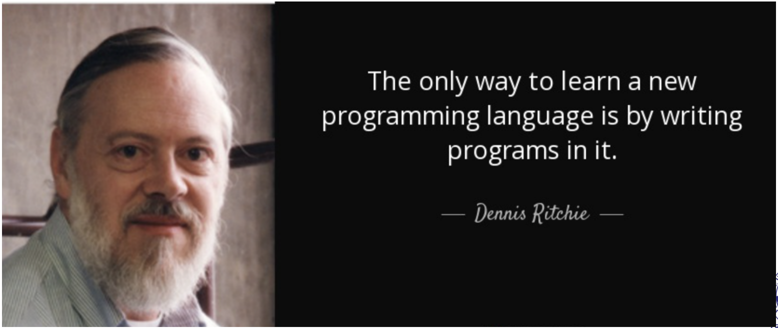
Embedded C Programming
Module-1: Introduction to C

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

C Programming - Overview

- General purpose programming language
- Developed by Dennis Ritchie
- Strengths - flexibility, efficiency, widespread use
- Used for embedded systems, OS, applications



Features of C Language

- Structured programming approach
- Direct low-level hardware access
- Rich set of built-in operators and functions

Example Program:

```
#include <stdio.h>
int main() {
    printf("Hello World \n");
    return 0;
}
```

Introduction to Embedded C

- Program for embedded devices/control, robotics etc.
- Programming for microcontrollers/MCU
- Tightly constrained resource usage
- Low level control of hardware

Difference: C vs Embedded C

C Language:

- High level language
- Large standard library
- Platform independent
- Dynamic memory allocation

Embedded C:

- Tightly constrained
- No standard library
- Platform specific
- Static allocation

Key constraints while programming for embedded systems.

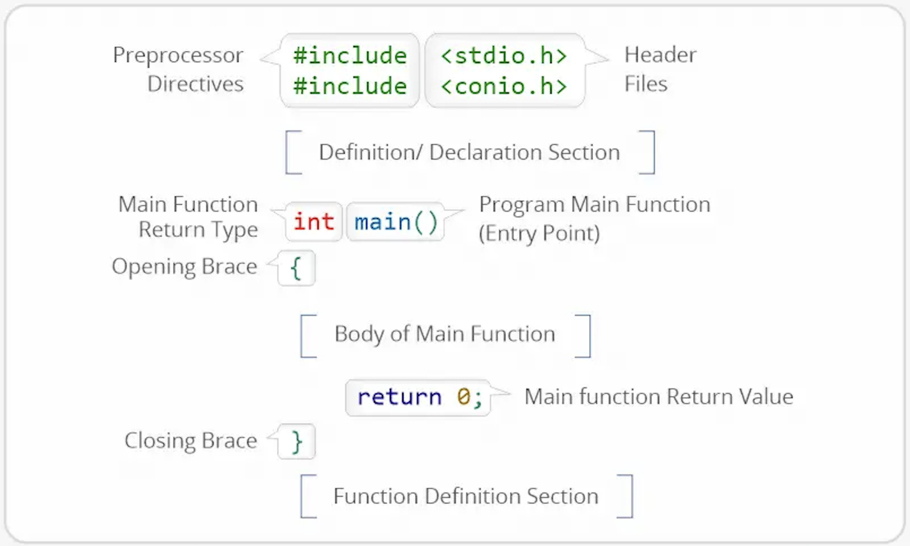
Basic C Program Structure

Structure:

1. Include necessary header files
2. Declare global variables
3. Define functions
4. Implement the main function

```c
#include <stdio.h>
// Declare global variables
int globalVar = 10;

// Function declaration
void myFunction();

int main() {
    // Implementation of main function
    printf("Hello, C Programming!");
    return 0;
}

// Function definition
void myFunction() {
// Implementation of myFunction
}
```

Basic C Program Structure



Embedded C Program Structure

- Interaction with hardware
- Memory considerations
- Real-time constraints

```c
#include <avr/io.h>

// Declare global variables
volatile uint8_t sensorValue;

// Function declaration
void initializeSensor();

int main() {
    // Implementation of main function
    initializeSensor();
    while(1) {
        // Real-time processing
        sensorValue = readSensor();
    }
}
void initializeSensor() {
    // Hardware initialization
}
uint8_t readSensor() {
    // Read data from sensor
}
```

Find the Output

```c
#include <stdio.h>
int main() {
    int x = 5, y = 3;
    printf("%d", x + y);
    return 0;
}
```

Find the Output

```c
#include <stdio.h>
int main() {
    int x = 5, y = 3;
    printf("%d", x + y);
    return 0;
}
```

Answer

The output of the code is 8.

