf("Less than: %d\n", x < y); // true
    printf("Greater than or equal to: %d\n", x >= y); // false
    printf("Less than or equal to: %d\n\n", x <= y); //true

    return 0;
}
```

Logical Operators:

- `&&` (logical AND)
- `||` (logical OR)
- `!` (logical NOT)

In [ ]:

```
#include <stdio.h>

int main() {
    // Logical Operators
    int p = 1, q = 0;

    // 1 AND 0
    printf("Logical AND: %d\n", p && q);

    // 1 OR 0
    printf("Logical OR: %d\n", p || q);

    // NOT 1
    printf("Logical NOT: %d\n\n", !p);

    return 0;
}
```

Assignment Operators:

- `=` (assignment)
- `+=` (addition assignment)
- `-=` (subtraction assignment)
- `*=` (multiplication assignment)
- `/=` (division assignment)
- `%=` (modulus assignment)

In [ ]:

```
#include <stdio.h>

int main() {
    // Assignment Operators
    int num = 5;
    num += 3;
    printf("Addition Assignment: %d\n", num);

    return 0;
}
```

Increment and Decrement Operators:

- `++` (increment)
- `--` (decrement)

In [ ]:

```
#include <stdio.h>

int main() {
    // Increment and Decrement Operators
    int count = 5;
    count++; // count =  count + 1
    printf("Increment: %d\n", count);
    count--; // count = count - 1
    printf("Decrement: %d\n\n", count);
    return 0;
}
```

Bitwise Operators:

- `&` (bitwise AND)
- `|` (bitwise OR)
- `^` (bitwise XOR)
- `~` (bitwise NOT)
- `<<` (left shift)
- `>>` (right shift)

In [ ]:

```
#include <stdio.h>

int main() {
   // Bitwise Operators
    unsigned int m = 12, n = 7;
    printf("Bitwise AND: %u\n", m & n);
/*
   1100
 & 0111
   ----
   0100
*/
    printf("Bitwise OR: %u\n", m | n);
/*
   1100
 | 0111
   ----
   1111
*/
    printf("Bitwise XOR: %u\n", m ^ n);
/*
   1100
 ^ 0111
   ----
   1011
*/
    printf("Bitwise NOT: %u\n", ~m);
/*
  1100
  ----
  0011

*/
    printf("Left Shift: %u\n", m << 2);
/*
   1100
<<
   ----
   110000

*/
    printf("Right Shift: %u\n\n", m >> 2);
/*
   1100
>>
   ----
   0011

*/
    return 0;
}
```

Conditional (Ternary) Operator:

- `condition ? expression_if_true : expression_if_false`

In [ ]:

```
#include <stdio.h>

int main() {
    // Conditional (Ternary) Operator
    int age = 20;
    printf("You are %s\n", (age >= 18) ? "an adult" : "a minor");


    return 0;
}
```

Managing Input and Output Operations in C

Standard Input/Output Functions

In C, managing input and output involves using standard functions from `stdio.h`. For standard output, `printf` is used to display formatted text, and for input, `scanf` allows formatted user input. Single characters can be handled using `getchar` for input and `putchar` for output. These functions are essential for basic interaction with the console.

In [ ]:

```
#include <stdio.h>

int main() {
    // printf for formatted output
    int num = 5;
    printf("The value of num is: %d\n", num);

    // scanf for formatted input
    int userInput;
    printf("Enter a number: ");
    scanf("%d", &userInput);

    // getchar and putchar for single characters
    char ch;
    printf("Enter a character: ");
    ch = getchar();
    putchar(ch);

    return 0;
}
```

File I/O Functions

For more comprehensive file operations, functions like `fopen`, `fclose`, `fwrite`, and `fread` are employed.

In [ ]:

```
#include <stdio.h>

int main() {
    // fopen, fclose, fwrite, and fread for file operations
    FILE *filePointer;
    filePointer = fopen("example.txt", "w");
    fprintf(filePointer, "Hello, File!");
    fclose(filePointer);

    // fgets and fputs for strings in files
    char buffer[100];
    filePointer = fopen("example.txt", "r");
    fgets(buffer, sizeof(buffer), filePointer);
    fclose(filePointer);

    return 0;
}
```

Error Handling

Error handling is facilitated by `perror` for printing error descriptions, and checking for end-of-file or errors during file processing can be done with `feof` and `ferror` functions. These operations collectively enable effective input and output management in C programs.

In [ ]:

```
#include <stdio.h>

int main() {
    // perror for error description
    FILE *filePointer;
    filePointer = fopen("nonexistentfile.txt", "r");
    if (filePointer == NULL) {
        perror("Error opening file");
    }

    // feof and ferror for end-of-file and error conditions
    filePointer = fopen("example.txt", "r");
    while (!feof(filePointer)) {
        int character = fgetc(filePointer);
        if (ferror(filePointer)) {
            perror("Error reading file");
            break;
        }
        // Process the character
    }
    fclose(filePointer);

    return 0;
}
```

Any Questions or Doubts?

Refer the Lectures/Tutorials GitHub Page