Learners Paradigm

Vaibhav Upadhyay

2026-12-05

Table of contents

Preface 3

1 Introduction 4

2 Summary 5

References 6

I init 7

3 Basics 8

4 C Basics 9

- 4.1 The Structure of a C Program 9
- 4.2 Variables and Data Types 12
  - 4.2.1 Basic Data Types 12
  - 4.2.2 Variable Declaration and Initialization 13
  - 4.2.3 Type Modifiers 13
  - 4.2.4 Constants 13
- 4.3 Type Conversion 14
- 4.4 Variable Scope 14
- 4.5 Memory Size and Range 15
- 4.6 Exercise: Working with Variables 15
- 4.7 Operators in C 16
- 4.8 Arithmetic Operators 16
  - 4.8.1 Example: 16
- 4.9 Assignment Operators 17
  - 4.9.1 Example: 18
- 4.10 Comparison Operators 18
  - 4.10.1 Example: 19
- 4.11 Logical Operators 19
  - 4.11.1 Example: 19
- 4.12 Bitwise Operators 20
  - 4.12.1 Example: 20

4.13 Miscellaneous Operators 20
4.13.1 Example: 21
4.14 Operator Precedence 21
4.15 Exercise: Temperature Converter 22
4.16 Control Structures 22
4.17 Conditional Statements 23
4.17.1 if Statement 23
4.17.2 if-else Statement 23
4.17.3 if-else if-else Statement 24
4.17.4 Nested if Statements 25
4.17.5 Conditional (Ternary) Operator 25
4.17.6 switch Statement 26
4.18 Loops 27
4.18.1 while Loop 27
4.18.2 do-while Loop 28
4.18.3 for Loop 28
4.18.4 Nested Loops 29
4.19 Jump Statements 30
4.19.1 break Statement 30
4.20 Functions 30

5 Pointers **32**

6 Pointers **33**
6.1 Introduction: 33
6.1.1 Pointers and Addresses 33

II 2_storing_data_elements **36**

7 Array **37**

8 Numbers etc. **38**
8.1 Basic Matrix multiplication 39
8.1.1 What is rendering? 39

9 Matrix **43**
9.0.1 Unified Memory 43
9.0.2 Diving into GPU programming. 43
9.0.3 What exactly is a .metal file? 48

III Scratching the Surface 52

10 Understanding Metal Pipeline 53

10.0.1 What exactly is a .metal file? 53

IV Design Elements 55

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

1 Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

2 Summary

In summary, this book has no content whatsoever.

References

Knuth, Donald E. 1984. "Literate Programming." *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.

Part I

init

3 Basics

4 C Basics

4.1 The Structure of a C Program

Let's start with a simple "Hello, World!" program:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

Let's break this down:

`#include <stdio.h>` - This is a preprocessor directive that includes the standard input/output library.

`int main()` - The main function, which is the entry point of every C program.

`printf("Hello, World!\n");` - A function call that prints text to the console.

`return 0;` - Returns 0 to the operating system, indicating successful execution.

`{ }` - Curly braces define code blocks.

Comments in C:

C supports two types of comments:

```c
// This is a single-line comment

/* This is a multi-line comment
   that spans across several lines */
```

Comments are ignored by the compiler and are used to explain your code.

Preprocessor Directives:

Preprocessor directives start with # and are processed before compilation:

```c
#include <stdio.h>  // Include a header file
#define PI 3.14159  // Define a constant
```

Common preprocessor directives:

`#include`- Includes a header file

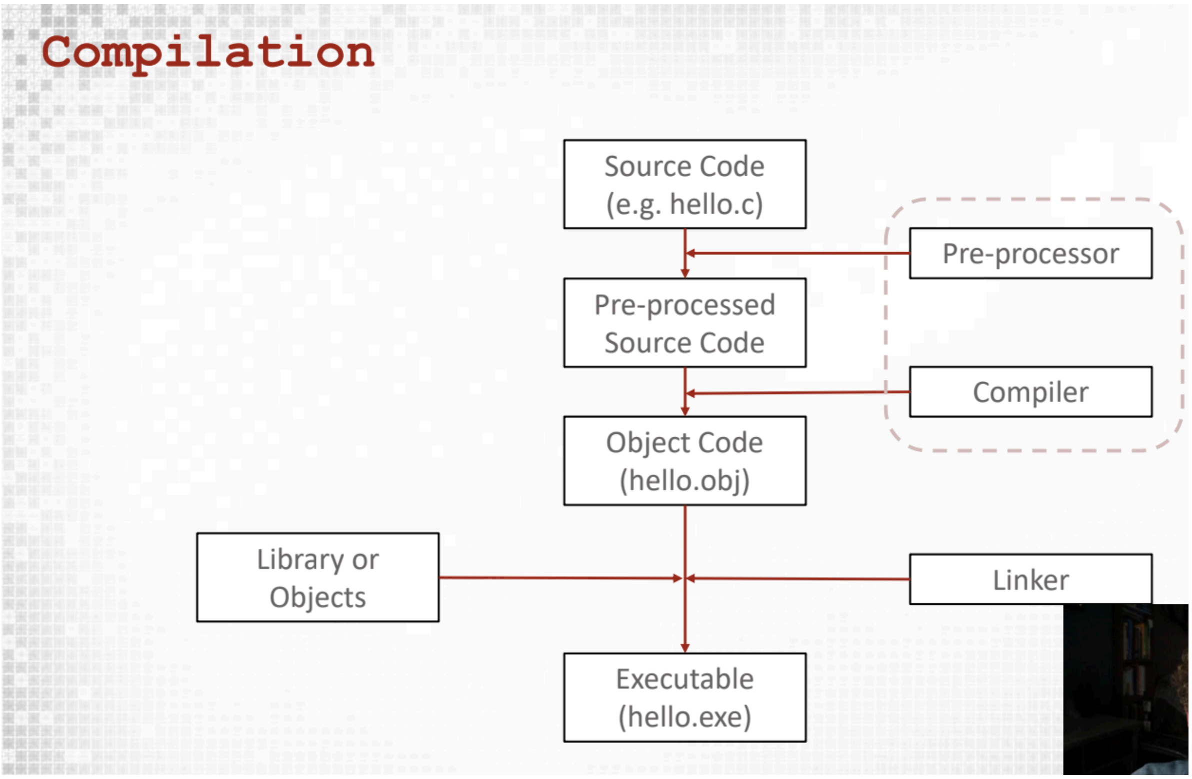
`#define`- Defines a constant or macro

‘`#ifdef, #ifndef, #endif` - Conditional compilation

Compilation Process:

The C compilation process involves four stages:

1. Preprocessing: Processes directives like #include and #define
2. Compilation: Converts source code to assembly code
3. Assembly: Converts assembly code to object code
4. Linking: Combines object files and libraries into an executable



Compilation

Source Code (e.g. hello.c)

Pre-processor

Pre-processed Source Code

Compiler

Object Code (hello.obj)

Library or Objects

Linker

Executable (hello.exe)

Figure 4.1: src : 3

Let’s revisit our “Hello World” program:

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

`printf()` is a function from the standard library that outputs formatted text

`\n` is an escape sequence representing a newline character

The semicolon ; ends each statement

Exercise: Writing our First C Program

Steps:

Create a file named greeting.c

Write a program that prints “Hello, [your name]!”