Introduction to Compilation Process

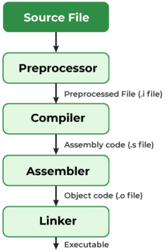
- Pre-processing
- Compilation
- Assembly
- Linking

```c
#include <stdio.h>
#define MAX 10

int main() {

int i = MAX;
printf("Maximum value is: %d", i);

return 0;
}
```

The compilation process converts source code to executable machine code.

Compilation Process

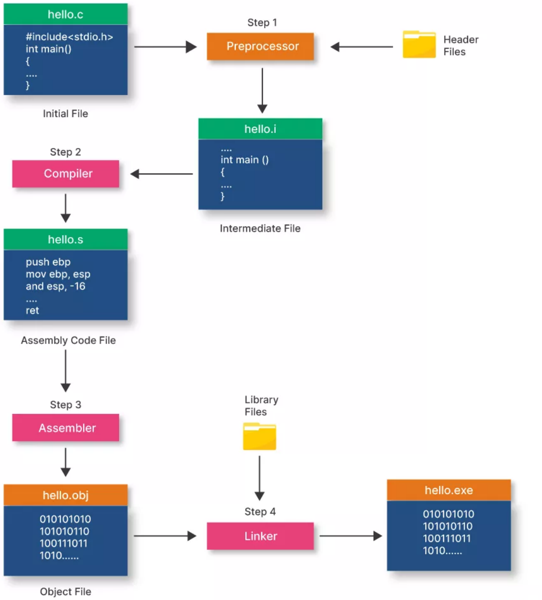
- Preprocessor handles directives (like ‘include‘), compiler translates C to Assembly, assembler to machine code, linker resolves references.
- Tools: C Compiler (e.g., gcc), Assembler (e.g., as), Linker (e.g., ld).



Terminal or GCC Compilation

gcc -E main.c -o main.i # Preprocessing
gcc -S main.i -o main.s # Compilation to assembly
as main.s -o main.o # Assembly to object code
ld main.o -o main # Linking object code to executable

Compilation Process



Comments

Single Line Comment

Used to provide descriptions and explanations in code.
Format: // Comments

Example

```
#include <stdio.h>
int main(){
// Print statement
printf("Hello World!");
return 0;
}
```

Comments help document the code functionality and are ignored by compiler.

Multi-line Comments

Syntax

Used to comment multiple lines or blocks of code.

Format:

/* This is a
multi-line
comment */

Example

```
  /* Comments message
across multiple
lines */
printf("Hello World!");
printf("From C program");
```

Multi-line comments are convenient for commenting code sections.

Identifiers

Naming Rules

- Start with letter or underscore
- Can have letters, digits, underscores
- Case sensitive index $\neq$ INDEX
- No whitespaces allowed

Example

```
#include <stdio.h>
int main() {
int final_count; // valid
int 123Invalid; // invalid
return 0;
}
```

Identifiers refer to user defined names for variables, functions etc. in C.

Variables

Declaring Variables

Specify data type and name: datatype name;

```
int count;
float price;
char code;
```

Initializing Variables

Assign initial value: datatype name = value;

```
int sum = 0;
float pie = 3.14;
char grade = 'A';
```

Variables represent memory locations to store program data.

In-depth Variable Types and Storage Classes

- **Global Variables**: Accessible throughout the program. Example: `int globalVar;`
- **Local Variables**: Accessible only within the function. Example: `void func() { int localVar; }`
- **Static Variables**: Retains value between function calls. Example: `static int staticVar;`
- **Register Variables**: Stored in CPU register for faster access. Example: `register int loopCounter;`

```
// Global Variable
int globalVar;

void function() {
// Local Variable
int localVar;

// Static Variable
static int staticVar = 0;
staticVar++;

// Register Variable
for(register int i = 0; i < 10; i++) {
    // Fast access within loop
}
}
```

Header Files

include

Includes external library contents in program:

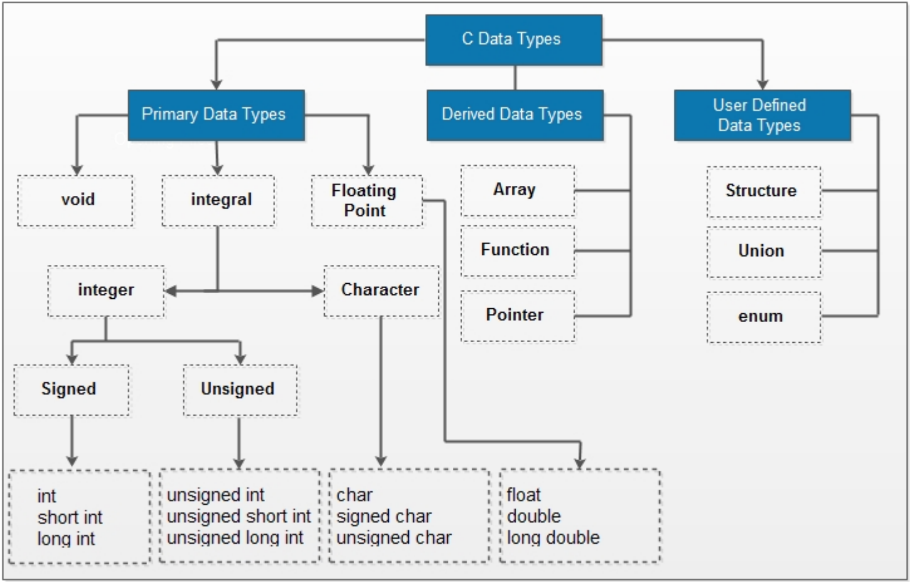
```
#include <stdio.h>
#include "myutils.h"
```

Commonly used library headers in C:

- stdio.h - standard I/O functions
- math.h - mathematical operations
- string.h - string handling

Header files contain reusable code functionalities.

Data Types


C Data Types
Primary Data Types
Derived Data Types
User Defined Data Types
void
integral
Floating Point
Array
Function
Pointer
Structure
Union
enum
integer
Character
Signed
Unsigned
int
short int
long int
unsigned int
unsigned short int
unsigned long int
char
signed char
unsigned char
float
double
long double

- Primitive types
  - int, float, double
  - char
- Derived types
  - Array, pointer
  - Structure, union

```
#include <stdio.h>
int main() {
// Primitive types
int num = 10;
float pi = 3.14;
double height = 79.234;
char x = 'A';

// Derived types
int arr[5];
struct student {
int id;
char name[20];
} s1;
int* ptr = &num;

union data {
int i;
float f;
} val;
return 0;
}
```

Data Types

Character Types	% Format	Size	Range
unsigned char	%c	1 byte	0 to 255
char	%c	1 byte	-128 to 127
signed char	%c	1 byte	-128 to 127
Integer Types			
unsigned short int	%hu	2 bytes	0 to 65,535
short int	%hd	2 bytes	-32,768 to 32,767
signed short int	%hd	2 bytes	-32,768 to 32,767
unsigned int	%u	2/4 bytes	0 to 65,535 or 0 to 4,294,967,295
int	%d	2/4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
signed int	%d	2/4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
long int	%ld	4/8 bytes	-2,147,483,648 to 2,147,483,647
unsigned long int	%lu	4/8 bytes	0 to 4,294,967,295 or 0 to 18,446,744,073,709,551,615
signed long int	%ld	4/8 bytes	-2,147,483,648 to 2,147,483,647
long long int	%lld	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long int	%llu	8 bytes	0 to 18,446,744,073,709,551,615
Float Types			
float	%f	4 bytes	$\pm$ 1.2E-38 to $\pm$ 3.4E+38
double	%lf	8 bytes	$\pm$ 2.3E-308 to $\pm$ 1.7E+308
long double	%Lf	12 bytes	$\pm$ 3.4E–4932 to $\pm$ 1.1E+4932

Exploring Data Types and Sizes

```
#include <stdio.h>
int main() {
    printf("Size of int: %lu bytes\n", sizeof(int));
    printf("Size of char: %lu byte\n", sizeof(char));
    return 0;
}
```

Exploring Data Types and Sizes

```
#include <stdio.h>
int main() {
    printf("Size of int: %lu bytes\n", sizeof(int));
    printf("Size of char: %lu byte\n", sizeof(char));
    return 0;
}
```

Question

What will be the output of this program?

Exploring Data Types and Sizes

```c
#include <stdio.h>
int main() {
    printf("Size of int: %lu bytes\n", sizeof(int));
    printf("Size of char: %lu byte\n", sizeof(char));
    return 0;
}
```

Question

What will be the output of this program?

Answer

The output will display the size of 'int' and 'char' in bytes. Typically, it will be "Size of int: 4 bytes" and "Size of char: 1 byte", but this can vary depending on the system architecture.

Arithmetic Operators

$+ \; - \; * \; / \; \%$

Arithmetic Operators

```
x = y + 2; // Addition
z = p - 5; // Subtraction
area = length * breadth; // Multiplication
q = a / b; // Division
r = 15 % 4;  // Modulus
```

Precedence

Follows order - Exponential > Multiplicative > Additive

```
a = b + (c * 2); // precedence
z = 3 * x / 5;

int num = 15;
float val = num; // typecasting
num + val;
```

Typecasting allows same operands to be treated as different types.

