C Programming: Deck 2

Data Types & Variables

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

Topics Covered

Why Data Types?

- Data types define what kind of data a variable can hold
- Determine how much memory is allocated
- Specify what operations can be performed
- Help compiler detect errors
- C is a statically-typed language - type must be declared

Categories of Data Types in C

1. **Basic/Primitive Types**
   - `int`, `char`, `float`, `double`
2. **Derived Types**
   - Arrays, Pointers, Structures, Unions
3. **Enumeration Type**
   - `enum`
4. **Void Type**
   - `void`

This deck focuses on basic/primitive types

Integer Type: int

- Stores whole numbers (no decimal point)
- Can be positive, negative, or zero
- Typically 4 bytes (32 bits) on most systems
- Range: -2,147,483,648 to 2,147,483,647 (on 32-bit systems)
- Default size depends on system architecture

Examples: -42, 0, 100, 2024

Character Type: char

- Stores a single character
- Typically 1 byte (8 bits)
- Enclosed in single quotes
- Internally stored as integer (ASCII value)
- Range: -128 to 127 (signed) or 0 to 255 (unsigned)

Examples: 'A', 'z', '5', '\n', '$'

Floating Point Type: float

- Stores decimal numbers (real numbers)
- Single precision floating-point
- Typically 4 bytes (32 bits)
- Precision: approximately 6-7 decimal digits
- Can represent very large or very small numbers

Examples: 3.14, -0.001, 2.5, 1.0

Double Precision Type: double

- Stores decimal numbers with higher precision
- Double precision floating-point
- Typically 8 bytes (64 bits)
- Precision: approximately 15-16 decimal digits
- More accurate than `float`
- Default type for decimal literals in C

Examples: 3.141592653589793, -0.00000001, 1e10

Basic Data Types Summary

Type	Size	Format	Use
`int`	4 bytes	`%d`	Whole numbers
`char`	1 byte	`%c`	Single character
`float`	4 bytes	`%f`	Decimals (low precision)
`double`	8 bytes	`%lf`	Decimals (high precision)

Note: Sizes may vary by system; these are typical values

What are Size Modifiers?

- Modify the size and range of basic data types
- Applied mainly to `int` type
- Can increase or decrease storage size
- Affect the range of values that can be stored

Modifiers:

- `short`
- `long`
- `long long`

short int

- Typically 2 bytes (16 bits)
- Range: -32,768 to 32,767
- Uses less memory than `int`
- Can write as `short int` or just `short`

Declaration:

- `short x;`
- `short int y;`

long int

- Typically 4 or 8 bytes (system dependent)
- On 32-bit: 4 bytes, same as `int`
- On 64-bit: 8 bytes, larger range
- Range: -2,147,483,648 to 2,147,483,647 (32-bit)
- Can write as `long int` or just `long`

Declaration:

- `long x;`
- `long int y;`

long long int

- At least 8 bytes (64 bits)
- Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- For very large numbers
- Introduced in C99 standard
- Can write as `long long int` or `long long`

Declaration:

- `long long x;`
- `long long int y;`

long double

- Extended precision floating-point
- Typically 10, 12, or 16 bytes (system dependent)
- Greater precision than `double`
- Precision: approximately 18-19+ decimal digits
- For high-precision scientific calculations

Declaration:

- `long double pi = 3.14159265358979323846L;`

Understanding Signed and Unsigned

Signed:

- Can store both positive and negative values
- Default for integer types
- One bit used for sign (+ or -)

Unsigned:

- Can store only non-negative values (0 and positive)
- All bits used for magnitude
- Doubles the positive range

Signed vs Unsigned: Range Comparison

Type	**Range**
`signed char`	-128 to 127
`unsigned char`	0 to 255
`signed short`	-32,768 to 32,767
`unsigned short`	0 to 65,535
`signed int`	-2,147,483,648 to 2,147,483,647
`unsigned int`	0 to 4,294,967,295
`signed long long`	-2^{63} to $2^{63}-1$
`unsigned long long`	0 to $2^{64}-1$

When to Use Unsigned?