Compile and run the program

Solution:

```c
#include <stdio.h>

int main() {
    printf("Hello, [your name]!\n");
    return 0;
}
```

Compile and run:

```
 #macOS
gcc greeting.c -o greeting
./greeting
```

4.2 Variables and Data Types

4.2.1 Basic Data Types

C provides several fundamental data types:

Type	Description	Format Specifier	Example
`int`	Integer	`%d` or `%i`	`int age = 25;`
`float`	Single-precision floating-point	`%f`	`float price = 19.99f;`
`double`	Double-precision floating-point	`%lf`	`double pi = 3.14159265359;`
`char`	Single character	`%c`	`char grade = 'A';`

4.2.2 Variable Declaration and Initialization

Variables must be declared before use:

```c
// Declaration only
int count;

// Declaration with initialization
int count = 0;

// Multiple variables of the same type
int x, y, z;
int a = 1, b = 2, c = 3;
```

4.2.3 Type Modifiers

Type modifiers alter the range or behavior of basic types:

Modifier	Description	Example
`short`	Reduced-size integer	`short int num = 123;`
`long`	Extended-size integer	`long int population = 7800000000L;`
`unsigned`	Non-negative values only	`unsigned int count = 50;`
`signed`	Positive and negative values (default)	`signed int temperature = -10;`

4.2.4 Constants

Constants are values that cannot be modified after declaration:

```c
// Using const qualifier
const float PI = 3.14159;

// Using #define preprocessor directive
#define MAX_SIZE 100
```

4.3 Type Conversion

C supports two types of type conversion:

1. **Implicit conversion** (automatic):

   ```
   int x = 10;
   double y = x;  // int implicitly converted to double
   ```

2. **Explicit conversion** (casting):

   ```
   double x = 10.5;
   int y = (int)x;  // x is explicitly cast to int, y = 10
   ```

4.4 Variable Scope

The scope of a variable determines where it can be accessed:

1. **Local variables** - Declared inside a function and accessible only within that function
2. **Global variables** - Declared outside any function and accessible throughout the program
3. **Block-level variables** - Declared inside a block and accessible only within that block

```
#include <stdio.h>

// Global variable
int globalVar = 100;

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

    {
        // Block-level variable
        int blockVar = 25;
        printf("Block variable: %d\n", blockVar);
    }

    // blockVar is not accessible here

    printf("Global variable: %d\n", globalVar);
    printf("Local variable: %d\n", localVar);
```

```
    return 0;
}
```

4.5 Memory Size and Range

The `sizeof()` operator returns the memory size (in bytes) of a data type or variable:

```
#include <stdio.h>
#include <limits.h>  // For integer limits
#include <float.h>   // For floating-point limits

int main() {
    printf("Size of int: %zu bytes\n", sizeof(int));
    printf("Range of int: %d to %d\n", INT_MIN, INT_MAX);

    printf("Size of float: %zu bytes\n", sizeof(float));
    printf("Range of float: %e to %e\n", FLT_MIN, FLT_MAX);

    return 0;
}
```

Output:

```
vaibhav@Mac src % gcc memory_size.c -o out
vaibhav@Mac src % ./out
Size of int: 4 bytes
Range of int: -2147483648 to 2147483647
Size of float: 4 bytes
Range of float: 1.175494e-38 to 3.402823e+38
```

4.6 Exercise: Working with Variables

Write a program that:

1. Declares variables for a person's name, age, and height
2. Initializes them with values
3. Prints them in a formatted way

Solution:

```
#include <stdio.h>

int main() {
    // Declare and initialize variables
    char name[50] = "John Doe";
    int age = 30;
    float height = 175.5;  // in cm

    // Print the values
    printf("Name: %s\n", name);
    printf("Age: %d years\n", age);
    printf("Height: %.1f cm\n", height);

    return 0;
}
```

4.7 Operators in C

4.8 Arithmetic Operators

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (remainder)	a % b
++	Increment	a++ or ++a
--	Decrement	a-- or --a

4.8.1 Example:

```
#include <stdio.h>

int main() {
    int a = 10, b = 3;
    printf("a + b = %d\n", a + b);  // 13
    printf("a - b = %d\n", a - b);  // 7
    printf("a * b = %d\n", a * b);  // 30
    printf("a / b = %d\n", a / b);  // 3 (integer division)
    printf("a %% b = %d\n", a % b); // 1 (remainder)

    int c = a++;  // c = 10, then a becomes 11
    int d = ++b;  // b becomes 4, then d = 4

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

    return 0;
}

/*
https://www.programiz.com/c-programming/online-compiler/
Output:
a + b = 13
a - b = 7
a * b = 30
a / b = 3
a % b = 1
After a++: a = 11, c = 10
After ++b: b = 4, d = 4


=== Code Execution Successful ===
*/
```

4.9 Assignment Operators

Operator	Description	Equivalent to
`=`	Simple assignment	`a = b`
`+=`	Add and assign	`a = a + b`
`-=`	Subtract and assign	`a = a - b`
`*=`	Multiply and assign	`a = a * b`
`/=`	Divide and assign	`a = a / b`
`%=`	Modulus and assign	`a = a % b`

4.9.1 Example:

```c
#include <stdio.h>

int main() {
    int x = 10;

    x += 5;  // x = x + 5
    printf("After x += 5: x = %d\n", x);  // 15

    x -= 3;  // x = x - 3
    printf("After x -= 3: x = %d\n", x);  // 12

    x *= 2;  // x = x * 2
    printf("After x *= 2: x = %d\n", x);  // 24

    x /= 4;  // x = x / 4
    printf("After x /= 4: x = %d\n", x);  // 6

    x %= 4;  // x = x % 4
    printf("After x %%= 4: x = %d\n", x);  // 2

    return 0;
}
```