Relational Operators

$> \; < \; >= \; <= \; != $

Chaining Relational Operators

```
if(a > 50)    // Greater than
printf("a is big");

if(b < 20)    // Less than
printf("b is small");

if(str == "test") // Equal to
printf("Strings matched");

if(i != 10)   // Not equal to
print("i is not 10");

if(age >= 18) // Greater than equal to
printf("Eligible");

if(marks <= 35) // Less than equal to
printf("Failed");

//Multiple conditional checks can be chained:
if(x > 5 && x < 10) {
...
}
```

Relational operators can be combined using logical operators.

Logical Operators

Used to combine multiple logical conditions:

- && - Logical AND
- || - Logical OR
- ! - Logical NOT

```
int a = 5;
int b = 10;

if(a < 8 && b >= 10) {
printf("AND condition met");
}

if(a < 8 || b < 5) {
printf("OR condition met");
}

if(!(b == 15)) {
printf("NOT condition met");
}
```

Logical operators are used to implement conditional logic.

Bitwise Operators

Used to manipulate individual bits:

- & - Bitwise AND : Compares bits
- | - Bitwise OR : Makes bits 1 if set in either
- ^- Bitwise XOR : Makes bits 1 if different
- ~ - Bitwise NOT : Inverts all bits

```
int a = 12; // 0000 1100
int b = 25; // 0001 1001

int c = a & b; // 0000 1000
int d = a | b; // 0001 1101

int e = a ^ 5; // 0000 1101
int f = ~a; // 1111 0011
```

Bitwise operators perform operations directly on binary representations.

Other Operators

Some other operators in C:

- sizeof() - size of type/variable
- & - Address of variable
- ? : - Ternary conditional
- , - Comma separates expressions

```
int a;
float b;

printf("Size of int is %d", sizeof(a));
printf("Address of b is %x", &b);

int x = 5;
int res = (x > 2) ? 10 : 0; // Ternary operator

int y = 1, z = 15; // Comma separates
```

C provides special operators for certain usage contexts.

Complex Challenge: Data Types and Operators

```c
#include <stdio.h>
int main() {
    unsigned int x = 5;
    int y = -8;
    printf("Result: %s\n", x > y ? "True" : "False");
    return 0;
}
```

Complex Challenge: Data Types and Operators

```c
#include <stdio.h>
int main() {
    unsigned int x = 5;
    int y = -8;
    printf("Result: %s\n", x > y ? "True" : "False");
    return 0;
}
```

Question

What will be the output of this code and why?

Complex Challenge: Data Types and Operators

```
#include <stdio.h>
int main() {
    unsigned int x = 5;
    int y = -8;
    printf("Result: %s\n", x > y ? "True" : "False");
    return 0;
}
```

Question

What will be the output of this code and why?

Answer

The output is "False". This is because when comparing an unsigned int with an int, the int is implicitly converted to unsigned int. So, -8 is converted to a large unsigned int value 8, which is greater than 5.

Type Promotion and Arithmetic Operations

```c
#include <stdio.h>
int main() {
    short a = 32767; // Max value for short
    short b = a + 1;
    printf("Result: %d\n", b);
    return 0;
}
```

Type Promotion and Arithmetic Operations

```
#include <stdio.h>
int main() {
    short a = 32767; // Max value for short
    short b = a + 1;
    printf("Result: %d\n", b);
    return 0;
}
```

Question

What will be the output, and why is this output observed?

Type Promotion and Arithmetic Operations

```
#include <stdio.h>
int main() {
    short a = 32767; // Max value for short
    short b = a + 1;
    printf("Result: %d\n", b);
    return 0;
}
```

Question

What will be the output, and why is this output observed?

Answer

The output is -32768. In the expression 'a + 1', 'a' is first promoted to an int and then added to 1. The result overflows the range of short, and when it is stored back in 'b', it wraps around to the minimum value for a short.

Order of Operations and Associativity

Operator	Order (Highest to Lowest)	Associativity
() [] -> .	1st Level	Left to Right
! ++ -- + (type) - (type) *	2nd Level	Right to Left
* / %	3rd Level	Left to Right
+ -	4th Level	Left to Right
<< >>	5th Level	Left to Right
< <= > >=	6th Level	Left to Right
== !=	7th Level	Left to Right
&	8th Level	Left to Right
^	9th Level	Left to Right
|	10th Level	Left to Right
&&	11th Level	Left to Right
||	12th Level	Left to Right
?:	13th Level	Right to Left
= += -= *= /= %= &= ^= |= <<= >>=	14th Level	Right to Left
,	15th Level	Left to Right