Use unsigned when:

- Value will never be negative (e.g., count, size, age)
- Need larger positive range
- Working with bit operations

Declaration:

```
1 unsigned int age = 25;
2 unsigned long population = 8000000000UL;
3 unsigned char byte = 255;
```

sizeof() Operator

- Returns the size of a type or variable in bytes
- Compile-time operator (evaluated at compile time)
- Returns value of type `size_t` (unsigned integer)
- Syntax: `sizeof(type)` or `sizeof variable`

Examples:

```
1 sizeof(int)
2 sizeof(char)
3 sizeof(double)
4 sizeof(x)     // where x is a variable
```

What is a Variable?

- Named storage location in memory
- Holds data that can change during program execution
- Must be declared before use
- Has a type, name, and value

Variable Declaration Syntax:

```
datatype variable_name;
```

Variable Declaration

Simple Declaration:

```
1 int age;
2 char grade;
3 float price;
4 double pi;
```

Multiple Variables of Same Type:

```
1 int x, y, z;
2 float a, b, c;
```

Variable Initialization

Declaration with Initialization:

```
int age = 25;
char grade = 'A';
float price = 99.99;
double pi = 3.14159;
```

Multiple Variables with Initialization:

```
int x = 10, y = 20, z = 30;
float a = 1.5, b = 2.5;
```

Variable Naming Rules

Valid Names:

- Must start with letter (a-z, A-Z) or underscore (_)
- Can contain letters, digits (0-9), and underscores
- Case-sensitive (age $\neq$ `Age`)
- Cannot use C keywords (int, if, while, etc.)

Good Examples: `age`, `student_name`, `marks1`, `_temp`
Bad Examples: `2age`, `student-name`, `int`, `my$money`

What are Constants?

- Fixed values that cannot be changed during program execution
- Improve code readability
- Make code easier to maintain
- Two ways to define constants in C:
  1. Using `const` keyword
  2. Using `#define` preprocessor directive

Constants using const Keyword

Syntax:

```
const datatype variable_name = value;
```

Examples:

```
const int MAX_SIZE = 100;
const float PI = 3.14159;
const char GRADE = 'A';
```

- Value cannot be modified later
- Attempting to modify causes compilation error
- Type-safe (compiler checks type)

Constants using #define

Syntax:

```
#define IDENTIFIER value
```

Examples:

```
#define MAX_SIZE 100
#define PI 3.14159
#define NEWLINE '\n'
```

- Preprocessor replaces all occurrences before compilation
- No semicolon at the end
- No type checking
- Convention: use UPPERCASE names

const vs #define

Feature	**const**	**#define**
Type checking	Yes	No
Memory allocation	Uses memory	Text replacement
Scope	Has scope	Global
Debugging	Can debug	Cannot debug
Usage	Modern C	Traditional C

Program 1: Basic Variable Usage

```c
#include <stdio.h>

int main() {
    int age = 25;
    char grade = 'A';
    float height = 5.9;
    double pi = 3.14159;

    printf("Age: %d\n", age);
    printf("Grade: %c\n", grade);
    printf("Height: %f\n", height);
    printf("Pi: %lf\n", pi);

    return 0;
}
```

Program 1: Output

```
Age: 25
Grade: A
Height: 5.900000
Pi: 3.141590
```

Observations:

- %d for integers
- %c for characters
- %f for float (shows 6 decimal places by default)
- %lf for double

Program 2: sizeof() Operator

```c
#include <stdio.h>

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

    return 0;
}
```

Program 2: Output

```
Size of char: 1 byte(s)
Size of int: 4 byte(s)
Size of float: 4 byte(s)
Size of double: 8 byte(s)
Size of long long: 8 byte(s)
```

Notes:

- %lu used for size_t (unsigned long)
- Output may vary on different systems
- Shows typical values on 64-bit systems

Program 3: All Integer Types Sizes

```c
#include <stdio.h>

int main() {
    printf("short: %lu bytes\n", sizeof(short));
    printf("int: %lu bytes\n", sizeof(int));
    printf("long: %lu bytes\n", sizeof(long));
    printf("long long: %lu bytes\n",
           sizeof(long long));

    printf("\nSigned vs Unsigned:\n");
    printf("signed int: %lu bytes\n",
           sizeof(signed int));
    printf("unsigned int: %lu bytes\n",
           sizeof(unsigned int));

    return 0;
}
```

Program 3: Output

```
short: 2 bytes
int: 4 bytes
long: 8 bytes
long long: 8 bytes

Signed vs Unsigned:
signed int: 4 bytes
unsigned int: 4 bytes
```

Observation:

- Signed and unsigned have same size
- Only the range differs, not the memory

Program 4: Signed vs Unsigned Range

```c
#include <stdio.h>

int main() {
    signed char sc = -128;
    unsigned char uc = 255;

    printf("Signed char: %d\n", sc);
    printf("Unsigned char: %u\n", uc);

    signed int si = -2147483648;
    unsigned int ui = 4294967295U;

    printf("Signed int: %d\n", si);
    printf("Unsigned int: %u\n", ui);

    return 0;
}
```

Program 4: Output

```
Signed char: -128
Unsigned char: 255
Signed int: -2147483648
Unsigned int: 4294967295
```

Notes:

- %u for unsigned integers
- U suffix for unsigned literals
- Demonstrates minimum and maximum values

Program 5: Integer Overflow

```c
#include <stdio.h>

int main() {
    signed char x = 127;      // Max value
    printf("Before overflow: %d\n", x);

    x = x + 1;                // Overflow
    printf("After overflow: %d\n", x);

    unsigned char y = 255;    // Max value
    printf("\nBefore overflow: %u\n", y);

    y = y + 1;                // Wraps to 0
    printf("After overflow: %u\n", y);

    return 0;
}
```

Program 5: Output

```
Before overflow: 127
After overflow: -128

Before overflow: 255
After overflow: 0
```

Observations:

- Signed overflow wraps from max to min
- Unsigned overflow wraps from max to 0
- This is undefined behavior for signed types

Program 6: Integer Underflow

```c
#include <stdio.h>

int main() {
    signed char x = -128;    // Min value
    printf("Before underflow: %d\n", x);

    x = x - 1;               // Underflow
    printf("After underflow: %d\n", x);

    unsigned char y = 0;     // Min value
    printf("\nBefore underflow: %u\n", y);

    y = y - 1;               // Wraps to 255
    printf("After underflow: %u\n", y);

    return 0;
}
```

Program 6: Output

```
Before underflow: -128
After underflow: 127

Before underflow: 0
After underflow: 255
```

Observations:

- Signed underflow wraps from min to max
- Unsigned underflow wraps from 0 to max
- Demonstrates circular nature of integer storage

Program 7: Float Precision

```c
#include <stdio.h>

int main() {
    float f = 3.14159265358979323846;
    double d = 3.14159265358979323846;
    long double ld = 3.14159265358979323846L;

    printf("float: %.20f\n", f);
    printf("double: %.20lf\n", d);
    printf("long double: %.20Lf\n", ld);

    return 0;
}
```

Note: .20 specifies 20 decimal places

Program 7: Output

```
float: 3.141592741012573242 19
double: 3.141592653589793116 00
long double: 3.141592653589793238 51
```

Observations:

- `float` loses precision after 6-7 digits
- `double` maintains precision up to 15-16 digits
- `long double` has highest precision
- Notice the rounding errors due to binary representation

Program 8: Character and ASCII

```c
#include <stdio.h>

int main() {
    char ch1 = 'A';
    char ch2 = 65;     // ASCII value of 'A'
    char ch3 = 'a';

    printf("Character: %c, ASCII: %d\n", ch1, ch1);
    printf("Character: %c, ASCII: %d\n", ch2, ch2);
    printf("Character: %c, ASCII: %d\n", ch3, ch3);

    printf("\nDifference: %d\n", ch3 - ch1);

    return 0;
}
```

Program 8: Output

```
Character: A, ASCII: 65
Character: A, ASCII: 65
Character: a, ASCII: 97
Difference: 32
```

Observations:

- Characters are stored as integers (ASCII values)
- %c displays character, %d displays ASCII
- 'A' = 65, 'a' = 97
- Difference between lowercase and uppercase is 32

Program 9: Constants with const

```c
#include <stdio.h>

int main() {
    const int MAX_STUDENTS = 50;
    const float PI = 3.14159;

    printf("Maximum students: %d\n", MAX_STUDENTS);
    printf("Value of PI: %f\n", PI);

    // MAX_STUDENTS = 100;  // ERROR! Cannot modify

    printf("These values cannot be changed!\n");

    return 0;
}
```

Program 9: Output

```
Maximum students: 50
Value of PI: 3.141590
These values cannot be changed!
```

Notes:

- `const` prevents modification
- Compiler error if you try to change the value
- Good practice for values that shouldn't change