4.10 Comparison Operators

Operator	Description	Example
`==`	Equal to	`a == b`
`!=`	Not equal to	`a != b`
`>`	Greater than	`a > b`
`<`	Less than	`a < b`
`>=`	Greater than or equal to	`a >= b`
`<=`	Less than or equal to	`a <= b`

4.10.1 Example:

```
#include <stdio.h>

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

    printf("a == b: %d\n", a == b);  // 0 (false)
    printf("a != b: %d\n", a != b);  // 1 (true)
    printf("a > b: %d\n", a > b);    // 0 (false)
    printf("a < b: %d\n", a < b);    // 1 (true)
    printf("a >= b: %d\n", a >= b);  // 0 (false)
    printf("a <= b: %d\n", a <= b);  // 1 (true)

    return 0;
}
```

4.11 Logical Operators

Operator	Description	Example
&&	Logical AND	a && b
|	Logical OR	a | b
!	Logical NOT	!a

4.11.1 Example:

```
#include <stdio.h>

int main() {
    int a = 1, b = 0;  // 1 is true, 0 is false in C

    printf("a && b: %d\n", a && b);    // 0 (false)
    printf("a || b: %d\n", a || b);    // 1 (true)
    printf("!a: %d\n", !a);            // 0 (false)
    printf("!b: %d\n", !b);            // 1 (true)
    return 0;
}
```

4.12 Bitwise Operators

Operator	Description	Example
&	Bitwise AND	a & b
|	Bitwise OR	a | b
^	Bitwise XOR	a ^ b
~	Bitwise NOT	~a
<<	Left shift	a << n
>>	Right shift	a >> n

4.12.1 Example:

```
#include <stdio.h>

int main() {
    unsigned int a = 60;  // 00111100 in binary
    unsigned int b = 13;  // 00001101 in binary

    printf("a & b = %u\n", a & b);    // 12 (00001100)
    printf("a | b = %u\n", a | b);    // 61 (00111101)
    printf("a ^ b = %u\n", a ^ b);    // 49 (00110001)
    printf("~a = %u\n", ~a);          // 4294967235 (11...00011)
    printf("a << 2 = %u\n", a << 2);  // 240 (11110000)
    printf("a >> 2 = %u\n", a >> 2);  // 15 (00001111)

    return 0;
}
```

4.13 Miscellaneous Operators

Operator	Description	Example
sizeof	Size of variable/type	sizeof(a)

Operator	Description	Example
`&`	Address of variable	`&a`
`*`	Pointer to variable	`*a`
`?:`	Conditional operator	`a ? b : c`

4.13.1 Example:

```c
#include <stdio.h>

int main() {
    int a = 10;
    int *ptr = &a;  // ptr holds the address of a

    printf("Size of int: %zu bytes\n", sizeof(int));
    printf("Address of a: %p\n", &a);
    printf("Value at address stored in ptr: %d\n", *ptr);

    // Conditional operator
    int max = (a > 20) ? a : 20;
    printf("Max of a and 20: %d\n", max);

    return 0;
}
```

4.14 Operator Precedence

Operators have different precedence levels that determine the order of evaluation:

1. Postfix `++ -- () [] . ->`
2. Prefix `++ -- + - ! ~ (type) * & sizeof`
3. Multiplicative `* / %`
4. Additive `+ -`
5. Shift `<< >>`
6. Relational `< <= > >=`
7. Equality `== !=`
8. Bitwise AND `&`
9. Bitwise XOR `^`
10. Bitwise OR `|`
11. Logical AND `&&`

12. Logical OR `||`
13. Conditional `?:`
14. Assignment `= += -= *= /= %=` etc.

When in doubt, use parentheses to explicitly specify the order of operations.

4.15 Exercise: Temperature Converter

Write a program that converts temperature from Fahrenheit to Celsius and vice versa.

Solution:

```c
#include <stdio.h>

int main() {
    float fahrenheit, celsius;

    // Fahrenheit to Celsius
    printf("Enter temperature in Fahrenheit: ");
    scanf("%f", &fahrenheit);

    celsius = (fahrenheit - 32) * 5 / 9;
    printf("%.2f°F = %.2f°C\n", fahrenheit, celsius);

    // Celsius to Fahrenheit
    printf("\nEnter temperature in Celsius: ");
    scanf("%f", &celsius);

    fahrenheit = (celsius * 9 / 5) + 32;
    printf("%.2f°C = %.2f°F\n", celsius, fahrenheit);

    return 0;
}
```

4.16 Control Structures

Control structures allow you to control the flow of execution in your program based on conditions or to repeat a block of code.

4.17 Conditional Statements

4.17.1 if Statement

The `if` statement executes a block of code if a specified condition is true:

```c
if (condition) {
    // Code to execute if condition is true
}
```

Example:

```c
#include <stdio.h>

int main() {
    int age = 20;

    if (age >= 18) {
        printf("You are an adult.\n");
    }

    return 0;
}
```

4.17.2 if-else Statement

The `if-else` statement executes one block if the condition is true and another if it's false:

```c
if (condition) {
    // Code to execute if condition is true
} else {
    // Code to execute if condition is false
}
```

Example:

```c
#include <stdio.h>

int main() {
    int age = 15;

    if (age >= 18) {
        printf("You are an adult.\n");
    } else {
        printf("You are a minor.\n");
    }

    return 0;
}
```

4.17.3 if-else if-else Statement

You can check multiple conditions using `if-else if-else`:

```c
if (condition1) {
    // Code to execute if condition1 is true
} else if (condition2) {
    // Code to execute if condition1 is false and condition2 is true
} else {
    // Code to execute if all conditions are false
}
```

Example:

```c
#include <stdio.h>

int main() {
    int score = 85;

    if (score >= 90) {
        printf("Grade: A\n");
    } else if (score >= 80) {
        printf("Grade: B\n");
    } else if (score >= 70) {
        printf("Grade: C\n");
    } else if (score >= 60) {
        printf("Grade: D\n");
    } else {
        printf("Grade: F\n");
    }

    return 0;
}
```

4.17.4 Nested if Statements

You can place `if` statements inside other `if` or `else` blocks:

```
if (condition1) {
    if (condition2) {
        // Code to execute if both conditions are true
    }
}
```