Order of Operations - Example Codes

Example 1:

```
int x = 5;
int y = x + 3 * 2; // y = 11, not 16
```

Example 2:

```
int a = 5;
int b = a++ + 2; // b = 7, a becomes 6
```

Example 3:

```
int m = 3;
int n = 2 * ++m; // n = 8, m becomes 4
```

Example 4:

```
bool p = true;
bool q = !p; // q = false
```

Example 5:

```
int i = 4; int j = 5;
int k = i * (j - 2) + 6 / 2 - 3; // k = 4 * (5 - 2) + 3 - 3 = 12
```

Debugging Challenge: Order of operation

```
#include <stdio.h>
int main() {
    int a = 10, b = 5, c = 5;
    int result = a / b * c;
    printf("Result: %d\n", result);
    return 0;
}
```

Debugging Challenge: Order of operation

```c
#include <stdio.h>
int main() {
    int a = 10, b = 5, c = 5;
    int result = a / b * c;
    printf("Result: %d\n", result);
    return 0;
}
```

Answer

The output of the code is "Result: 10".
The expression is evaluated as $(a\ /\ b) * c = (10\ /\ 5) * 5 = 2 * 5 = 10$.

Debugging Question: Operators

```
#include <stdio.h>
int main() {
    int i = 5;
    printf("%d %d %d\n", i++, i, ++i);
    return 0;
}
```

Debugging Question: Operators

```
#include <stdio.h>
int main() {
    int i = 5;
    printf("%d %d %d\n", i++, i, ++i);
    return 0;
}
```

Answer

The output of this code is undefined due to the sequence point rule in C. The order of evaluation of expressions involving post-increment and pre-increment operators in the same statement is not defined, which leads to undefined behavior.

Find the Output: Operators Challenge

```
#include <stdio.h>
int main() {
    int x = 2, y = 3, z = 4;
    int result = x + y * z / x - y;
    printf("Result: %d\n", result);
    return 0;
}
```

Find the Output: Operators Challenge

```c
#include <stdio.h>
int main() {
    int x = 2, y = 3, z = 4;
    int result = x + y * z / x - y;
    printf("Result: %d\n", result);
    return 0;
}
```

Answer

The output of the code is 3. The expression is evaluated as follows: - Multiplication and division have higher precedence than addition and subtraction and are evaluated from left to right. - So, y * z / x is evaluated first to get 6, then x + 6 - y results in 3.

Format Specifiers

Syntax

Used with `printf()` and `scanf()` for formatted I/O:

```
printf("Format string", var1, var2);
scanf("Format string", &var1, &var2);
```

Some commonly used specifiers:

- %c - character
- %d - integer
- %f - float
- %s - string

Format specifiers allow displaying outputs in the desired format.

Format Specifiers

Some additional specifiers:

- %ld - long integer
- %lf - double float
- %Lf - long double
- %x - hex integer
- %o - octal integer

```
#include <stdio.h>

int main() {
int num = 10;
long int lnum = 15000000;
float flt = 1.234567;
double dbl = 1.23456789;

printf("Integer: %d\n", num);
printf("Long Integer: %ld\n", lnum);
printf("Float: %f\n", flt);
printf("Double: %lf\n", dbl);
printf("Hexadecimal: %x\n", num);
printf("Octal: %o\n", num);

return 0;
}
```

Format specifiers provide flexibility to print different data types.

Format Specifiers

Format specifiers provide fine grained control over textual output.

Customizing and formatting output:

- width and precision
- padding and signs

```c
#include <stdio.h>
int main() {
printf("Padding: %-6d\n", 12);
printf("Precision: %.4f\n", 123.4567);
printf("Width: %6d\n", 1234);
printf("Sign: %+d | %+d\n", 10, -10);
return 0;
}
```

Latex — -zsh — 80×24

```
(base) markkandan@Markkandans-MBP Latex % ./formatspecifiers2
Padding: 12
Precision: 123.4567
Width:   1234
Sign: +10 | -10
(base) markkandan@Markkandans-MBP Latex %
```

Understanding Format Specifiers Challenge

```
#include <stdio.h>
int main() {
    float num = 12345.6789;
    printf("Output: %.2f and %e\n", num, num);
    return 0;
}
```

Understanding Format Specifiers Challenge

```c
#include <stdio.h>
int main() {
    float num = 12345.6789;
    printf("Output: %.2f and %e\n", num, num);
    return 0;
}
```

Question

What will be the output of this code, considering the format specifiers used?

Understanding Format Specifiers Challenge

```c
#include <stdio.h>
int main() {
    float num = 12345.6789;
    printf("Output: %.2f and %e\n", num, num);
    return 0;
}
```

Question

What will be the output of this code, considering the format specifiers used?

Answer

The output will be "Output: 12345.68 and 1.234568e+04". The first specifier 'formats the float in scientific notation.

