XCELERIUM

Digital Design and Verification Training

Deep Dive into C and Compiler

Agenda

- Advanced C Concepts - Pointers, Memory, Structures, and File I/O

- Compilation Process

C Language

Digital Design and
Verification Training

Advanced C Concepts - Pointers

- Definition: A pointer is a variable that stores the memory address of another variable
- Syntax:

```
int x = 10;
int *ptr = &x; // ptr holds the address of x
```

- Declaring and initializing pointers

```
int *p; // Declare a pointer to an int
char *str; // Declare a pointer to a char
double *dptr = NULL; // Initialize a pointer to null
```

Advanced C Concepts - Pointers

- Accessing the value pointed to by a pointer
- Example:

```
int x = 42;
int *ptr = &x;
printf("Value of x: %d\n", x); // 42
printf("Value pointed to by ptr: %d\n", *ptr); // 42
*ptr = 100; // Change the value of x through the pointer
printf("New value of x: %d\n", x); // 100
```

Advanced C Concepts - Pointers

- Incrementing and decrementing pointers
- Example:

```
int arr[] = {10, 20, 30, 40, 50};
int *p = arr; // p points to the first element of arr
printf("%d ", *p); // 10
printf("%d ", *(p+1)); // 20
printf("%d ", *(p+2)); // 30
p++; // Move to the next element
printf("%d ", *p); // 20
```

Advanced C Concepts - Pointers

- Relationship between arrays and pointers
- Array name as a pointer to its first element

```c
int numbers[] = {1, 2, 3, 4, 5};
int *ptr = numbers; // ptr points to the first element
for (int i = 0; i < 5; i++)
{ printf("%d ", *(ptr + i)); // Access elements using pointer arithmetic
}
printf("\n"); // Equivalent array notation
for (int i = 0; i < 5; i++)
{ printf("%d ", numbers[i]); }
```

Advanced C Concepts - Pointers

Strings in C are arrays of characters terminated by '\0'.

They can be manipulated using pointers.

```
char str[] = "Hello";
char *p = str;


printf("%s\n", str);   // using array
printf("%s\n", p);     // using pointer
```

Advanced C Concepts - Pointers

Pointers can iterate over characters until '\0' is reached.

```
char str[] = "Pointers";
char *p = str;

while (*p != '\0') {
    printf("%c ", *p);
    p++;
}
// Output: P o i n t e r s
```

Advanced C Concepts - Pointers

Comparing Array Index vs. Pointer Notation

```
char str[] = "World";
for (int i = 0; str[i] != '\0'; i++)
    printf("%c", str[i]);   // array indexing


char *p = str;
while (*p)
    printf("%c", *p++);     // pointer notation
```

Advanced C Concepts - Pointers

Practical Demo

```
void magic_func(char *s) {
    char *end = s + strlen(s) - 1;
    while (s < end) {
        char temp = *s;
        *s++ = *end;
        *end-- = temp;
    }
}
```

Advanced C Concepts - Pointers

- Accessing elements of 2D arrays using pointers
- Example

```
int matrix[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
int (*p)[4] = matrix; // Pointer to an array of 4 integers
printf("%d ", *(*(p+1)+2)); // Accesses matrix[1][2], prints 7
printf("%d ", p[1][2]); // Equivalent array notation
```

Function Pointers - Basics

- Definition: Pointers that point to functions
- Syntax:

```
return_type (*pointer_name)(parameter_types);
```

- Example:

```
int add(int a, int b) { return a + b; }

int (*func_ptr)(int, int);

func_ptr = add;

printf("Sum: %d\n", func_ptr(5, 3)); // Calls add(5, 3), prints 8
```

Function Pointers – Example

```
int compare_ints(const void* a, const void* b)
{ return (*(int*)a - *(int*)b); }


int main() {
int numbers[] = {42, 13, 7, 87, 35};
int n = sizeof(numbers) / sizeof(numbers[0]);
qsort(numbers, n, sizeof(int), compare_ints);
for (int i = 0; i < n; i++)
{ printf("%d ", numbers[i]); }
printf("\n"); return 0; }
```