Example:

```
#include <stdio.h>

int main() {
    int age = 25;
    char hasLicense = 'Y';

    if (age >= 18) {
        if (hasLicense == 'Y') {
            printf("You can drive.\n");
        } else {
            printf("You need a license to drive.\n");
        }
    } else {
        printf("You are too young to drive.\n");
    }

    return 0;
}
```

4.17.5 Conditional (Ternary) Operator

The conditional operator is a shorthand for simple if-else statements:

```
result = (condition) ? value_if_true : value_if_false;
```

Example:

```c
#include <stdio.h>

int main() {
    int age = 20;
    char *status = (age >= 18) ? "adult" : "minor";

    printf("You are a %s.\n", status);

    return 0;
}
```

4.17.6 switch Statement

The `switch` statement selects one of many code blocks to execute:

```c
switch (expression) {
    case value1:
        // Code to execute if expression equals value1
        break;
    case value2:
        // Code to execute if expression equals value2
        break;
    default:
        // Code to execute if expression doesn't match any case
}
```

Example:

```c
#include <stdio.h>

int main() {
    int day = 3;

    switch (day) {
        case 1:
            printf("Monday\n");
            break;
        case 2:
            printf("Tuesday\n");
            break;
        case 3:
            printf("Wednesday\n");
            break;
        case 4:
            printf("Thursday\n");
            break;
        case 5:
            printf("Friday\n");
            break;
        case 6:
            printf("Saturday\n");
            break;
        case 7:
            printf("Sunday\n");
            break;
        default:
            printf("Invalid day\n");
    }

    return 0;
}
```

4.18 Loops

4.18.1 while Loop

The `while` loop executes a block of code as long as a specified condition is true:

```
while (condition) {
    // Code to execute while condition is true
}
```

Example:

```
#include <stdio.h>

int main() {
    int i = 1;

    while (i <= 5) {
        printf("%d ", i);
        i++;
    }
    // Output: 1 2 3 4 5

    return 0;
}
```

4.18.2 do-while Loop

The `do-while` loop is similar to the `while` loop, but it executes the code block once before checking the condition:

```
do {
    // Code to execute at least once
} while (condition);
```

Example:

```
#include <stdio.h>

int main() {
    int i = 1;

    do {
        printf("%d ", i);
        i++;
    } while (i <= 5);
    // Output: 1 2 3 4 5

    return 0;
}
```

4.18.3 for Loop

The `for` loop provides a concise way to write the loop structure:

```
for (initialization; condition; increment/decrement) {
    // Code to execute while condition is true
}
```

Example:

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; i++) {
        printf("%d ", i);
    }
    // Output: 1 2 3 4 5

    return 0;
}
```

4.18.4 Nested Loops

You can place loops inside other loops:

```c
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        // Code to execute
    }
}
```

Example:

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 3; i++) {
        for (int j = 1; j <= 3; j++) {
            printf("(%d,%d) ", i, j);
        }
        printf("\n");
    }

    return 0;
}
```

Output:

```
(1,1) (1,2) (1,3)
(2,1) (2,2) (2,3)
(3,1) (3,2) (3,3)
```

4.19 Jump Statements

4.19.1 break Statement

The `break` statement terminates the innermost enclosing loop or switch statement:

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 6) {
            break;  // Exit the loop when i equals 6
        }
        printf("%d ", i);
    }
    // Output: 1 2 3 4 5

    return 0;
```

4.20 Functions

A function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. With properly designed functions, it is possible to ignore how a job is done; knowing what is done is sufficient. C makes the sue of functions easy, convinient and efficient; you will often see a short function defined and called only once, just because it clarifies some piece of code. ![src]

A function definition has this form:

```c
return-type function-name(parameter declarations, if any)
{
 declarations
 statements
}
```

Rewrite function to calculate temperature

```c
#include <stdio.h>

// Function declaration (prototype)
int convert_temperature(float fahr);

int main() {
    int result = convert_temperature(5.0);
    printf("celsius: %d\n", result);
    return 0;
}

// Function definition
int convert_temperature(float fahr) {
    float celsius = 5*(fahr-32)/9;
    return celsius;
}
```

If you notice in above example there's something known as function declaration. This declaration, which is called a function prototype, has to agree with the definition and uses of power. It is an error if the definition of a function or any uses of it do not agree with its prototype. parameter names need not agree. Indeed, parameter names are optional in a function prototype, so for the prototype we could have written

```c
int convert_temperature(float);
```

Further Reading:

1. Chapter 4, C by K&R

5 Pointers

6 Pointers

6.1 Introduction:

Pointers require a separate chapter altogether because I feel pointers in C provides a powerfu mechanism to interact with memory management.

In short, `pointers` are variables which store memory address to other variables.

6.1.1 Pointers and Addresses

From C by K&R:

```
Let us begin with a simplified picture of how memory is organized. A typical
machine has an array of consecutively numbered or addressed memory cells that
may be manipulated individually or in contiguous groups. One common situation
is that any byte can be a char, a pair of one-byte cells can be treated as a
short integer, and four adjacent bytes form a long. A pointer is a group of
cells (often two or four) that can hold an address. So if c is a char and p
is a pointer that points to it, we could represent the situation this way:
```

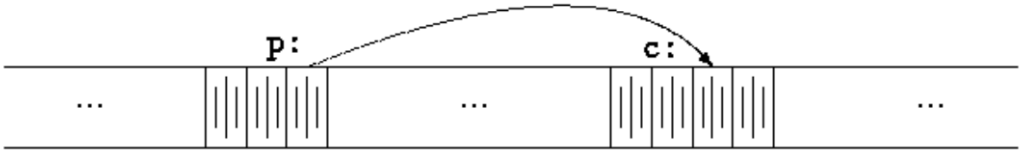


Figure 6.1: src : C by K&R

In order to make variable ‘p’ point to the memory address of variable ‘c’

```
p = &c;
```

The & operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or register variables.

Example code:

```
#include <stdio.h>

int main() {
    int a = 10; // integer a contains 10 as value
    int *p; //define a pointer variable p , * operator is the indirection or dereferencing op
            //accesses the object the pointer points to.

    p = &a; // this line means pointer p is said to 'point to' a
    printf("%d",p);

    return 0;
}

/*
From: https://www.programiz.com/c-programming/online-compiler/
Output:
1539960324

=== Code Execution Successful ===
*/
```

So in essence, `int x = 1` being a generic integer assignment can be operated on using a pointer variable as shown below