Escape Sequences

Escape sequences allow inserting special characters.Used with printf() and character arrays:

Common Escape Codes

- \n - new line - Used for new line
- \t - tab - Used for tab spacing
- \” - single quote - Prints double quotes
- \’ - double quote - Prints single quote

```c
#include <stdio.h>
int main() {
printf("Hello \n World \n");
printf("Name:\tJohn\n");
printf("\"Quotation\" marks\n");
char line[] = "Backslash\escaped";
printf("%s\n", line);
return 0;
}
```

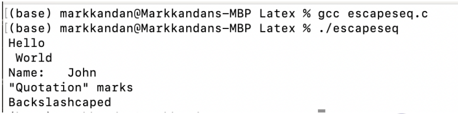


Figure: Output

Console I/O

scanf() and printf() are used for formatted console I/O.

Input

scanf("format", var_address);

Output

printf("format", var);

```
#include <stdio.h>
int main() {
int age;
float salary;
printf("Enter age: ");
scanf("%d", &age);
printf("Enter salary: ");
scanf("%f", &salary);
printf("Age: %d \n", age);
printf("Salary: %0.2f \n", salary);
return 0;
}
```

Enhanced I/O Operations

- **Formatted I/O**: `printf("Value: %d", a);` and `scanf("%d", &a);`
- **Unformatted I/O**: `getchar();` and `putchar();`
- **Common Pitfalls**: Buffer overflow, improper format specifiers.

```
(base) markkandan@Markkandans-MBP Latex % ./io_operations
Enter a number: 2000
You entered: 2000
Enter a string: Hello
You entered: Hello
```

Figure: Output

```c
#include <stdio.h>
int main() {
int number;
char str[100];

// Formatted Input
printf("Enter a number: ");
scanf("%d", &number);
printf("You entered: %d\n", number);

// Formatted Output
printf("Enter a string: ");
scanf("%s", str);
printf("You entered: %s\n", str);

// Unformatted I/O
char ch;
ch = getchar();   // Reads a character
putchar(ch);      // Writes a character
return 0;
}
```

Unformatted I/O Operations in C

```
#include <stdio.h>
int main() {
char ch;
printf("Enter a character: ");
ch = getchar();  // Reads a character from the standard input
printf("Character entered: ");
putchar(ch);     // Writes a character to the standard output
return 0;
}
```

- **getchar()**: Reads a single character from standard input. Waits for input if not available.
- **getch()**: Similar to getchar() but does not echo the character to the console. Often used in DOS-based systems.
- **putchar()**: Writes a single character to standard output.
- **putch()**: Similar to putchar() but used in DOS-based systems.

Best Practices and Coding Standards in C

- **Readability**: Use clear and meaningful variable names, consistent indentation.
- **Modularity**: Break down large problems into smaller, manageable functions.
- **Comments**: Document the code with necessary comments for better understanding.
- **Error Handling**: Implement comprehensive error handling for robustness.
- **Memory Management**: Avoid memory leaks by proper allocation and deallocation.
- **Code Reusability**: Write reusable code to enhance maintainability.

Best Practices and Coding Standards in C

```
#include <stdio.h>

int main() {
// Good practice: clear variable names
int totalItems = 10;
int processedItems = 0;

// Good practice: modular code
while (processedItems < totalItems) {
    // process each item
    processedItems++;
}

// Good practice: error checks and memory management
// Implement necessary checks and memory management

return 0;
}
```

Best Practices Question

Question

Why is it considered a best practice to initialize all variables in C before using them? Provide an example.

Best Practices Question

Question

Why is it considered a best practice to initialize all variables in C before using them? Provide an example.

Answer

Initializing variables prevents undefined behavior due to usage of uninitialized memory.
For example, without initialization, int x; printf("% d", x); might print any random value. Initializing with int x = 0; ensures 'x' has a defined, predictable value.

Common Errors and Troubleshooting in C

- Syntax errors: Issues with the code's structure, often caught by the compiler.
- Runtime errors: Errors that occur during the execution of the program, such as division by zero.
- Logic errors: Flaws in the program's logic leading to incorrect output despite successful compilation.
- Debugging tips: Use of debugger tools, reading compiler warnings, and code reviews.

```
#include <stdio.h>
int main() {
int a = 10, b = 0;
int result;

// Runtime error example: division by zero
if (b != 0) {
    result = a / b;
} else {
    printf("Error: Division by zero\n");
}

// Logic error example: incorrect condition
if (a = 10) { // Should be '==', not '='
    printf("a is 10\n");
}

return 0;
}
```

C Programs: Re-usability

Small reusable programs demonstrate language features:

- Math and prime checks
- String operations
- Sorting algorithms
- File handling

```
#include <stdio.h>

int factorial(int num) {
int f = 1;
for(int i=1; i<=num; i++) {
f *= i;
}
return f;
}

int main() {
int num;
printf("Enter a number: ");
scanf("%d", &num);
int result = factorial(num);
printf("The factorial of %d is %d", num, result);
return 0;
}
```

Modular programs effectively showcase constructs and libraries.