Program 10: Constants with #define

```c
#include <stdio.h>

#define MAX_SIZE 100
#define PI 3.14159
#define GREETING "Hello, World!"

int main() {
    printf("Max size: %d\n", MAX_SIZE);
    printf("PI value: %f\n", PI);
    printf("%s\n", GREETING);

    int array_size = MAX_SIZE;
    printf("Array size: %d\n", array_size);

    return 0;
}
```

Program 10: Output

```
Max size: 100
PI value: 3.141590
Hello, World!
Array size: 100
```

Notes:

- `#define` does text replacement
- No type checking
- Can define any constant (number, string, expression)
- Processed by preprocessor before compilation

Program 11: Variable Scope Example

```c
#include <stdio.h>

int global_var = 100;  // Global variable

int main() {
    int local_var = 50;  // Local variable

    printf("Global variable: %d\n", global_var);
    printf("Local variable: %d\n", local_var);

    {
        int block_var = 25;  // Block scope
        printf("Block variable: %d\n", block_var);
        printf("Can access local: %d\n", local_var);
    }
    // block_var not accessible here

    return 0;
}
```

Program 11: Output

```
Global variable: 100
Local variable: 50
Block variable: 25
Can access local: 50
```

Observations:

- Global variables declared outside all functions
- Local variables declared inside functions
- Block variables exist only within $\{\}$ braces
- Inner scopes can access outer variables

Program 12: Formatting Output

```c
#include <stdio.h>

int main() {
    int x = 5;
    float y = 3.14159;

    printf("Default int: %d\n", x);
    printf("Width 5: %5d\n", x);
    printf("Zero-padded: %05d\n", x);

    printf("\nDefault float: %f\n", y);
    printf("2 decimals: %.2f\n", y);
    printf("Width 10, 3 decimals: %10.3f\n", y);

    return 0;
}
```

Program 12: Output

```
Default int: 5
Width 5:     5
Zero-padded: 00005

Default float: 3.141590
2 decimals: 3.14
Width 10, 3 decimals:       3.142
```

Format Specifiers:

- %5d - minimum width of 5
- %05d - zero padding
- %.2f - 2 decimal places
- %10.3f - width 10, 3 decimals

Common Format Specifiers

Specifier	Type
%d or %i	int (signed)
%u	unsigned int
%c	char
%f	float
%lf	double
%Lf	long double
%ld	long int
%lld	long long int
%lu	unsigned long
%llu	unsigned long long
%x	hexadecimal (lowercase)
%X	hexadecimal (uppercase)
%o	octal

Important Points to Remember

1. Every variable must be declared with a type
2. Type determines size, range, and operations
3. `sizeof()` returns size in bytes
4. Signed types can be negative, unsigned cannot
5. Constants prevent accidental value changes
6. Use appropriate type for your data needs
7. Be careful with overflow and underflow
8. `float` has less precision than `double`

Common Mistakes

1. Using uninitialized variables
   - May contain garbage values
   - Always initialize before use
2. Integer overflow/underflow
   - Check ranges before operations
3. Using `%f` for double in `scanf()`
   - Use `%lf` for double in `scanf()`
   - `%f` works for `printf()` due to promotion
4. Comparing floats with $==$
   - Use tolerance due to precision issues

Try These!

1. Declare variables of all basic types and print their sizes
2. Write a program to swap two integer variables
3. Calculate the area of a circle using `const` for PI
4. Demonstrate overflow with different integer types
5. Print ASCII values of characters 'A' through 'Z'
6. Create a program showing precision difference between float and double

Sample Solution: Swap Two Numbers

```c
#include <stdio.h>

int main() {
    int a = 10, b = 20, temp;

    printf("Before swap: a = %d, b = %d\n", a, b);

    temp = a;
    a = b;
    b = temp;

    printf("After swap: a = %d, b = %d\n", a, b);

    return 0;
}
```

Sample Solution: Swap Output

```
Before swap: a = 10, b = 20
After swap: a = 20, b = 10
```

Sample Solution: Circle Area

```c
#include <stdio.h>

#define PI 3.14159

int main() {
    float radius = 5.0;
    float area;

    area = PI * radius * radius;

    printf("Radius: %.2f\n", radius);
    printf("Area: %.2f\n", area);

    return 0;
}
```

Sample Solution: Circle Area Output

```
Radius: 5.00
Area: 78.54
```

End of Deck 2

Questions?

Next: Deck 3 - Type Conversions