Common Pitfalls with Pointers

- Uninitialized pointers
- Dangling pointers
- Memory leaks

Best Practices for Using Pointers

- Always initialize pointers
- Check for NULL before dereferencing
- Use const when appropriate
- Be cautious with void pointers
- Free dynamically allocated memory

Dynamic Memory Allocation

- Why use dynamic memory allocation?
  - Flexible memory usage
  - Create data structures of variable size
- Functions: malloc(), calloc(), realloc(), free()

Dynamic Memory Allocation

- malloc()
  - Allocates specified number of bytes
  - Syntax: void *malloc(size_t size);
- Example:

```
int *ptr = (int *)malloc(5 * sizeof(int));
if (ptr == NULL)
{ printf("Memory allocation failed\n");
return 1; }
// Use the allocated memory
free(ptr); // Don't forget to free when done
```

Dynamic Memory Allocation

- calloc()
  - Allocates memory and initializes it to zero
  - Syntax: void *calloc(size_t num, size_t size);
- Example:

```
int *ptr = (int *)calloc(5, sizeof(int));
if (ptr == NULL)
{ printf("Memory allocation failed\n");
return 1; }
// Use the allocated memory
free(ptr);
```

Dynamic Memory Allocation

- realloc()
  - Resizes previously allocated memory
  - Syntax: void *realloc(void *ptr, size_t new_size);
- Example:

```
int *ptr = (int *)malloc(5 * sizeof(int));
// ... use the memory ...
ptr = (int *)realloc(ptr, 10 * sizeof(int));
if (ptr == NULL)
{ printf("Memory reallocation failed\n");
return 1; }
// Use the reallocated memory
free(ptr);
```

Structures

- Group related data items
- Syntax:

```
struct Person
{ char name[50];
int age;
float height; };

struct Person p1 = {"Alice", 30, 165.5};
printf("Name: %s, Age: %d, Height: %.1f\n", p1.name, p1.age, p1.height);
```

Structures

- Pointers to Structures
- Syntax:

```
struct Student { int roll; char name[20]; float gpa; };
struct Student s1 = {1, "Ali", 3.7};
```

```
struct Student *p = &s1;
printf("%s", p->name);
```

Structures

- Nested Structures
  - Structures within structures
- Example:

```
struct Address {
char street[50];
char city[30];
};
struct Employee {
char name[50];
struct Address addr;
};
struct Employee emp = {"Bob", {"123 Main St", "New York"}};
printf("Name: %s, Street: %s, City: %s\n", emp.name, emp.addr.street,
emp.addr.city);
```

Unions

- Shares memory among different data types
- Only one member can hold a value at a time
- Example:

```
union Data {
int i;
float f;
char str[20];
};

union Data data;
data.i = 10;
printf("Integer: %d\n", data.i);
data.f = 3.14;
printf("Float: %f\n", data.f);
```

Preprocessor Directives

Handled before compilation.

Categories: File inclusion, Macros, Conditional compilation

```
#define PI 3.14159
#define SQUARE(x) ((x)*(x))
```

Preprocessor Directives

Also used for Conditional Compilation

```
#ifdef DEBUG
  printf("Debug Mode");
#endif
```

File I/O - Basics

- Key functions: fopen(), fclose(), fread(), fwrite(), fprintf(), fscanf()
- File open modes: "r", "w", "a", "r+", "w+", "a+"
- Example:

```
FILE *file = fopen("example.txt", "w");
if (file == NULL) {
printf("Error opening file\n");
return 1; }

fprintf(file, "Hello, File I/O!\n");
fclose(file);
```

File I/O - Basics

- Reading from a file:

```
FILE *file = fopen("example.txt", "r");
char buffer[100];
while (fgets(buffer, sizeof(buffer), file) != NULL)
{
printf("%s", buffer);
}

fclose(file);
```

File I/O - Basics

- Writing to a file:

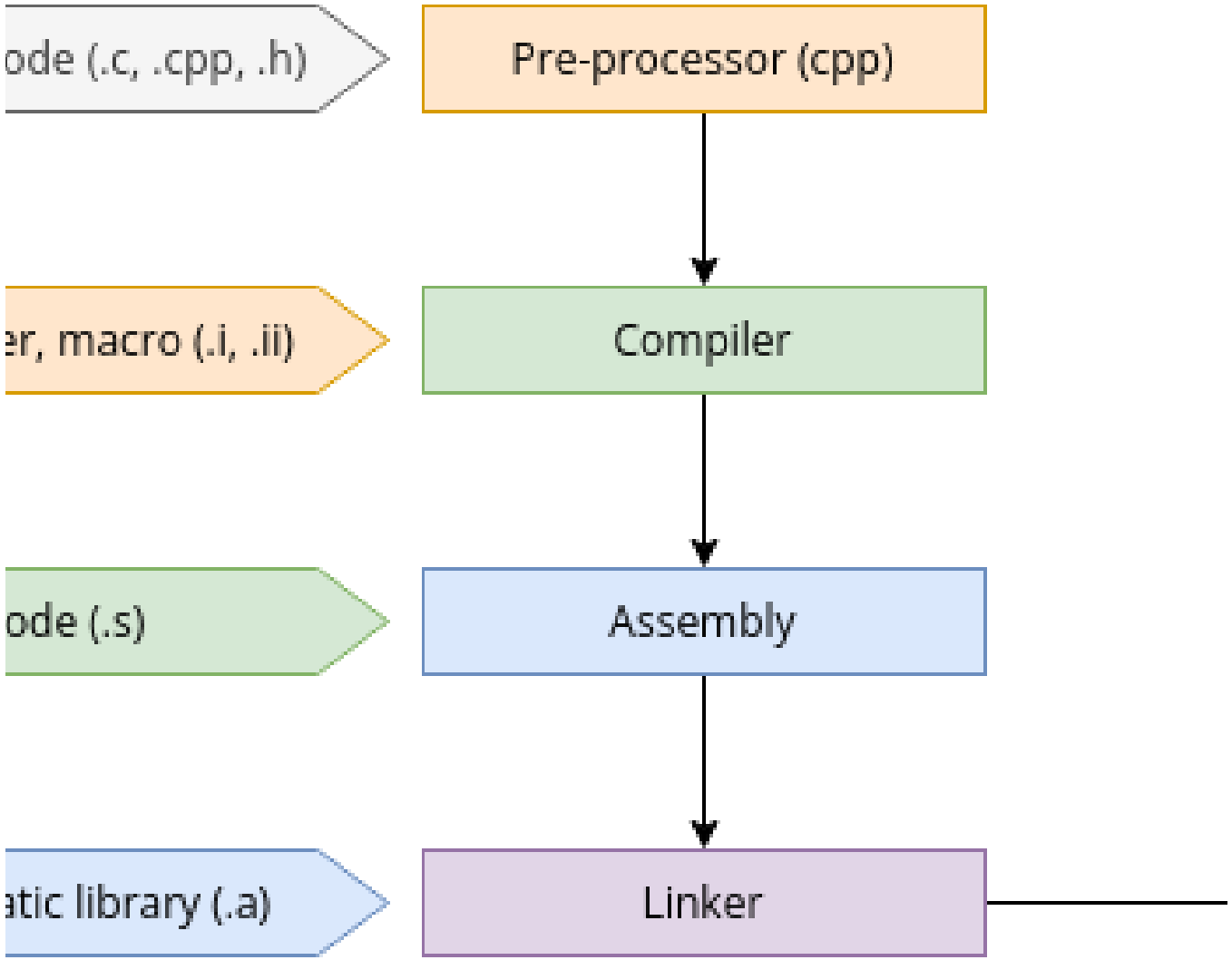
```
FILE *file = fopen("output.txt", "w");
for (int i = 1; i <= 5; i++) {
fprintf(file, "Line %d\n", i);
}

fclose(file);
```

File I/O - Basics

- File I/O Best Practices
  - Always check if file operations succeed
  - Close files when done
  - Use appropriate file modes
  - Handle errors gracefully
  - Consider using fflush() for immediate writing