C Programs : Modularization

Header files modularize functionality:

```
int add(int, int);
int factorial(int);
```

```
#include "math.h"
int main() {
int s = add(5, 10);
int f = factorial(5);
}
```

Header files and libraries enable code reuse across source files.

Function Call Challenge

```
#include <stdio.h>
void update(int x) {
    x = x + 10;
}
int main() {
    int a = 5;
    update(a);
    printf("a: %d\n", a);
    return 0;
}
```

Function Call Challenge

```c
#include <stdio.h>
void update(int x) {
    x = x + 10;
}
int main() {
    int a = 5;
    update(a);
    printf("a: %d\n", a);
    return 0;
}
```

Question

What is the value of 'a' after the function call and why?

Function Call Challenge

```c
#include <stdio.h>
void update(int x) {
    x = x + 10;
}
int main() {
    int a = 5;
    update(a);
    printf("a: %d\n", a);
    return 0;
}
```

Question

What is the value of 'a' after the function call and why?

Answer

The value of 'a' remains 5 after the function call. In C, function parameters are passed by value. Therefore, the function 'update' modifies a copy of 'a', not 'a' itself.

Function Call Challenge: Update with Pointer

```
#include <stdio.h>
void update(int *x) {
    *x = *x + 10;
}
int main() {
    int a = 5;
    update(&a);
    printf("a: %d\n", a);
    return 0;
}
```

Function Call Challenge: Update with Pointer

```c
#include <stdio.h>
void update(int *x) {
    *x = *x + 10;
}
int main() {
    int a = 5;
    update(&a);
    printf("a: %d\n", a);
    return 0;
}
```

Question

What is the value of 'a' after the function call now?

Function Call Challenge: Update with Pointer

```c
#include <stdio.h>
void update(int *x) {
    *x = *x + 10;
}
int main() {
    int a = 5;
    update(&a);
    printf("a: %d\n", a);
    return 0;
}
```

Question

What is the value of 'a' after the function call now?

Answer

Now the value of 'a' is 15 after the function call. The function 'update' uses a pointer to directly modify the value of 'a'.

File I/O

File handling allows data persistence across executions.

- Opening and closing files
- Reading and writing data
- Text vs Binary modes

```
#include <stdio.h>
int main() {
FILE *fptr;
fptr = fopen("file.txt","w");
fprintf(fptr,"Hello World!");
fclose(fptr);
fptr = fopen("file.txt","r");
char buffer[100];
fscanf(fptr,"%s", buffer);
printf("Data: %s", buffer);
return 0;
}
```

Basic File Operations in C

```
#include <stdio.h>
int main() {
FILE *fp;
fp = fopen("example.txt", "w");
if (fp == NULL) {
    perror("Error opening file");
    return -1;
}
fprintf(fp, "Hello, world!\n");
fclose(fp);
return 0;
}
```

- **Opening a File**: Use ‘fopen()‘ to open a file. Modes include "r", "w", "a".
- **Reading from a File**: Use ‘fscanf()‘ or ‘fgets()‘ for reading.
- **Writing to a File**: Use ‘fprintf()‘ or ‘fputs()‘ for writing.
- **Closing a File**: Always close a file using ‘fclose()‘.
- **Error Handling**: Check the return value of file operations for errors.

Debugging Challenge: File I/O

```
#include <stdio.h>
int main() {
    FILE *fp;
    fp = fopen("example.txt", "w");
    fprintf(fp, "%d %d %d", 5, 10, 15);
    fclose(fp);
    return 0;
}
```

Debugging Challenge: File I/O

```c
#include <stdio.h>
int main() {
    FILE *fp;
    fp = fopen("example.txt", "w");
    fprintf(fp, "%d %d %d", 5, 10, 15);
    fclose(fp);
    return 0;
}
```

Question

What is the content of "example.txt" after executing this program?

Debugging Challenge: File I/O

```
#include <stdio.h>
int main() {
    FILE *fp;
    fp = fopen("example.txt", "w");
    fprintf(fp, "%d %d %d", 5, 10, 15);
    fclose(fp);
    return 0;
}
```

Question

What is the content of "example.txt" after executing this program?

Answer

The content of "example.txt" will be "5 10 15". The program writes these three integers to the file separated by spaces using fprintf.

Preprocessors

Preprocessors handle meta programming logic.