```
#include <stdio.h>

int main() {
    int a = 10; // integer a contains 10 as value
    int *p; //define a pointer variable p , * operator is the indirection or dereferencing op
            //accesses the object the pointer points to.

    p = &a; // this line means pointer p is said to 'point to' a
    printf("%d \n",p);

    //use a third variable B
    int b = *p; //this assigns the value at memory address 'pointed by' p to integer variable

    printf("B is : %d\n",b);

    *p=1;
    printf("a is : %d\n",a);
    printf("b is : %d\n",b);


    return 0;
}

/* output from https://www.programiz.com/c-programming/online-compiler/
1729765392
B is : 10
a is : 1
b is : 10


=== Code Execution Successful ===
*/
```

In no way this covers pointers, rather this is hardly an introduction. But the point it to learn by doing . I'll ensure to add some highlighted notes in further chapters where pointer play a key role.

Part II

2_storing_data_elements

7 Array

8 Numbers etc.

Let's think about this. Numbers have different way of getting represented. From Romans using letters (I, V, X, L, C, D, M) to Tally systems all were diffеrnt ways of putting thought of 'counts' in written form.

This brings us to the point of representation of numbers.

Table 8.1: Number representation

Number	1	2	3	4	5	10	100	1000
Roman	I	II	III	IV	V	X	C	M
Binary	1	10	11	100	101	1010	1100100	1111101000
Decimal	1	2	3	4	5	10	100	1000

The process of operating on different number representation is strongly associated with the need of grouping and reprenting larger numbers into compact forms. They key difference between these numbers is not just the symbol used for counts but also the `value perceived from each symbol` as shown in Table Table 8.1. [1]

8.0.0.1 Place holds Value

Roman numerals as good as it looks on fancy clock dials and book indexes still faced defeat when it came to writing larger numbers. The need for different letter arose when larger numbers were getting to difficult to record.

1. `'V'` was chosen to be 5 because it was harder to write `'IIIII'`.
2. Similarly `'X'` came in for 10 because it got harder to write 20 as `'VVVV'`.

But this approach still had one drawback. As the counting grew to thousands and tens of thousands there were lots of repetitions of the smaller denominations. `The oldest noteworthy inscription containing numerals representing very large numbers is on the Columna Rostrata, a monument erected in the Roman Forum to commemorate a victory in 260 bce over Carthage during the First Punic War. In this column a symbol for 100,000, which was an early form of (((I))), was repeated 23 times, making 2,300,000.`. [1]

Clearly there was need of more sophisticated ways which gave birth to the idea of "place value systems".

8.1 Basic Matrix multiplication

```
import <stdio.h>
```

References:

1. numerals and numeral systems

Reference: https://www.kodeco.com/books/metal-by-tutorials/v2.0/chapters/1-hello-metal

8.1.1 What is rendering?

The processing of an outline image using color and shading to make it appear solid and three-dimensional.

The process of Metal rendering is much the same no matter the size and complexity of your app, and you'll become very familiar with the following sequence of drawing your models on the screen:

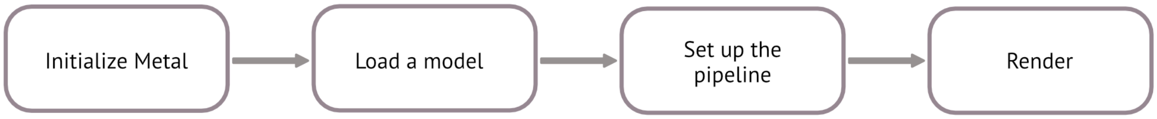


Figure 8.1: img

See Figure 9.1 for an illustration.

```
import PlaygroundSupport
import MetalKit
//PlaygroundSupport lets you see live views in the assistant editor, and MetalKit is a
//framework that makes using Metal easier
```

```
//check for whether GPU is available
guard let device = MTLCreateSystemDefaultDevice() else {
  fatalError("GPU is not supported")
}


/**
 This code configures an MTKView for the Metal renderer. MTKView is a subclass of
 NSView on macOS and of UIView on iOS. MTLClearColor represents an RGBA value
 - in this case, cream. The color value is stored in clearColor and is used to set the
 color of the view.

 */
let frame = CGRect(x: 0, y: 0, width: 600, height: 600)
let view = MTKView(frame: frame, device: device)

view.clearColor = MTLClearColor(red: 1, green: 1, blue: 0.8, alpha: 1)



/**
 Model I/O is a framework that integrates with Metal and SceneKit. Its main purpose
 is to load 3D models that were created in apps like Blender or Maya, and to set up
 data buffers for easier rendering.
 Instead of loading a 3D model, you'll load a Model I/O basic 3D shape, also called a
 primitive. A 3D primitive is typically a cube, a sphere, a cylinder or a torus.
 */
// 1
let allocator = MTKMeshBufferAllocator(device: device)
// 2
let mdlMesh = MDLMesh(sphereWithExtent: [0.75, 0.75, 0.75],
                      segments: [100, 100],
                      inwardNormals: false,
                      geometryType: .triangles,
                      allocator: allocator)
// 3
let mesh = try MTKMesh(mesh: mdlMesh, device: device)

guard let commandQueue = device.makeCommandQueue() else {
  fatalError("Could not create a command queue")
}

/**
```

```
 Ideally shader file should be separate but for now we are going with string format
 */
let shader = """
#include <metal_stdlib>
using namespace metal;

struct VertexIn {
  float4 position [[ attribute(0) ]];
};

vertex float4 vertex_main(const VertexIn vertex_in [[ stage_in ]]) {
  return vertex_in.position;
}

fragment float4 fragment_main() {
  return float4(1, 0, 0, 1);
}
"""

let library = try device.makeLibrary(source: shader, options: nil)
let vertexFunction = library.makeFunction(name: "vertex_main")
let fragmentFunction = library.makeFunction(name: "fragment_main")

let pipelineDescriptor = MTLRenderPipelineDescriptor()
pipelineDescriptor.colorAttachments[0].pixelFormat = .bgra8Unorm
pipelineDescriptor.vertexFunction = vertexFunction
pipelineDescriptor.fragmentFunction = fragmentFunction


pipelineDescriptor.vertexDescriptor =
     MTKMetalVertexDescriptorFromModelIO(mesh.vertexDescriptor)

let pipelineState =
  try device.makeRenderPipelineState(descriptor: pipelineDescriptor)