Compilation of C Program

Digital Design and Verification Training

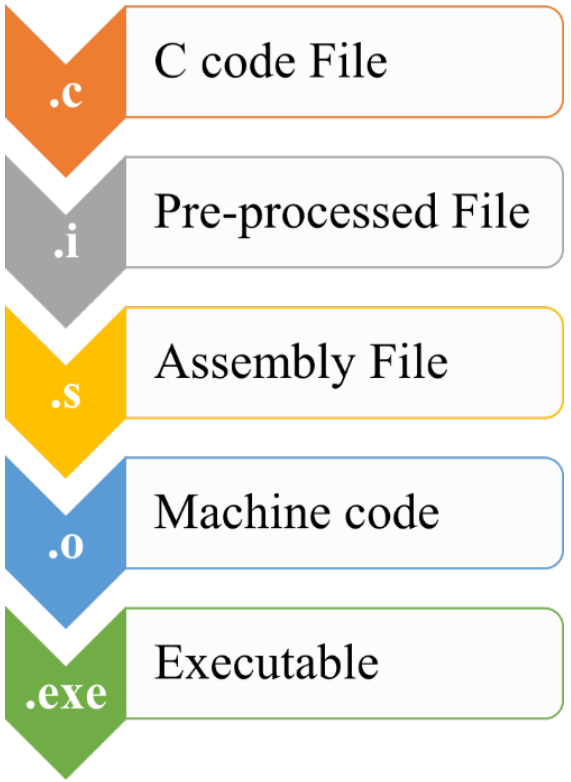


Assembler Vs Compiler

- If the user program is written in assembly language, then we use an assembler to convert it to an object code, which is also called machine code.
- If the user program is written in a high-level language, we will take the example of a C program, then we use a compiler to convert it to the machine code.

Compilation Process

- Preprocessing
- Compilation
- Assembly
- Linking



Compilation Process

- Preprocessing
  - First stage of compilation
  - Handled by the preprocessor
  - Processes directives like #include, #define, #ifdef, etc.
- Preprocessing - What It Does
  - Includes header files
  - Expands macros
  - Removes comments
  - Handles conditional compilation directives

Preprocessing - Example

- Original code:

```
#include <stdio.h>
#define MAX 100

int main() {
printf("Max is %d\n", MAX);
return 0;
}
```

- After preprocessing:

```
// Contents of stdio.h inserted
here

Int main() {
printf("Max is %d\n", 100);
return 0;
}
```

Preprocessing - GCC Command

```
gcc -E source.c -o source.i
```

Compilation

- Second stage of the process
- Converts preprocessed code to assembly language
- Performs syntax checking and optimization
- Parses the preprocessed code
- Performs semantic analysis
- Generates assembly code

Compilation - Example

- Original code:

```
int add(int a, int b)
{
return a + b;
}
```

- After preprocessing:

```
add:
        pushl %ebp
       movl %esp, %ebp
       movl 8(%ebp), %edx
       movl 12(%ebp), %eax
      addl %edx, %eax
      popl %ebp ret
```

Preprocessing - GCC Command

```
gcc -S source.i -o source.s
```

Assembly

- Third stage of compilation
- Converts assembly code to machine code
- Produces object files

Assembly - Object File Content

- Machine code
- Symbol table
- Relocation information
- Debugging information (if compiled with -g)

Linking

- Final stage of compilation
- Resolves external references
- Combines object files into a single executable
- Resolves symbols (function calls, variable references)
- Assigns final addresses to functions and variables
- Links with standard libraries

Linker Types

- Static Linking
  - Libraries are embedded in the executable
  - Larger file size, but self-contained
- Dynamic Linking
  - Libraries are linked at runtime
  - Smaller executable, but requires libraries to be present

Compilation steps using gcc

- Pre-processing: gcc -E example.c -o example.i
  - Output pre-processed file example.i can be viewed using any text editor (Notepad++)
- Compilation: arm-none-eabi-gcc -S example.i
  - Output assembly file can be viewed using any text editor (Notepad++)
- Linking: gcc -c example.s
  - In command prompt using: objdump -D example.o
- Creating Executable: gcc -o out example.o
  - In command prompt: out