- Directives evaluated before compilation
- #include, #define, #ifdef etc.
- Macro expansions
- File inclusion
- Conditional compilation

```
#include <stdio.h>
#define PRINT printf
#define SQUARE(x) x*x

int main() {
PRINT("In main function\n");
int num=5;
PRINT("Square of %d is %d", num, SQUARE(num));
eturn 0;
}
```

Preprocessors : File Inclusion

Preprocessors insert contents of file during compilation.

- include $< file >$ - Search built-in directories
- include "file" - Search current directory
- include $< file.h >$ - Header files convention

```
#include <stdio.h>
int main() {
printf("Standard library");
#include "userdefs.h"
printcustom();
return 0;
}
```

This demonstrates:

- Inclusion of stdio.h from built-in folders
- Inclusion of userdefs.h from current folder
- Calling custom function after inclusion

Preprocessors: Macro Arguments

Macros can make code more readable and maintainable.

Define macros accepting parameters:

define MACRO (arg1, arg2)
(arg1 + arg2)

```c
#include <stdio.h>

#define MIN(x,y) ((x) < (y) ? (x) : (y))

int main() {
int a = 10, b = 5;
int small = MIN(a, b); //Macro invocation
printf("Minimum of %d & %d is: %d", a, b, small);
return 0;
}
```

Understanding Macros

```
#include <stdio.h>
#define SQUARE(x) (x*x)
int main() {
    int a = 4, b = 2;
    int result = SQUARE(a + b);
    printf("Result: %d\n", result);
    return 0;
}
```

Understanding Macros

```
#include <stdio.h>
#define SQUARE(x) (x*x)
int main() {
    int a = 4, b = 2;
    int result = SQUARE(a + b);
    printf("Result: %d\n", result);
    return 0;
}
```

Question

What is the output of this program and why?

Understanding Macros

```c
#include <stdio.h>
#define SQUARE(x) (x*x)
int main() {
    int a = 4, b = 2;
    int result = SQUARE(a + b);
    printf("Result: %d\n", result);
    return 0;
}
```

Question

What is the output of this program and why?

Answer

The output is 14, not 36. The macro expands to (a + b * a + b), which is equivalent to (4 + 2 * 4 + 2) due to macro substitution leading to unexpected results without proper parentheses.

Preprocessors: $\#if - \#else - \#endif$

Conditionally include code sections during compilation.

{#if CONDITION
//code A
else
//code B
#endif }

This shows:

- DEBUG macro definition as flag
- Code inside if block prints when defined
- Alternate code in else prints when not defined

```
#define DEBUG

int main() {

#if DEBUG
printf("In debug mode\n");
#else
printf("Debug disabled\n");

#endif
// Rest of code
}
#endif
```

Predefined Macros

Commonly available predefined macros are:

- __FILE__ - Current filename
- __LINE__ - Current line number
- __DATE__ - Compilation date
- __TIME__ - Compilation time

```
#include <stdio.h>
int main() {
printf("Compiled at line %d of file %s \n", LINE, FILE );
printf("On date: %s time: %s\n", DATE, TIME);
return 0;
}
```

FILE and **LINE** for displaying context

DATE and **TIME** for compilation timestamps

Other interesting predefined macros are:

STDC - Conformance level indicator

FUNCTION- Prints function name

Sequential Statements

- Modular program structure
  - Functions, headers, libraries
- Sequence of statements execute top to bottom
  - Code blocks, conditionals
- Input, process, output, style
- Program structure best practices
- Statements execute sequentially
- Overall program flow and stages
- Systematic sequence of steps solve problem.

```c
#include <stdio.h>
// Function declaration
void readInput();
int main() {
// Initialize
int num;

// Read input
readInput();

// Process
num = num * 2;

// Display output
printf("%d", num);

return 0;
}

// Define function
void readInput() {
scanf("%d", &num);
}
```

Modular Programs

Functions follow calling conventions for parameter passing.

Functions encapsulate logic:

- Declaration in header file
- Definition with logic
- Call from multiple places

This demonstrates:

- Function declaration in header
- Calling declared function from main()
- Definition separate from usage

```
// In header.h
int add(int, int);

// In main.c
#include "header.h"

int main() {
int sum = add(5, 10);
printf("Sum=%d",sum);
}

// In add.c
int add(int a, int b) {
return a+b;
}
```

Version Control Basics

- Version control systems track changes to files over time.
- Git is a distributed version control system widely used in software development.
- Key operations: 'git init', 'git add', 'git commit', 'git push'.
- Benefits: Collaboration, backup, history, and branch management.

```
// Command line snippets that show basic Git commands
// Example: Initializing a new Git repository
git init

// Adding a file to the staging area
git add filename.c

// Committing changes with a message
git commit -m "Initial commit"

// Pushing changes to a remote repository
git push origin main
```