// 1
guard let commandBuffer = commandQueue.makeCommandBuffer(),
// 2
  let renderPassDescriptor = view.currentRenderPassDescriptor,
// 3
  let renderEncoder = commandBuffer.makeRenderCommandEncoder(descriptor:
    renderPassDescriptor)
else { fatalError() }
```

```
renderEncoder.setRenderPipelineState(pipelineState)

renderEncoder.setVertexBuffer(mesh.vertexBuffers[0].buffer,
                              offset: 0, index: 0)

guard let submesh = mesh.submeshes.first else {
  fatalError()
}


renderEncoder.drawIndexedPrimitives(type: .triangle,
                          indexCount: submesh.indexCount,
                          indexType: submesh.indexType,
                          indexBuffer: submesh.indexBuffer.buffer,
                          indexBufferOffset: 0)


// 1
renderEncoder.endEncoding()
// 2
guard let drawable = view.currentDrawable else {
  fatalError()
}
// 3
commandBuffer.present(drawable)
commandBuffer.commit()


PlaygroundPage.current.liveView = view
```

9 Matrix

9.0.1 Unified Memory

In order to get familiar with concept of unified memory, let's think about how CPU and GPU works under the hood. Each time CPU or GPU are called with a task, there is a 'quick' memory which loads/reloads data from secondary memory to quick accessible memory i.e. RAM. This is more like bringing data closer to compute. Now traditionally, the CPU and GPU had their rescpective RAM memories but unified memory brought the concept of shared pool of memory between GPU and CPU.

```
Unified Memory creates a pool of managed memory that is shared between the
CPU and GPU, bridging the CPU-GPU divide. Managed memory is accessible to
both the CPU and GPU using a single pointer. The key is that the system
automatically migrates data allocated in Unified Memory between host and
device so that it looks like CPU memory to code running on the CPU, and like
GPU memory to code running on the GPU.
```

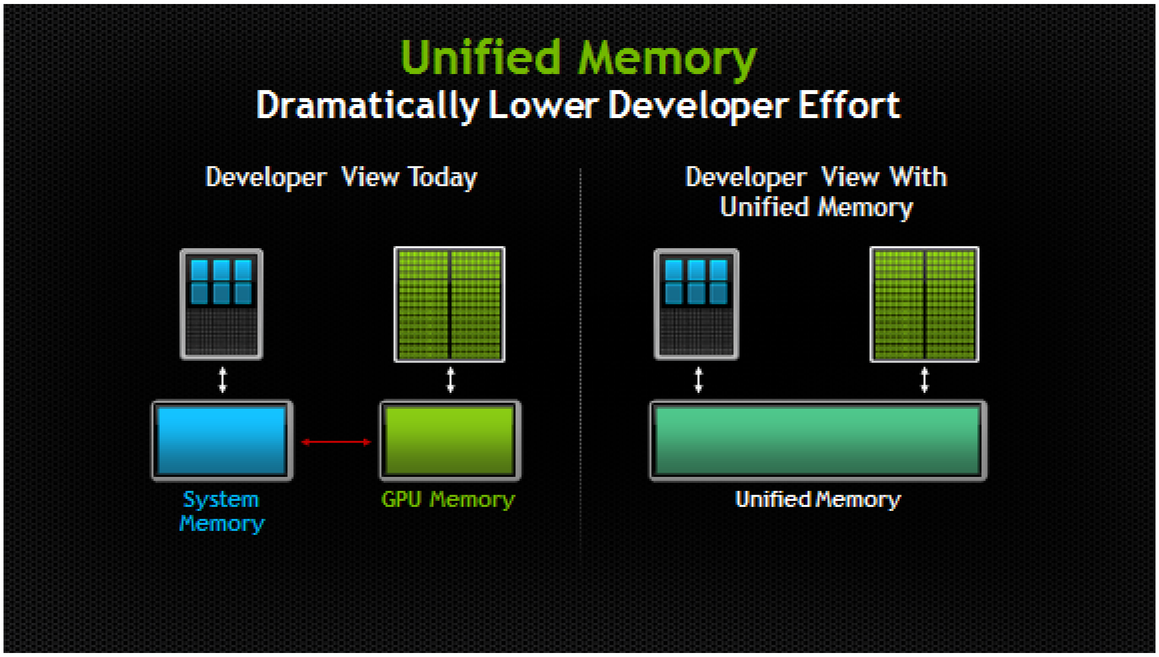
See Figure 9.1 for an illustration. src

9.0.2 Diving into GPU programming.

Let's start with basic vector addition in C++.

```
#include <iostream>
#include <math.h>

void add(int n,float *x ,float *y)
{
    for(int i = 0;i <n; i++)
    {
```



Unified Memory

Dramatically Lower Developer Effort

Developer View Today

Developer View With Unified Memory

Figure 9.1: Unified Memory

```
        y[i] = x[i] + y[i];
    }
}



int main (void)
{
    int N = 1<<20; //1M elements

    float *x = new float[N];
    float *y = new float[N];

    for (int i =0; i<N;i++)
    {
        x[i]=1.0f;
        y[i]=2.0f;
    }

    //add using cpu
    add(N, x ,y);
    // Check for errors (all values should be 3.0f)
     float maxError = 0.0f;
     for (int i = 0; i < N; i++)
       maxError = fmax(maxError, fabs(y[i]-3.0f));
     std::cout << "Max error: " << maxError << std::endl;

     // Free memory
     delete [] x;
     delete [] y;

     return 0;
    }
```

A quick run in local cpp environment . Refer this to setup C++ compiler on your local machine

```
g++ matrix_mul.cpp -o add
./add
Output:
Max error: 0
```

Now the naive addition let's walk through the C++ code **step by step** to understand what it does and how it works.

Header Files

```
#include <iostream>
#include <math.h>
```

- `#include <iostream>`: Used for input and output, e.g., `std::cout`.
- `#include <math.h>`: Includes math functions like `fabs()` and `fmax()`.

Function: `add`

```
void add(int n, float *x, float *y)
{
    for (int i = 0; i < n; i++)
    {
        y[i] = x[i] + y[i];
    }
}
```

This function:

- Takes `n` elements of two float arrays: `x` and `y`.
- For each index `i`, it adds `x[i]` to `y[i]` and stores the result back in `y[i]`.

So, it's **adding vectors** `x` and `y` element-wise, storing the result in `y`.

Main Function

```
int main (void)
```

The entry point of the program.

Step 1: Declare and Allocate Memory

```
int N = 1 << 20; // 1 million elements
float *x = new float[N];
float *y = new float[N];
```

- `1 << 20` means `2^20 = 1,048,576`.
- Allocates two arrays `x` and `y` of size 1 million using `new`.

Step 2: Initialize Arrays

```
for (int i = 0; i < N; i++)
{
    x[i] = 1.0f;
    y[i] = 2.0f;
}
```

- Fills the `x` array with `1.0f` and the `y` array with `2.0f`.
- So before addition:
  - `x = [1.0, 1.0, ..., 1.0]`
  - `y = [2.0, 2.0, ..., 2.0]`

Step 3: Add Arrays Using CPU

```
add(N, x, y);
```

Calls the `add` function:

- `y[i] = x[i] + y[i] = 1.0 + 2.0 = 3.0` Now `y = [3.0, 3.0, ..., 3.0]`

Step 4: Check for Errors

```
float maxError = 0.0f;
for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i] - 3.0f));
```

- Checks that each value in `y` is exactly `3.0`.
- `fabs(y[i] - 3.0f)` measures the absolute error.
- `fmax` keeps the **maximum** error encountered.
- For correct execution, `maxError` should be `0.0`.

Step 5: Print Result

```
std::cout << "Max error: " << maxError << std::endl;
```

Prints the maximum error (should be 0.0 if all additions were perfect).

Step 6: Free Allocated Memory

```
delete [] x;
delete [] y;
```

- Frees the dynamically allocated memory to prevent memory leaks.

Now to run this code on GPU cores (using Metal framework), it requires some changes.

9.0.3 What exactly is a .metal file?

A .metal file (aka shader) is Apple's native built alternative to OpenGL (deprecated).

1. The first change is to create a .metal file. In the context of Apple GPUs, "metal file" typically refers to as Metal Shader (Metal Shader Language), which is a small program that runs on the GPU to perform rendering operations. MSL is a variant of C++ designed for GPU programming. In Metal, code that runs on GPUs is called a shader, because historically they were first used to calculate colors in 3D graphics. Metal is Apple's low-level graphics API that allows developers to harness the full power of the GPU for various tasks, including rendering, image processing, and compute operations.

```
#include <metal_stdlib>
using namespace metal;

kernel void vector_add(
    const device float *x [[buffer(0)]],
    const device float *y [[buffer(1)]],
    device float *result [[buffer(2)]],
    uint id [[thread_position_in_grid]]
) {
    result[id] = x[id] + y[id];
}
```

Unpacking above program.

Step 2: Set Up C++/Objective-C++ Host Code Save this as main.mm (note .mm for Objective-C++):

```
#import <Metal/Metal.h>
#include <iostream>
#include <vector>

int main() {
    const int N = 1 << 20;

    id<MTLDevice> device = MTLCreateSystemDefaultDevice();
    id<MTLCommandQueue> queue = [device newCommandQueue];

    // Load Metal library
    NSError *error = nil;
    NSString *path = @"addVectors.metallib"; // Precompiled Metal library
    id<MTLLibrary> library = [device newLibraryWithFile:path error:&error];
    id<MTLFunction> func = [library newFunctionWithName:@"vector_add"];
    id<MTLComputePipelineState> pipeline = [device newComputePipelineStateWithFunction:func e

    // Host arrays
    std::vector<float> x(N, 1.0f), y(N, 2.0f), result(N);

    // GPU buffers
    id<MTLBuffer> xBuf = [device newBufferWithBytes:x.data() length:N * sizeof(float) options
    id<MTLBuffer> yBuf = [device newBufferWithBytes:y.data() length:N * sizeof(float) options
    id<MTLBuffer> resBuf = [device newBufferWithLength:N * sizeof(float) options:0];

    id<MTLCommandBuffer> cmdBuf = [queue commandBuffer];
    id<MTLComputeCommandEncoder> encoder = [cmdBuf computeCommandEncoder];
    [encoder setComputePipelineState:pipeline];
    [encoder setBuffer:xBuf offset:0 atIndex:0];
    [encoder setBuffer:yBuf offset:0 atIndex:1];
    [encoder setBuffer:resBuf offset:0 atIndex:2];

    MTLSize gridSize = MTLSizeMake(N, 1, 1);
    NSUInteger threadGroupSize = pipeline.maxTotalThreadsPerThreadgroup;
    MTLSize threadgroupSize = MTLSizeMake(threadGroupSize, 1, 1);

    [encoder dispatchThreads:gridSize threadsPerThreadgroup:threadgroupSize];
    [encoder endEncoding];
    [cmdBuf commit];
    [cmdBuf waitUntilCompleted];

    float *resData = (float *)resBuf.contents;
    float maxError = 0.0f;
    for (int i = 0; i < N; i++) {
        float err = fabs(resData[i] - 3.0f);
        if (err > maxError) maxError = err;
    }
    std::cout << "Max error: " << maxError << std::endl;

    return 0;
}
```

9.0.3.1 Step 3: Compile and Run

9.0.3.1.1 A. Compile the Metal shader:

```
xcrun -sdk macosx metal -c addVectors.metal -o addVectors.air
xcrun -sdk macosx metallib addVectors.air -o addVectors.metallib
```

9.0.3.1.2 B. Compile the main program:

```
clang++ -std=c++17 -framework Metal -framework Foundation main.mm -o vector_add
```

9.0.3.1.3 C. Run it:

```
./vector_add
```

You should see:

```
Max error: 0
```

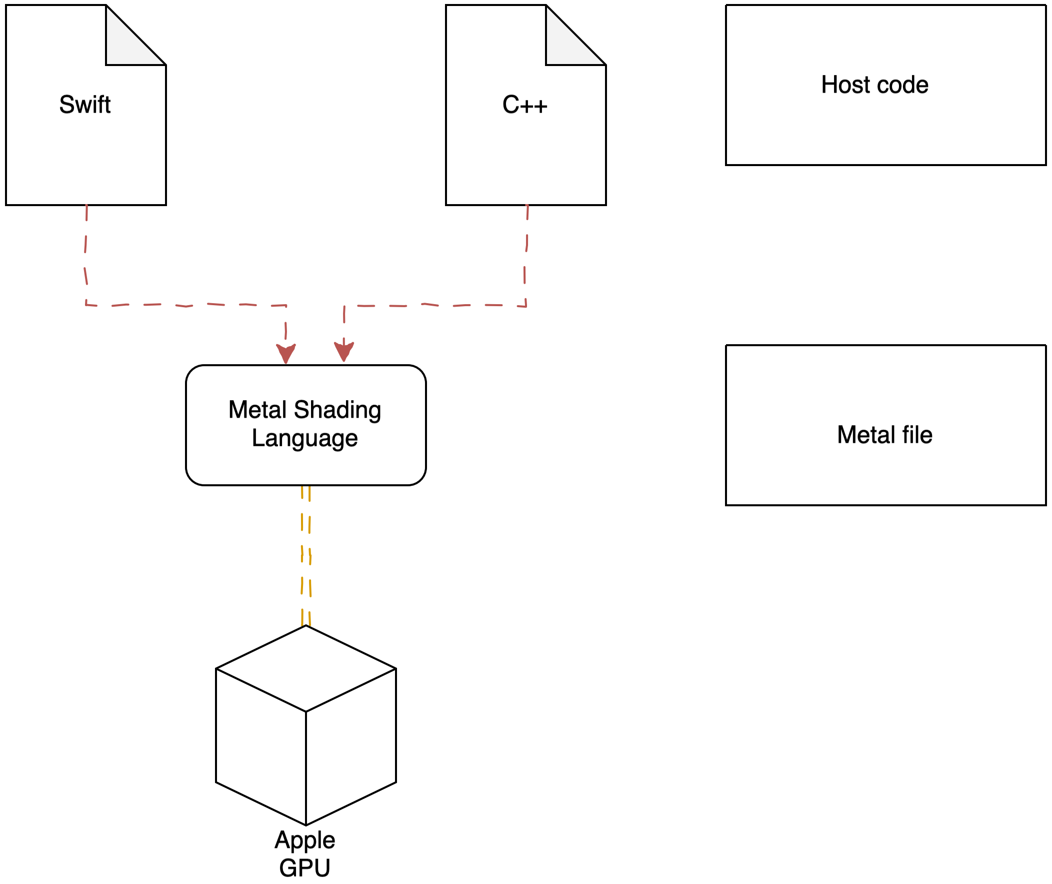


Figure 9.2: MSL

9.0.3.2 Template

Component	Purpose	Changes
`.metal` shader	Runs on GPU	Change `result[id] = ...` logic
`.metallib`	Compiled shader	Recompile when shader changes
`main.mm`	Host-side driver	Usually unchanged (optional name changes)
`clang++` + `xcrun`	Compile tools	is Fixed

Now Let's bind everything up as a resuable piece for plugging in metal shaders for other operations.

Part III

Scratching the Surface

10 Understanding Metal Pipeline

So let's take it from the top.

10.0.1 What exactly is a .metal file?

A .metal file (aka shader) is Apple's native built alternative to OpenGL (deprecated).

1. The first change is to create a .metal file. In the context of Apple GPUs, "metal file" typically refers to as Metal Shader (Metal Shader Language), which is a small program that runs on the GPU to perform rendering operations. MSL is a variant of C++ designed for GPU programming. In Metal, code that runs on GPUs is called a shader, because historically they were first used to calculate colors in 3D graphics. Metal is Apple's low-level graphics API that allows developers to harness the full power of the GPU for various tasks, including rendering, image processing, and compute operations.

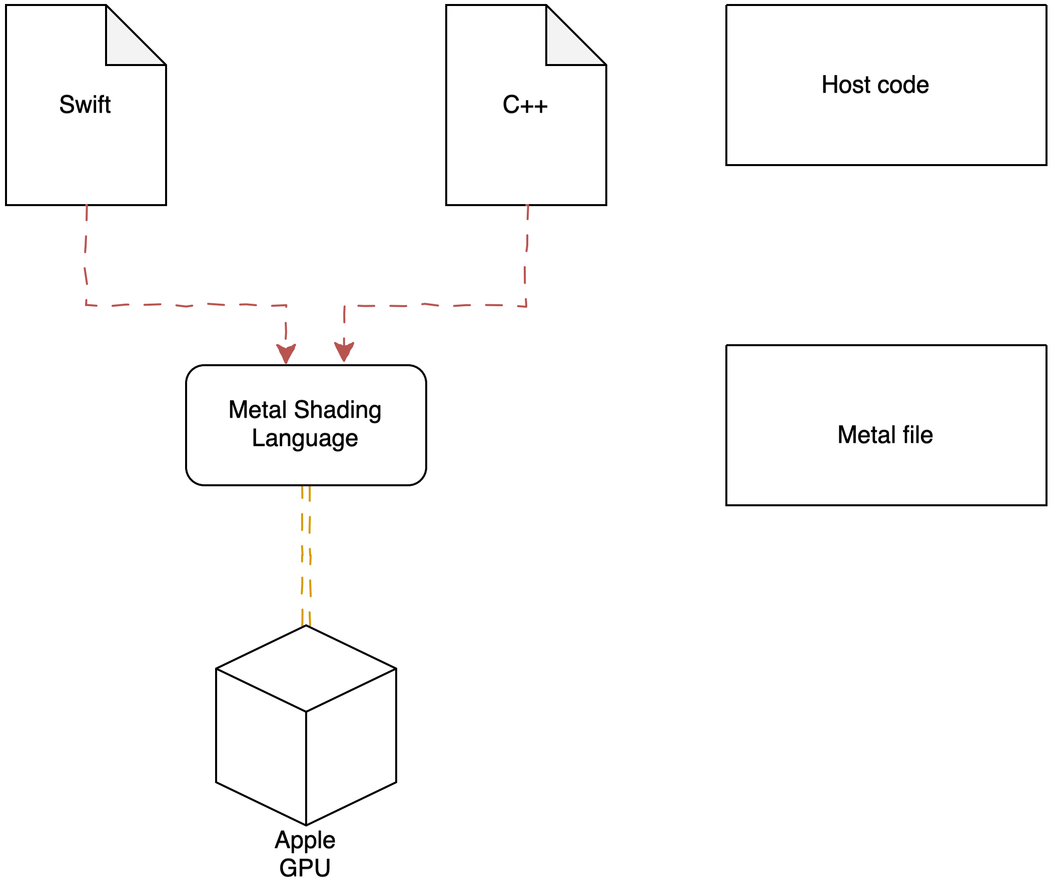


Figure 10.1: MSL

Resources:

1. youtube video on how GPU's work
2. 30 days of Metal
3. Apple docs
4. Apple Metal Language specification

Part IV

Design Elements

11